

**School of Electronic
Engineering and
Computer Science**

Science without Borders (2012/13):
3-Months Project Report

**Dynamic
Graph
Computations
using Parallel
Distributed
Computing
Solutions**



MARCO AURELIO
BARBOSA FAGNANI
GOMES LOTZ

20/08/2013

Abstract:

One of the main problems that modern computation suffers is to easily perform computation on large graphs. Google published the Pregel paper trying to propose a computational model to solve this dilemma. Later, an open-source implementation of Pregel, called Giraph, was created.

After describing the current scenario for this kind of computation, the author briefly compares a few recent graph distributed frameworks and depicts his experiences in the Giraph framework. In these experiences are included implemented graph search applications (such as two Breadth First Search variations), data treatment scripts, benchmarks and comparisons between Giraph and Hadoop for single-node configuration.

Dynamic graphs are finally approached with an implemented solution and probable minor improvements to this solution. A proposal of possible further improvements is described in details. Among these details, are included considerations about parallelism, communication reduction and load balancing.

Introduction

When Google introduced the MapReduce paper, the implementations that came from it – among them Hadoop – solved many of the big data computation problems. Among these problems, one can point the fact that about 80% of the big data is unstructured (Zettaset, 2013). For this reason, MapReduce presents a possibility to perform structuration, allowing analysis (in example data mining) to be done over the computation results. Graphs, however, due to its structure presented a suboptimum performance for this approach. This mainly derives from its innate intensive I/O behaviour and the fact that transversing the graph would require required many chained MapReduce jobs. Taking this characteristic into consideration, Google released the Pregel paper with the “think-like-a-vertex” proposal.

In the moment that the project was first introduced to the author, the proposal was to use a Pregel implementation in order to develop features in the open-source framework that would allow the dynamic graph analysis. By dynamic graphs, one can define as graphs that are being updated from an outer source while the framework is performing computations over it. In other words, computations over a dynamic environment. A dynamic environment is an environment which can change while an analysing agent is deliberating about it. (Wiggins, 2013)

Current distributed graph computation frameworks do not support the dynamic graph processing. Many real life situations can be modelled, however, by dynamic graphs, among them one can point: social networks, communication networks and VLSI designs. In the scope of this document, an environment is considered dynamic when the input may change while the job is being performed. This report is the final product of the research project performed by the author for the Science Without Borders programme, of the Brazilian government.

In the document, the author begins comparing distributed graph frameworks that implement Pregel. After this, Giraph is detailed and the author experiences with it are described. Among these is how the framework was configured, how to perform test computations and the Giraph user community mail list.

Once the framework was mastered, a few implementations of consolidated search algorithms were done – in this case two implementation of breadth-first search. These applications were used as benchmarks for comparing the performance of single-node installations of Giraph and then the performance gain between Giraph and Hadoop for the same data set.

In the last section, the author shows an implementation of a dynamic graph computation. An implementation of this kind of computation is detailed in flowcharts and improvements are suggested. A possible further improvements approach to add dynamic graphs computations to the framework is then proposed and its features described in details.

1. Related Works

1.1 Scenario

1.1.1 MapReduce

MapReduce was originally mentioned in a Google paper (Dean & Ghemawat, 2004) as a solution to make computations in big data using a parallel approach and commodity-computer clusters. The document describes how a parallel framework would behave using the Map and Reduce functions from functional programming over large data sets.

In this solution there would be two main steps – called Map and Reduce –, with an optional step between the first and the second – called Combine. The Map step would run first, do computations in the input key-value pair and generate a new output key-value. The Reduce step would assemble all values of the same key, performing other computations over it. As a result, this last step would output key and value pairs. One of the most trivial applications of MapReduce is to implement word counts¹. The algorithm to write one is described in the original paper.

In order to avoid too much network traffic, the paper describes how the framework should try to maintain the data locality. This means that it should always try to make sure that a machine running Map jobs has the data in its memory/local storage, avoiding fetch it from the network. Aiming to reduce the network through put of a mapper, the optional step, described before, is used. The Combiner performs computations on the output of the mappers in a given machine before sending it to the Reducers – that may be in another machine.

The document also describes how the elements of the framework should behave in case of faults. These elements, in the paper, are called as worker and master. They will be divided into more specific elements in open-source implementations.

Since the Google has only described the approach in the paper and not released its proprietary software, many open-source frameworks were created in order to implement the model.

1.1.2 Apache Hadoop and HDFS

One of the most popular frameworks is named Hadoop². This framework is an open-source software framework, maintained by the Apache foundation. It follows many of the premises shown in the Google's MapReduce Paper and it focus in commodity hardware for the cluster. It was created by Doug Cutting and Mike Cafarella in 2005, while they were working in Yahoo! Company.

Between the companies that use Hadoop, one can list (Apache Foundation, 2013):

¹ A word count is an algorithm that takes a text as an input and counts how many times a given word appears in the text.

² Hadoop is a short for High-Availability Distributed Object-Oriented Platform. The original name, however, came from name of Doug Cutting son's toy elephant. This is also the reasons why the elephant is the project symbol. (Vance, 2009)

- Adobe: uses Hadoop and HBase³ in structured data storage and in order to perform internal processing.
- Facebook: uses in order to store copies to internal log, also uses as a source for reporting/analytics and machine learning.
- Spotify: uses for data aggregation, content generation, analysis and reports. (Bie, 2012)

In order to implement the framework, a complete new file system, named HDFS – that stands for Hadoop Distributed File System –, had to be created. The file system is composed of a Namenode, a SecondaryNamenode and several Datanodes, which implement the master/slave relationship describe in the MapReduce Paper. A Datanode contains blocks of data. By default, in the HDFS, all the data is split in 64 Mbytes blocks and these blocks are replicated among the Datanodes. The standard replication factor is 3, which means that one should find the exactly the same data block in three different Datanodes (Yahoo! Inc, 2013). Some Datanodes also perform the MapReduce jobs.

As an attempt to keep track of all the replicated data blocks, the Namenode stores all the meta-data information and is a single point of failure in the system. All its stored information is related to which Datanode holds which blocks of the file system. The user can define checkpoints, where all the information existing in the Namenode is copied to the SecondaryNamenode. In case of a Namenode failure, one can then return the system to the last checkpoint available in the SecondaryNamenode. All the computations between the failure and the checkpoint are lost. The Namenode and the SecondaryNamenode do not store blocks of data nor perform computations, since they should use all their performance to preserve the framework integrity.

Usually in the same machine that implements the Namenode, a daemon called JobTracker runs. The JobTracker acts as the master described in the MapReduce paper, coordinating the TaskTrackers (that are the slave nodes that perform the map reduce computation). The JobTracker receives heartbeats from every single TaskTracker in order to verify that all mappers and reducers are working correctly. In the case of a TaskTracker failure, the JobTracker analyses the cause of the failure and determines if a new TaskTracker should be allocated or if the job should be declared as failed. (White, 2012)

Many companies decided to provide Hadoop as a commercial platform for big data computation, among them one can list Cloudera (Cloudera Inc, 2013), that is the producer of the version currently installed in the Informatics Teaching Laboratory (ITL) machines of Queen Mary – University of London.

1.1.3 MapReduce 2.0 and YARN

After the release of Hadoop version 0.20.203 the complete framework was restructured and a new framework was created. This framework is called MapReduce 2.0 (MRv2) or YARN (Apache Foundation, 2013). The short YARN

³ Hbase is an open-source distributed database, maintained by Apache and written in Java. It is modelled after the Google's BigTable paper. (Fae & all, 2006)

stands for “Yet Another Resources Negotiator” (Monash, 2012). This modification has as a main goal to make the MapReduce component less dependent of Hadoop. Furthermore, this will improve the possibility of using programming interfaces other than MapReduce over it and increase the scalability of the framework.

As mentioned in (Monash, 2012) in July 23rd of 2012: “YARN is in alpha. YARN is expected to be in production at year-end, give or take”. In the moment that this document is being written, YARN stills in alpha⁴, more than half a year after the expected release date. In June of 2012, Cloudera decided to include the unstable alpha YARN version in its products, warning consumers away from actually using it. The ITL laboratory has this alpha distribution of Hadoop running.

One of the main characteristics of YARN is that the original JobTracker was split into two daemons: the Resource Manager and the Application Master. The first monitors the cluster status, indicating which resources are available to which nodes. It arbitrates the resources among all the applications in the system. The last manages the parallel execution of any job. In Yarn this is done separately for each individual job. It is a framework specific library and negotiates the resources from the Resource Manager and works with the Node Manager to execute and monitor tasks. (Apache Foundation, 2013)

The TaskTracker was transformed in the Node Manager, that with the Resource Manager compose the data-computation framework. It is a machine individual framework agent that takes care of the containers by monitoring the resource usage (memory, disk, CPU, network) and reporting the results to the Scheduler that is inside the Resource Manager.

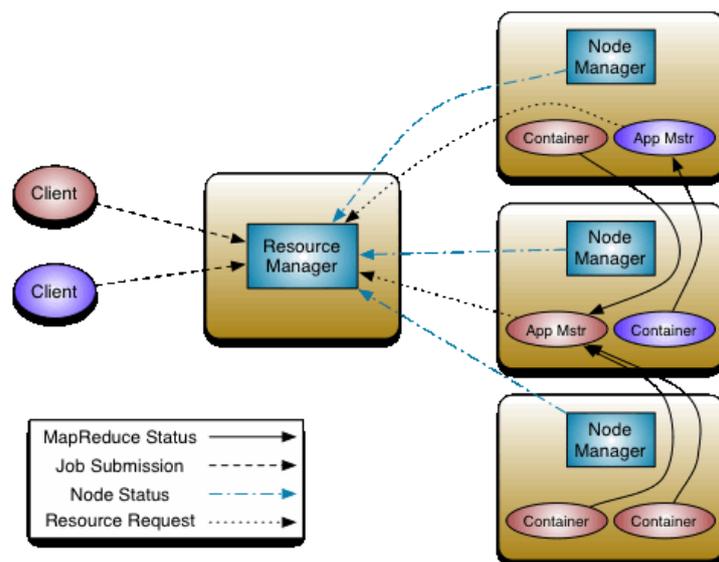


Figure 1. YARN Framework (Apache Foundation, 2013)

⁴ Named alpha release phase in a development life cycle, it is the first phase where the software tests begin. Usually this moment of development is characterized by tests using white box techniques (which the description surpasses the scope of this document). After this, additional validation is performed using a black box technique, by another team. Moving to the black box testing is called as alpha release (PCMAG, 2013).

The ResourceManager daemon has several components; among them one can point: the Scheduler and the ApplicationsManager. The first allocates the resource for the various running applications and takes into considerations capacity and queues constraints. The second is a basic scheduler, without tracking status or monitoring an application. It only performs the scheduling taking into account the resource requirements of an application, by taking an abstract notion of the ResourceContainer – which incorporates elements such memory, CPU, disk and network. According to (Apache Foundation, 2013), in the early versions only memory is supported. At the moment that the present document it being written, no modifications were done into the framework and it still supports only memory scheduling (Ryza, 2013).

The ApplicationsManager does the job-submissions acceptance work. It makes the negotiation with the first container for executing the application specific ApplicationMaster and also provides the service for restarting the ApplicationMaster container on failure – therefore helping to solve one of the single points of failure that was present in the original Hadoop framework.

One of the major features of YARN is that it is backward compatible with Hadoop non-YARN. For this reason, applications that were developed for the previous versions do not need to be recompiled in order to run on a YARN cluster.

1.1.4 Graphs

As mentioned by (Lotz, 2013): “Graphs, in computer science, are an abstract data type that implements the graph and hypergraph concepts from mathematics. This data structure has a finite – and possible mutable – set of ordered pairs consisting of edges (or arcs), and nodes (or vertices). An edge can be described mathematically as $edge(x,y)$ which is said to go from x to y . There may be some data structures associated with edges, called edge value. This structure can represent numeric attributes (for example costs) or just a symbolic label. Also there is the important definition of $adjacent(G, x, y)$, that is a function that tests for a given graph data structure G if the $edge(x,y)$ exists.”

A graph can be classified in many types taking into consideration its structure. In example, an undirected graph is a graph in which the edges do not take into account the orientation, as opposed to the direct graphs in which they take.

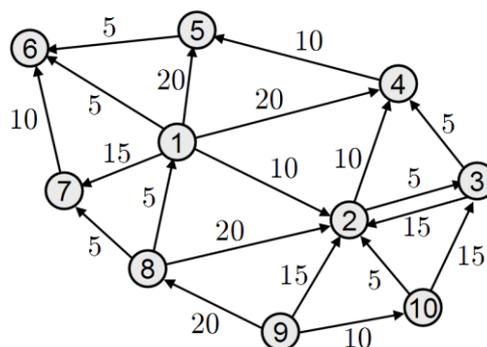


Figure 2. Example of a directed graph (Carvalho, 2009)

1.1.5 Graph Analysis Problem

In the big data analysis, graphs are usually known as hard to compute. (Malewicz & all, 2010) points that the main factors that contribute to these difficulties are:

1. Poor locality of memory access.
2. Very little work per vertex.
3. Changing degree of parallelism.
4. Running over many machines makes the problem worse.

Before the introduction of Pregel (Malewicz & all, 2010), the state-of-art solutions to this problem were:

- The implementation of a customized framework: that demands lots of engineering effort.
- Implementation of a graph analysis algorithm using the MapReduce platform: this proves to be inefficient, since it has to store too much data after each MapReduce job and requires too much communication between the stages (which may cause excessive I/O traffic). Aside from that, it is quite hard to implement graph algorithms, since there is no option to make calculations node/vertex-oriented (Stylianou, 2013).

In order to completely solve the I/O problem, one may use a single computer graph solution. This would, however, require not commodity hardware and would not be scalable. The existing distributed graph systems that existed before the Pregel introduction were not fault tolerant, which may cause the loss of huge amount of work in the case of any failure.

1.1.6 Pregel

In order to create a fault-tolerant graph-computation framework, Google implemented Pregel⁵. The project source-code is, however, proprietary and only a high-level description of the solution was released in the form of a paper. In this document, which was released only in 2010, an explanation using algorithms illustrates how the C++ code of the framework behaves in a job. The main propose of Pregel is to think in the graph computation as a vertex.

