Distributed Social Network Analyzer

Joel Cemeli Sanchez

June 2017

Contents

1 Introduction **7**

2 Apache Spark **8**
2.1 Introduction . . . 8
2.2 Core . . . 8
2.3 Cluster Manager and Distributed Storage System . . . 8
2.4 MLlib . . . 9
2.4.1 Provided tools . . . 9
2.5 GraphX . . . 10
2.5.1 Structure . . . 10
2.6 RDD . . . 10
2.6.1 Characterization . . . 11

3 Apache Hadoop **12**
3.1 History . . . 12
3.2 Modules . . . 13
3.3 Why Hadoop? . . . 14
3.4 How to setup . . . 14
3.4.1 HDFS . . . 15
3.4.2 Architecture . . . 15
3.4.3 Configure and run on Fedora . . . 15
3.4.4 Accessibility . . . 18
3.5 YARN . . . 18
3.5.1 Architecture . . . 19
3.5.2 Configure and run on Fedora . . . 20
3.5.3 Accessibility . . . 21
3.6 Connect with spark . . . 21

4 Scala programming language **22**
4.1 Introduction . . . 22
4.2 Description . . . 22
4.3 Main features . . . 22
4.4 Compiling . . . 23
4.4.1 Program structure . . . 23
4.4.2 Sbt file and External dependencies . . . 24

5 Graphs: structure and operations **25**
5.1 Graph RDD . . . 25
5.2 Graph structure . . . 25
5.3 Operating with graphs: Pregel . . . 26
5.3.1 How it works? . . . 26
5.3.2 SuperSteps . . . 26
5.3.3 Code structure . . . 26

6 Json parsing **28**
6.1 Introduction . . . 28
6.2 How it works? . . . 28
6.3 Example . . . 28

7 Conversation downloaders **29**
7.1 Introduction . . . 29
7.2 Obtaining conversations . . . 29
7.2.1 Twitter Downloader . . . 29
7.2.2 Reddit Downloader . . . 29
7.3 Behaviour . . . 29
7.4 Download Manager . . . 30

8 SocialGraph **31**
8.1 What is a SocialGraph? . . . 31
8.2 Data Structure . . . 31
8.2.1 Vertex . . . 31
8.2.2 Edges . . . 31
8.2.3 Graph . . . 32
8.3 Creating a graph . . . 32
8.3.1 From raw data . . . 32
8.3.2 From Json . . . 32
8.4 SubGraphs . . . 32
8.4.1 How it works? . . . 32

9 TwitterGraph **33**
9.1 Introduction . . . 33
9.2 Main behaviour . . . 33
9.3 Conversations in Twitter . . . 33
9.4 Mentions and extra connections . . . 33
9.5 Example . . . 34
9.6 Weight . . . 35

10 RedditGraph **36**
10.1 Introduction . . . 36
10.2 Messages structure . . . 36

11 Graph Representations **37**
11.1 What and why? . . . 37
11.2 Representations . . . 37

12 Pregel algorithms **38**
12.1 Starting with pregel . . . 38
12.2 Remember peculiarities . . . 38
12.3 First algorithms . . . 38
12.3.1 Minimum Distance . . . 38
12.3.2 Maximum Distance . . . 39
12.3.3 Average Distance . . . 40

13 Reasoning calculus **43**
13.1 Introduction . . . 43
13.2 Defeaters Counter Algorithm . . . 43
13.3 Original implementation, with message counting . . . 44

14 Tests **46**
14.1 Introduction . . . 46
14.2 Basic Tests . . . 46
14.2.1 SocialGraph . . . 46
14.2.2 TwitterGraph . . . 47
14.2.3 Reddit . . . 47
14.3 Distributed Graph Tests . . . 47
14.4 Graph MPC Tests . . . 47
14.5 Serialization Tests . . . 47

15 Structure and Behaviour **48**
15.1 Graph . . . 48
15.2 Defeaters Calculus Flow . . . 48

16 Obtained results **51**
16.1 The results . . . 51
16.2 Commented results . . . 51
16.2.1 Time . . . 51
16.2.2 Messages . . . 51

Appendices **57**

A Research paper **58**

List of Figures

2.1 Triplet representation from [15] 10

3.1 Products and frameworks built on top of YARN from [1] 13
3.2 HDFS Architecture from [2] 16
3.3 YARN Architecture from [3] 19

15.1 UML diagram for all graph Modules 49
15.2 Flow diagram from graph creation to defeaters calculation 50

16.1 Graphic representing the results for Non Distributed execution of Original Defeaters Counter 53
16.2 Graphic representing the results for Distributed execution of Original Defeaters Counter 54
16.3 Graphic representing the results for Non Distributed execution of Optimized proposal for Defeaters Counter 54
16.4 Graphic representing the results for Distributed execution of Optimized proposal for Defeaters Counter 55
16.5 Graphic representing the messages sent by execution of Original Defeaters Counter 55
16.6 Graphic representing the messages sent by Distributed execution of Optimized proposal for Defeaters Counter 56

List of Tables

16.1 Original Non-Distributed 52
16.2 Original Distributed 52
16.3 Optimized Non-Distributed 52
16.4 Optimized Distributed 53

Acknowledgments

Special agreements to dr.Jordi Planes and dr.Ramon Bejar for proposing the project, giving us all the needed resources, and their interest on the progress of the project. Also to Cristian Sanahuja, with whom I have been working on the project for all these months and it has been very helpful.

Chapter 1

Introduction

This project consists on a set of tools to analyze SocialNetwork users interactions. Also, a cluster was build and configured, and some experiments was performed on it with this project, that are also commented on the document.

The entire software consist on different parts, abstractions of social network graphs as SocialGraphs, downloaders managers, classifiers, multiple graphs representations, operations over the graphs.

It can be seen as two main parts, **Learning** implemented by Cristian Sanahuja, that determines the intention of the interactions between users in a conversation and **Graphs** implemented by Joel Cemeli, that generates graphs from downloaded conversations, and perform calculus with them using pregel algorithm design.

Also there will be explained all the environment used, programming languages, and get deep on the Graphs part, its structure and computation, with special mention to pregel, a Google programming technique for graphs. The results of the experiments are shown and explained, comparing two algorithms on a single-node and on multiple nodes.

A brief guide it's included in order to program using this set of tools, and also expanding them, and a few recommendations.

Finally a paper about this reasoning was sent to IJCAI (International Joint Conference on Articial Intelligence) on Melbourne, Australia, that can be found on the annex. See page 58

Chapter 2

Apache Spark

2.1 Introduction

It's a general-purpose open-source cluster computing system centered on RDD (Resilient Distributed Dataset) originally developed at the University of California, Berkeley's AMPLab, on 2009.

It provides high-level API for the following programming languages: Java, Python, R and Scala. It gives high-level tools like SparkSQL for SQL and structured data processing, Spark Streaming and MLlib for machine learning and GraphX for graph processing (the last two are used in the project).

It be used in standalone mode on a single machine, in order to do develop and testing, on this mode, the distributed storage is not required. In cluster mode the distributed storage is mandatory, the program and all the fields will be taken from there.

2.2 Core

The core consists on a task dispatching, scheduling and basic I/O functionalities, centered on the RDD abstration and operations with them. Every operation on an RDD like map, filer or reduce and joins produce a new RDD, as this structures are inmutable (in order to be treated distributed).

2.3 Cluster Manager and Distributed Storage System

The RDD's represents an improve for the latency, compared to Map Reduce, is reduced in several orders of magnitude, because in it's case this algorithms visit their dataset multiple times in a loop, and interactive/exploratory data analysis.

Spark provides a default cluster manager, but not distributed storage (Spark can work without it, in order to develop and test on a single machine). On both cases Spark supports externals systems:

Cluster Manager support:

- Native Spark Cluster
- Hadoop YARN
- Apache Mesos

Distributed Storage System interface with:

- Hadoop Distributed File System (HDFS)
- MapR File System (MapR-FS)
- Cassandra
- OpenStack Swift
- Amazon S3
- Kudu
- Custom implementation

Spark can interface with all distributed storage systems. Also, it means that a custom solution can be implemented with the same interface.

2.4 MLlib

MLlib (*Machine Learning library*) it's part of Spark's API. It provides common learning algorithms and utilities like classification, regression, clustering, collaborative fileting, dimensionality reduction and also underlying optimization primitives.

It's supported on all Spark compatible programming languages (Java, Scala, Python and R).

The purpose of this library is to provide practical machine learning scalable and easy at high level.

2.4.1 Provided tools

- **ML Algorithms**: Common learning algorithms (classification, regression, clustering, collborative filtering...)
- **Featurization**: Extraction of features, transformation, dimensionality reduction and selection.
- **Pipelines**: Constructing, evaluating and turning ML pipelines.
- **Persistence**: saving and load algorithms, models and pipelines.
- **Utilities**: linear algebra, statistics, data handling, etc...

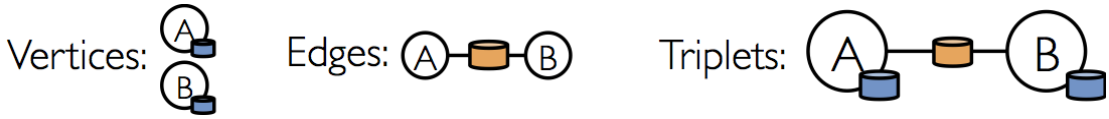


Figure 2.1: Triplet representation from [15]

2.5 GraphX

It's the high-level API that provides graph computing on Spark and the reason because of the project is written in Scala. This component provides and entire library extending the Spark RDD with a Graph abstraction for creating directed multigraphs as RDD's and working with them on parallel computation with some utilities and functionalities like pregel. It's basic behaviour allows to create a cluster with one master and multiple slaves and submit jobs to it.

GraphX allows to create inmutable graphs as RDD's (in order to operate them on the cluster). Also it provides a bunch of features and attributes to that Graphs.

Main provided features:

- Information about the Graph
- Views of the graph as collections
- Functions for caching graphs
- Change the partitioning heuristic
- Transform vertex and edge attributes
- Modify the graph structure (getting subgraphs, or alternative representations, graph is an RDD, so it's inmutable).
- Join RDDs with the graph
- Aggregate information about adjacent triplets
- Iterative graph-parallel computation
- Basic graph algorithms

2.5.1 Structure

Graphs are composed by two RDD's, the vertex and the edges, and when creating a graph, this converge into a "linked" representation of that RDD's, that can be seen as a set of triplets, with two nodes (with it's properties), linked with an edge that can also has its own properties. See figure 2.1

2.6 RDD

An RDD is a Resilient Distributed Dataset, in other words, a read-only multiset of data items distributed over a cluster (maintained in a fault-tolerant way) that can be operated on in parallel.

Is the basic abstraction in Spark and it contains all the basic operations available on them like map, filer and persist. Special types of RDD (like key-value pairs) contains extra methods such as PairRDDFunctions...

2.6.1 Characterization

Each RDD is defined by five main properties:

- List of partitions
- A function for computing each split
- A list of dependencies on other RDDs
- Partitioner for key-value RDDs **(optionally)**
- A list of preferred locations to compute each split **(optionally)**

Spark allows each RDD to implement its own way of computing itself, also user can implement custom RDDs.

Chapter 3

Apache Hadoop

Apache Hadoop it is a framework used to distribute and process large amounts of data. It is also an open-source technology and Java-based programming.

One of the key design elements was to think that failures are part of the routine, this is why all modules try to handle automatically them. Hadoop bases its computing in Commodity cluster computing.

3.1 History

What we know today as Hadoop was started back in 2002 by then-Internet Archive search director Doug Cutting and University of Washington graduate student Mike Cafarella. They started a project called Nutch that was intended to be a better search engine.

They started to develop Nutch that was able to crawl and index hundreds of millions of page. However the deployment was not easy and only could run across a handful of machines plus someone needed to check it all day to make sure it did not fall.

On 2003 paper called "Google File System" was published. And later on 2004 one called "MapReduce: Simplified Data Processing on Large Clusters". These papers would be revelatory to them so they started to develop and underlying framework where they generalize all the steps they were doing manually to be automated.

On 2006 Cutting went to work with Yahoo, which was also impressed by the Google File System and MapReduce papers and wanted to build an open source technology based on them. So Cutting created a new project named Hadoop and included the distributed file-system renamed HDFS (Hadoop Distributed File System) and MapReduce of Nutch. To have on April the first release of Hadoop.

At this point Yahoo started a transition to use Hadoop and on 2007 they said to have Hadoop running on 1000 nodes. Around this time, Twitter, Facebook, LinkedIn and many others started using Hadoop and contributing back to the open source ecosystem.

At the same time they found out that MapReduce had too many responsibilities. It was built above HDFS (Hadoop distributed file system) layer and its charges were to assigning cluster

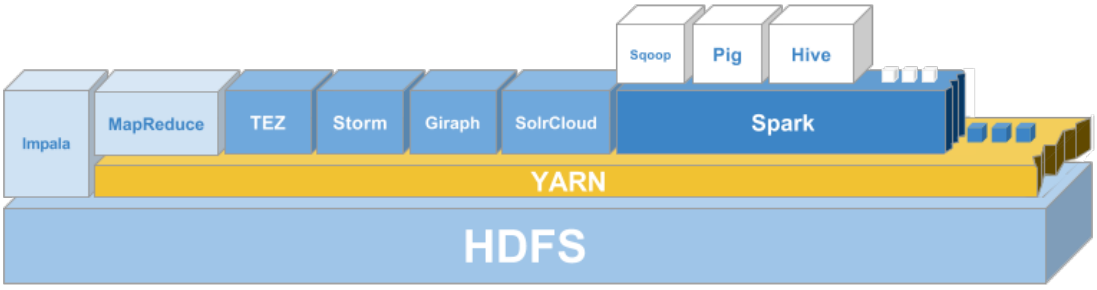


Figure 3.1: Products and frameworks built on top of YARN from [1]

resources and managing job execution, doing data processing and interfacing towards clients (API). That caused that other high level frameworks had to be build over MapReduce.

In order to solve that MapReduce was split, pulling out the code-base and decoupling cluster operations from data pipeline. Then to generalize processing capability, the resource management, work-flow management and fault-tolerance components were removed from MapReduce, and transferred into YARN. That is how YARN became a new layer between HDFS and MapReduce where new frameworks could be developed as Spark was later, check the Figure 2.1 on page 10.

Since then to nowadays Hadoop have keep evolving, in 2012 Yahoo's Hadoop Cluster have 42.000 nodes and Hadoop contributors reaches 1200. Hadoop it is used by more than 1000 companies in which we can find companies as: Google, Facebook, Twitter, Amazon, IBM, etc.

3.2 Modules

The Hadoop framework have at least 4 separate modules here we will give a brief description about them:

- Hadoop Common: contains all the libraries needed by other modules, as it provides abstraction of the underlying operative system and file system.
- Hadoop Distributed File System (HDFS): as its name indicates it is a distributed file system which is scalable and portable.
- Hadoop YARN (Yet Another Resource Negotiator): by its name a technology that administrates resources. It is a modification of old MapReduce that turns YARN into a tool of cluster management that allow to launch more applications with different kinds of processing. Sometimes referred as MapReduce 2.0
- Hadoop MapReduce: an implementation of the MapReduce programming model for large-scale data processing. Since the rework runs over YARN.

3.3 Why Hadoop?

As we have seen in the previous section Hadoop have different modules. In this project along with Spark we have used HDFS and YARN. But what are the advantages that these modules offer?

In first place we need will focus on HDFS. When we were first executing our jobs with Spark we encounter the problem or the inconvenient to need to place each file accessed by our program into all nodes on the same path. To avoid that behavior we use HDFS that allows us to place the files there and they will be accessible by all nodes making the manual task of placing files into all nodes not only less incommode but unnecessary.

And in second place we will explain why to use YARN. As we have mentioned before, Spark comes with its own standalone cluster which is ideal to start working and practise as its deploy it is trivial. However, when the things become more tedious and complex having YARN or Mesos resource managers are a must.

We have decide to use YARN in other to approach to a real situation in a business where a YARN cluster may also be present and Spark can run over it with no difficulties. Although to our small cluster where no multiple applications are launched we would had enough with standalone.

3.4 How to setup

In this chapter we will explain more accurately how install Hadoop on a Linux environment and more specifically on a Fedora system. On the following chapters of HDFS and YARN we will detail more how to configure each module.

First of all we will download the binary package of Hadoop, you can do it from this `http://hadoop.apache.org/releases.html` on June, 2017. We have used the version 2.7.3.

Once we have download it we just need to unpack it in our home and set some environment variables. In order to add this variables we need to edit the */.bashrc* file and append the following lines:

```
export JAVA_HOME=/usr/lib/jvm/jre-1.8.0-openjdk
export HADOOP_PREFIX=/home/{path-to-hadoop}/hadoop-2.7.3
export HADOOP_HOME=$HADOOP_PREFIX
export HADOOP_COMMON_HOME=$HADOOP_PREFIX
export HADOOP_CONF_DIR=$HADOOP_PREFIX/etc/hadoop
export HADOOP_HDFS_HOME=$HADOOP_PREFIX
export HADOOP_MAPRED_HOME=$HADOOP_PREFIX
export HADOOP_YARN_HOME=$HADOOP_PREFIX
```

And we can see we just need to specify some variables of Hadoop and where the Java Home is. With this we have Hadoop ready to be configured and executed. Check the sections 3.4.3 and 3.5.2 to see how to configure HDFS and YARN correspondingly.

A site note, when running HDFS and YARN on nodes we may encounter problems with like *'Connection Refused'* to solve that it is necessary to able the remote log-in. Under Fedora we did that going to Settings, Sharing and enable Remote Log-in.

3.4.1 HDFS

The Hadoop distributed file system it is a distributed file system primary used by Hadoop applications. This file system it is designed to run on commodity hardware which means it is designed to have highly fault-tolerant and be able to run on low-cost hardware. It is written in Java which makes it supported on all major platforms. And supports shell-like commands to interact with.

3.4.2 Architecture

The architecture of the HDFS is the normal architecture of a cluster. We have a master and various slaves. So in essence we have to kind of nodes the NameNode and the DataNode. Check the Figure 3.1 on page 13 to see a graphical representation of the architecture.

The NameNode is a master node that manages the file system name-space, saves the metadata and regulates the access to files. Usually we have one of this nodes per cluster.

The DataNode a slave node that manage and stores the files. Usually we have one DataNode in each node we want to store files.

Internally the files are split in blocks and stored in DataNodes, the blocks of the files are replicated for fault tolerance, we can configure the replication factor of each file and change it later.

Periodically the NameNode receive a Heartbeat and a Blockreport. The Heartbead means the DataNode is working properly and the Blockreport is a list of the blocks it contains. If some DataNode fails NameNode will order to replicate the necessary files to other DataNode. Note that in this architecture the files never go through the NameNode.

3.4.3 Configure and run on Fedora

Once we have download Hadoop like we explained on section 3.4 we need to change some configuration to make it working. These configurations will be done just by changing some files.

Before starting we need to remember the difference between the NameNode and the DataNode. Typically we will want one node the be the master and the rest to be the slaves so we have to make different configurations on each node.

The two files we have to change are the next:

- `$HADOOP_PREFIX/etc/hadoop/hdfs-site.xml`
- `$HADOOP_PREFIX/etc/hadoop/core-site.xml`

First we will start with the configuration in the *core-site.xml* as it is the same on master and slaves. We need to add the following code:

```
<configuration>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://udlnet-05-108.udl.net/</value>
```

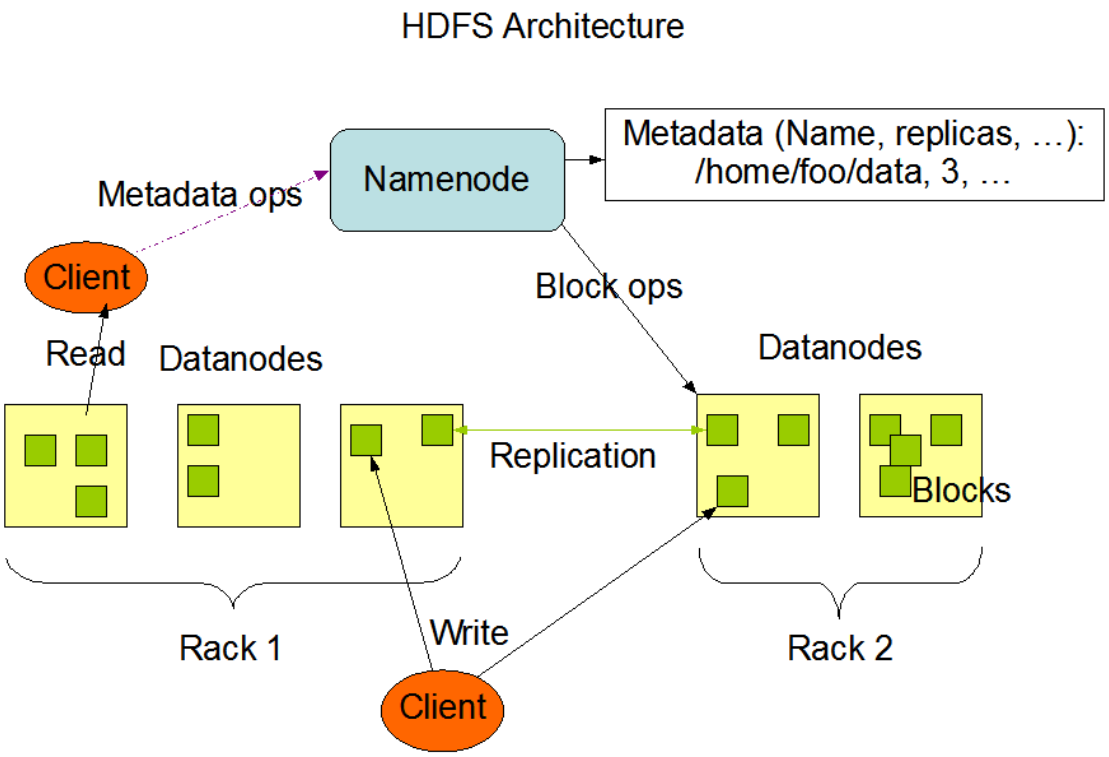


Figure 3.2: HDFS Architecture from [2]

```
        <description>NameNode URI</description>
    </property>
</configuration>
```

With this lines we are specifying which is the address of the master where the NameNode will be running. As example we show the configuration we used, we should change the text inside the value label to our master address.

Once we do it we need to modify the *hdfs-site.xml* file. In this file we will specify the NameNode and the DataNode.

```
<configuration>
    <property>
        <name>dfs.datanode.data.dir</name>
        <value>file:////home/scala-cluster/hadoop-2.7.3/hdfs/datanode</value>
        <description>Comma separated list of paths on the local filesystem of a DataNode
            where it should store its blocks.</description>
    </property>

    <property>
        <name>dfs.namenode.name.dir</name>
        <value>file:////home/scala-cluster/hadoop-2.7.3/hdfs/namenode</value>
        <description>Path on the local filesystem where the NameNode stores the namespace
             and transaction logs persistently.</description>
    </property>
</configuration>
```

As we see in the lines above we put the address where our DataNode and our NameNode are. At this point we need to understand that putting the property of NameNode on a slave it is not necessary as it will not make any difference and we could avoid it. As well if we just want one node to have a NameNode and not a DataNode to store information we could remove the other entry. That is why we said we needed different configurations depending what we want in the node, however it is also true that we can just copy the code above to the master and slaves and everything will just work.

Next step after configuration is to make HDFS be alive. The very first thing we need to do is to give format to the HDFS with the following command:

```
$HADOOP_PREFIX/bin/hdfs namenode -format
```

After that we have a set of scripts in Hadoop folder inside sbin folder. There are two ways we can start the HDFS. First with this command:

```
$HADOOP_PREFIX/sbin/start-dfs.sh
```

This command will try to start the NameNode and the DataNode daemons, this command can be used if a node contains both types of nodes. However it is recommended to start the daemons separately as not in all nodes we need to start NameNode. Actually if we try to execute the above command on a slave after master is already running we will receive the output of daemon already started as we will be trying to start a NameNode of the master. In conclusion it is better to use the commands below:

- To start the NameNode:

```
$HADOOP_PREFIX/sbin/hadoop-daemon.sh start namenode
```

- To start the DataNode:

```
$HADOOP_PREFIX/sbin/hadoop-daemon.sh start datanode
```

3.4.4 Accessibility

Now we have configure successfully our HDFS and have it running we may want to know how to check its state and how to interact with. In this section we will present the web interface and the shell commands.

The web interface is a web we can access with any browser on the localhost:50070 from the master and also from the slaves replacing localhost for the IP address of the master. I.e: `http://udl-net-05-108.udl.net:50070`. In that web we can obtain some information about the HDFS, things as the file system itself, the different folders and files there are, and the DataNodes running.

HDFS also offers a shell-like commands. In Hadoop folder we can run the command *bin/hdfs dfs* to execute different commands similar to shell. Here we will list the most command:

- To create a directory named folder:

```
$HADOOP_PREFIX/bin/hdfs dfs -mkdir /folder
```

- To remove a directory named in this path named /folder2:

```
$HADOOP_PREFIX/bin/hdfs dfs -rm -R /path/folder2
```

- To upload local files to the HDFS:

```
$HADOOP_PREFIX/bin/hdfs -put /localpath/localfile.txt /hdfspath/
```

Note that when Spark will look up for files in the file system will look up in the following directory /user/username/ where username is the current logged user. So if we want to upload files that then we will access from Spark we should upload them inside a folder the path /user/username/ directories we have to create previously.

3.5 YARN

YARN is a cluster technology that controls the resource management and job scheduling/monitoring. As explained briefly before that module comes from the separation of old MapReduce module where resource management and scheduling capabilities where together mixed with data processing limiting the type of application we could submit. This modification brings a new way to applications to use Hadoop system resources. Like HDFS and all Hadoop modules it is made to run on commodity hardware.

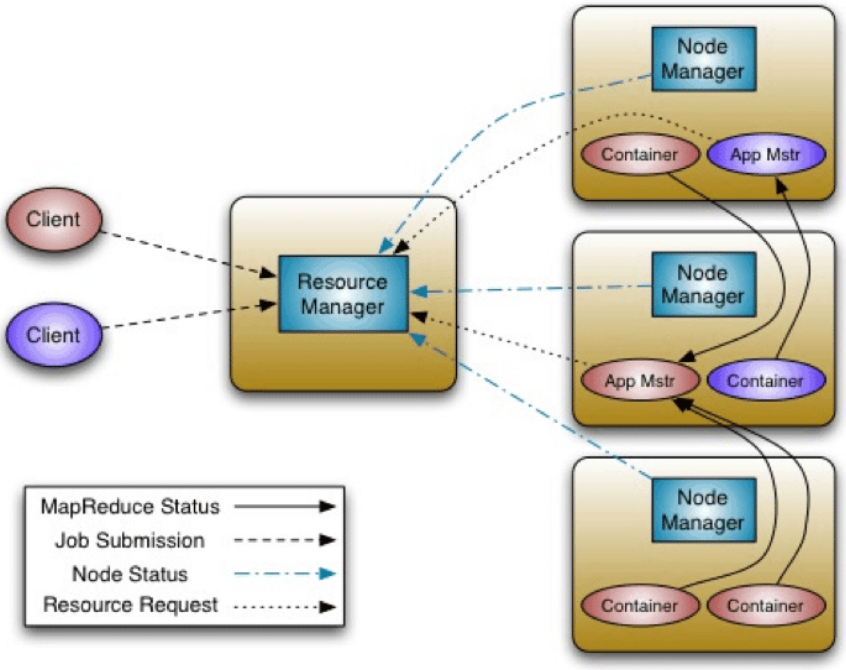


Figure 3.3: YARN Architecture from [3]

3.5.1 Architecture

Like in HDFS the YARN architecture is quite similar in the fact that is a cluster architecture and as a cluster architecture it have two separated entities, the master and the slaves. In this case we have two different daemons, the ResourceManager and the NodeManager. Check the Figure 3.2 on page 16 for the first approach.

The ResourceManager is the daemon in charge to arbitrate resources among all applications. It has two main components:

- Scheduler: is responsible for allocating resources to the various running applications. It performs its scheduling function based on the resource requirements of the applications. Its policy, responsible of sharing resources among applications, it is pluggable which means that we put and set different policies like CapacityScheduler and the FairScheduler.
- ApplicationsManager: is responsible for accepting job-submissions, negotiating the first container for executing the application specific ApplicationMaster. The per-application ApplicationMaster has the responsibility of negotiating appropriate resource containers from the Scheduler, tracking their status and monitoring for progress.

The NodeManager is the daemon responsible for containers, monitoring their resource usage and reporting it to the ResourceManager/Scheduler.

3.5.2 Configure and run on Fedora

To configure YARN is pretty much like we did with HDFS with the difference that we have to modify other file and specify other properties. In this case we have to modify this file:

- `$HADOOP_PREFIX/etc/hadoop/yarn-site.xml`

In this file we have to include the following lines:

```
<configuration>
    <property>
        <name>yarn.resourcemanager.hostname</name>
        <value>udlnet-05-108.udl.net</value>
        <description>The hostname of the RM.</description>
    </property>
    <property>
        <name>yarn.scheduler.minimum-allocation-mb</name>
        <value>128</value>
        <description>Minimum limit of memory to allocate to each container request at the
             Resource Manager.</description>
    </property>
    <property>
        <name>yarn.scheduler.maximum-allocation-mb</name>
        <value>2048</value>
        <description>Maximum limit of memory to allocate to each container request at the
             Resource Manager.</description>
    </property>
    <property>
        <name>yarn.scheduler.minimum-allocation-vcores</name>
        <value>1</value>
        <description>The minimum allocation for every container request at the RM, in
            terms of virtual CPU cores. Requests lower than this won't take effect, and
            the specified value will get allocated the minimum.</description>
    </property>
    <property>
        <name>yarn.scheduler.maximum-allocation-vcores</name>
        <value>2</value>
        <description>The maximum allocation for every container request at the RM, in
            terms of virtual CPU cores. Requests higher than this won't take effect, and
            will get capped to this value.</description>
    </property>
    <property>
        <name>yarn.nodemanager.resource.memory-mb</name>
        <value>4096</value>
        <description>Physical memory, in MB, to be made available to running containers</
            description>
    </property>
    <property>
        <name>yarn.nodemanager.resource.cpu-vcores</name>
        <value>4</value>
        <description>Number of CPU cores that can be allocated for containers.</
            description>
    </property>
</configuration>
```

Now we know the YARN architecture we can understand quite well what these properties are and if need to be on master on slave. Let us review these properties.