Pregel makes use of the Bulk Synchronous Parallel (BSP) abstract computer, which is a bridging model⁶ in order to produce parallel algorithms. This model was proposed in (Valiant, 1990). A real implementation of a BSP machine is a group of processors connected by a communication network. The processors should have

⁵ According to (Malewicz & all, 2010): "The name honours Leonhard Euler. The Bridges of Königsberg - which inspired his famous theorem - spanned the Pregel river."

⁶ According to (Valiant, 1990): "A bridging model 'is intended neither as a hardware nor a programming model but something in between'".

access to fast local memory and may also contain multiple (and different) computation threads. In this machine, the BSP computation is made in a series of individual stages, called *supersteps*. Each one of this superstep has three components:

1. Concurrent Computation: In this stage, the computations in each processor are done. Each individual computation does not depend on the other computation, which is why it is named asynchronous, and should only make use of the values stored in its processor memory.
2. Communication: After the end of the previous step, the processors (and its processes) exchange information (also called messages) between them. This communication should take form as a one side *put* and *get* calls (asynchronous communication), instead of two side *send* and *receive* calls (that are called blocking communication).
3. The Synchronization Barrier: After a process reaches this point, it waits all the other processes – even the one in others processors – to finish their communication actions. Once all the communication is over, another superstep begins, with the updated values stored in memory in order to feed the new computation.

One can visualize the three steps of a BSP computation in the figure below:

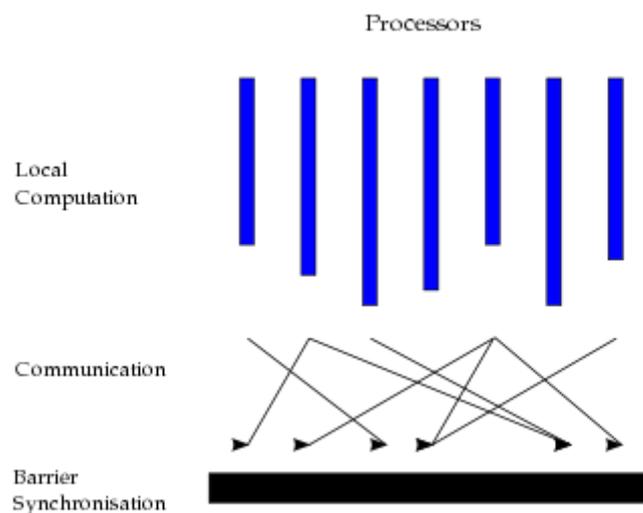


Figure 3. BSP model (Jungblut, 2011)

In Google's Pregel, the graph vertices are split between the processes running the job. These processes are called *workers*. The workers are managed by the *master* node. In a single machine implementation, there may be more than a single worker and the number of workers is limited to the memory of that machine. In a cluster, there may be several machines with multiple workers inside each one of them. The number of workers inside a machine is only limited by the resources that the machine can allocate to the workers. One may easily visualize that Pregel job is

a Map only job, since there are nothing to be reduced. Each worker receives several nodes from the master in order to perform the computation. Each node is computed individually and its computation, just like in the BSP model, does not take into account result of computations that are happening concurrently.

Each vertex can be described as a state machine with two basic states: active and inactive. In the first superstep, all vertices are active. Each vertex, after performing its own computation and sending its messages to the other vertices, votes to halt (which changes the state of that vertex to inactive). If an inactive vertex receives a message in the beginning of a superstep, it becomes active once again and performs new computations. A Pregel job will not stop until all vertices have voted to halt or a stop condition has been achieved – in example the maximum number of supersteps has been reached.

One may visualize below an illustration of the vertex state machine:

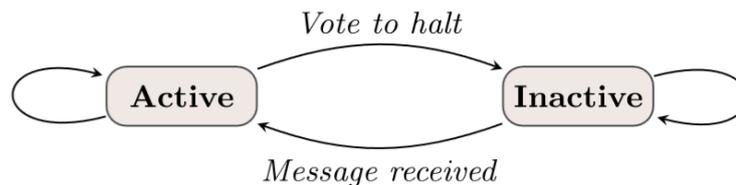


Figure 4. Vertex State Machine (Malewicz & all, 2010)

One of the most characteristic features of Pregel is the use of aggregators, which can be defined as a mechanism for global communications, monitoring and data. Each vertex can send data to one or more aggregator in a given superstep. The framework combines the values sent by all the vertices and make calculations over it using the master. The result of the calculations of each aggregator is available to the vertices of the beginning of the next superstep. The aggregators run inside the Master node, which may create huge I/O traffic depending on the algorithm running and the number of vertices sending values to it.

Similar to the MapReduce framework, combiners are also available in Pregel. They are used in order to reduce the amount of messages received by a vertex in the beginning of each superstep. This reduces the messages traffic in the framework, optimizing the network usage.

The fault-tolerance problem in Pregel is solved through the checkpoint system, which is very costly since it uses persistent storage. Through this system, a user can define certain supersteps to be checkpoints. In the checkpoints, the Master performs the following procedures:

- Asks the workers to save their state into persistent storage. The stored data includes the Vertex Identification, value, edges values and incoming messages.
- Master saves the aggregator value (if there is an aggregator) into persistent storage. (Peixiang, 2010)

After this procedure, every single time that the Master detects a failure in the workers it can restore the job to the stored state. All the computations done from the moment of the checkpoint to the failure will be lost and have to be redone.

One can easily visualize the Pregel Master-Worker framework in the figure below:

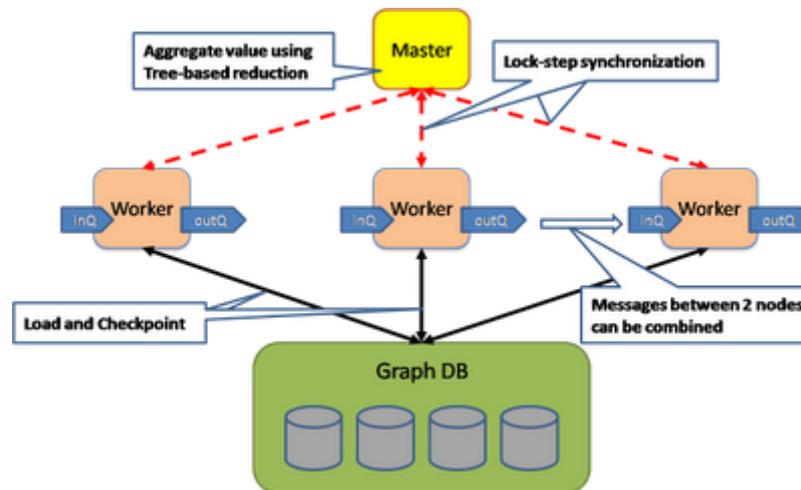


Figure 5. Pregel Master-Worker framework (Ho, 2010)

1.1.7 Apache Giraph

Giraph is an open-source implementation of Google's Pregel. It was originally donated to the Apache Foundation by Yahoo (Homan, 2012) and its first version (v0.1) was released in February of 2012. By the time this document is being written, the framework still quite new – about a year old. Nowadays Twitter, LinkedIn and Facebook uses Giraph in order to process most of the social circles data.

Currently, the Giraph computations hijack the Hadoop framework in order to run as a Hadoop Map only job. For this reason, a stable Hadoop version is required for Giraph⁷. The recommended stable builds uses Hadoop 0.20.203. Apache ZooKeeper is used in the framework in order to elect a master between the workers. One may notice that any worker can act as master and one will automatically take over if the current master fails (Apache Foundation, 2013)

The Giraph framework provides several ways to implement the graph algorithm. The framework comes with several pre-made format reader/writers in order to produce the desired input and output. It also supports to save each vertex state in disk after each superstep, rather than storing it into volatile memory. (Apache Foundation, 2013)

The current Giraph version is 1.1 and is available in GitHub⁸ webpage (Apache Foundation, 2013). In order to install Giraph, one has to first configure a

⁷ This fact may change with the introduction of YARN, that should provide the whole Hadoop resource managing framework, leaving the MapReduce to be an add-on.

⁸ GitHub is a web-based hosting service in order to allow software development using the git revision control system, that works doing versioning.

Hadoop cluster and then use Apache Maven⁹ to build all the content. There are several Maven profiles for Giraph, in order to configure it to several Hadoop versions. There are even profiles for installing Giraph over YARN. The author experiences with it will be presented later in this document.

2 Activities

Before start the dynamic graph implementation, the author had to first select a framework to be used. In this section, the criteria for the framework selection are delineated, followed by the experiences of how to setup the environment.

Once the framework is running, a few implementations of classic graph search are done, followed by some experiments and comparisons between Giraph and Hadoop. Finally, a dynamic computation implementation is introduced, with possible improvements and further improvements model proposal.

2.1 Graph Processing Frameworks comparison

In the first two weeks of the project, the researcher had to select and discuss with his supervisor which Pregel open-source implementation would be used in this project. After reading the papers of Hama (Sangwon & al, n.d.), Mizan (Khayyat & al, 2013), Sedge (Yang & al, 2013) and Giraph (Martella, 2012), the author decided to select Giraph.

Hama was not selected because it stands for Hadoop Matrix and is a BSP implementation. For this reason Pregel can be implemented over it. It is not, however, the main goal of the HAMA framework to implement Pregel. Mizan was left apart because it is too recent to be used. Its paper was only released in late April 2013. In open-source projects, the newer tend to be the more unstable/with lesser documentation.

Only two projects then were left: Giraph and Sedge. Giraph was select to be used due to its increasing number of contributors and the companies that were already running clusters based on it – that included Facebook, Twitter and LinkedIn. Due to the industrial acceptance of this framework, it will probably become a standard for graph data processing in the next years.

2.2 Apache Giraph system

Right after deciding the framework to be used, the author had to become used with the Giraph framework and the framework that it was going to run upon, Hadoop.

⁹ Apache Maven has similar function to Apache Ant. Both are build automation tools for java projects. Their implementations and concepts, however, differ.

2.2.1 Configuring the Hadoop framework

One of the main problems of open-source projects is the lack of up-to-date documentation (Stylianou, 2013). Since the author was using his own computer, it was required for him to set a single-node pseudo-distributed setup of Hadoop running over Ubuntu 13.04 Linux. There are many tutorials on the internet about how to setup Hadoop, and most of them are out-of-date.

One of the most recurring problems of a badly installed Hadoop 0.20.203 machine is the JAVAHOME error. This error consists in Hadoop not being able to find where is the path to the Java Virtual Machine installation in that machine, even with an environmental variable correctly set in the scripts. After a few badly-succeeded installation attempts, the author finally found an up-to-date tutorial to install that Hadoop version (Apache Foundation, 2012).

Once the Hadoop framework was up and running, it was time to start implementing some common applications in it in order to get used with the syntax and interface. The student took a couple of weeks in order to implement all the laboratories available in Queen Mary - High Performance Computing module (Cuadrado & White, 2013) and get familiarized with all the idiosyncrasies associated with the platform. Between the programs, one can list many variants of word count applications, stocks analysers and an implementation of a non-Pregel graph analysis tool. This last one will be used for benchmarking the single-node setup later in this document. The time invested in this part was also to learn about the functionalities of the HDFS, that is one of the main input/output sources for Giraph.

2.2.2 The users mail list

One of the main ways to learn about an open-source project is the mail list. In this list, there were many useful posts, most of them complaining about the unstable behaviour of certain distributions of Hadoop, mainly YARN (Apache Foundation, 2013). There were also many posts of new applications of Giraph and problems that many authors were finding. Reading those e-mails helped the author to find quick solutions to many problems that he found while creating Giraph applications.

2.2.3 Giraph for Hadoop 0.20.203 setup

After getting used with Hadoop, the time came to install Giraph. Unfortunately, since it is a really new platform, took some time for the author to find a tutorial of how to install it. Actually, so far there is only one tutorial available in the internet (Stylianou, 2013). Taking into consideration this fact, this is probably because the developers assume that any possible user must be already experienced with GitHub projects – that is where most of the open-source projects are stored – and Apache Maven. Even this tutorial is quite incomplete and many configurations had to be performed. Among the not listed configurations one can point the fact that Giraph framework requires at least 4 Mappers running per job. For this reason, one has to

manually modify the already installed Hadoop framework in order to be able to run it. This is a common mistake that many non-experienced users report in the mail list.

While using Maven for installation, there was a constant error happening during the installation of HBase and thus restricting the installation of Giraph. In order to solve this, the author submitted a ticket, JIRA-700¹⁰, to the developers and skipped any test/uses of the HBase for the framework.

Once Giraph was up and running considerably stable, the author and his supervisor decided to start using YARN as the base for Giraph, since it is the Hadoop distribution used in the ITL computers.

2.2.4 Giraph for Yarn setup

Again, the installation of Hadoop YARN was not trivial. Since it is an alpha version, many of the tutorials available in the web were inconsistent. After setting the Hadoop framework, there were some small modifications in the syntax of a few HDFS commands and scripts – that started being considered deprecated by the framework. For this reason, in order to control the start and stop of the HDFS daemons and reduce the amount of command line code every time that YARN was started/stopped, the researched had to write a few basic shell scripts. These scripts are available in the appendix.

By the time that Hadoop YARN was configured, the author read a few emails from the Giraph-user list that indicated problems in Pure-YARN profile installation when using Maven. The solutions were available in a recently released JIRA Giraph-688¹¹. This time, there were no errors in the installation differently from what happened with Hadoop 0.2.203

The main test for Giraph is an example called PageRankBenchmark. This benchmark, however, was not running correctly in the YARN distribution. All the other classical examples were also having problems with the ZooKeeper. Due to the reduced age of the project, there were no reports of this problem in the internet and the author had to send it to the user mail list. A few days later, one of the developers (Apache Foundation, 2013) answered that it would be required to launch a standalone version of ZooKeeper and that the Zookeeper it was not currently integrated with YARN Giraph. Since this would be a non-trivial procedure, the supervisor oriented the author to return to the Hadoop non-YARN distribution.