The first property is mandatory on all nodes, there we specify which is the host name of the ResourceManager that all nodes need to know. The 4 next are properties related with scheduler and as we have seen before it is one of the components of the Resource Manager that is why these properties only need to be on master even nothing it is going to stop working if we put them as well in slaves. And the last two properties are referred to NodeManager as each slave will have a NodeManager we have to configure these properties with different values fitting each node properties.

If the way we configure YARN was quite similar in what the process refers to how we did with HDFS, executing YARN it is also quite the same as HDFS.

In this case we need to run two daemons, ResourceManager on master and NodeManager on slaves. Note that we could also run a NodeManager on master if we want. These are the commands to run the daemons:

- To start the ResourceManager:

  ```
  $HADOOP_PREFIX/sbin/hadoop-daemon.sh start resourcemanager
  ```

- To start the NodeManager:

  ```
  $HADOOP_PREFIX/sbin/hadoop-daemon.sh start nodemanager
  ```

  And we have YARN running. Review next section to check the YARN status.

3.5.3 Accessibility

Now that we have YARN properly configured and running we can check its properties, applications we have submit, nodes connected, etc, on the web interface. To browser that web we need to connect to the IP address of the ResourceManager with the port by default 8088, do not confuse it with 8080 which is the default port of standalone Spark cluster. I.e: `http://udlnet-05-108.udl.net:8088`

3.6 Connect with spark

After running HDFS and YARN next step is to know how to make Spark to use them.

To use HDFS as simple as it seems once we configure the environment variables as we explained in section 3.4, Spark will use by default HDFS. That means if we do not specify the route of the file with either *file://* or *hdfs://*, Spark will search the file inside the HDFS we configured.

To execute a Spark application over YARN we just need to change the value of the *–master* when we do a*/spark-submit* to the key value yarn and Spark will already submit the application to the configured YARN cluster.

Chapter 4

Scala programming language

4.1 Introduction

The reason because this project is on Scala is GraphX, that only offers support for this programming language in the moment that this project has start (it's announced that it will be available for other programming languages in a future).

So, what is Scala? Scala it's a Functional Object-oriented programming language that it compiles into java bytecode. What it means? That the bytecode generated by scala compiler and java compiler it's the same, so it gives interoperability between both languages.

4.2 Description

Scala is the acronym for "Scalable Language" because it "grows with you", it means that it works with one-line expressions (in order to test, play or practice) but it also can afford something larger mission critical systems (it's used by companies like Twitter, LinkedIn or Intel).

Scala feels like an scripting language, easy to program to anyone that has programmed with any Object-Oriented programming language before. Scala is **pure-bred** object-oriented language. Conceptually we can assume every value as an object and every operation as a method call. Also the language support advanced component architectures. As an object-oriented language it support the traditional programming patterns.

Scala is defined also as a **full-blown** functional language but with a conventional syntax. It provides mutable and inmutable structures, so that means that it's easier to migrate from a "Java without semicolons" to a more functional style using those inmutable structures.

4.3 Main features

Scala offer an interesting set of advantages, as said before, the first of them is the interoperability with java. Also it offers other advantages:

- **Type inference**, it means that the compiler can detect the type of the variables.
- **Traits**: multiple traits can be mixed into a class, so their interface and their behaviour can be mixed, also it allows to a single class extend multiple classes.
- **Pattern matching**: it provides inmutable classes known as "case classes". Data structures like Lists can be matched to this classes.
- **High-Order functions**: in scala, functions are values, so them can be defined as anonymous functions in a very compact way.
- **Concurrency Distribution**: a result of a future operation can be used as an operand before finish. It's useful for expensive computation operations that are computed asynchronously. Also data-parallel operations are used on collections and actors for concurrency.

4.4 Compiling

This project was compiled with **sbt**, version 0.13.13. This, is an multiplatform interactive build tool available for Linux, Mac and Windows. This compiler can generate the jar files from a well structured scala program and can include external dependencies.

These external dependencies are references from an sbt file, that also contains the build configuration.

4.4.1 Program structure

In order to be compiled by sbt scala programs must have and specific structure of folders, where code and resources are placed.

Scala program structure:

```
project
----/build.sbt
----/LICENSE
----/project
--------/build.properties
--------/build.scala
--------/plugins.sbt
----/README.md
----/sbt
----/src
--------/main
------------/scala
----------------/Main.scala
```

All the generated binaries will be placed on *target* folder on the root of the project.

4.4.2 Sbt file and External dependencies

The sbt file (known as built definition) contains the build configuration and dependencies in order to let to sbt all the information to compile properly the program.

Is not mandatory to import the external dependencies manually on the code. Them can be stored on an external repository, and included on sbt file on the root of the project. On the compiling time, sbt will check them, and download the missing ones (they will only be downloaded the first time), and include them automatically on the project.

A programmer only have to include the new dependency repository on the *sbt* file and use as another import on the code.

The project file:

```
name := "Social Network Analyzer"

version := "1.0"

scalaVersion := "2.11.7"

libraryDependencies += "org.apache.spark" %% "spark-core" % "2.1.0"
libraryDependencies += "org.apache.spark" %% "spark-graphx" % "2.1.0"
libraryDependencies += "org.json4s" %% "json4s-jackson" % "3.2.11"
libraryDependencies += "org.apache.spark" %% "spark-mllib" % "2.1.0"
```

Chapter 5

Graphs: structure and operations

5.1 Graph RDD

The graph it's also an RDD, they are defined with the properties of vertices and edges that it contains. It provides useful fields and operations, like retrieving properties from the graph like number of vertices / edges, provide RDD that represents the vertices with each number of in / out degrees, etc...

The most common access to this RDD is using the "triplets", that provides two related vertices.

```
val graph: Graph[(String,(Int,Bool)), String]

graph.triplets.map(triplet =>
triplet.srcAttr._1 + " is the " + triplet.attr + " of " + triplet.dstAttr._1
)
```

5.2 Graph structure

A graph in graphx, its created from two RDD's that represent vertices and edges, so there are two RDD's filled of this info, but how are vertices and edges represented?

Vertices: the vertices are structures composed by two field, the first (and the mandatory) it's always a **Long** that represents the VertexId. Then the second field are defined by user, and can be basic type or an structure.

For example in our case:

```
val vertices: RDD[(VertexID, (String, (Int,Bool))]
```

Edges: edges are the structures that creates relationships between the vertices, and them has three fields, the first two are **Longs** that represents the nodes that are connecting, they can represent directed graphs (first long is "from" and second is "to"), and finally the third field represent the "property", usually represented by an String, but like in the vertex RDD, this field is defined by the programmer and it can be a basic type or an structure.

```
val relationships: RDD[Edge[String]]
```

5.3 Operating with graphs: Pregel

Pregel is a programming model in order to solve large-scale graph problems by doing parallel computation on graphs with "synchronizing" points called "superSteps". It wants to provide easy programming concept dividing the pregel algorithms in three parts and also be more efficient than MapReduce.

5.3.1 How it works?

Pregel executes it's entire code multiple times, it consist in nodes sending messages to each other, and updating the nodes state, finally it will return a new graph with the final state of the nodes.

First, all the nodes receive an initial message, and after this, all the nodes will sent or not a message to each neighbor, after that all messages that a node has received will be joined into one (like in reduce) and it will process it.

After executing this code, if only one message has been sent, all the code will be executed again until all the nodes don't send any message to each other.

5.3.2 SuperSteps

Pregel algorithms will execute it's code entirely multiple times. There are a synchronization points between this executions called **SuperSteps**, why we need them? They are necessary in order to know if pregel algorithm has finished. If no node has sent any messages pregel consider that the algorithm is finished.

5.3.3 Code structure

Pregel algorithms in GraphX are structured in three sections:

State updating: A message (initial or a normal message after joining them), that has the same structure as the node of the graph, on this section this node will evaluate this message and will update it's state.

Sending messages: Every node will check all their neighbors, evaluate them and send a message or not (it can send a message to one node, and nothing to another).

Joining messages: After sending messages step, all the messages for every node will be joined into one with a programmer-defined criteria on the same way of reduce.

```
def minDistance() : Int = {
  val graph = m_representations.integerBool(0, true)
  val sssp = graph.pregel((0, true))(
    (id, dist, newDist) => {
```

```
      if(dist._1 == newDist._1 && dist._1 != 0){
        (dist._1, false)
      }
      else{
        newDist
      }
    },
    triplet => {
      if(triplet.dstAttr._2){
        Iterator((triplet.dstId, (triplet.srcAttr._1+1, true)))
      }
      else{
        Iterator.empty
      }
    },
    (a,b) => {
      if(a._1 < b._1) a
      else b
    }
  )
  val distance = sssp.vertices.reduce((a,b) => {
    if(a._2._1 > b._2._1) a
    else b
  })
  println(sssp.vertices.collect.mkString("\n"))
  distance._2._1
}
```

Chapter 6

Json parsing

6.1 Introduction

As a first note before start talking about the project, all the jsons are parsed using the library json4s (json for Scala) available on: http://json4s.org/.

6.2 How it works?

This utility allows parsing a json directly if the names of the fields are the same of the parameters of a case class (a class with inmutable instances in scala). The case class can contain only a subset of parameters of the json.

6.3 Example

Given the following Json:

```
{"id":12654849684, "message":"This is a message!",
"author":"Joel", "likes":4, "answers":19}
```

Can be matched, in example, with the following case classes:

```
case class BasicInfo(id : Long, message : String, author : String)

case class MessageImpact (message : String, likes : Int, answers : Int)

case class FullInfo(id : Long, message: String, author : String,
likes : Int, answers : Int)
```

Chapter 7

Conversation downloaders

7.1 Introduction

In order to automatize all the process we've implemented two downloaders. TwitterDownloader and RedditDownloader. Both are implemented in python and called from the scala project.

The python twitter downloader was originally implemented by Carles Mateu, from the Universitat de Lleida, and modified by Joel Cemeli in order to allow to be called from spark scala project.

By the other hand, the python reddit conversations downloader was implemented entirely by Cristian Sanahuja.

7.2 Obtaining conversations

Every social network has its own structure for the conversations, for example, on twitter, the conversation tree it's only a concept. In this case, the Reddit and Twitter downloader have different behaviours:

7.2.1 Twitter Downloader

The download of a twitter conversation starts from the root message. Starting from the root identifier, the collector gets the answers, and the answers to the answers. So finally returns a file filled of all the jsons of the messages that are on all the branches of the conversations.

7.2.2 Reddit Downloader

On Reddit the conversations start given an initial post, and people answer directly to the post or to another response. So the root will always be the post, and the branches are created with the responses (and responses to the responses).

7.3 Behaviour

Other programs can be called from an spark project via RDD. The program cannot be called with arguments, so, all the communication between Spark and Python programs are performed by the standard input. So, first of all, all the input lines must be put into a list, in this case,

only one line, the ID of the conversation, then add the source code file to the SparkContext. Finally is used a pipe RDD, inputs are distributed to the nodes, and execute the program with that inputs.

The results are also taken from the output and placed into an RDD when calling the *collect()* instruction, that also, forces spark to pause until the program halts.

7.4 Download Manager

Inside the project, the **Download Managers** are the responsible to do all this process, **TwitterDownloader** and **RedditDownloader** encapsulates all the behaviour. So calling them with a desired identifier of the root of the conversation stores the received results and return the path where this file is placed.

Chapter 8

SocialGraph

8.1 What is a SocialGraph?

SocialGraphs is the common abstraction for the graphs, providing multiple methods to create a new graph, serializing, giving multiple representations, and can be operated with pregel. Also SocialGraph is extendable, in the project two extensions are provided: TwitterGraph and RedditGraph, allowing to parse twitter/reddit conversations directly.

It acts as a builder and also as a method provider.

8.2 Data Structure

So, how it's the data structured in SocialGraph?

8.2.1 Vertex

On SocialGraphs the Vertex are named as **Messages**, that are a case class with the basic parameters that we need to identify a Vertex on the graph as a message in a Social Network discussion. They are stored on a Map, identified by their own id and the following structure:

```
case class Message(id: Long, var weight: Float, author: String,
message: String, properties: Map[String, Any])
```

ID: Store the unique identifier of the message on the conversation.
Weight: Represents the score that have the message on the conversation.
Author: The author of the message.
Message: The text of the message.
Properties: As SocialGraph is a common representation for all the SocialNetwork graphs it provides this field to store extra-values that could be relevant. In the provided extensions (TwitterGraph and RedditGraph) this map is filled with all the fields of the json of the message.

8.2.2 Edges

The edges are named **Connections**, are also stored on a map, identified by are a tuple that represents the *from to* identifiers, and in this case, they are stored as Edges with a String property, that represents the intention of that connection: *noone, attack* and *support*.

```
case class Connection (origin: Long, destination: Long, intention: String)
```

8.2.3 Graph

Graph on GraphX it's also an RDD, so SocialGraph generates it on a lazy way. The first time that user tries to get a graph, SocialGraph generates it and return. The other times, "makeGraph" option should be called in order to refresh the graph representation when getting it.

In order to create the graph, SocialGraph, transform the Message and Connections maps into RDD's using "parallelize" methods, then create a "default relation", and, with this three components, generates the graph with GraphX, and store it for requests.

8.3 Creating a graph

8.3.1 From raw data

SocialGraph provides the necessary methods in order to generate a graph message to message and connection to connection.

8.3.2 From Json

SocialGraph are serialized in a specific Json format, writing a Json with the same fields allows SocialGraph to parse them and create a graph.

8.4 SubGraphs

As are explained, the connections on the graph has a field that notes the intention of that interaction. In order to perform operations using only specific connections of the graph, it's provided a method to get a subgraph with some of the connections based on their intention.

This sub-graphs can use 1 to 3 intentions, (using 3 is the original graph).

8.4.1 How it works?

It iterates all the connections of the SocialGraph, and checks the intention, if the intention is one of the requested, this connection is added to the new graph connections RDD and also add the both nodes that are linked with that connection (if are not added yet), a map is used to store the vertices of the new graph in order to check fast if it's already in the graph RDD.

Chapter 9

TwitterGraph

9.1 Introduction

TwitterGraph it's the first of the two provided extensions for SocialGraph. This extensions allow users add **specific rules** when making the graph. Every Social Network has its own behaviour, so we probably want for example make extra-connections, based in the content of the message, or calculate the weight on an specific way (with specific properties that only have that Social Network).

9.2 Main behaviour

As SocialGraph, the main behaviour of this extension is parsing a conversation, message by message, from json, or loaded from a file that contains the json. It fills the Messages and Connections maps, so then it can be treated as a normal SocialGraph (retrieving the common graph representation) and perform operations over it.

9.3 Conversations in Twitter

On Twitter the conversations are only a concept. We've got a main tweet, a second tweet answers it, then a third tweet answers the second. This creates a tree of tweets pointing to another, but also we found the **mentions**, a message can reference to another user (that has participated or not on the conversation), so on TwitterGraph this is used as extra-information for increase the precision of the graph.