2.3 Benchmarking and Testing Giraph installation

By the time the researcher returned to the Hadoop 0.2.203 version, there was already a solution for the JIRA GIRAPH-700 ticket. With it was possible to perform a

¹⁰ Available at: <https://issues.apache.org/jira/browse/GIRAPH-700>

¹¹ Available at: <https://issues.apache.org/jira/browse/GIRAPH-688>

complete installation of the framework without the HBase error. As mentioned before, one of the main tests to check if the Giraph framework is up and running correctly is the PageRankBenchmark application. In this application, the user enters the number of workers, the number of edges, number of vertices and the number of supersteps to be performed. The platform generates the input/output data by its own. It has the following command line syntax:

```
hadoop jar path/to/giraph_file/giraph_jar_file.jar
org.apache.giraph.benchmark.PageRankBenchmark -e 1 -s 3 -v -V 5000000 -w 3
```

Table 1. Giraph PageRankBenchmark command line call

The syntax calls the Hadoop daemon and specifies that the job is in a jar file. After that, it specifies the jar and the class that is going to run. The PageRankBenchmark class takes a few arguments: -e indicates the number of vertices; -s the number of supersteps until stop; -v enables verbose; -V the number of vertices; -w the number of workers.

A classic output for this call can be seen here:

```
hduser@prometheus-UX32A:/usr/local/giraph/giraph-core/target$ hadoop jar giraph-1.1.0-
SNAPSHOT-for-hadoop-0.20.203.0-jar-with-dependencies.jar
org.apache.giraph.benchmark.PageRankBenchmark -e 1 -s 3 -v -V 500 -w 3
13/08/05 19:26:53 INFO job.GiraphJob: run: Since checkpointing is disabled (default), do not allow
any task retries (setting mapred.map.max.attempts = 0, old value = 4)
13/08/05 19:27:11 WARN bsp.BspOutputFormat: checkOutputSpecs: ImmutableOutputCommitter will
not check anything
13/08/05 19:27:11 INFO mapred.JobClient: Running job: job_201308051926_0001
13/08/05 19:27:12 INFO mapred.JobClient: map 0% reduce 0%
13/08/05 19:27:33 INFO mapred.JobClient: map 25% reduce 0%
13/08/05 19:27:36 INFO mapred.JobClient: map 50% reduce 0%
13/08/05 19:27:39 INFO mapred.JobClient: map 75% reduce 0%
13/08/05 19:27:42 INFO mapred.JobClient: map 100% reduce 0%
13/08/05 19:27:47 INFO mapred.JobClient: Job complete: job_201308051926_0001
13/08/05 19:27:47 INFO mapred.JobClient: Counters: 30
13/08/05 19:27:47 INFO mapred.JobClient: Job Counters
13/08/05 19:27:47 INFO mapred.JobClient: SLOTS_MILLIS_MAPS=55468
13/08/05 19:27:47 INFO mapred.JobClient: Total time spent by all reduces waiting after reserving
slots (ms)=0
13/08/05 19:27:47 INFO mapred.JobClient: Total time spent by all maps waiting after reserving
slots (ms)=0
13/08/05 19:27:47 INFO mapred.JobClient: Launched map tasks=4
13/08/05 19:27:47 INFO mapred.JobClient: SLOTS_MILLIS_REDUCE=0
13/08/05 19:27:47 INFO mapred.JobClient: Giraph Timers
13/08/05 19:27:47 INFO mapred.JobClient: Superstep 0 PageRankComputation (ms)=466
13/08/05 19:27:47 INFO mapred.JobClient: Superstep 3 PageRankComputation (ms)=565
13/08/05 19:27:47 INFO mapred.JobClient: Superstep 1 PageRankComputation (ms)=823
13/08/05 19:27:47 INFO mapred.JobClient: Input superstep (ms)=565
13/08/05 19:27:47 INFO mapred.JobClient: Total (ms)=8821
13/08/05 19:27:47 INFO mapred.JobClient: Shutdown (ms)=90
13/08/05 19:27:47 INFO mapred.JobClient: Superstep 2 PageRankComputation (ms)=611
```

13/08/05 19:27:47 INFO mapred.JobClient:	Setup (ms)=5697
13/08/05 19:27:47 INFO mapred.JobClient:	Graph Stats
13/08/05 19:27:47 INFO mapred.JobClient:	Aggregate edges=500
13/08/05 19:27:47 INFO mapred.JobClient:	Sent message bytes=0
13/08/05 19:27:47 INFO mapred.JobClient:	Superstep=4
13/08/05 19:27:47 INFO mapred.JobClient:	Last checkpointed superstep=0
13/08/05 19:27:47 INFO mapred.JobClient:	Current workers=3
13/08/05 19:27:47 INFO mapred.JobClient:	Current master task partition=0
13/08/05 19:27:47 INFO mapred.JobClient:	Sent messages=0
13/08/05 19:27:47 INFO mapred.JobClient:	Aggregate finished vertices=500
13/08/05 19:27:47 INFO mapred.JobClient:	Aggregate vertices=500
13/08/05 19:27:47 INFO mapred.JobClient:	File Output Format Counters
13/08/05 19:27:47 INFO mapred.JobClient:	Bytes Written=0
13/08/05 19:27:47 INFO mapred.JobClient:	FileSystemCounters
13/08/05 19:27:47 INFO mapred.JobClient:	HDFS_BYTES_READ=176
13/08/05 19:27:47 INFO mapred.JobClient:	FILE_BYTES_WRITTEN=87576
13/08/05 19:27:47 INFO mapred.JobClient:	File Input Format Counters
13/08/05 19:27:47 INFO mapred.JobClient:	Bytes Read=0
13/08/05 19:27:47 INFO mapred.JobClient:	Map-Reduce Framework
13/08/05 19:27:47 INFO mapred.JobClient:	Map input records=4
13/08/05 19:27:47 INFO mapred.JobClient:	Spilled Records=0
13/08/05 19:27:47 INFO mapred.JobClient:	Map output records=0
13/08/05 19:27:47 INFO mapred.JobClient:	SPLIT_RAW_BYTES=176

Table 2. Classic output of PageRankBenchmark call

Following the only available tutorial so far on the internet, the author also ran the SimplePageRankComputation (Stylianou, 2013) and the SimpleShortestPathsComputation (Stylianou, 2013) with the following input data set:

[0,0,[[1,1],[3,3]]
[1,0,[[0,1],[2,2],[3,1]]]
[2,0,[[1,2],[4,4]]]
[3,0,[[0,3],[1,1],[4,4]]]
[4,0,[[3,4],[2,4]]]

Table 3. Giraph PageRank and ShortestPath examples input data

The data set format is called JSON format and has the following syntax:
[vertex id, vertex value, [[id of the vertex that this edge points to, edge value], [...]]

The outputs for the SimplePageRankComputation and the SimpleShortestPathsComputation, were respectively:

0 0.16682289373110673
4 0.17098446073203238
2 0.17098446073203238
1 0.24178880797750443
3 0.24178880797750443

Table 4. Giraph SimplePageRankComputation output data

0 1.0
2 2.0

1 0.0
3 1.0
4 5.0

Table 5. Giraph SimpleShortestPathsComputation output data

2.4 Analysing Giraph source code

Before starting the year abroad through the Science without Borders project, the author of this document had no experience with UNIX systems or Java. He had, however, experience with C++, that is an object-oriented language in a few aspects similar to Java. After taking the module of Software Tools for Engineers, the author started getting closely familiarized with UNIX systems.

Many modules like High Performance Computing and Computational Genomics demanded that the author learned Java language during the second semester. Due to the reduced size of the projects there was no need for an Integrated Development Environment (IDE)¹². Therefore much of the code was done in software as gedit or nano.

The Giraph framework, however, consists in more than 6075 files. For this reason, the choice of an IDE was required. There were two possible choices between the more popular: Eclipse or NetBeans. The first was selected due to its high consolidated use for Java development in the industry. After a few tutorials in order to master the IDE, the supervisor helped the researcher to correctly set all the configurations in order to work.

Among the many classes and packages available in Giraph, the author found that there are a reduced number of packages and classes of paramount importance. Among them, once can say:

- `Org.apache.giraph.graph`: it contains the `Vertex.java`, which is the basic class used to perform the computation. Any class that is going to have a compute method must implement it from this class. Also it defines how to create a vertex, which has four attributes: Vertex Id, Vertex Value, List of edges and values and messages destined to that vertex.
- `Org.apache.giraph.io.formats`: has the classes that can be directly used or derived in order to read and write files.
- `Org.apache.giraph.master`: Has the master classes, which performs computations between the supersteps, following the Pregel description.
- `Org.apache.giraph.aggregators`: Contains implementations of the aggregators originally described in the Pregel article. Also includes basic computations has maximum and minimum values.
- `Org.apache.giraph`: Contains the `GiraphRunner` class, that is used to parse command line options into the framework and configures Giraph jobs.

¹² Also knows as Interactive Development Environment, consists in a software application that provides features in order to turn the software development easier to the computer programmers.

- `Org.apache.giraph.graph.VertexResolver`: It allows a user defined solution for solving multiple modifications requests for the same vertex in the same superstep. Through this, the Giraph user may define how the framework should behave and the request precedence.

2.5 Running new Giraph Code

Giraph jobs require access to many classes that are inside the original Giraph and Hadoop Java Archive (JAR). For this reason, a path to those archives should be specified. In order to create a simple solution, the author dumped the Hadoop Java Archive inside the original Giraph Java Archive. In order to easily solve the dependencies between the written applications and the libraries, was created a java package named `uk.co.qmul.giraph` that would have all the user made applications and would be inserted in the previous mentioned java archive.

The first application was written, inserted in the modified Java Archive and successfully submit as a Giraph job. In order to automatize this production, the author created the script named `updateJar.sh`, that is in the appendices. This script takes as an argument the location of the java class files and the name of the package. After that, it updates the Giraph jar file with those classes. One should configure the script itself to fits its own purposes/file system architecture.

In parallel with these activities, there was the need to extract the Java documents from Giraph and Hadoop. Giraph website has a broken link to those documents, thus the only possible way to have access to them was to extract them from the git repository. The author created a script, using Unix Shell and AWK, that seeks a specific file extension and copies it to the target directory, keeping the original directory structure. This script is available in the appendices, named `getJavaDocs.sh`.

Later during the development of Giraph applications, the author studied how Hadoop and Giraph code executed in order to submit a job. With this, the author found that it was possible to define an option in the Hadoop calling script. The option `-libjars [path]` would not require the complete jar to be submitted in every computation, since the jars required could be referenced through this. The large jar solution was kept because the `-libjars` option was still unstable in the moment that this document was being written (Orlando, et al., 2013).

2.6 Giraph Breadth-First Search

2.6.1 Classical Breadth-First Search Algorithm

Breadth-First Search is one of the most well-known ways to search Graph structures. It performs the search in a graph by going across the search tree several

times, replacing, in each iteration, the analysed node by its children. Between the advantages of this type of search, one can list:

- It will always find a solution to the problem, if there is one.
- It always guarantees the shortest action sequence to that solution – one may note that the solution may not be optimal, since it does not take into account possible weights in the graph edges.

And as disadvantages, one can point that:

- It may cause, in case of too great search spaces, a combinatory explosion into the search agenda.

In Artificial Intelligence, an agenda is list of tasks that should be performed by an agent. In the BFS approach, the agenda should be a queuing function that would remove the current node to be expanded from the front of the agenda and then append its children in the back of the agenda. One may note that the agenda is always rotating, since it is always dealing (and thus removing) with its first element (node).

The figure bellow, from (Stuart J. Russell, 2009), elucidates what is the sequence of actions performed.

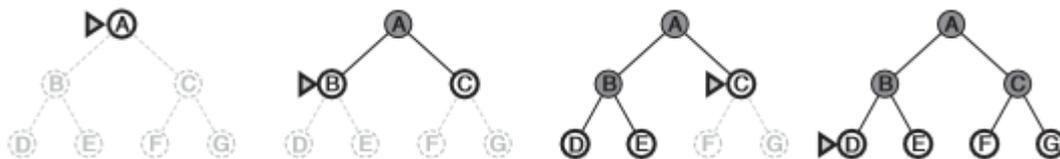


Figure 6. Breadth-First Search Sequence (Stuart J. Russell, 2009)

2.6.2 Breadth-First Search as a Pregel Algorithm

Since Pregel has the “think like a vertex” approach, a Breadth First Search should also be vertex oriented. In this scenario, one should define if it is a target oriented or a structure oriented search. The two variations will be described later.

The starting procedure for both is the same: in the first iteration every single vertex that is not the start vertex should vote to halt (which makes them to become inactive, the only way to become active is thought receiving a message). The start vertex is recognized by its vertex Id. For this reason, the only vertex that will perform the compute method in the first superstep will be the start vertex.

The start vertex, at the end of the first superstep, sends messages to all adjacent nodes and then vote to halt. One may realise that the content of the messages is not really important, since what really matters is the message itself, that will make the nodes receiving them to become active.

For a target oriented search, the new activated nodes will check if they are the target vertex and may perform two actions: in the negative case, it will send messages to its adjacent nodes; in the positive case, it will tell an aggregator that the

target node has been found. The negative scenario makes the computation continues until the target vertex is found or a defined number of maximum supersteps is reached. After finding the target vertex, there may be other vertices that are receiving messages. For this reason, it is important for them to know that the computation should be finished. This is done by the aggregator mentioned before. The nodes, in the beginning of each superstep, check the aggregator to see if the target has been found and, in the positive case, they vote to halt without sending messages. This makes the computation converge, since there are no more active vertices.

For structure oriented search, each node value is started in the first superstep with the maximum value possible to be attributed to an integer. When that node becomes active, it means that the vertex received a message. The now active vertex checks its current value. If it is the maximum integer value, it changes that value to the current superstep number – since in each superstep, a new depth should be explored – otherwise it just votes to halt because it has already been explored in a previous superstep. This allows all nodes to find out their minimum depth from the target vertex.

In the next sections, experiments were performed in order to benchmark the performance of Giraph for single node setup. Although Giraph benchmarks should be performed in a distributed node, even in single node setup it shows considerable gain of performance over Hadoop implementation. All the data was computed three times, in order to reduce possible metrological errors. One should keep in mind that the measured times are just to illustrate the performance. In order to create a more serious benchmark to the framework, one should do the comparison in a controlled computer system, with distributed node setup and similar datasets for Hadoop and Giraph.

The specifications of the machine used in these benchmarks are:

- 4 Gigabytes Memory, DDR3 @ 1600MHz
- Intel Core i7-3517U @ 1.90GHz, 4 cores.
- Ubuntu 13.04, 64 bits.