9.4 Mentions and extra connections

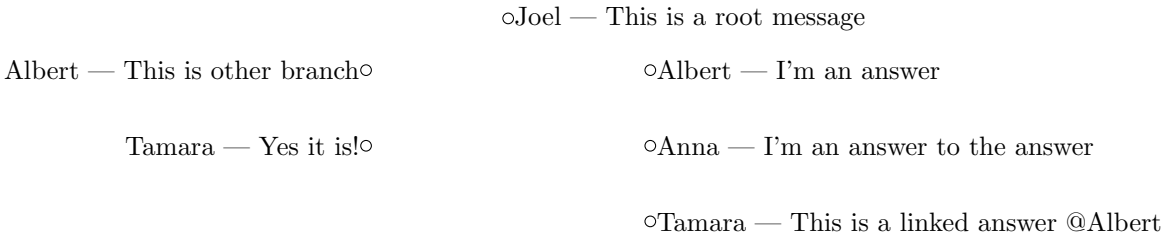
The main part of this extra rules we found the extra connections. The tweets are parsed in order, from the original to the leafs. Every tweet, except the root, are answering a previous tweet, that is the first connection made. After that parser will read the mentions on that tweet, if it got something it will look for the most recent tweet of the mentioned author on the **same branch**, if it exists, a new connection will be made.

9.5 Example

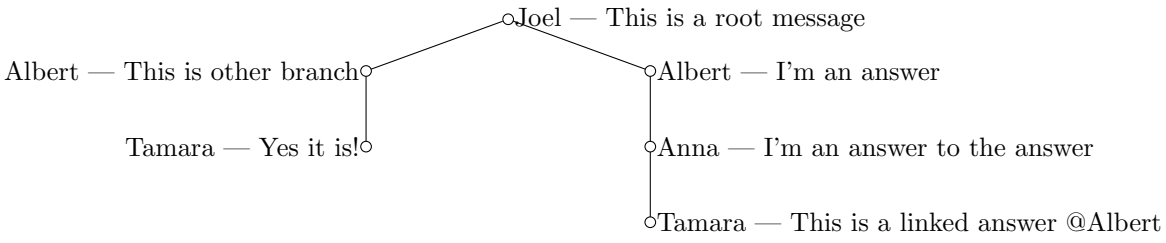
Given the following Json:

The result graph will be:

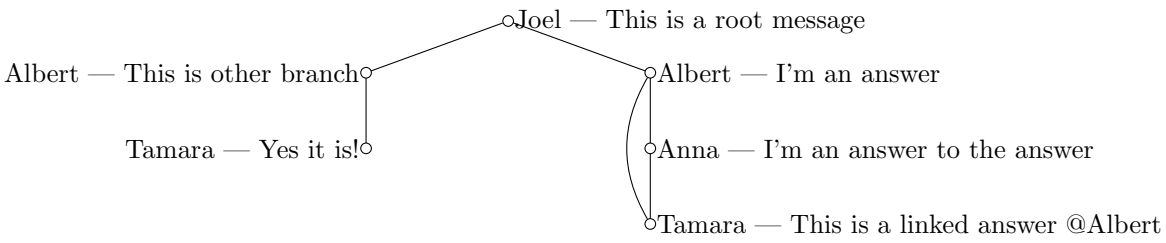
First we have the nodes of the graph.



Then the links are made.



When the message with the mention is reached, the parser looks inside the same branch trying to find the last message of the mentioned author.



9.6 Weight

The weight of every message will be calculated with the following parameters: the followers of the author, the number of retweets of this message and the favorites it have. the operation will be the following:

$Weight = \log_2(followers + 20 * retweets + 40 * favorites + 1)$
The logarithm is performed in order to don't differentiate to much between the messages.

Chapter 10

RedditGraph

10.1 Introduction

RedditGraph, provides the same interface allowing to parse entire conversations from it's jsons. In this case the parsing it's more simple because on reddit we don't take in account the mentions. Why? Mentions exists on reddit, but the mentioned name it's different from the author name. So it's hard to find the message that user is referencing to with that mention.

10.2 Messages structure

In this case the structure of the messages it's simpler, providing only an ID for every message, author, text and parent (which message it's answering to). And parsed to that class:

```
case class Reddit_Temp(id: Long, author_name: String, text: String, parent : Long)
```

Chapter 11

Graph Representations

11.1 What and why?

Graph Representations are a set of methods that can convert a SocialGraph into another representation. This representations maintains the same structure as the original graph (vertices and connections), but the vertices properties are different. This is because, in pregel, the messages and the states are represented with vertices, and in order to send or store specific information the alternative representations of the graph are needed.

11.2 Representations

Some representations are provided based on the needs when implementing pregel algorithms. This representations are:

- integerOnly
- integerInteger
- integerBool
- integerIntegerBool
- floatIntegerBool
- floatIntegerLongBool

The names of the representations corresponds to each params. Also some of this representations has some in common, for example, all the representations has a version with an extra field: a bool, that it's useful in pregel to indicate an 'end' state. Also all the default values for these representations are defined by the user when calling these methods.

Also on the last two representations has two extra-features:

- First value as weight
- Leaves mark.

It can be set to make that the first value it's the weight of every node. And also the leaves' bool default value can be set sepparatedly from the rest of the nodes.

Chapter 12

Pregel algorithms

12.1 Starting with pregel

All the calculus on the graphs are made using pregel. As is said on the **Pregel** chapter on the **Environment part**, Spark provides a pregel implementation that has some peculiarities.

12.2 Remember peculiarities

As explained in the chapter quoted, pregel on Spark structure its algorithms in three parts: *State updating*, *Sending messages* and *Joining messages*.

When calling pregel an initial user-defined message will always be sent, also it's possible to configure the message sending directions (inner-way or outter).

12.3 First algorithms

These algorithms was implemented in order to provide some basic functionalities and also get started in pregel algorithms implementation.

12.3.1 Minimum Distance

Explanation: Social graph are trees, in this case, all the nodes are pointing to a parent node (except the root). All of them will send their accumulated distance to their parent + 1 (because of the distance between them and the parent). Following the next logic, and knowing that messages will be sent from a node to their parents, we know that the minimmum distance at the end of the algorithm will be placed on the root node of the graph. Also, on this case a bool it's required in order to indicate if node it's in its final state or can be updated with a lower distance.

Basics:
Requires representation: Integer-Bool
Initialization: Integer - 0, Bool - True
Initial message: (0, true)
Return value: Integer

Updating states:
A state is finished if its distance is different from 0 and receive the same distance twice. In other case, if the received distance is different from the current vertice distance, stores the new.

Sending messages:
Every vertice checks for everyneighbor if it's finished. If it is, don't send message, if not, it sends its distance + 1.

Joining messages:
When reducing it get's the minimum distance for each pair of messages, getting finally the minimum distance at that point.

Result:
In order to get the minimum distance of all the distances from all the nodes it's made via reduce. It get's the bigger distance of all the minimum distances. Why? Because after executing this algorithm the minimum distance can be found on the root message, that will has the bigger distance (the other nodes will has only intermediate distances).

12.3.2 Maximum Distance

Explanation: All of them will send their accumulated distance to their parent + 1 (because of the distance between them and the parent). Following the next logic, and knowing that messages will be sent from a node to their parents, we know that the maximum distance at the end of the algorithm will be placed on the root node of the graph. In this case, the state boolean it's not required, because, the distance received will be always increasing, and the vertices will stop sending messages to a neighbor if the destiny node distance is higher than its own.

Basics:
Requires representation: Only Integer
Initialization: Integer - 0
Initial message: (0)
Return value: Integer

Updating states:
Always update the state with the received state.

Sending messages: Every vertex checks for every neighbor if it has a lower distance. If it's the case it sends its distance + 1, if not, it don't send a message.

Joining messages: When reducing it get's the maximum distance for each pair of messages, getting finally the maximum distance at that point.

Result:
In order to get the maximum distance of all the distances from all the nodes it's made via

reduce. It get's the bigger distance of all the distances because after executing this algorithm the maximum distance can be found on the root message, that will has the bigger distance (the other nodes will has only intermediate distances).

12.3.3 Average Distance

This is the more complicated of the three algorithms, and it got a lot of failed implementations before that last version.

Explanation: In this case the integer-integer-bool representation is used. The first integer represents the accumulated distance, the second represents how many branches are using that connection, and the bool represents if a node has finished it's calculus. This is done in order to sum all the distances from every leaf to the root.

Basics:
Requires representation: Integer Integer Bool
Initialization: (0, 0, true)
Initial message: (-1, -1, true)
Return value: Integer

Updating states:
Handling initial message: Initial message is tagged as -1, in order to be handled and "maintain" and initialize the nodes.
If it's not the firs message there are two ways:

If both integer values received are the same that the two stored on the current vertex, they will be the final values and the node will be marked as finished, else, the received values will be stored on the current vertex.

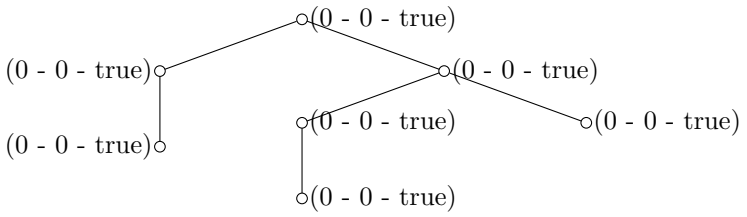
Sending messages: For every neighbor node that it's not marked as finished, if it's distance is 0, it will send the next message: (1, 1, true) this means adding one distance and "subscribing" a new branch to all the connections from parent to the root; else, the message would be (firstValue + secondValue, secondValue, true), counting that connection as many times as branches are "subscribed" to that node.

Joining messages: All the accumulated distance and subscribed branches are summed separatedly.

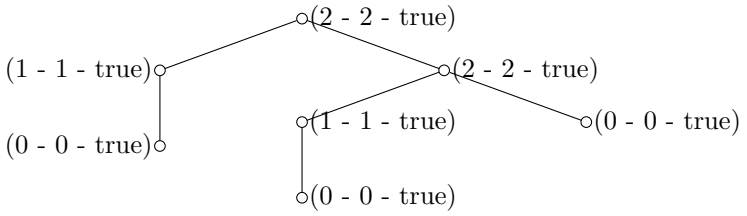
Result:
As on the minimum and maximum distance, the biggest distance is found via reduce, that can be found on the root. But in this case this is not the final result, it must be divided by the number of leafs the tree has, this number is obtained with a property that got the graphs on GraphX, that made that all the graphs contains a RDDs with special properties: RDD with all the vertices, another with inDegrees information (if a vertex has not inDegree it won't be on the RDD) and another similar to the last one but with outDegrees. So, taking the vertices.Count (in order to get how many vertices are placed in this RDD), and subtract inDegrees.count to it, only the leafs will remain.

Example:

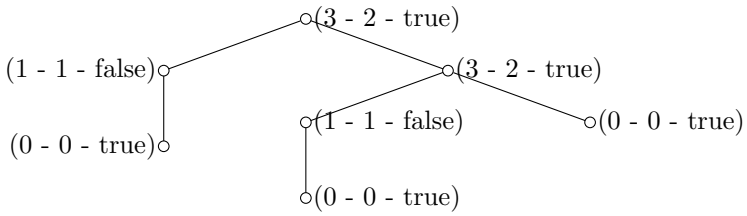
SuperStep 0:



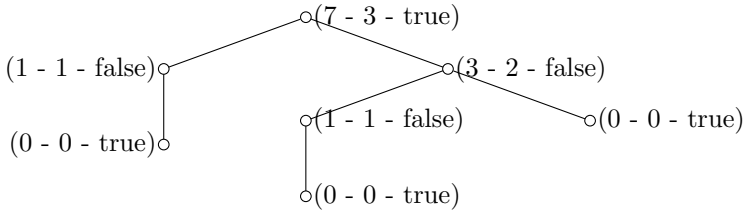
SuperStep 1:



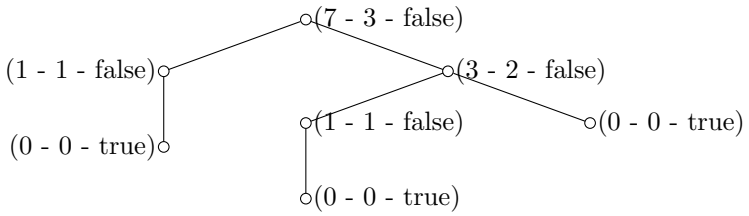
SuperStep 2:



SuperStep 3:



SuperStep 4:



Chapter 13

Reasoning calculus

13.1 Introduction

The last piece of this machine is the reasoning calculus, the place, where all the other pieces converge and make sense. At this state the project provides defeaters calculus. It provides two versions of the algorithm.

Both versions are provided in two different methods, one on "standard" way, and the other counting the total messages sent during pregel.

13.2 Defeaters Counter Algorithm

The main idea about this algorithm is: taking the attack subgraph, count how many messages are "winning", so first of all we must define what we understand by "winning":

Winner node: is that one, that is not defeated by anybody. For example, all the leaves are "winner" messages.

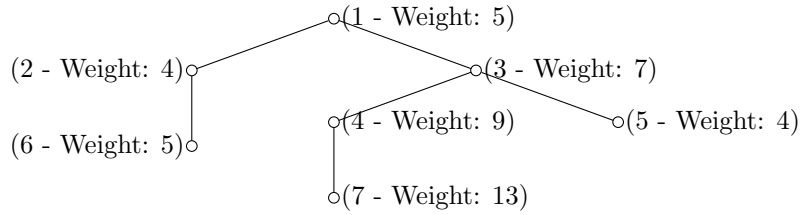
Defeated node: is that one that has at least one node attacking to it that has a bigger weight and this attacker node isn't defeated.

Note: if a node is defeated it cannot defeat other nodes.

Example:

On the following graph we have 4 winner nodes and 3 defeated ones.

- Winner: 3, 5, 6, 7
- Defeated: 1, 2, 4



- Node 1 (Defeated): Defeated by 3, that would be defeated by 4, but it's annulated by 7.
- Node 2 (Defeated): Defeated by 6.
- Node 3 (Winner): Would be defeated by 4, but it's defeated by 7.
- Node 4 (Defeated): Defeated by 7.
- Node 5 (Winner): Not defeats node 3, but it's not defeated.
- Node 6 (Winner): Defeats 2, not defeated by any node.
- Node 7 (Winner): Defeats 4, not defeated by any node.

13.3 Original implementation, with message counting

The first prefel implementation of the algorithm:

Basics:
Required subgraph: Attack only
Requires representation: Float - Integer - Long - Bool
Initialization: Float - Weight, Integer - 0, Long - 0, Bool (False for all nodes but the leaves)
Initial message: (0, -1, 0, false)
Return value: Integer

Updating states:
In this case the algorithm distinguishes between initial message and the rest of messages, because the initial messages on pregel are mandatory.

If the received message is the initial message simply ignore it (maintain the same node state).

In other case, the message has four fields:

- Float - Weight
- Integer - Number of defeaters of the node.
- Long - Number of messages received by the node.
- Bool - (True -¿ Finished, False -¿ Unfinished)

The new state will be:

- Weight - Still the same.
- Defeaters - Received defeaters value
- Messages - Current messages value + received messages value
- Bool - If the numbers of defeaters is the same as previous received, or all the neighbours has finished.

Sending messages:

Here we found three cases:

Vote to finish: If this node is finished and the destiny node too, it won't send any message to it.

Defeat: If the node is not defeated and its weight is bigger than destiny's weight, the message that it will send it will indicate that it is a defeater (1 on that value). On the finished field it will add its own value.

Non-defeater: In any other case, it will send a message, indicating 0 on defeaters field. On the finished field it will add its own value.

Note: On two last cases, the node is sending a message, so it will add 1 on the messages field.

Joining messages:

- Weight - 0 (The nodes always maintains the same)
- Defeaters - Sum all the defeaters values.
- Messages - Sum all the messages values.
- Finished - AND union of all the values (Only true if all the nodes are finished).

.

Result:
Counting messages: The messages will be counted by summing all of the message fields by reducing them.

Counting winner nodes: This will be performed with a map-reduce operation. On the map, all the nodes are converted into an integer 0 if they has 1 or more defeaters or 1 if they are not defeated, and summed on reduce.

Chapter 14

Tests

14.1 Introduction

All the classes and methods are tested. The most of the methods couldn't be tested on a traditional way like unitary tests.

The tests are distributed in four sections:

- Basic Tests
- Distributed Graph Tests
- Graph MPC Tests
- Serialization Tests

14.2 Basic Tests

On this class are performed the earlier tests, and all the functionalities for SocialGraph, also some basic functionalities for the specific graph implementations.

Creating SocialGraph and specific graphs from jsons and also message by message.

14.2.1 SocialGraph

Creating a graph message by message (also the connections), from json, and basic operations:

- List triplets.
- Get subgraph.
- Update punctuation of nodes.

14.2.2 TwitterGraph

Parse different conversations from jsons:

- Single tweet.
- Tweet simple conversation.
- MultireferenceConversation

14.2.3 Reddit

Parse a conversation from it's json.

14.3 Distributed Graph Tests

This class performs operations using SocialGraph and TwitterGraph. On this class are two kind of tests, the first, are full controlled graphs (graphs which we know the results of the operations, and on the other hand, real graphs).

Also this class provides a method to download and parse a multireferenced graph, also these methods was tested on this class.

The operations tested on the both ways are the following:

- Maximum distance of the graph.
- Minimum distance of the graph.
- Average distance of the graph.
- Original Defeaters Counter algorithm
- Optimized proposal Defeaters Counter algorithm.

14.4 Graph MPC Tests

This class only tests the classification of the edges of a graph. It's tested with a Twitter conversation.

14.5 Serialization Tests

This class provides tests about saving/loading graphs. It simply create a graph, serialize it, and deserialized, and make sure if both are the same.

Chapter 15

Structure and Behaviour

15.1 Graph

On this diagram are displayed all the current graph implementations, also the classes relative to pattern matching (this is indicated by placing them on a field with the same name). See figure 15.1

15.2 Defeaters Calculus Flow

This flow diagram represents the basic behaviour (in standalone mode)* of how all the modules work. On this diagram the MPC creation is not represented, only explains all the graph part, from graph downloading and creation to graph calculus. See figure 15.2

*On distributed mode, the downloader and parser caller must be called only on the master, separately from the Distributed program.

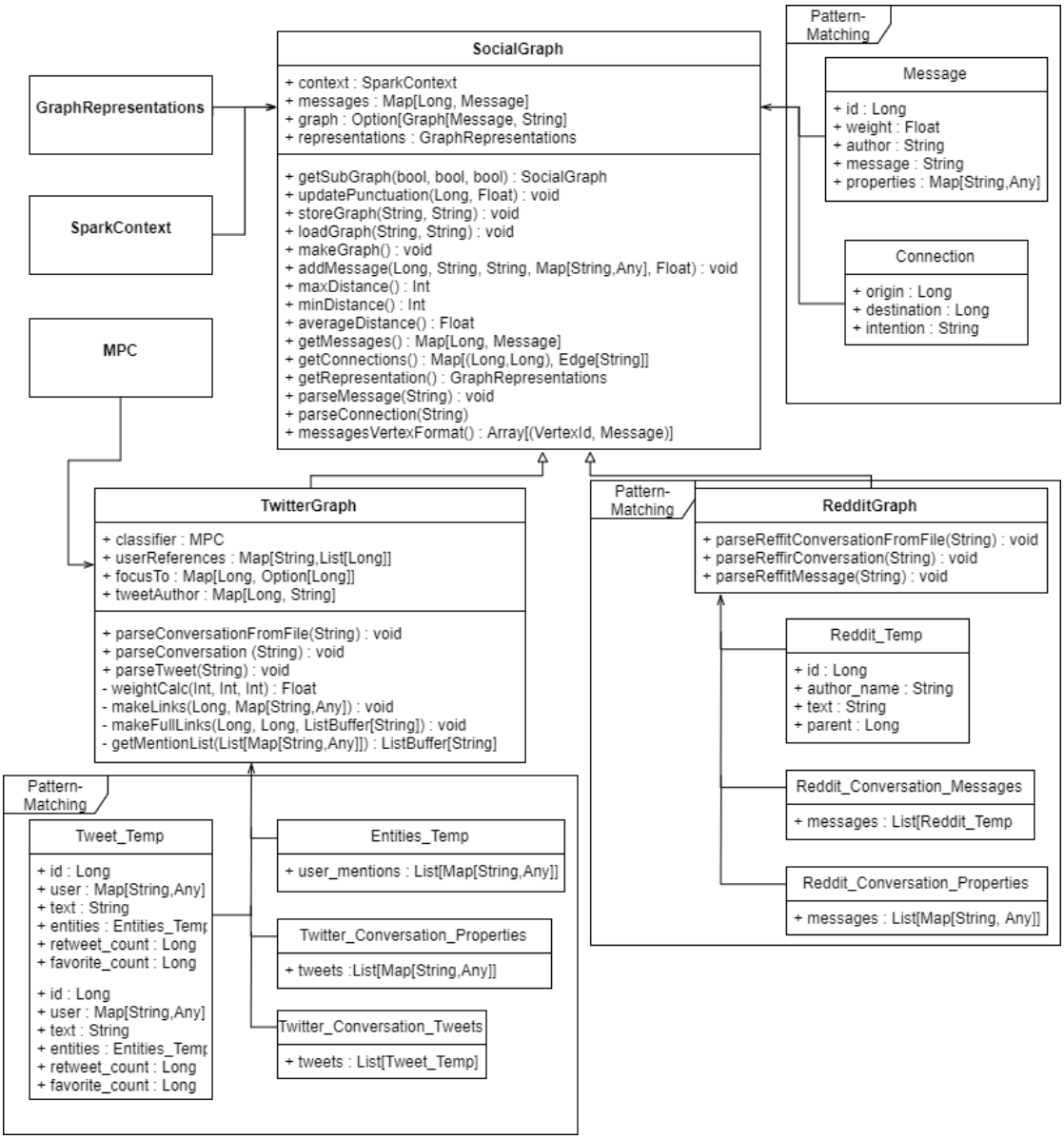


Figure 15.1: UML diagram for all graph Modules

.

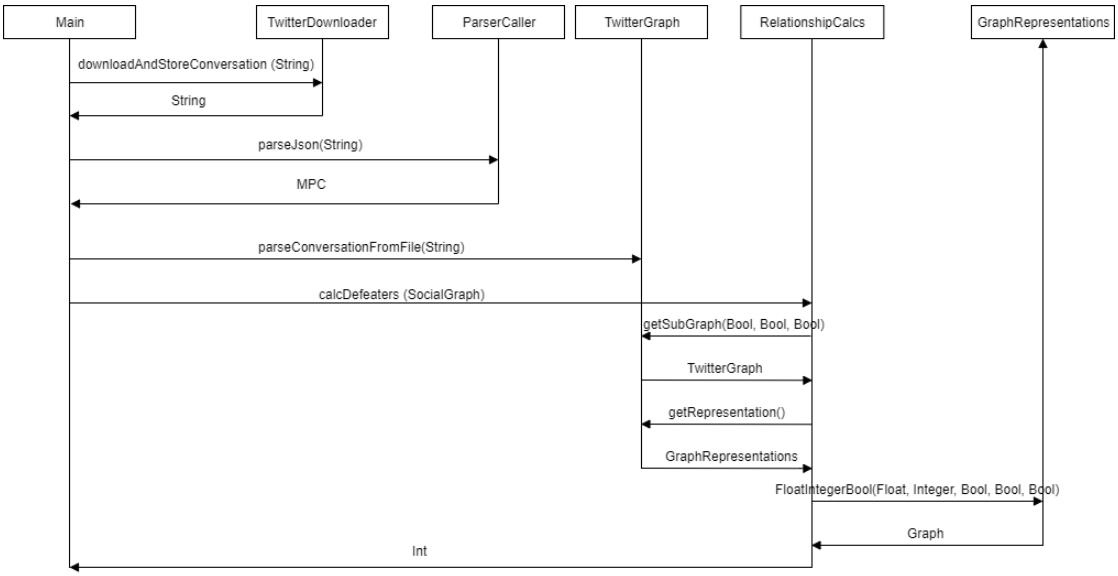


Figure 15.2: Flow diagram from graph creation to defeaters calculation

.

Chapter 16

Obtained results

16.1 The results

On this part, are placed all the results obtained for defeaters counter algorithms (the orignal and the optimize proposal). This are tested with a set of conversations sorted from small ones to big conversations in order to see, how the algorithms perform the increase of conversation sizes.

Both algorithms are tested in two different contexts: single machine, and distributed (4 machines).

On the graphs there are some peaks, this is because, the number of nodes does not represent the real attack nodes of the graph.

16.2 Commented results

16.2.1 Time

In terms of time, the original version and the optimized proposal (in non-distributed and distributed experiments) has very similar results, the differences probably are because of the Spark environment (user-time + sys-time it's bigger than real-time, that probably means that spark process is not taken in account in real-time, but it is on the user-time and system-time, where all the processes time are summed).

Also on this cases the distributed versions take more time to execute, this is because, on this cases there are more work in terms of message-passing than computing (that probably wouldn't happen on great conversations). On these cases the time taken in account is the time that took master (the other nodes time are not summed, in order to compare with single-node versions).

16.2.2 Messages

In terms of messages the optimized proposal is more stable and reach numbers several times smaller than the original, with a minimum of 1 to a maximum of 569, the original goes from 10 to 23935. On a distributed system, sending messages could consume the most of the time, so

Table 16.1: Original Non-Distributed

Discussions	#nodes	#util nodes	#accepted	#edges	#attacks	real time	user time	sys time	#messages
867743234046480384	19	11	10	21	10	0m9.921s	0m14.946s	0m0.588s	10
867742319088467968	20	21	19	23	23	0m12.169s	0m18.790s	0m0.706s	26
867841549085929472	31	14	13	38	14	0m11.849s	0m18.014s	0m0.693s	14
865615862291718144	39	24	17	80	45	0m12.389s	0m18.996s	0m0.729s	100
867627523059961856	63	35	29	110	48	0m13.171s	0m20.570s	0m0.876s	59
574324656905281538	73	74	56	165	142	0m14.957s	0m22.731s	0m0.850s	206
867756958337683457	137	136	129	173	159	0m16.987s	0m25.818s	0m1.016s	181
867651560406478848	152	121	120	159	126	0m9.260s	0m14.890s	0m0.474s	126
868086232932376580	158	158	152	196	175	0m17.032s	0m25.039s	0m0.925s	175
867751574436761600	212	87	73	325	131	0m15.365s	0m23.163s	0m0.916s	179
866895801494228993	379	255	182	1408	882	0m20.900s	0m29.268s	0m1.101s	4518
867494068917608448	386	318	229	2615	1905	0m21.846s	0m30.874s	0m1.221s	12225
865925019007954944	446	426	404	882	749	0m23.699s	0m33.344s	0m1.337s	1024
867052819878203394	519	421	279	2657	1936	0m24.685s	0m33.765s	0m1.440s	6117
867494287122059264	589	491	313	4559	3062	0m26.599s	0m36.717s	0m1.539s	13226
866624474568953857	788	763	529	3829	2864	0m32.412s	0m42.509s	0m1.871s	8822
867693522195034112	897	876	615	5160	4306	0m33.074s	0m42.585s	0m1.923s	16293
868072573002821635	931	779	544	6373	4505	0m32.461s	0m42.981s	0m1.855s	23935

Table 16.2: Original Distributed

Discussions	#nodes	#util nodes	#accepted	#edges	#attacks	real time	user time	sys time	#messages
867743234046480384	19	11	10	21	10	0m13.983s	0m14.096s	0m0.544s	10
867742319088467968	20	21	19	23	23	0m12.329s	0m13.840s	0m0.590s	26
867841549085929472	31	14	13	38	14	0m15.664s	0m19.756s	0m0.681s	14
865615862291718144	39	24	17	80	45	0m16.016s	0m21.010s	0m0.713s	100
867627523059961856	63	35	29	110	48	0m16.504s	0m21.014s	0m0.711s	59
574324656905281538	73	74	56	165	142	0m17.663s	0m23.012s	0m0.796s	206
867756958337683457	137	136	129	173	159	0m20.040s	0m26.028s	0m1.018s	181
867651560406478848	152	121	120	159	126	0m13.023s	0m13.113s	0m0.469s	126
868086232932376580	158	158	152	196	175	0m19.912s	0m25.121s	0m0.926s	175
867751574436761600	212	87	73	325	131	0m17.665s	0m23.500s	0m0.882s	179
866895801494228993	379	255	182	1408	882	0m22.488s	0m28.441s	0m1.121s	4518
867494068917608448	386	318	229	2615	1905	0m27.034s	0m32.934s	0m1.306s	12225
865925019007954944	446	426	404	882	749	0m11.887s	0m13.523s	0m0.526s	1024
867052819878203394	519	421	279	2657	1936	0m31.650s	0m36.495s	0m1.383s	6117
867494287122059264	589	491	313	4559	3062	0m32.140s	0m37.171s	0m1.551s	13226
866624474568953857	788	763	529	3829	2864	0m38.405s	0m41.374s	0m1.732s	8822
867693522195034112	897	876	615	5160	4306	1m39.571s	0m44.385s	0m1.914s	16293
868072573002821635	931	779	544	6373	4505	0m36.548s	0m42.402s	0m1.757s	23935

Table 16.3: Optimized Non-Distributed

Discussions	#nodes	#util nodes	#accepted	#edges	#attacks	real time	user time	sys time	#messages
867743234046480384	19	11	10	21	10	0m9.675s	0m14.390s	0m0.554s	1
867742319088467968	20	21	19	23	23	0m13.897s	0m20.021s	0m0.703s	2
867841549085929472	31	14	13	38	14	0m12.659s	0m18.610s	0m0.673s	2
865615862291718144	39	24	17	80	45	0m13.144s	0m19.225s	0m0.747s	14
867627523059961856	63	35	29	110	48	0m13.798s	0m19.862s	0m0.809s	7
574324656905281538	73	74	56	165	142	0m15.596s	0m22.438s	0m0.835s	17
867756958337683457	137	136	129	173	159	0m18.182s	0m25.387s	0m0.916s	7
867651560406478848	152	121	120	159	126	0m9.512s	0m14.004s	0m0.590s	1
868086232932376580	158	158	152	196	175	0m18.848s	0m25.835s	0m0.940s	8
867751574436761600	212	87	73	325	131	0m16.316s	0m22.884s	0m0.897s	22
866895801494228993	379	255	182	1408	882	0m22.595s	0m30.143s	0m1.136s	129
867494068917608448	386	318	229	2615	1905	0m23.647s	0m30.514s	0m1.227s	233
865925019007954944	446	426	404	882	749	0m24.919s	0m32.449s	0m1.256s	29
867052819878203394	519	421	279	2657	1936	0m26.568s	0m34.325s	0m1.263s	343
867494287122059264	589	491	313	4559	3062	0m30.657s	0m37.816s	0m1.438s	498
866624474568953857	788	763	529	3829	2864	0m35.954s	0m42.449s	0m1.939s	379
867693522195034112	897	876	615	5160	4306	0m40.115s	0m45.440s	0m1.803s	604
868072573002821635	931	779	544	6373	4505	0m34.873s	0m42.909s	0m1.632s	569