2.6.3 Running Breadth-First Search Target-Oriented using Giraph

The code of breadth first search target-oriented implementation is available in the appendices. It is target oriented because it looks for a defined node before finishing the computation, rather than just checking the minimum depth that a node is from the source. One may visualize in the code that there is the necessity of registering the aggregators in the MasterCompute method. This is due to the fact that the master of the computation may wish to change aggregators values. In each Superstep, first the Master compute method is executed, followed then by the Workers supersteps methods and vertex methods, respectively.

In the example, a Google Plus dataset, used in the High Performance Computing module (Cuadrado & White, 2013), was modified in order to be in the

JSON format. The dataset has 10000 vertices and 121210 edges. The source vertex is the vertex 1 and it takes 4 supersteps until it finds the target vertex 708.

Increasing the number of workers, in this specific configuration, makes the computation take more time to complete. It may, however, be more useful for larger datasets in a cluster wide Giraph installation.

Number of Workers	Execution Number Time (ms)			Average (ms)
	1	2	3	
1	6686	6213	5898	6265
2	7005	7374	7052	7143
3	9910	10571	10075	10185
4	14043	14277	14041	14120

Table 6. Time measures BFS target-oriented on Google Plus data.

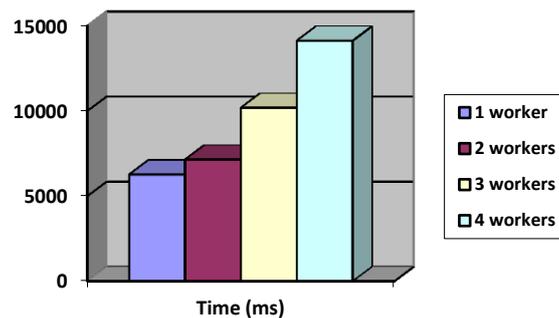


Figure 7. Average time per worker for BFS target-oriented for Google Plus data.

It was also made experiments with a larger dataset, with 100000 vertices and 1659270 edges. There original data set is larger and was reduced in size by the author of this document. The original can be found at (Leskovec, 2009).

Once loading the whole non-modified dataset into the experiments machine memory was not possible (it resulted into the out-of-memory error), there were two possible solutions for the problem: Making out-of-core computations – which are beyond the scope of this document – or just reducing the dataset. The original AWK script used to convert the Google Plus data to JSON format was modified and is available in the appendix. With this modification, it now removes nodes over a user-defined ID, as well as any edges to that node. The edges references had to be removed from the input edges due to the default vertex resolving of Giraph framework. If a vertex that does not exist receives a message, it will be created by the framework. This would create the out-of-memory error after a few computation steps.

In the end of this computation, 99630 vertices were computed, taking 6 supersteps until the computation finishes, finding the target vertex. The difference between the input and the computed number of vertices is because there are vertices that are not reached before the end of the computation at all, never

receiving messages before the end of the computation. The results of the experiment are given below:

Number of Workers	Execution Number Time (ms)			Average (ms)
	1	2	3	
1	12867	10581	11735	11727
2	10876	14197	11012	12028
3	23643	12852	16541	17678
4	25073	27883	31862	28272

Table 7. Time measures BFS target-oriented on journal data.

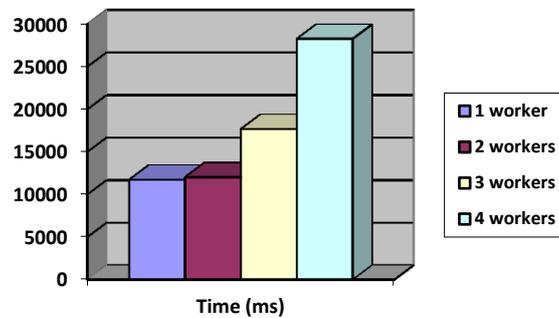


Figure 8. Average time per worker for BFS target-oriented for journal data.

2.6.4 Running Breadth-First Search Structure-Oriented using Giraph

As described before, this method uses the current superstep number to calculate the minimum depth of all the nodes in a breadth-first search starting in the source vertex. For the Google plus data mentioned before, all connected vertex have their depth calculated in 5 supersteps. One can easily visualize that there is also no improve of performance for increasing the number of workers. This is due to the single-node configuration used in the system. The code used is in the appendix. The start vertex used was the vertex with the ID 708.

Number of Workers	Execution Number Time (ms)			Average (ms)
	1	2	3	
1	6668	6653	6057	6459
2	7443	6936	7427	7268
3	11319	7803	10282	9801
4	14584	14054	13998	14212

Table 8. Time measures BFS Structure-oriented on Google Plus data

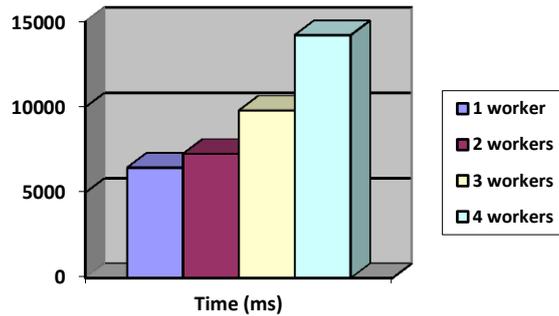


Figure 9. Average time per worker for BFS structure-oriented for Google Plus data.

For the same reduced journal dataset described before, it solves the structure in 13 supersteps, aggregating 1659270 edges and 99940 vertices.

Number of Workers	Execution Number Time (ms)			Average (ms)
	1	2	3	
1	11312	12344	12139	11931
2	14204	13993	13695	13964
3	24091	24332	23731	24051
4	27545	26794	28931	27756

Table 9. Time measures BFS structure-oriented on journal data.

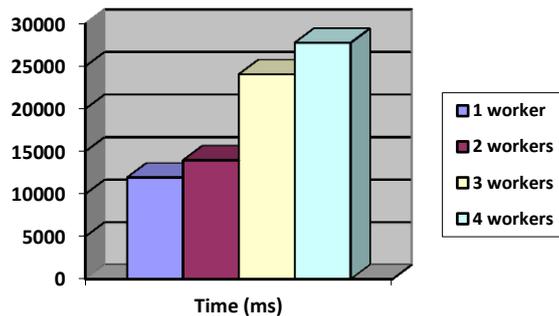


Figure 10. Average time per worker for BFS structure-oriented journal data

2.6.5 Comparison with the same implementation in MapReduce

In the first semester of 2013, the students of the module of high performance computing in Queen Mary – University of London had to develop a Hadoop MapReduce solution to perform graph breadth-first search. The code processed graphs in order to achieve the same result that the algorithm implemented in Giraph for breadth-first search structure-oriented, which means that it gave, as an output, the minimum depth of a vertex from the target vertex. It was used the Google was dataset. There was, however, a pre-processing of the input data in order to make the

dataset compatible with the second MapReduce job. The time of this pre-processing is not going to be measured into the time comparisons.

As expected, in order to process the whole graph, many chained MapReduce jobs would have to be submitted until a goal was reached. A single MapReduce job performs the same role that a superstep in the Giraph approach.

The table below shows the timing of a single map reduce job. One may note that, due to the single-node configuration, only one map task was launched.

Number of Mappers	Number of Reducers	Execution Number Time (ms)			Average (ms)
		1	2	3	
1	1	34206	32104	33234	33181
1	2	35213	36118	38153	36494
1	3	42254	42205	41183	41880
1	4	44225	46275	44245	44915

Table 10. Hadoop times for multiple Reducers jobs

Assuming – it is, however, a rough approximation – that Giraph has equal timing per superstep one can calculate the average time per superstep. From table 9, the average superstep time would be:

$$\frac{\text{total time}}{\text{Number of supersteps}} = \frac{6459 \text{ ms}}{5} = 1291 \text{ ms}$$

Equation 1. Average superstep time

The figure below illustrates a comparison of the timing taken to process a superstep by Hadoop and by Giraph, for the same algorithm and structure. One can see that actually the whole computation in Giraph is finished before a single superstep is processed in MapReduce. Other algorithms may increase the Hadoop performance, at the cost of increase structure/code complexity. According to this measures, Giraph breadth-first search structure oriented implementation is at least 25 times faster the same implementation on MapReduce.

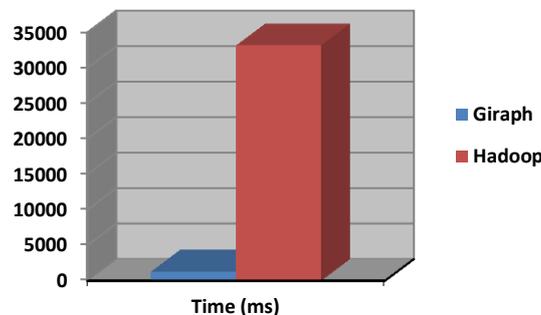


Figure 11. Time comparison between Giraph superstep and Hadoop job time

2.7 Dynamic Graph on Giraph

Once the author was already used with normal Giraph coding, it was time to modify the current applications in order to allow the dynamic input. Firstly, the author got in contact with Claudio Martella, who made the first presentation about Giraph (Martella, 2012) in FOSDEM 2012.

Through this conversation, the author confirmed that the framework did not provide any type of support to dynamic graph computation, as was expected. However, in order to create this, Martella suggested that one could try using an injector vertex in order to get new input data.

2.7.1 First Dynamic Graph Application

In order to implement the dynamic graph and still be coherent with the Pregel logic, the author had to deliberate about how the job should behave. Since the framework uses an object-oriented programming language, the author implemented a user defined master computation class with an observer object inside. In the initialization of the master (that happens before the beginning of the first superstep), the observer object gets a timestamp. This stamp indicates what time the tracked files were modified, in Coordinated Universal Time¹³. In the first version, only one object is tracked, but this can be easily modified to a variable amount of objects.

In the beginning of each superstep, the first thing to be executed by the framework is the compute method in the master. In this method, the file observer checks if there was a modification of the timestamp of the tracked files. If a modification is confirmed – by comparing the new and old time stamps –, it uses an aggregator to signalize to the workers that a modification was detected in the file system. Another aggregator provides the path of the file that was modified. The path will be used by the injector vertex later.

In the procedure of dynamic input analysis described above, one of the aggregators is of Boolean type and another one is of Text type. The Boolean type aggregator is called *underUpdate* and it should be true from the moment that an update is detected in the tracked files to the moment that the injector finishes injecting the new vertices. All the other moments it should remain false. The Text type is used to store the path.

During a pre superstep routine, called before the first superstep, the injector vertex is created. This may cause race conditions, a possible solution for this is discussed later. The injector node has a special vertex identification, used to differentiate that vertex from all the others. In the implemented solution, it is the vertex with ID -1. This may vary for another graph data set. Although the request to create a new vertex happens in the beginning of the superstep, the vertex is only

¹³ UTC is used in UNIX systems in order to provide the correct time to file systems. It counts the amount of milliseconds from first of January of 1970.

created, by the framework, after the superstep is finished. For this reason, one may assert that it takes the complete first superstep to correctly initialize the system.

After the setup phase, the worker sequence of actions was modified. In order to do this, the author changed the *org.apache.giraph.graph.ComputeCallable* class. With this modification, the worker, before running the compute of that vertex, checks if the computation has the `underUpdate` flag set. If this flag is true, it makes all vertices that are not the injector ask for their removal. This is important because even vertices that are inactive will perform this action. If the vertex analyse this flag during the computation time, vertices that are not active will not request to be removed.

The injector vertex will, in normal conditions, be always voting to halt. If the `underUpdate` flag is true, it will call the `inject` method of the computation class. The abstract class *org.apache.giraph.graph.computation* also was modified. Now it contains an `inject` method that is user defined.

Using the approach described above, the compute method of the vertices should not worry about the modification of update flags or the vertex identification (by this one should understand differentiate normal and injector vertices). A injector vertex will only be processed when there is an update in the file system, remaining halted in all other supersteps.

The injector vertex stays idle during the removal superstep. Any new injected vertex with the same ID of a vertex that demanded removal may generate conflicts on the framework default vertex resolving routine. There is a solution for this problem using a vertex resolver and will be described later in this document.

On the following superstep, it is expected to be only the injector vertex in the framework. The injector then accesses the path, given by the Path aggregator, and uses a modified JSON reader to generate new vertices from the designated input file. There are several ways to check if the framework is ready for injection, one of them is forcing the injector to take two supersteps for the injection. Another possibility is getting the number of vertices in the computation. If there is only the injector vertex, then new vertices may be added.

One may easily visualize that these vertices will only be added to the framework in the end of the current superstep. The master waits two supersteps before returning to file tracking on a file modification was flagged. Any solution that needs that graphs modifications be solved at any moment of the superstep may harm the BSP structure, since it would require that the processes do not be independent between then during the computation itself.

In approach detailed above and the code available in the appendices of this document, takes two supersteps in order to update the job vertices with the new values. It may be possible to solve in in only one superstep, this solution will be discussed later in this document.

The File observer was created in the Master Compute class and not into the worker itself in order to preserve the load balance between workers. This may, however, increase the non-parallel part of the code. If there are many file timestamps being analysed, the best solution would be to distribute them between workers. One

should realize that if only one worker had the “observer node” and had to check the file system in each superstep, it would increase its I/O when compared to other workers, therefore taking more time to complete its computation. An overloaded worker would increase the time per superstep due to its own extra load and thus increase the idle time of the other workers, which is not desired.

In this solution, only the worker with the injector vertex runs the special injection method. Since all nodes, in this approach, check the same aggregator, there is no vertex unbalancing. Taking into consideration that the injector vertex is inside a fixed worker, the injected vertices will be allocated to that same worker. It is desirable, thus, that the framework performs some sort of load balancing on the injected vertex. Distributing those new injected vertex among workers would to keep the balance among them. This was not implemented in the current solution and only the default framework procedure is taken into consideration.