Table 16.4: Optimized Distributed

Discussions	#nodes	#util nodes	#accepted	#edges	#attacks	real time	user time	sys time	#messages
867743234046480384	19	11	10	21	10	0m11.213s	0m12.894s	0m0.460s	1
867742319088467968	20	21	19	23	23	0m12.291s	0m12.990s	0m0.479s	2
867841549085929472	31	14	13	38	14	0m16.367s	0m20.220s	0m0.755s	2
865615862291718144	39	24	17	80	45	0m16.223s	0m20.738s	0m0.828s	14
867627523059961856	63	35	29	110	48	0m17.631s	0m21.461s	0m0.735s	7
574324656905281538	73	74	56	165	142	0m17.175s	0m22.842s	0m0.803s	17
867756958337683457	137	136	129	173	159	0m20.845s	0m25.083s	0m1.020s	7
867651560406478848	152	121	120	159	126	0m13.071s	0m12.697s	0m0.515s	1
868086232932376580	158	158	152	196	175	0m21.706s	0m26.603s	0m0.993s	8
867751574436761600	212	87	73	325	131	0m17.917s	0m22.216s	0m0.794s	22
866895801494228993	379	255	182	1408	882	0m23.882s	0m28.920s	0m1.174s	129
867494068917608448	386	318	229	2615	1905	0m26.278s	0m32.560s	0m1.222s	233
865925019007954944	446	426	404	882	749	0m11.758s	0m13.745s	0m0.481s	29
867052819878203394	519	421	279	2657	1936	0m30.237s	0m33.396s	0m1.396s	343
867494287122059264	589	491	313	4559	3062	0m33.634s	0m37.313s	0m1.488s	498
866624474568953857	788	763	529	3829	2864	0m38.167s	0m41.735s	0m1.788s	379
867693522195034112	897	876	615	5160	4306	0m38.540s	0m40.800s	0m1.765s	604
868072573002821635	931	779	544	6373	4505	0m37.258s	0m39.379s	0m1.710s	569

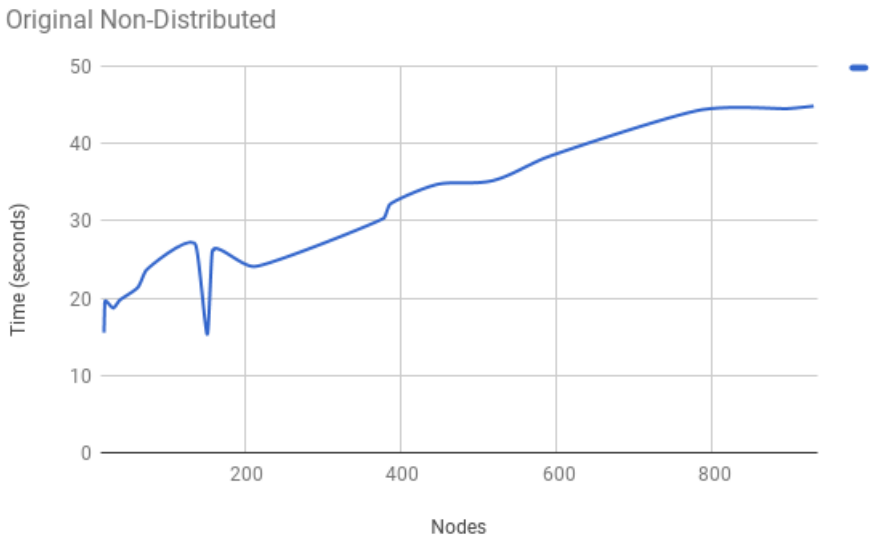


Figure 16.1: Graphic representing the results for Non Distributed execution of Original Defeaters Counter

.

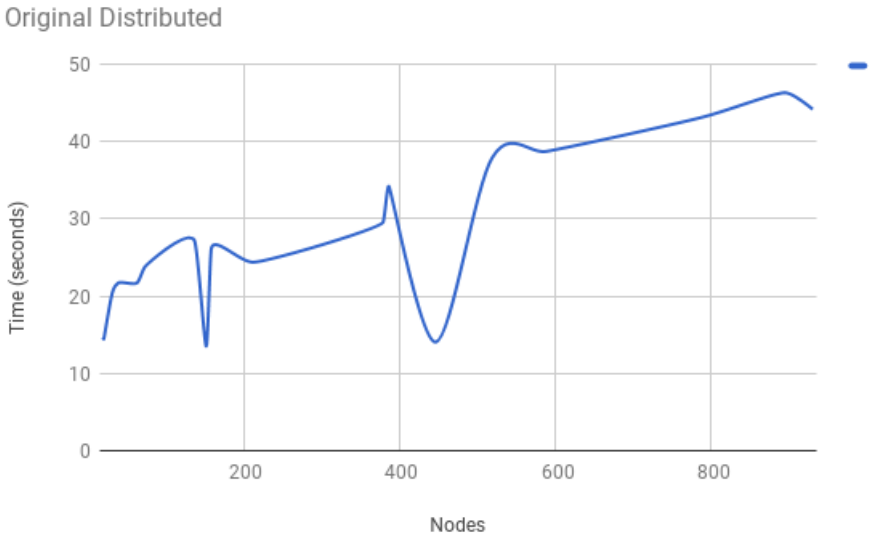


Figure 16.2: Graphic representing the results for Distributed execution of Original Defeaters Counter

.

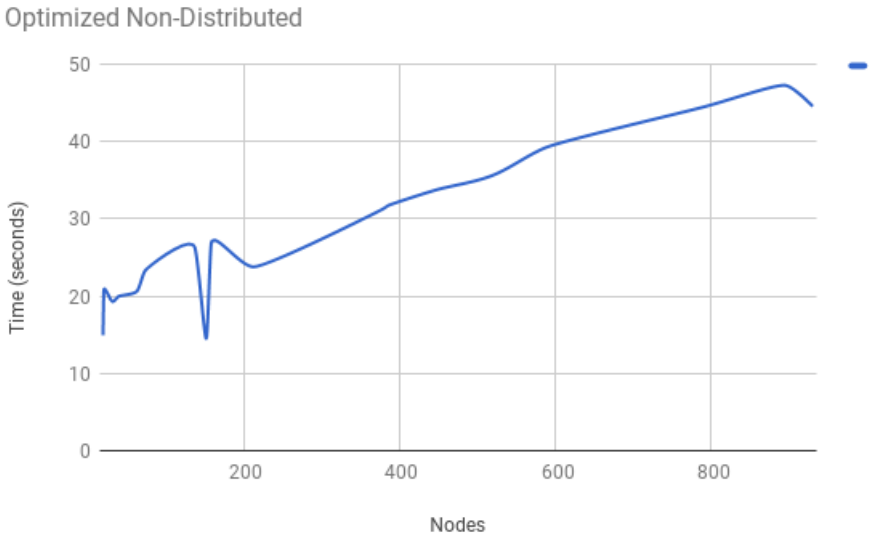


Figure 16.3: Graphic representing the results for Non Distributed execution of Optimized proposal for Defeaters Counter

.

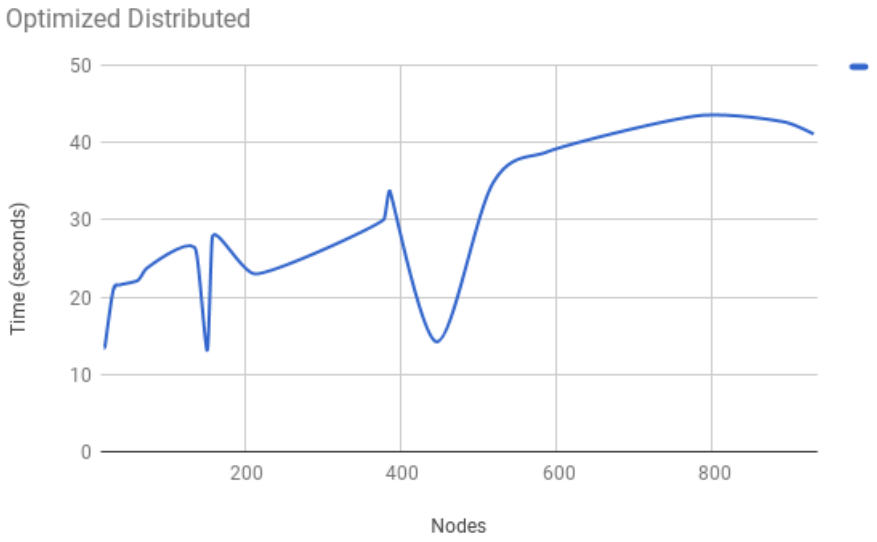


Figure 16.4: Graphic representing the results for Distributed execution of Optimized proposal for Defeaters Counter

.

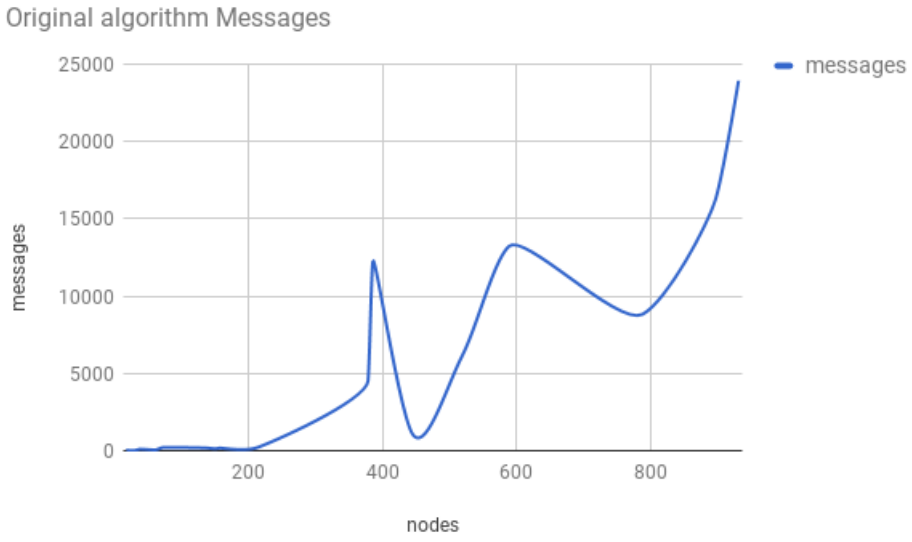


Figure 16.5: Graphic representing the messages sent by execution of Original Defeaters Counter

.

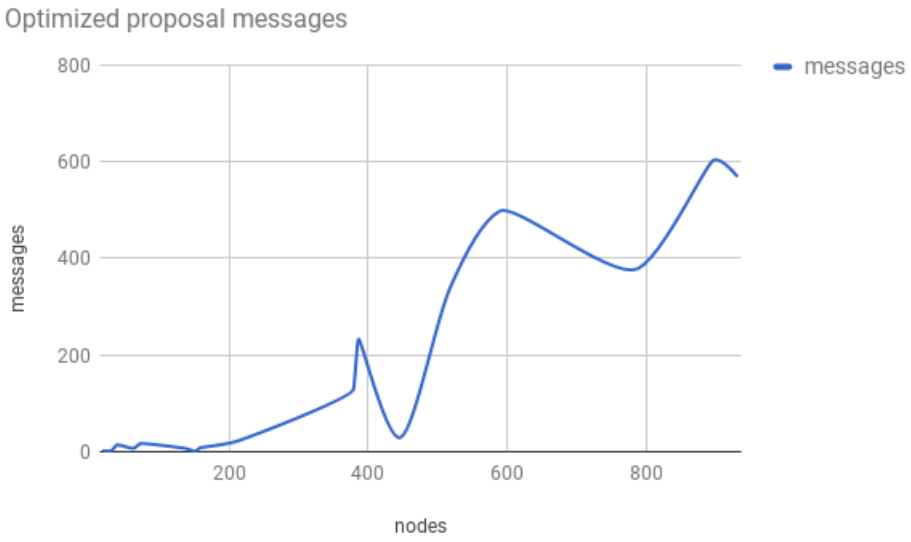


Figure 16.6: Graphic representing the messages sent by Distributed execution of Optimized proposal for Defeaters Counter

.

the optimized proposal probably would have a better behaviour on a conversations with a grater size. See graphics 16.5 and 16.6

Appendices

Appendix A

Research paper

A Distributed Approach for the Analysis of Discussions in Twitter

blind review

Abstract

In a recent work we have developed an argumentative approach for discovering relevant opinions in Twitter. A Twitter discussion is modeled as a weighted argument graph where each node denotes a tweet, each edge denotes a criticism relationship between a pair of tweets of the discussion and each node is attached with a weight that denotes the social relevance of the corresponding tweet in the discussion. In the social network Twitter, a tweet always refers to previous tweets in the discussion, so the obtained underlying argument graph is acyclic. Based on this structural feature, in this work, we introduce a distributed algorithm for computing the set of globally accepted opinions of a Twitter discussion. The set of accepted opinions is extracted by mapping the weighted argument graph into a valued argumentation framework and it is computed as the biggest set of tweets of the discussion that satisfies that it is consistent.

1 Motivation and Antecedents

In order to understand what are the major accepted and rejected opinions in different domains by Twitter users, in a recent work [Alsinet *et al.*, 2017] we have developed a system for analysis of discussions in Twitter.

The system architecture has two main components: a discussion retrieval and a reasoning system. The discussion retrieval component allows us to move from a discussion in Twitter (a set of tweets) in natural language to a weighted graph which is computed taking into account criticism relationships between tweets and three different attributes of a tweet: the number of followers of the author, the number of retweets and the number of favorites. The reasoning system component maps the weighted graph into a weighted argumentation framework and the set of socially accepted tweets in the discussion is evaluated from the weight assigned to each tweet and the criticism relationships between the tweets of the discussion, and it is computed as the ideal semantics [Dung *et al.*, 2007] of a valued abstract argumentation framework [Bench-Capon, 2003].

In abstract argumentation [Dung, 1995], a graph is used to represent a set of arguments and counterarguments. Each node is an argument and each edge denotes an attack between arguments. Several different kinds of semantics for abstract argumentation frameworks have been proposed that highlight different aspects of argumentation (for reviews see [Bench-Capon and Dunne, 2007; Besnard and Hunter, 2001; Rahwan and Simari, 2009]). Usually, semantics are given to abstract argumentation frameworks in terms of extensions. For a specific extension an argument is either accepted, rejected, or undecided and, usually, there is a set of extensions that is consistent with the semantic context.

The system developed in [Alsinet *et al.*, 2017] builds a weighted argument graph for a Twitter discussion, where each node denotes a tweet, each edge denotes a criticism relationship between a pair of tweets of the discussion and each node is attached with a weight that denotes the social relevance of the corresponding tweet in the discussion and it is computed from some tweet's attributes. In the social network Twitter, a tweet always answers or refers to previous tweets in the discussion, so the obtained underlying argument graph is acyclic.

Based on the fact that the graphs we obtain are acyclic, in this work, we introduce and investigate a distributed implementation of the skeptical approach based on the ideal semantics of a valued abstract argumentation framework. The ideal semantics for valued argumentation guarantees that the set of tweets in the solution is the maximal set of tweets that satisfies that it is consistent, in the sense that there are no defeaters among them, and that all of the tweets outside the solution are defeated by a tweet within the solution. That is, if a tweet outside the solution defeats a tweet within the solution, it is, in turn, defeated by another tweet within the solution. In other words, the solution is the biggest consistent set of tweets that defeats any defeaters outside the solution.

The defeat relationship between tweets is evaluated by criticism relationships between tweets and taking into account a social valuation function that for each tweet considers different information sources from the social network, such as the number of followers of the author, the number of retweets and the number of favorites. The distributed approach can be of special relevance for assessing Twitter discussions that involve a large number of tweets and the system can be applied in fields where identifying groups of tweets globally compatible or consistent, but at the same time that are widely accepted, is of particular interest, such as for instance for the

assistance and guidance of marketing and policy makers.

After this introduction, in the next section, we formalize the structure to model Twitter discussions and, in Section 3, we define the reasoning model for computing their solutions. Then, in Section 4, we present a distributed strategy for the implementation of the reasoning model based on the ideal semantics for a valued abstract argumentation framework. We end the paper with some conclusions and a discussion of future work.

2 A weighted graph for Twitter discussions

Following the approach proposed in [Alsinet *et al.*, 2017], in this section, we introduce a computation structure (a weighted graph) to represent a Twitter discussion considering only criticism relationships between pairs of tweets.

Definition 1 *(Twitter Discussion) A Twitter discussion Γ is a non-empty set of tweets. A tweet $t \in \Gamma$ is a tuple $t = (m, a, fl, r, fv)$, where m is the up to 140 characters long message of the tweet, a is the author's identifier of the tweet, $r \in \mathbb{N}$ is the number of retweets and $fv \in \mathbb{N}$ is the number of favorites. Let $t_1 = (m_1, a_1, fl_1, r_1, fv_1)$ and $t_2 = (m_2, a_2, fl_2, r_2, fv_2)$ be a pair of tweets of a Twitter discussion Γ. We say that t_1 answers t_2 iff t_1 is a reply to the tweet t_2 or t_1 mentions (refers to) tweet t_2.*

Definition 2 *(Discussion Graph) The Discussion Graph* (DisG) *for a Twitter discussion Γ is the directed graph (T, E) such that*

- *for every tweet in Γ there is a node in T and*
- *if tweet $t_1 = (m_1, a_1, fl_1, r_1, fv_1)$ answers tweet $t_2 = (m_2, a_2, fl_2, r_2, fv_2)$, with $a_1 \neq a_2$, and m_1 criticizes the claim expressed in m_2, there is a directed edge (t_1, t_2) in E.*

Only the nodes and edges obtained by applying this process belong to T and E, respectively.

Although our system allows us to analyze any discussion in Twitter (set of tweets), in this work we deal with discussions where a tweet answers previous tweets in the discussion. Moreover, in our approach, $(t_1, t_2) \in E$ iff $t1$ answers t_2 and the message of tweet t_1 does not agree with the claim expressed in the message of tweet t_2. So, the answers between tweets whose messages are not classified as criticisms, do not give rise to edges and, therefore, some nodes (tweets) of a discussion graph can be disconnected. Thus, we can find nodes for which the input or the output degree, or both, is zero.

Since the social network we are considering in this work is Twitter, every tweet of a discussion can reply at most one tweet, but can mention many tweets, and all of them are prior in the discussion. So, every tweet can answer and, in turn, can criticize many prior tweets of the discussion, and thus, every tweet can criticize many prior tweets from a same author and from different authors.

From an implementation point of view, in order to check if a tweet criticizes another tweet, we can use some of the components we have developed in [Alsinet *et al.*, 2017]. On the one hand, to check if a tweet t_1 replies a tweet t_2, the system can use the data structure in the JSON format provided by the Twitter API. In particular, this fact can be easily checked from the attribute `in_reply_to_status_id` of the object of t_1, which provides the tweet identifier to which t_1 replies. To check the set of mentions of t_1, the system searches for all authors mentions in the message of t_1. Every mention of an author is stored in the message with a label of the form: `@<author_identifier>`. So, t_1 mentions t_2 whenever the author's identifier of t_2 is in the set of mentions of t_1.

On the other hand, to check if a tweet does not agree with the claim expressed in a different tweet, the system uses an automatic labeling system based on Support Vector Machines (SVM). Our SVM model for labeling relations between tweets considers different attributes obtained from the tweets of an answer (t_1, t_2). On the one hand, we have attributes that count the number of occurrences of relevant words in the tweets t_1 and t_2. We have considered two kinds of words: regular words and stopwords. We have considered the inclusion of stopwords as attributes because the typical tweet is very short and the fraction of stopwords that can be giving information about the kind of answer could be relevant. For example, in the next tweet

```
@ponpimpampum @LL_Sosa Jajajaja...!!!
```

the stopwords `...` and `!!!` give information about the sentiment associated to the tweet.

On the other hand, we also consider attributes that have to be computed from the text and from the additional information that comes with the tweets. In particular, for each tweet these attributes are the number of images and the number of URLs mentioned in the tweet, the number of positive and negative emoticons and the sentiment expressed by the tweet. Our labeling system incorporates a sentiment analysis computation module [Hansen *et al.*, 2011; Nielsen, 2011] that given the set of words in a tweet it provides a sentiment value in the range $[-5, +5]$, where -5 is the most negative sentiment and +5 is the most positive sentiment. Finally, the sentiment value is incremented (or decremented) considering every positive (negative) emoticon.

Since SVM follows a supervised learning approach, we first have to train a model from an already labeled data set of answers. To this aim, we have collected a set of several Twitter discussions, on the Spanish language, and we have manually labeled the answers in the discussions to be able to train a SVM labeling model for Spanish discussions.

The training collection contains 12 discussions and a total of 582 pairs of tweets (answers). We have considered the creation of SVM models with different number of regular words (w) and different number of stopwords (s). The words in the collection are sorted by number of occurrences, and for an SVM model with w regular words, we select the first w most frequent regular words. The stopwords have been obtained from the natural language toolkit (NLTK) [Bird, 2006] and have been also ordered by number of occurrences, so we also select the s most frequent stopwords.

Using this training collection, we have trained four models with different values for w and s, in order to get a good labeling model. In order to compute the sentiment for the

tweets in this collection, we have taken the AFINN data[1] used in [Hansen *et al.*, 2011; Nielsen, 2011] and we have translated the words to Spanish. See [Alsinet *et al.*, 2017] for a detailed description of the SVM training model that we have implemented for checking criticism relationships between tweets from Spanish Twitter discussions.

Definition 3 *(Weighted Discussion Graph) A weighted discussion graph* (WDisG) *for a Twitter discussion* Γ *is a tuple* $\langle T, E, R, W\rangle$*, where*

- (T, E) *is the* DisG *graph for* Γ*,*
- R *is a nonempty set of ordered values and*
- W *is a weighting scheme* $W : T \rightarrow R$ *that assigns a weight value in* R *to each tweet in* T*, representing the social relevance of the tweet.*

Regarding the implementation, we instantiate the set of ordered values R to the natural numbers $\mathbb{N}$ and, for each node with tweet $t = (m, a, fl, r, fv)$, we consider the following weighting scheme:

$$W(t) = \lfloor \log_2(fl + 20 * r + 40 * fv + 1) \rfloor,$$

which takes into account not only the number of followers of the author, but also the number of retweets and favorites. This function allows us to quantify the orders of magnitude of the social relevance of tweets following the statistics about tweets and retweets defined in [Bild *et al.*, 2015], trying to give each attribute a weight proportional to its relevance. From the statistics shown in [Bild *et al.*, 2015], we observe that on weighting with twenty times the value of retweets and forty times the value of favorites, the magnitudes of the three attributes are comparable and one attribute does not dominate the others, since the number of followers is usually much bigger than the number of retweets and favorites. We finally compute the $\log_2$ function of the combined value, since we want to consider that one tweet is more relevant than another only if such combined weight is at least two times bigger for the first tweet. We will refer to this weighting scheme as `fl1r20fv40`.

Figure 1 shows a WDisG graph instance for a Twitter discussion obtained from the political domain using the `fl1r20fv40` weighting scheme. Each tweet is represented as a node and each criticism answer as an edge. The root tweet of the discussion is labeled with 0 (the tweet that starts the discussion) and the other nodes are labeled with consecutive identifiers according to the temporal generation order of the tweets in the social network. The nodes that appear disconnected in the graph correspond to tweets that have participated in the discussion in response to other tweets, but have not been classified as criticisms answers (nodes 3 and 5). The discussion has a simple structure, possibly one of the most frequent in Twitter. A root tweet starts the discussion and some answers criticize it, and there are not many criticisms between non-root tweets. The discussion contains 13 tweets and 14 criticisms answers. Nodes are colored in *blue scale*, where the darkness of the color is directly proportional to its weight with respect to the maximum value in the discussion.

[1] `http://www2.compute.dtu.dk/~faan/data/AFINN.zip`

Since our reasoning model defines the set of socially accepted tweets of a discussion as a set of tweets without effective conflicts, the solution is mainly defined by tweets that generate or receive criticism of other tweets and it is presented in next section.

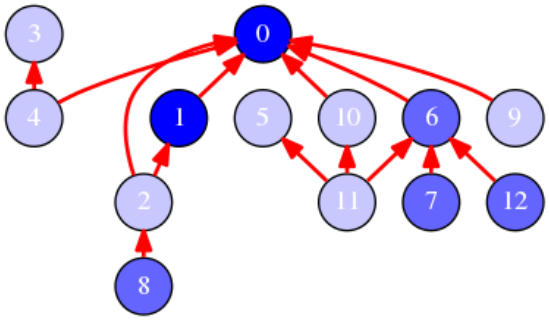


Figure 1: WDisG graph instance.

3 An argumentation-based reasoning model

Once we have introduced our formal representation model of Twitter discussions with criticism relationships between pairs of tweets, the next key component is the definition of the reasoning model used to obtain the set of socially accepted tweets. To this end, we use a valued abstract argumentation framework [Bench-Capon, 2003] for modeling the weighted argumentation problem associated with a WDisG graph and ideal semantics [Dung *et al.*, 2007] for defining the solution (the set of socially accepted tweets).

A valued argumentation framework (VAF) is a tuple $\langle A, \mathit{attacks}, R, \mathit{Val}, \mathit{Valpref}\rangle$ where A is a set of arguments, *attacks* is an irreflexive binary relation on A, R is a nonempty set of values, *Val* is a valuation function $\mathit{Val} : A \rightarrow R$ that assigns to each argument in A a weight value in R, and $\mathit{Valpref} \subseteq R \times R$ is a preference relation on R (transitive, irreflexive and asymmetric), reflecting the value preferences of arguments.

Given a Twitter discussion Γ and its WDisG graph $G = \langle T, E, \mathbb{N}, \texttt{fl1r20fv40}\rangle$ based on the weighting scheme $\texttt{fl1r20fv40} : T \rightarrow \mathbb{N}$, the VAF for G is the tuple $\mathcal{F} = \langle T, E, \mathbb{N}, \texttt{fl1r20fv40}, >\rangle$, where each tweet in T results in an argument of $\mathcal{F}$, each criticism answer in E results in an attack between the arguments of $\mathcal{F}$, $\mathbb{N}$ is the set of valuations or weights of the arguments, the weighting scheme `fl1r20fv40` is the weighting valuation function of $\mathcal{F}$, and the order relation $>$ of $\mathbb{N}$ is the preference relation between the valuations or weights of the arguments; i.e. a tweet t_2 is more valued than a tweet t_1 whenever $\texttt{fl1r20fv40}(t_2) > \texttt{fl1r20fv40}(t_1)$.

Then, a *defeat* relation (or effective attack relation) between tweets (arguments) is defined as follows: $\mathit{defeats} = \{(t_1, t_2) \in E \mid \texttt{fl1r20fv40}(t_2) \not> \texttt{fl1r20fv40}(t_1)\}$.

Moreover, a set of tweets (arguments) $S \subseteq T$ is *conflict-free* if for all $t_1, t_2 \in S, (t_1, t_2) \notin \mathit{defeats}$, and a conflict-free set of tweets $S \subseteq T$ is *maximally admissible* if for all $t_1 \notin S$, $S \cup \{t_1\}$ is not conflict-free and, for all $t_2 \in S$,

if $(t_1, t_2) \in$ *defeats* then there exists $t_3 \in S$ such that $(t_3, t_1) \in$ *defeats*. Finally, the *set of socially accepted tweets* of Γ, referred as the *solution* of Γ, is computed as the largest admissible conflict-free set of tweets $S \subseteq T$ in the intersection of all maximally admissible conflict-free sets. Remark that the tweets of a discussion that do not generate nor receive criticism, are always part of the solution.

Figure 2 shows the VAF solution for the WDisG graph instance of Figure 1. The nodes colored in blue are the tweets in the solution and the nodes colored in gray are the rejected tweets, where the darkness of the color is directly proportional to its weight. According to the `fl1r20fv40` valuation function, the tweets of the discussion are stratified in three levels denoting their relevance in the discussion. For each level, we find the following sets of tweets: level 0: $\{3, 2, 4, 5, 9, 10, 11\}$, level 1: $\{6, 7, 8, 12\}$, level 2: $\{0, 1\}$, being level 0 the lowest level and level 2 the highest one. The solution contains 7 tweets (Tweets 1, 4, 7, 8, 9, 11 and 12) of the 13 tweets of the discussion, and 6 tweets are rejected (Tweets 0, 2, 3, 5, 6 and 10). On the one hand, Tweet 1 defeats the root tweet, since Tweet 1 attacks the root tweet and both have the same weight. The same happens with Tweets 3, 5, 6 and 10, since Tweet 4 attacks Tweet 3, Tweet 11 attacks Tweets 5 and 10, and Tweets 7 and 12 attack Tweet 6. On the other hand, Tweet 8 defeats Tweet 2, since Tweet 8 attacks Tweet 2 and Tweet 8 is heavier than Tweet 2. Finally, Tweets 1, 4, 7, 8, 9, 11 and 12 are accepted since they do not have any defeater in the solution. Notice that in this discussion with high controversy around Tweets 0 and 6 (with a high number of attacks), we end up rejecting both tweets due to the weight they get during the discussion.

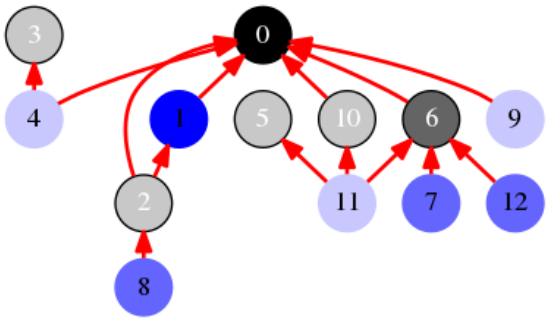


Figure 2: WDisG graph solution.