Follows a flow chart that indicates the decision making process of the master and the vertices:

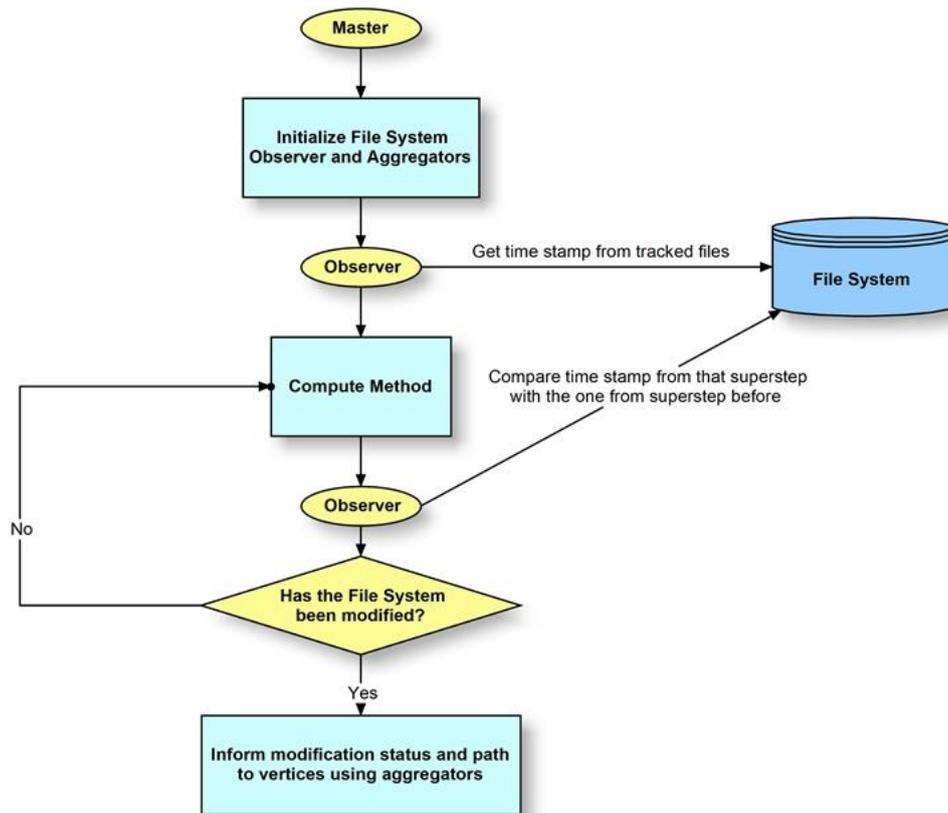


Figure 12. Master decision making flowchart

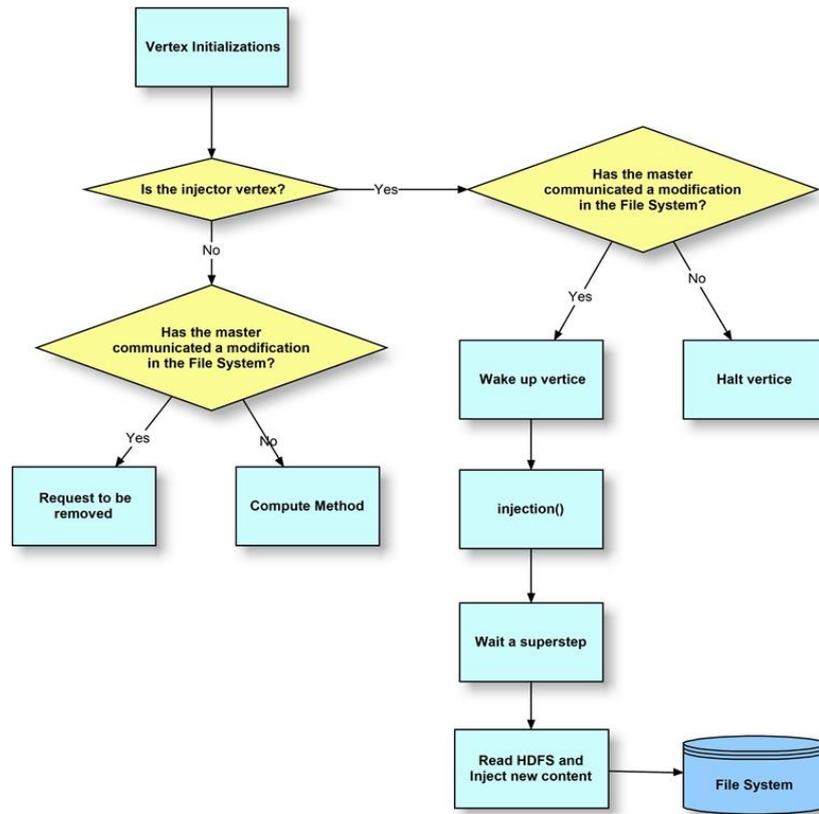


Figure 13. Vertex decision making flowchart

2.7.3 Implementation time analysis

In order verify that the first implementation already presents a performance gain compared with submitting a brand-new Giraph job, the author made two experiments. The experiments consisted on running three times the dynamic graph application in two different datasets. It was performed using two workers in the single-node configuration.

The execution had 33 supersteps, where vertices requested removal in supersteps that ended with number 9 and were injected in the next superstep. In other words, an update in the framework dataset takes two supersteps. For the Google database, used in the Breadth-First Search experiments, one can achieve the results bellow. The time is measured in milliseconds, the delete field is the removal superstep time and the add field is the time of the injection superstep.

Execution Number	Update 1		Update 2		Update 3		Setup time	Shutdown time	Input superstep
	delete	add	delete	add	delete	add			
1	678	1155	611	810	888	744	6262	66	1062
2	667	1299	619	814	488	721	3144	89	1113
3	722	1221	4788	801	499	744	2681	78	1123

Table 11. Dynamic Graph application timing for Google plus dataset

From the data above, one can easily sum the removal and injection times per application and achieve the average value. The resulting value is then compared with the sum of the setup, shutdown and input superstep times. A factor of speed is compared by the following formula:

$$Performance\ Factor = \frac{Framework\ setup\ time}{Total\ update\ time}$$

Execution Number	Average Update time			Framework Setup time	Performance Factor
	delete	add	total		
1	725.6	903	1628.6	7390	4.8
2	591.3	944.6	1536	4346	2.8
3	2003	955.3	2958.3	3882	1.31

Table 12. Performance Gain for Google plus dataset

From the values of the experiments above, one can see that using the dynamic graph application implemented by the author for reduced datasets is at least 31.2% faster than just submitting a new job. In order to check the conditions for a larger dataset, the reduced journal input described in the Breadth-First search was used to perform the same experiments. The results can be seen in the table below:

Execution Number	Update 1		Update 2		Update 3		Setup time	Shutdown time	Input superstep
	delete	add	delete	add	delete	add			
1	1831	3327	778	3498	833	3276	3124	234	2697
2	1221	4767	744	3777	800	3320	3150	146	2691
3	1342	4555	766	4077	1477	3066	3103	200	2349

Table 13. Dynamic Graph application timing for reduced journal dataset

Execution Number	Average Update time			Framework Setup time	Performance Factor
	delete	add	total		
1	1147.3	3367	4514.3	6055	1.34
2	921.6	3954.6	4876.2	5987	1.22
3	1195	3899.3	5094.3	5655	1.11

Table 14. Performance Gain for the reduced journal dataset

The results in the tables above show that – as expected – there is an improvement over submitting the job once again. For the journal dataset, injection is at least 11% faster than resubmitting. One can easily increase the performance of the current dynamic graph application by reducing the number of supersteps used for a dataset upgrade. The reduction from two to one superstep may increase considerably the performance factor. This may be done by using a vertex resolver. In the next section, the author proposes these basic modifications. After that, it is proposed additional modification in order to increase the scalability of this solution.

2.7.3 Suggested Modifications to Improve Performance

There are a few minor modifications that would increase the performance of the current implementation of the dynamic graph analysis:

1. Create the injector vertex in the Master: This would remove the race condition between multiple workers to create the injector vertex in the first superstep. This way, one can guarantee that only one request to create the injector would be done. Furthermore, it is possible to use a pre-processing first superstep. At the end of this superstep, the master elects a vertex based on its content, in example the vertex with minimum ID in the computation, and makes this vertex add the injection feature in parallel to its behaviour. This would remove the necessity of having a vertex only for injection, but may generate other problems that may have to be solved in the case of that injector vertex requesting removal by user code.
2. Communicate end of update section: Instead of making the framework master wait for two supersteps before assuming that the framework has finished injecting new vertices, one could use aggregators. In this solution, the underUpdate aggregator would be set to false at the end of the injection routine. The master would wait that condition to return to file tracking. The current implementation waits two supersteps: The first superstep is used only for vertex removals. The following is used on to inject new content. This prevents concurrent requests of adding and removing a vertex in the same superstep.
3. Implement a vertex resolver: A vertex resolver is used to determine, in the case of multiple events of addition/modification/removal of a vertex in the same superstep, which action to take. With the help of this procedure, one could add and remove vertex in the same superstep. The solution is simple. In the case of a removal request in parallel with an add request for the same vertex, modify the vertex to become the new vertex. Otherwise, if there is only the removal request, remove the vertex. This can be considered one of the most important changes for minor modifications, since it would reduce the computation time to only one superstep, therefore reducing the time when compared to submitting a new job.

2.7.3 Further Improvements Proposal

In order to detail even further performance improvements, there are a few modifications that should be performed:

1. **Master:** the master should, aside from taking timestamps of the monitored files, track the splits/HDFS blocks – if the number of files and their number of block is reduced (in order to maintain the non-parallel part to a minimum). If the number of files is larger or they have many blocks, each worker should

know which file/splits are being worked by it. This can be configured by the master in the first superstep. From then on, each worker would check the modification on its own respective files/splits. By doing this, there will be no communication between workers and the analysis would be balanced.

Each worker should have its own injector agent. The worker would activate the injection agent only when the files/splits being tracked by that worker were modified. In the case of automatic load balancing, a most complex solution to find which vertex should be removed has to be implemented. This is due to the possibility that many vertices that came from that split are not in the current worker anymore. This can be solved by a vertex variable, where the vertex saves its worker number in an attribute. Every vertex could check in the beginning of the superstep if their worker asked for a database update and then report to it.

2. **Worker:** The worker should be able to compare the original split vertices with the current new one. This reduces the mutations, since only vertices that were actually modified would be treated in this process. It may be interesting to the worker to perform the injection as a worker, not as an injector node inside of it. This would help avoiding the injector identification problem. This is one of the major modifications in order to improve the scalability of the original implementation.
3. **Implementation of the VertexResolver method:** This method should solve problems with graph mutations. By the use of this method, it may not be required to take two supersteps to modify the dataset, only one superstep from the moment the system detects a file modification. The Giraph framework already has this class, that handles graph mutations in a user defined way. One possible solution may be found in (Gabriellova & Cetindil, 2011). This solution is not optimal for the current problem and should be modified depending on the user approach of the solution. A possible solution would be perform the following:
 - a. In the case of auto load-balancing, check if that vertex detected, during its superstep, that his original injector worker detected file system modification.
 - b. Check messages to that vertex. If it does not receive messages from its worker injector agent, it means that that vertex is no longer in the new data base. Remove the vertex. If it received and does not exist, create that vertex. If it receives and exists:
 - c. Prune that vertex edges
 - d. Update that vertex value
 - e. Update vertex edges

One should modify the vertex resolver to take into consideration each scenario and computation. Sometimes the user may wish to retain some data of the old vertex state.

- 4. Modification of the GiraphRunner options:** GiraphRunner is a class used by Giraph to call used defined jobs. It has many options, among them the `-mc` (that defines the class that is going to perform the master compute) and `-wc` (that defines a worker context). A new option should be added called `-di`, which specifies to the framework that Giraph is going to work with non-static graph inputs. This would force the user to define a file observer type object and define methods in a DynamicMasterCompute (or worker) in order so support this kind of computations.
- 5. Fault Tolerance:** In order to provide fault tolerance in dynamic computation, it is of paramount importance that the workers also record the metadata of the block/splits that they are tracking (i.e. timestamps) in the moment that a checkpoints is reached. Although all computation, between the moment of the creation of the checkpoint and the fault, will be lost, at least one has a point to resume the computation without simple restart. After a fault event, a setup superstep should be performed. In this superstep, the workers would compare the files that are being tracked. If there was a modification into one of the tracked blocks/splits, the worker taking care of that data section should perform an update of its dataset.

A flowchart that represents how the system should behave in this approach is given below:

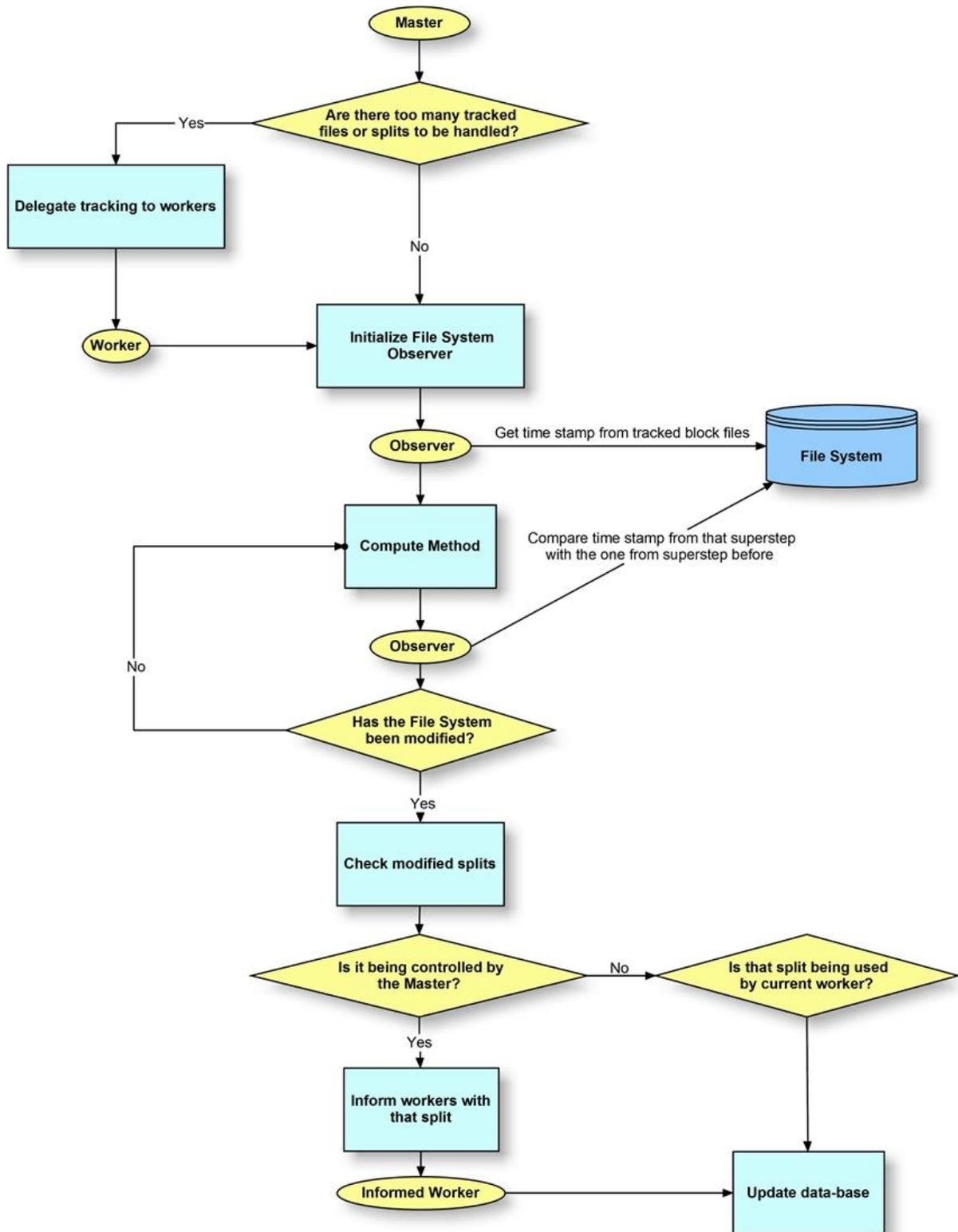


Figure 14. Further improvements for Master/Worker behaviours

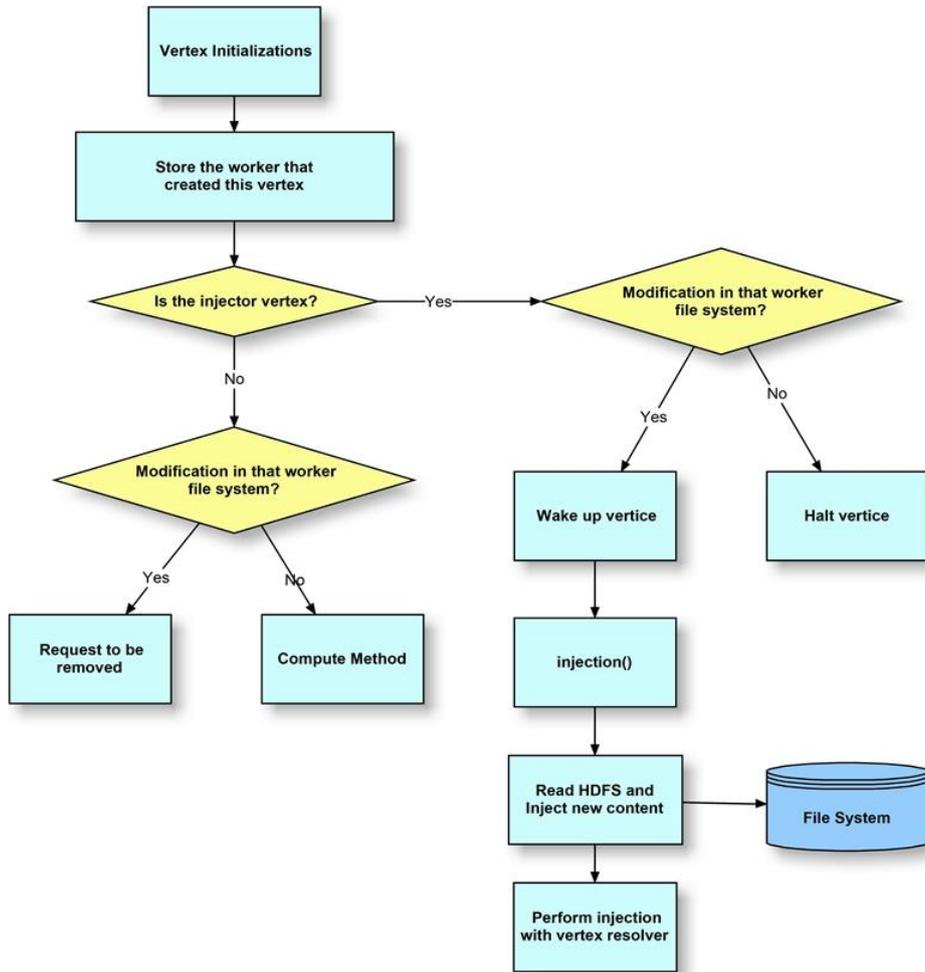


Figure 15. Further improvements for vertex behaviour

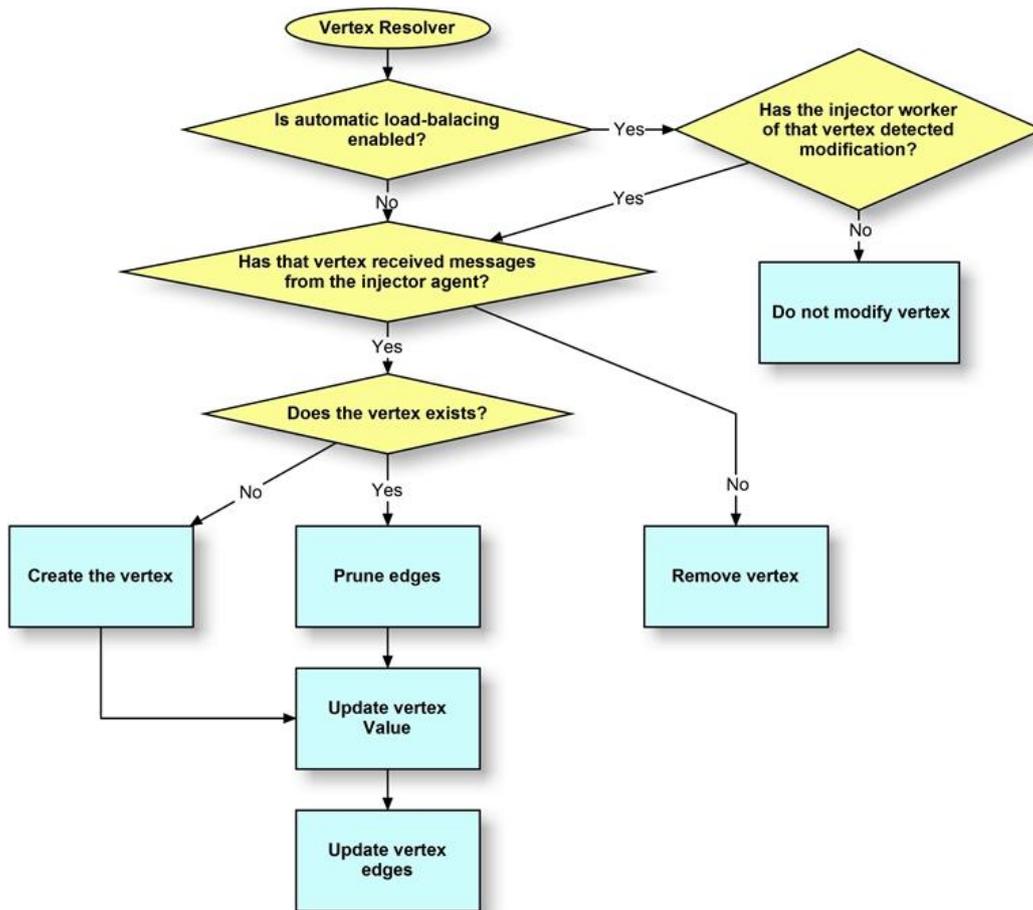


Figure 16. Possible vertex resolver behaviour

Conclusion

Although this document is originally intended to be a report to describe the author activities in the Science without Borders scheme, it may have surpassed its original goals. In the beginning of the documentation, the author described how one could setup a working version of Giraph, with some basic benchmarks to test the installation. This can be used as reference in a possible – and hereby suggested by the author – addition of Giraph as a component to the High Performance Computing module syllabus. The accelerated increase of Giraph importance in industry, referenced many times during this report, may create a good differential for Queen Mary University students.

After describing the scenario and the setup stages, the author depicted how one could implement well-known graph search solution, as breadth-first search. The result were then used to confirm the superiority of Giraph over Hadoop for this kind of task, considering the configuration used. Many graphs had to be modified in order to be used by Giraph standard JSON reader. While analysing it, one can visualize how to easily adapt the script to another dataset, in order to produce the desired JSON file.

Dynamics graphs are finally approached in the last section of the document, describing how the author could modify the framework in order to turn their computation possible. In order to produce this, during the research part, many other people involved with the Giraph framework development were incited – due to author questions - to deliberate about the possibilities of this new feature. This document will serve as a first description of how to implement this feature for the Giraph platform.

As a last topic, the author decided to analyse how to perform further improvements in the computation model, using the Pregel prism. This was done taking into account elements as parallelism, reduced communication between workers, load balancing and fault tolerance. This description may well be used to implement support for dynamic graphs straight into Giraph framework, rather than modifying an application in order to perform this.

Bibliography

Apache Foundation, 2012. *Single Node Setup*. [Online]

Available at: http://hadoop.apache.org/docs/stable/single_node_setup.html

[Accessed 3 August 2013].

Apache Foundation, 2013. *Building and Testing*. [Online]

Available at: <http://giraph.apache.org/build.html>

[Accessed 2 August 2013].

Apache Foundation, 2013. *Giraph-user mailing list archives: July 2013*. [Online]

Available at: http://mail-archives.apache.org/mod_mbox/giraph-user/201307.mbox/browser

[Accessed 3 August 2013].

Apache Foundation, 2013. *Implementation*. [Online]

Available at: <http://giraph.apache.org/implementation.html>

[Accessed 2 August 2013].

Apache Foundation, 2013. *Mailing List Archives: user@giraph.apache.org*. [Online]

Available at: http://mail-archives.apache.org/mod_mbox/giraph-user/

[Accessed 3 August 2013].

Apache Foundation, 2013. *Out-of-core Giraph*. [Online]

Available at: <http://giraph.apache.org/ooc.html>

[Accessed 2 August 2013].

Apache Foundation, 2013. *PoweredBy*. [Online]

Available at: <http://wiki.apache.org/hadoop/PoweredBy>

[Accessed 25 July 2013].

Apache Foundation, 2013. *YARN*. [Online]

Available at: <http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html>

[Accessed 1 August 2013].

Author, U., 2009. *Optimal Account Balancing II*. [Online]

Available at: <http://stochastix.wordpress.com/2009/07/28/>

[Accessed 1 August 2013].

Bie, W., 2012. *Hadoop at Spotify*. [Online]

Available at: <http://files.meetup.com/5139282/SHUG%20%20-%20Hadoop%20at%20Spotify.pdf>

[Accessed 1 August 2013].

Carvalho, R., 2009. *Optimal Account Balancing II | Rod Carvalho*. [Online]
Available at: <http://stochastix.wordpress.com/2009/07/28/optimal-account-balancing-ii/>
[Accessed 3 August 2013].

Cloudera Inc, 2013. *CDH*. [Online]
Available at: <http://www.cloudera.com/content/cloudera/en/products/cdh.html>
[Accessed 1 August 2013].

Cuadrado, F. & White, G., 2013. *High Performance Computing Laboratories*. [Online]
Available at: <http://qmplus.qmul.ac.uk/course/view.php?id=2409>
[Accessed 3 August 2013].

Dean, J. & Ghemawat, S., 2004. *MapReduce: Simplified Data Processing on Large Clusters*. [Online]
Available at: <http://research.google.com/archive/mapreduce.html>
[Accessed 1 August 2013].

Fae, C. & all, e., 2006. *Bigtable: A Distributed Storage System for Structured Data*. [Online]
Available at: <http://wiki.apache.org/hadoop/PoweredBy>
[Accessed 25 July 2013].

Gabrielova, E. & Cetindil, I., 2011. *graph-algorithm-ports-giraph-hyracks*. [Online]
Available at: <https://code.google.com/p/graph-algorithm-ports-giraph-hyracks/source/browse/trunk/src/main/java/org/apache/giraph/graph/VertexResolver.java?r=11>
[Accessed 8 August 2013].

Homan, J., 2012. *Apache Giraph, a framework for large-scale graph processing on Hadoop*. [Online]
Available at: <http://engineering.linkedin.com/open-source/apache-giraph-framework-large-scale-graph-processing-hadoop-reaches-01-milestone>
[Accessed 1 August 2013].

Ho, R., 2010. *Google Pregel Graph Processing*. [Online]
Available at: <http://horicky.blogspot.co.uk/2010/07/google-pregel-graph-processing.html>
[Accessed 1 August 2013].

Jungblut, T., 2011. *Back to Blogosphere and how BSP works*. [Online]
Available at: http://codingwiththomas.blogspot.co.uk/2011_04_01_archive.html
[Accessed 3 August 2013].

Khayyat, Z. & al, e., 2013. *Mizan: A System for Dynamic Load Balancing in Large-Scale Graph processing*. Prage, Eurosys'13.

Leskovec, j., 2009. *SNAP: Network datasets: LiveJournal social network*. [Online]
Available at: <http://snap.stanford.edu/data/soc-LiveJournal1.html>
[Accessed 23 August 2013].

Lotz, M., 2013. *Dynamic Graph Computation using Parallel Distributed Computing Solutions*. London: Queen Mary - University of London.

Lotz, M., 2013. *In-memory graph processing*. London: Queen Mary - University of London.

Lotz, M., 2013. *Problem building/installing HBase usin Maven*. [Online]
Available at: <https://issues.apache.org/jira/browse/GIRAPH-700>
[Accessed 3 August 2013].

Malewicz, G. & all, e., 2010. *Pregel: a system for large-scale graph processing*. [Online]
Available at: <http://www.slideshare.net/shatteredNirvana/pregel-a-system-for-largescale-graph-processing>
[Accessed 1 August 2013].

Malewicz, G. & all, e., 2010. *Pregel: a system for large-scale graph processing*. [Online]
Available at: http://kowshik.github.io/JPregel/pregel_paper.pdf
[Accessed 1 August 2010].

Martella, C., 2012. *Apache Giraph: Distributed Graph Processing in the Cloud*. [Online]
Available at: <http://www.youtube.com/watch?v=3ZrqPEIPRe4>
[Accessed 3 August 2013].

Monash, C., 2012. *Hadoop YARN- Beyond MapReduce*. [Online]
Available at: <http://www.dbms2.com/2012/07/23/hadoop-yarn-beyond-mapreduce>
[Accessed 1 August 2013].

Orlando, K., Martella, C. & al, e., 2013. *Help needed for Running my own java programs in Giraph*. [Online]
Available at: http://mail-archives.apache.org/mod_mbox/giraph-user/201308.mbox/%3CCAFJOoJefOhp2cUgGU8pXf4QdC3b4jZbPit6WYwhcUt7SJ%3DvbLQ%40mail.gmail.com%3E
[Accessed 27 August 2013].

PCMAG, 2013. *Alpha version definition from PC Magazine Encyclopedia*. [Online]
Available at: <http://www.pcmag.com/encyclopedia/term/37675/alpha-version>
[Accessed 1 August 2013].

Peixiang, Z., 2010. *Pregel: a system for large-scale graph processing*. [Online] Available at: <https://wiki.engr.illinois.edu/download/attachments/188588798/pregel.pdf?version=1> [Accessed 1 August 2013].

Reisman, E., 2013. *Make sure Giraph builds against all compatible YARN-enabled Hadoop versions, warns if none set, works w/ new 1.1.0 line*. [Online] Available at: <https://issues.apache.org/jira/browse/GIRAPH-688> [Accessed 3 August 2013].

Ryza, S., 2013. *Improvements in the Hadoop YARN Fair Scheduler*. [Online] Available at: <http://blog.cloudera.com/blog/2013/06/improvements-in-the-hadoop-yarn-fair-scheduler/> [Accessed 1 August 2013].

Sangwon, S. & al, e., n.d. *HAMA: An Efficient Matrix Computation with the MapReduce Framework*. s.l., 2nd IEEE International Conference on Cloud Computing Technology and Science.

Stuart J. Russell, P. N., 2009. *Artificial Intelligence: A Modern Approach*. 3rd ed. s.l.:Prentice Hall.

Stylianou, M., 2013. *How to set up Apache Giraph*. [Online] Available at: <http://marsty5.wordpress.com/2013/04/03/how-to-set-up-apache-giraph/> [Accessed 3 August 2013].

Stylianou, M., 2013. *How to set-up apache giraph*. [Online] Available at: <http://marsty5.wordpress.com/2013/04/03/how-to-set-up-apache-giraph/> [Accessed 02 August 2013].

Stylianou, M., 2013. *I set up Apache Giraph - Now what?*. [Online] Available at: <http://marsty5.wordpress.com/2013/04/29/i-set-up-apache-giraph-now-what/> [Accessed 3 August 2013].

Stylianou, M., 2013. *run example in giraph pagerank*. [Online] Available at: <http://marsty5.wordpress.com/2013/05/29/run-example-in-giraph-pagerank/> [Accessed 3 August 2013].

Stylianou, M., 2013. *run example in giraph shortest-paths*. [Online] Available at: <http://marsty5.wordpress.com/2013/04/29/run-example-in-giraph-shortest-paths/> [Accessed 3 August 2013].

Stylianou, M., 2013. *Understanding Pregel*. [Online]
Available at: <http://marsty5.wordpress.com/2013/03/03/understanding-pregel/>
[Accessed 1 August 2013].

Valiant, L., 1990. A Bridging model for parallel computation. *Communications of the ACM*, August.33(8).

Vance, A., 2009. *Hadoop, a Free Software Program, Finds Uses Beyond Search*. [Online]
Available at: <http://www.nytimes.com/2009/03/17/technology/business-computing/17cloud.html>
[Accessed 1 08 2013].

White, T., 2012. *Hadoop: The Definitive Guide*. 3rd Edition ed. s.l.:O'Reilly.

Wiggins, G. A., 2013. *ELE 611: Artificial Intelligence lecture slides*. London: Queen Mary - University of London.

Yahoo! Inc, 2013. *Hadoop Tutorial from Yahoo!*. [Online]
Available at: <http://developer.yahoo.com/hadoop/tutorial/module2.html>
[Accessed 1 August 2013].

Yang, S. & al, e., 2013. *SEDGE: A Self Evolving Distributed Graph Processing Environment*. Santa Barbara: Computer Science Department.

Zettaset, 2013. *What is Big Data and Hadoop? - Zettaset*. [Online]
Available at: <http://www.zettaset.com/info-center/what-is-big-data-and-hadoop.php>
[Accessed 2 September 2013].

Appendices

1. YARN HDFS scripts

```
echo "[INFO] Procedure scripted by Marco Aurelio Lotz"
echo ""
echo "[INFO] Changing Directory to $YARN_HOME/sbin"
cd $YARN_HOME/sbin
echo "[INFO] Starting DFS Daemons:"
./start-dfs.sh &&
echo "[INFO] DONE: Starting DFS Daemons."
echo "[INFO] Starting YARN Daemons:"
./start-yarn.sh &&
echo "[INFO] DONE: Starting YARN Daemons."
echo "[INFO] Starting History Server."
./mr-jobhistory-daemon.sh start historyserver &&
echo "[INFO] DONE: Starting History Server."
```

Table 15. Start-all script

```
# Scripted Created by Marco Aurelio Lotz in order to automatize stop-all process.
echo "[INFO] Procedure scripted by Marco Aurelio Lotz"
echo ""
echo "[INFO] Changing Directory to $YARN_HOME/sbin"
cd $YARN_HOME/sbin
echo "[INFO] Stopping DFS Daemons:"
./stop-dfs.sh &&
echo "[INFO] DONE: Stopping DFS Daemons."
echo "[INFO] Stopping YARN Daemons:"
./stop-yarn.sh &&
echo "[INFO] DONE: Stopping YARN Daemons."
echo "[INFO] Stopping History Server."
./mr-jobhistory-daemon.sh stop historyserver &&
echo "[INFO] DONE: Stopping History Server."
```

Table 16. Stop-all script

2. updateJar Script

```
# Created by Marco Aurelio Lotz
# in order to automatize the jar updates.

PACKAGEADDRESS=$1
LASTPACKAGENAME=$2
JARFILEADDRESS="/usr/local/giraph/giraph-examples/target"
JARNAME="giraph-examples-prometheus.jar"
PACKAGESYNTAX="uk/co/qmul/giraph/"
PACKAGEDISTANATION="/home/hduser/Documents/codesInCore/"
MAINPACKAGE="uk"

echo "[INFO] Giraph Jar updater by Marco Aurelio Lotz"
echo "[INFO] This script assumes that there is a giraph jar to be updated in the same directory that the script is in."
echo "[INFO] Then it copies this jar to the destination"
echo "[INFO] The package address is the address where the bin folder is"

if [ $# -eq 2 ]; then
    echo "[INFO] Copying the Package classes from
: "$PACKAGEADDRESS"bin/$PACKAGESYNTAX$LASTPACKAGENAME"
    echo "[INFO] to: $PACKAGEDISTANATION$PACKAGESYNTAX"
    cp -r $PACKAGEADDRESS/bin/"$PACKAGESYNTAX$LASTPACKAGENAME
$PACKAGEDISTANATION$PACKAGESYNTAX

    echo "[INFO] Done."

    echo "[INFO] Updating Jar file"
    jar uf giraph-examples-prometheus.jar $MAINPACKAGE &&
    echo "[INFO] Done."

    echo "[INFO] Replacing old jar file with new one in $JARFILEADDRESS"
    cp giraph-examples-prometheus.jar $JARFILEADDRESS &&
    echo "[INFO] Done."
```

```

        echo ""
        echo "[INFO] SUCCESS!"
else
        echo "Syntax: [script] [path to bin source][Last Package name]"
fi

```

Table 17. Script used to update jar files

3. *getJavaDocs Script*

```

# Created by Marco Aurelio Lotz

INPUTDIR=$1
OUTPUTDIR=$2
TARGETEXTENSION=$3

function copyEXTENSION(){
    for i in * ; do
        if [ -d $i ]; then
            mkdir $OUTPUTDIR$i #Creates that dir in the output
            cd $i
            OUTPUTDIR="$OUTPUTDIR$i/" #Updates the output path
            copyEXTENSION #Run the function recursively
            cd .. #Return to parent directory
            OUTPUTDIR=`echo "$OUTPUTDIR"|awk -F / '{ for (i=1; i <= NF-2; i++){ tmp = tmp $i "/" } print
tmp}`
            # Removes the last element in the output path
        elif [ -f $i ]; then
            EXTENSION=`echo "$i"|awk -F . '{print $NF}'` #get file extension
            if [ $EXTENSION = "$TARGETEXTENSION" ]; then
                cp $i $OUTPUTDIR
            fi
        fi
    done
}

if [ $# -eq 3 ]; then
    echo "Script written by Marco Aurelio Lotz"
    echo "This script extracts recursively all the .[extension] documents"
    echo "from the specified folder and copies to the destination folder"

    OUTPUTDIR=`pwd`/"$OUTPUTDIR #creates the requirement of the script to be in the output dir
    echo "looking for .$TARGETEXTENSION in $INPUTDIR..."
    cd $INPUTDIR
    copyEXTENSION
    echo "The script has finished copying the files"
else
    echo "Wrong usage."
    echo "use: scriptname [input dir] [output dir] [extension type]"
    echo "example: ./getDocs input output java"
    echo "Note: Requires the script to be in the output dir"
    echo "Note: In a posterior version, implement parser and remove empty dirs"
fi

```

Table 18. Script used to extract java docs

4. *Breadth-First Search Target-Oriented*

```

/*
 * Licensed to the Apache Software Foundation (ASF) under one
 * or more contributor license agreements. See the NOTICE file
 * distributed with this work for additional information
 * regarding copyright ownership. The ASF licenses this file

```

```

* to you under the Apache License, Version 2.0 (the
* "License"); you may not use this file except in compliance
* with the License. You may obtain a copy of the License at
*
* http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*/

package uk.co.qmul.giraph.targetbfs;

import org.apache.giraph.Algorithm;
import org.apache.giraph.graph.BasicComputation;
import org.apache.giraph.conf.LongConfOption;
import org.apache.giraph.edge.Edge;
import org.apache.giraph.graph.Vertex;
import org.apache.giraph.master.DefaultMasterCompute;
import org.apache.hadoop.io.DoubleWritable;
import org.apache.hadoop.io.FloatWritable;
import org.apache.hadoop.io.LongWritable;
//import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.giraph.aggregators.DoubleMaxAggregator;

import java.io.IOException;

/**
 * Demonstrates the basic Pregel Breadth First Search.
 */
/**
 * SimpleBFSComputation is the basic class with the configurations for a BFS
 * search and the basic configurations for it. The Target means that it looks
 * for a target vertex into the structure.
 *
 * @author Marco Aurelio Lotz
 */
@Algorithm(name = "Breadth First Search Target oriented", description =
"Uses Breadth First Search from a source vertex to find target vertex")
public class SimpleBFSTargetComputation
    extends
    BasicComputation<LongWritable, DoubleWritable, FloatWritable, DoubleWritable> {

    /**
     * Define a maximum number of supersteps
     */
    public final int MAX_SUPERSTEPS = 50;

    /**
     * Status aggregator, in this case a max is used. When the solution is
     * found, the aggregator receives the value 1.
     */
    private static String MAX_AGG = "max";

    /**
     * Indicates the first vertex to be computed in superstep 0.
     */
    public static final LongConfOption START_ID = new LongConfOption(
        "SimpleBFSComputation.START_ID", 0,
        "Is the first vertex to be computed");

    /**
     * Indicates the target vertex. Once the vertex as been found, the
     * computation can converge.
     */
    public static final LongConfOption TARGET_ID = new LongConfOption(
        "SimpleBFSComputation.TARGET_ID", 708,
        "Indicates which is the target node");

    /** Class logger */
    private static final Logger LOG = Logger
        .getLogger(SimpleBFSTargetComputation.class);

```

```

/**
 * Target Vertex Found flag definition. Is going to be compared with the
 * aggregator value by the vertices
 */
public static final DoubleWritable FOUND = new DoubleWritable(1d);

/**
 * Target Vertex Missing flag definition. Is going to be compared with the
 * aggregator value by the vertices
 */
public static final DoubleWritable MISSING = new DoubleWritable(0d);

/**
 * Is this vertex the target vertex?
 *
 * @param vertex
 * @return true if the analysed node is the target
 */
private boolean isTarget(Vertex<LongWritable, ?, ?> vertex) {
    return vertex.getId().get() == TARGET_ID.get(getConf());
}

/**
 * Is this vertex the start vertex?
 *
 * @param vertex
 * @return true if analysed node is the start vertex
 */
private boolean isStart(Vertex<LongWritable, ?, ?> vertex) {
    return vertex.getId().get() == START_ID.get(getConf());
}

/**
 * Send messages to all the connected vertices. The content of the messages
 * is not important, since just the event of receiving a message removes the
 * vertex from the inactive status.
 *
 * @param vertex
 */
public void BFSMessages(
    Vertex<LongWritable, DoubleWritable, FloatWritable> vertex) {
    for (Edge<LongWritable, FloatWritable> edge : vertex.getEdges()) {
        sendMessage(edge.getTargetVertexId(), new DoubleWritable(1d));
    }
}

@Override
public void compute(
    Vertex<LongWritable, DoubleWritable, FloatWritable> vertex,
    Iterable<DoubleWritable> messages) throws IOException {

    // If the target node was found, stop computation and save values.
    // This was implemented to prevent graph where vertices have more than
    // one edge to run
    // until the the maximum superstep possible.
    if (FOUND.get() == ((DoubleWritable) getAggregatedValue(MAX_AGG)).get()) {
        if (!isStart(vertex) && (!isTarget(vertex))) { // in order to keep

            // track begin and

            // end values
            vertex.setValue(new DoubleWritable(getSuperstep())); // so one
            // can know in which superstep it was halted if there was a vertex
            // being computed.
            if (LOG.isInfoEnabled()) {
                LOG.info("Target found! Converging computation: Vertex ID: "
                    + vertex.getId() + " voted to halt.");
            }
        }
    }

    // If it reaches the maximum number of Supersteps just voteToHalt
    // (forces converge of the code)
    else if (!(getSuperstep() == MAX_SUPERSTEPS)) {