In [Alsinet *et al.*, 2017] we implemented a reasoning system for computing the VAF solution of a WDisG graph based on the algorithm for computing the ideal extension for an argumentation framework presented in [Dunne, 2008], but adapting it to work with valued arguments. Regarding the implementation we used an approach based on Answer Set Programming (ASP) available in the argumentation system ASPARTIX [Egly *et al.*, 2008], but we extended it to work with VAFs, as the current implementation in ASPARTIX only works with non-valued arguments. To develop such extension we modified the manifold ASP program explained in [Faber and Woltran, 2009] incorporating:

- the *valuation function* for arguments,
- the *preference relation* between argument valuations and
- the *defeat relation* relation between arguments (effective attack).

Now in this work, we define a distributed strategy for implementing the underlying reasoning algorithm for computing the ideal extension for a VAF. The algorithm takes a WDisG graph of a Twitter discussion and outputs the set of accepted tweets based on the valuation function, the preference relation and the computation of the largest admissible conflict-free set of tweets according to the defeat relation.

4 Distributed skeptical output computation

We design the distributed strategy for computing the solution of a WDisG graph using the distributed model of computation of Pregel [Malewicz *et al.*, 2010]. This model is appropriate for our problem, because the input for a Pregel algorithm is a directed graph, where the nodes can be in different states, and the goal of a distributed algorithm in Pregel is to compute the state of each node based on the state of the nodes' neighbors. Any Pregel algorithm starts initializing each node to some initial state. Then, the distributed computation follows a sequence of *supersteps* separated by global synchronization points until the algorithm finishes a point where every node is happy with its current state. This computation model is actually inspired by Valiant's Bulk Synchronous Parallel model [Valiant, 2011].

Within each superstep the nodes compute their state in parallel, executing a specific function that computes the new state of the node taking into account the possible messages sent by the nodes' neighbors in the *previous* superstep. Then, as a byproduct of the computation of the node state, some messages may be sent to some of the nodes' neighbors, that they will be processed in the next superstep. The idea is that the messages are used by nodes to indicate some change in their state, that could have some influence on the state of their neighbors in the next superstep. The superstep finishes when every node has computed its state and has sent the necessary messages to its neighbors.

In the computation model of Pregel, the input can be any directed graph. However, the algorithm we present here works only with discussion graphs that are acyclic, such that it allows us to solve discussions of big size with an efficient polynomial time distributed algorithm, where the state of each node depends only on the state of its attacking nodes. Observe that in the case of Twitter, where a tweet only answers previous tweets, the discussion graphs are always acyclic, so restricting the algorithm to consider only acyclic discussion graphs is not a real restriction for Twitter.

For the distributed computation of the skeptical output for a discussion acyclic graph, we propose a Pregel algorithm where each node can be in two states: accepted or not accepted (rejected), but a node also stores a defeaters count, to keep track of the number of *accepted defeaters* the node has, to actually compute its acceptance state. Initially, every node starts in the rejected state and with its defeaters counter equal to zero.

Algorithm 1 shows the pseudocode of the function used by

Algorithm 1 Compute the acceptance state for a node a

```
1: procedure a.CompAcceptance(i)                    ▷ Update acceptance state of a at superstep i
2:     a.storeCurrentAcceptanceState()
3:     for all msg ∈ a.received(i-1) do             ▷ Check defeaters count update
4:         if (msg.type == attacker) then a.updateDefeaters( msg.value )
5:     a.updateAcceptanceState()
6:     if ( a.StateChanged() ) then                 ▷ a changed to accepted or to not accepted
7:         sendToAllNeighbors( msg( attacker, (a.weight(), a.isAccepted() ) ) )
8:     else
9:         a.VoteToHalt()                           ▷ Vote to finish distributed computation
```

each node a to compute its state in a superstep i. [2] It works as follows. In the superstep i, a node a checks if its state has to be changed after updating its defeaters counter. The node a updates its defeaters counter based on the received messages (from the previous superstep $i-1$) from any nodes v connected with a with an incoming edge (v, a). These messages are processed in the for loop of lines 3 and 4. There are two possible changes that every message can produce on the defeaters counter of a. On the one hand, if in the previous superstep a node v, and such that (v, a) is an edge of the graph, changed its state to accepted then v sent a message to a of type $attacker$ and with value $(v.weight(), +1)$, indicating to a that its defeaters counter should be increased by one but only if v is a defeater of a (if $v.weight() \geq a.weight()$). On the other hand, such node v could instead have changed its state to not accepted, so in that case the message sent to a will be also of type $attacker$ but with value $(v.weight(), -1)$, indicating that its defeaters counter should be decreased by one if v is a defeater of a. The possible addition or subtraction to its defeaters counter, caused by a message msg sent by a node v is checked (and performed when needed) by calling the function $a.updateDefeaters(msg.value)$, where the value has the format indicated previously. Then, after updating the defeaters counter the state of a is updated calling the function $a.updateAcceptanceState()$, that checks if the updated defeaters counter is greater than zero. Finally, we call the function $a.StateChanged()$ (in line 6) to check whether the state of a changed (from not accepted to accepted or viceversa). If it changed, a sends an $attacker$ message to all its outgoing neighbors with value $(a.weight(), a.isAccepted())$, where the function $a.isAccepted()$ will return either +1 or -1 depending on whether the state of a is accepted or not accepted. These sent messages will be processed by the outgoing neighbors of a when the next superstep begins. If the state of a did not change, then a votes towards finishing the distributed computation of the skeptical output (line 9), but only when all the nodes agree to finish, in a same superstep, will the algorithm finish.

Consider the execution of the algorithm when solving the example discussion we have presented in Section 2, and whose solution is shown on Figure 2:

1. Before the first superstep, all the nodes start in the not accepted state and with defeaters counter equal to zero, so in the first superstep, as there are no incoming messages for any node, all of them will change their state to accepted. Then, any node with outgoing edges (all except 3 and 0), sends a message to its attacked nodes with value $(node.weight(), +1)$. Observe that nodes $4, 7, 8, 9, 11$ and 12 will remain accepted for the rest of the execution of the algorithm, as they will never receive any messages.

2. In the second superstep, the received messages produce that nodes $0, 1, 2, 3, 5, 6$ and 10 change to not accepted, as they receive messages from attacking accepted nodes that defeat them, so each one of these nodes (except 0 and 3) will send a message with value $(node.weight(), -1)$. Nodes $2, 3, 5, 6$ and 10 will remain not accepted for the rest of the execution of the algorithm.

3. In the third superstep, the messages sent in the previous superstep produce that nodes 0 and 1 change to accepted, and so node 1 will send a message to node 0 with value $(1.weight(), +1)$. Node 1 will remain accepted for the rest of the execution.

4. Finally, in the fourth superstep the message received by node 0 from 1, as 1 is a defeater for 0, will change the state of 0 to rejected, being this its final state and the end of the execution of the algorithm as no more messages are sent by any nodes.

Observe that the number of supersteps is equal to the maximum path length of the discussion graph, and although one can think about variants of this algorithm where less messages are sent, it seems that in the worst case it is not possible the have less supersteps than the maximum path length of the discussion graph. So, for typical Twitter discussions, where one has many branches in the discussion graph but not too deep, this algorithm may solve big discussions.

We have started to evaluate the performance of this algorithm on real Twitter discussions of different sizes, working with a small Spark cluster with five computers working with Linux. A table with preliminary results for a test set of Twitter discussions can be found at the URL: `https://www.dropbox.com/s/cf3ivkvpfcvgg4g/table.pdf?dl=0`.

[2] The pseudocode is written using object oriented notation, as the Pregel API is written in C++. However, our actual implementation is based on the Pregel implementation found on the Spark distributed programming framework, graphX, that is written in Scala.

5 Conclusions and future work

In this paper we introduce a distributed system for mining the set of globally accepted tweets of a Twitter discussion. We model discussions with a weighted argumentation graph, where each node denotes a tweet, each edge denotes a criticism relationship between a pair of tweets of the discussion and each node is attached with a weight, that denotes the social relevance of the corresponding tweet in the discussion and it is computed from some tweet's attributes, such as the number of followers of the author, the number of retweets and the number of favorites.

The set of accepted tweets is defined following an skeptical approach based on the ideal semantics of a valued argumentation framework. The ideal semantics for valued argumentation guarantees that the set of tweets in the solution is the maximal set of tweets that satisfies that it is consistent, in the sense that there are no defeaters among them, and that all of the tweets outside the solution are defeated by a tweet within the solution.

Our distributed strategy is based on the distributed computation model of Pregel which is appropriate for our problem, because the input for a Pregel algorithm is a directed graph, where the nodes can be in different states, and the goal is to compute the state of each node based on the state of the nodes' neighbors.

In our system each node (tweet) can be in two states, accepted or not accepted (rejected), and the algorithm works only with discussion graphs that are acyclic, such that it allows us to solve discussions of big size with an efficient polynomial time distributed algorithm, where the state of each node depends only on the state of its attacking nodes. As far as we know, the system is the first distributed implementation of an argumentative reasoning algorithm for social network analysis.

As future work, we plan to extend the representation model to consider support relationships between tweets and also to explore an efficient implementation of the distributed algorithm for bipartite graphs; i.e. graphs with no odd cycles.

References

[Alsinet *et al.*, 2017] Teresa Alsinet, Josep Argelich, Ramón Béjar, Cèsar Fernández, Carles Mateu, and Jordi Planes. Weighted argumentation for analysis of discussions in Twitter. *Int. J. Approx. Reasoning*, 85:21–35, 2017.

[Bench-Capon and Dunne, 2007] Trevor J. M. Bench-Capon and Paul E. Dunne. Argumentation in artificial intelligence. *Artif. Intell.*, 171(10-15):619–641, 2007.

[Bench-Capon, 2003] Trevor J. M. Bench-Capon. Persuasion in practical argument using value-based argumentation frameworks. *Journal of Logic and Computation*, 13(3):429–448, 2003.

[Besnard and Hunter, 2001] Philippe Besnard and Anthony Hunter. A logic-based theory of deductive arguments. *Artif. Intell.*, 128(1-2):203–235, 2001.

[Bild *et al.*, 2015] David R. Bild, Yue Liu, Robert P. Dick, Zhuoqing Morley Mao, and Dan S. Wallach. Aggregate characterization of user behavior in Twitter and analysis of the retweet graph. *ACM Trans. Internet Techn.*, 15(1):4:1–4:24, 2015.

[Bird, 2006] Steven Bird. NLTK: the natural language toolkit. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics, ACL 2006, Sydney, Australia*, pages 17–21, 2006.

[Dung *et al.*, 2007] Phan Minh Dung, Paolo Mancarella, and Francesca Toni. Computing ideal sceptical argumentation. *Artif. Intell.*, 171(10-15):642–674, 2007.

[Dung, 1995] Phan Minh Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial Intelligence*, 77(2):321 – 357, 1995.

[Dunne, 2008] Paul E. Dunne. The computational complexity of ideal semantics I: abstract argumentation frameworks. In *Proceedings of Computational Models of Argument, COMMA 2008, Toulouse, France*, pages 147–158, 2008.

[Egly *et al.*, 2008] Uwe Egly, Sarah Alice Gaggl, and Stefan Woltran. Aspartix: Implementing argumentation frameworks using answer-set programming. In *Proceedings of the 24th International Conference on Logic Programming, ICLP 2008*, pages 734–738, 2008.

[Faber and Woltran, 2009] Wolfgang Faber and Stefan Woltran. Manifold answer-set programs for meta-reasoning. In *Proceedings of Logic Programming and Nonmonotonic Reasoning, LPNMR 2009*, pages 115–128, 2009.

[Hansen *et al.*, 2011] Lars Kai Hansen, Adam Arvidsson, Finn Arup Nielsen, Elanor Colleoni, and Michael Etter. Good friends, bad news - affect and virality in Twitter. In *International Workshop on Social Computing, Network, and Services (SocialComNet 2011)*, 2011.

[Malewicz *et al.*, 2010] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010*, pages 135–146, 2010.

[Nielsen, 2011] Finn Arup Nielsen. A new anew: Evaluation of a word list for sentiment analysis in microblogs. In *Proceedings of the ESWC2011 Workshop on 'Making Sense of Microposts'*, pages 93–98, 2011.

[Rahwan and Simari, 2009] Iyad Rahwan and Guillermo R. Simari. *Argumentation in Artificial Intelligence*. Springer Publishing Company, 1st edition, 2009.

[Valiant, 2011] Leslie G. Valiant. A bridging model for multi-core computing. *J. Comput. Syst. Sci.*, 77(1):154–166, 2011.

Glossary

API Application Programming Interface. Set of subroutine definitions, protocols, and tools for building application software. 13

Commodity cluster computing it is known as the usage of personal computers widely available for purchase to make a cluster. It is preferable to have more low-performance, low-cost computers than fewer high-performance, high-cost computers. 12

Functional Functional programming is a programming paradigm that computationally works by evaluating mathematical functions, and it don't allow the changing states and mutable data.. 22

Mesos an open-source software project from Apache to manage computer cluster. 14

Object-oriented Is a programming paradigm based on conceptual objects. That "objects" can contains data, and functions. Instances can be created from this objects, an it can be seen as data-structures with code.. 22

Bibliography

[1] Marko Bonaci, Products and frameworks built on top of YARN, 2015. [Image] Retrieved from URL https://medium.com/@markobonaci/the-history-of-hadoop-68984a11704 [Online; accessed June, 2017].

[2] Hadoop Apache Docs, HDFS Architecture. [Image] Retrieved from URL https://hadoop.apache.org/docs/r2.7.2/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html [Online; accessed June, 2017].

[3] Hadoop Apache Docs, YARN Architecture. [Image] Retrieved from URL https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html [Online; accessed June, 2017].

[4] Hadoop Apache Docs, Hadoop Docs. URL: https://hadoop.apache.org/docs [Online; accessed June, 2017].

[5] Alex JF, Hadoop YARN Installation. URL: http://www.alexjf.net/blog/distributed-systems/hadoop-yarn-installation-definitive-guide/ [Online; accessed May, 2017].

[6] Wikipedia, Commodity computing. URL: https://en.wikipedia.org/wiki/Commodity_computing [Online; accessed May, 2017].

[7] Wikipedia, Apache Hadoop. URL: https://en.wikipedia.org/wiki/Apache_Hadoop [Online; accessed May, 2017].

[8] Search Cloud Computing, What is Hadoop?. URL: http://searchcloudcomputing.techtarget.com/definition/Hadoop [Online; accessed May, 2017].

[9] Gigaom, The history of Hadoop. URL: https://gigaom.com/2013/03/04/the-history-of-hadoop-from-4-nodes-to-the-future-of-data/ [Online; accessed May, 2017].

[10] Hortonworks, Apache Hadoop Yarn. URL: https://es.hortonworks.com/apache/yarn/ [Online; accessed May, 2017].

[11] Search Data Center, Apache Hadoop Yarn. URL: http://searchdatacenter.techtarget.com/es/definicion/Apache-Hadoop-YARN-Yet-Another-Resource-Negotiator [Online; accessed May, 2017].

[12] Spark, MLlib. URL: https://spark.apache.org/docs/latest/ml-guide.html[Online; accessed July, 2017]

[13] Spark, RDD. URL: https://spark.apache.org/docs/1.6.2/api/java/org/apache/spark/rdd/RDD.html[Online; accessed July, 2017]

[14] Wikipedia, Apache Spark. URL: https://en.wikipedia.org/wiki/Apache_Spark[Online; accessed July, 2017]

[15] Spark, GraphX. URL: https://spark.apache.org/docs/latest/graphx-programming-guide.htm[Online; accessed July, 2017]