```

```

// if it is the first Superstep.
if (getSuperstep() == 0) { // Others vertices will become activate
// when they receive
messages.
    if (isStart(vertex)) {
        // Identify the start vertex in the output with the value
        vertex.setValue(new DoubleWritable(10d));
        BFSMessages(vertex);
        if (LOG.isInfoEnabled()) {
            LOG.info("Vertex ID: " + vertex.getId()
                + " [START]: Start node found!");
            //LOG.info("The maximum number of Supersteps is: "
            // + TEST_MAX_SUPERSTEPS);
        }
    }
}

// if it is not the first Superstep (Superstep 0) :
// Check vertex ID

else if (isTarget(vertex)) {
    if (LOG.isInfoEnabled()) {
        LOG.info("Target vertex found! Vertex ID: "
            + vertex.getId() + " is the target node.");
    }

    // So other vertices can know that the target was found. Used
    // graphs with branch factor larger than 1
    aggregate(MAX_AGG, FOUND);
    if (LOG.isInfoEnabled()) {
        LOG.info("Vertex ID: "
            + vertex.getId()
            + " [TARGET]: informed aggregator to converge
computation.");
    }
    // BFSMessages(vertex); //debug purposes, to test the stop
    // condition detection for non-directed graphs
    vertex.setValue(new DoubleWritable(100d)); // shows the found

    // target in the

    // output with the

    // value 100
    }

    else {
        // Non-target vertices:
        // Send messages to all connected vertices.

        BFSMessages(vertex);

        // send 0 to the aggregator. Shows that current vertex is not
        // the target.
        aggregate(MAX_AGG, MISSING);
    }
}
vertex.voteToHalt();
}

/**
 * Master compute associated with {@link SimpleBFSComputation}. It registers
 * the required max aggregator.
 */
public static class SimpleBFSMasterCompute extends DefaultMasterCompute {
    @Override
    public void initialize() throws InstantiationException,
        IllegalAccessException {
        registerPersistentAggregator(MAX_AGG, DoubleMaxAggregator.class);
    }
}
}

```

Table 19. Breadth-First Search Target-Oriented in Giraph

5. Breadth-First Search Structure-Oriented

```
/*
 * Licensed to the Apache Software Foundation (ASF) under one
 * or more contributor license agreements. See the NOTICE file
 * distributed with this work for additional information
 * regarding copyright ownership. The ASF licenses this file
 * to you under the Apache License, Version 2.0 (the
 * "License"); you may not use this file except in compliance
 * with the License. You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package uk.co.qmul.giraph.structurebfs;

import org.apache.giraph.Algorithm;
import org.apache.giraph.graph.BasicComputation;
import org.apache.giraph.conf.LongConfOption;
import org.apache.giraph.edge.Edge;
import org.apache.giraph.graph.Vertex;
import org.apache.hadoop.io.DoubleWritable;
import org.apache.hadoop.io.FloatWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.log4j.Logger;

import java.io.IOException;

/**
 * Demonstrates the basic Pregel Breadth First Search applied to graph
 * structure.
 *
 * SimpleBFSSComputation is the basic class with the configurations for a BFS
 * search and the basic configurations for it. The "Structure" means that it
 * computes the whole structure in order to check the depth from the start
 * vertex
 *
 * @author Marco Aurelio Lotz
 */
@Algorithm(name = "Breadth First Search Structural oriented", description = "Uses Breadth First Search from a source vertex
to calculate depths")
public class SimpleBFSStructureComputation
    extends
        BasicComputation<LongWritable, DoubleWritable, FloatWritable, DoubleWritable> {
    /**
     * Define a maximum number of supersteps
     */
    public final int MAX_SUPERSTEPS = 9999;

    /**
     * Indicates the first vertex to be computed in superstep 0.
     */
    public static final LongConfOption START_ID = new LongConfOption(
        "SimpleBFSStructureComputation.START_ID", 8,
        "Is the first vertex to be computed");

    /** Class logger */
    private static final Logger LOG = Logger
        .getLogger(SimpleBFSStructureComputation.class);

    /**
     * Is this vertex the start vertex?
     *
     * @param vertex
     */
}
```

```

    * @return true if analysed node is the start vertex
    */
    private boolean isStart(Vertex<LongWritable, ?, ?> vertex) {
        return vertex.getId().get() == START_ID.get(getConf());
    }

    /**
     * Send messages to all the connected vertices. The content of the messages
     * is not important, since just the event of receiving a message removes the
     * vertex from the inactive status.
     *
     * @param vertex
     */
    public void BFSMessages(
        Vertex<LongWritable, DoubleWritable, FloatWritable> vertex) {
        for (Edge<LongWritable, FloatWritable> edge : vertex.getEdges()) {
            sendMessage(edge.getTargetVertexId(), new DoubleWritable(1d));
        }
    }

    @Override
    public void compute(
        Vertex<LongWritable, DoubleWritable, FloatWritable> vertex,
        Iterable<DoubleWritable> messages) throws IOException {

        // Forces convergence in maximum superstep
        if (!(getSuperstep() == MAX_SUPERSTEPS)) {
            // Only start vertex should work in the first superstep
            // All the other should vote to halt and wait for
            // messages.
            if (getSuperstep() == 0) {
                if (isStart(vertex)) {
                    vertex.setValue(new DoubleWritable(getSuperstep()));
                    BFSMessages(vertex);
                    if (LOG.isInfoEnabled()) {
                        LOG.info("[Start Vertex] Vertex ID: " + vertex.getId());
                    }
                } else { // Initialise with infinite depth other vertex
                    vertex.setValue(new DoubleWritable(Integer.MAX_VALUE));
                }
            }

            // if it is not the first Superstep (Superstep 0) :
            // Check vertex ID

            else {
                // It is the first time that this vertex is being computed
                if (vertex.getValue().get() == Integer.MAX_VALUE) {
                    // The depth has the same value that the superstep
                    vertex.setValue(new DoubleWritable(getSuperstep()));
                    // Continue on the structure
                    BFSMessages(vertex);
                }
                // Else this vertex was already analysed in a previous
                // iteration.
            }
            vertex.voteToHalt();
        }
    }
}

```

Table 20. Breadth-First Search Structure-Oriented in Giraph

6. ToJSON Script

```

# AWK Script written by Marco Aurelio Lotz
# Converts a graph structure to JSON graph
# format.

BEGIN{
    vertexValue=0;
    edgeValue=0;
}

```

```

FS=" ";
vertexId=-1;
outString = "";
maxId=100000; #Maximum referred Id in output
}
{
if ( NR >= 5 ) #The first 4 lines have unuseful information
{
    newVertexId=$1;
    TargetVertex=$2;
    TargetVertex++;
    TargetVertex--;
    newVertexId++;
    newVertexId--;
    #Does not generate nodes higher than the maximum
    if (newVertexId <= maxId)
    {
        if (newVertexId != vertexId)
        {
            #Doesnt print in the first iteration
            if (outString != "")
            {
                outString= outString "]]";
                print(outString); #print old outString
            }
            outString=["newVertexId","vertexValue","];
            vertexId = newVertexId;
            firstEdge = 1;
        }

        #There is some trash coming from the input
        #Force number comparisson.
        #Prevent reference to outEdges to nodes higher then
        #maxId
        if (TargetVertex < maxId)
        {
            if (firstEdge == 1)
            {
                outString= outString [" TargetVertex ", " edgeValue"];
                firstEdge = 0;
            }
            else
            {
                outString= outString ",[" TargetVertex ", " edgeValue"];
            }
        }
    }
}
}
}

```

Table 21. Script used to convert Journal to JSON format

7. Hadoop Breadth-First Search Code

```

package bfs;

import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;
import org.apache.hadoop.io.ArrayWritable;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.io.WritableComparable;

public class BFSNode implements WritableComparable<BFSNode> {

    public static String DISTANCE_INFO = "DISTANCE_INFORMATION_ONLY";
    private Text id;
    private ArrayWritable dest;
    private IntWritable distance;

```

```

public BFSNode() {
    ArrayWritable aw = new ArrayWritable(Text.class);
    aw.set(new Writable[0]);
    set(new Text(), aw, new IntWritable());
}

public BFSNode(Text id, ArrayWritable dest, IntWritable distance) {
    set(id, dest, distance);
}

public void set(Text id, ArrayWritable dest, IntWritable distance) {
    this.id = id;
    this.dest = dest;
    this.distance = distance;
}

public void set(String id, String[] dest, int distance) {
    this.id.set(id);
    Text[] values = new Text[dest.length];

    for (int i = 0; i < dest.length; i++) {
        values[i] = new Text(dest[i]);
    }
    this.dest.set(values);
    this.distance.set(distance);
}

public void setId(String id) {
    this.id.set(id);
}

public void setDistance(int distance) {
    this.distance.set(distance);
}

public String getId() {
    return id.toString();
}

public String[] getDest() {
    Writable[] arr = (Writable[]) dest.get();
    String[] ldest = new String[arr.length];
    for (int i = 0; i < arr.length; i++) {
        ldest[i] = ((Text) arr[i]).toString();
    }
    return ldest;
}

public int getDistance() {
    return distance.get();
}

@Override
public void write(DataOutput out) throws IOException {
    id.write(out);
    dest.write(out);
    distance.write(out);
}

@Override
public void readFields(DataInput in) throws IOException {
    id.readFields(in);
    dest.readFields(in);
    distance.readFields(in);
}

```

```

@Override
public int compareTo(BFSNode node) {
    String self = this.getId();

    return self.compareTo(node.getId());
}

@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((dest == null) ? 0 : dest.hashCode());
    result = prime * result
        + ((distance == null) ? 0 : distance.hashCode());
    result = prime * result + ((id == null) ? 0 : id.hashCode());
    return result;
}

@Override
public boolean equals(Object obj) {
    if (obj instanceof BFSNode) {
        BFSNode node = (BFSNode) obj;
        return this.getId().equals(node.getId());
    }
    return false;
}
}

```

Table 22. BFSNode java code

```

package bfs;

import java.io.IOException;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

/* Mapper function:
 * input: Text (id), BFSNode (node)
 * output: Text (id), BFSNode (node)
 */

public class IterateBFSMapper extends Mapper<Text, BFSNode, Text, BFSNode> {

    private final BFSNode distanceNode = new BFSNode();
    private Text key = new Text();

    /*
     * Setup method is always called before the map method by the framework
     * http:
     * //hadoop.apache.org/docs/current/api/org/apache/hadoop/mapreduce/Mapper
     * .html
     */
    protected void setup(Context context) throws IOException,
        InterruptedException {
        distanceNode.setId(BFSNode.DISTANCE_INFO);
    };

    public void map(Text nid, BFSNode node, Context context)
        throws IOException, InterruptedException {

        context.write(nid, node);

        if (node.getDistance() < Integer.MAX_VALUE) {
            context.getCounter(IterateBFS.Counters.ReachableNodesAtMap)
                .increment(1);
        }
    }
}

```

```

        String[] dest = node.getDest();
        int dist = node.getDistance() + 1;

        for (int i = 0; i < dest.length; i++) {
            String neighbor = dest[i];
            distanceNode.setDistance(dist);
            key.set(neighbor);
            context.write(key, distanceNode);
        }
    }
}

```

Table 23. Mapper java code

```

package bfs;

import java.io.IOException;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

/* Reducer function:
 * input: Text (id), BFSNode (node)
 * output: Text (id), BFSNode (node)
 */

public class IterateBFSReducer extends
    Reducer<Text, Iterable<BFSNode>, Text, BFSNode> {

    private final BFSNode emittedNode = new BFSNode();
    Text NodeId = new Text();
    int minDistance = Integer.MAX_VALUE;

    public void reduce(Text nid, Iterable<BFSNode> nodes, Context context)
        throws IOException, InterruptedException {

        context.getCounter(IterateBFS.Counters.ReachableNodesAtReduce)
            .increment(1);

        for (BFSNode node : nodes) {
            if (node instanceof BFSNode) {
                if (node.getId() != BFSNode.DISTANCE_INFO) {
                    emittedNode.set(node.getId(), node.getDest(),
                        node.getDistance());
                }
                if (node.getDistance() < minDistance) {
                    minDistance = node.getDistance();
                }
            }
        }
        emittedNode.setDistance(minDistance);
        NodeId.set(emittedNode.getId());
        context.write(NodeId, emittedNode);
    }
}

```

Table 24. Reducer java code

```

package bfs;

import java.util.Arrays;
import org.apache.commons.lang.StringUtils;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.SequenceFileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat;
import org.apache.log4j.*;

```

```

public class IterateBFS {

    /** Class logger */
    private static final Logger LOG = Logger
        .getLogger(IterateBFS.class);

    static enum Counters {
        ReachableNodesAtMap, ReachableNodesAtReduce
    };

    public static void runJob(String[] input, String output) throws Exception {

        Configuration conf = new Configuration();
        Job job = new Job(conf);

        conf.set("mapreduce.child.java.opts", "-Xmx2048m");

        job.setJarByClass(IterateBFS.class);
        job.setMapperClass(IterateBFSMapper.class);
        job.setReducerClass(IterateBFSReducer.class);
        job.setNumReduceTasks(4);
        job.setInputFormatClass(SequenceFileInputFormat.class);
        job.setOutputFormatClass(SequenceFileOutputFormat.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(BFSNode.class);

        Path outputPath = new Path(output);

        FileInputFormat.setInputPaths(job, StringUtils.join(input, ","));
        FileOutputFormat.setOutputPath(job, outputPath);

        outputPath.getFileSystem(conf).delete(outputPath, true);
        job.waitForCompletion(true);
    }

    public static void main(String[] args) throws Exception {
        long time = System.currentTimeMillis();
        runJob(Arrays.copyOfRange(args, 0, args.length - 1),
            args[args.length - 1]);
        time = System.currentTimeMillis() - time;
        LOG.info("Took "+ time +" ms to complete the computation");
    }
}

```

Table 25. Job configuration and submission java code

8. Dynamic Graph application

```

/*
 * Licensed to the Apache Software Foundation (ASF) under one
 * or more contributor license agreements. See the NOTICE file
 * distributed with this work for additional information
 * regarding copyright ownership. The ASF licenses this file
 * to you under the Apache License, Version 2.0 (the
 * "License"); you may not use this file except in compliance
 * with the License. You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

```

```

package org.apache.giraph.graph;

import org.apache.giraph.comm.WorkerClientRequestProcessor;
import org.apache.giraph.conf.DefaultImmutableClassesGiraphConfigurable;
import org.apache.giraph.conf.TypesHolder;
import org.apache.giraph.edge.Edge;
import org.apache.giraph.edge.OutEdges;
import org.apache.giraph.worker.WorkerAggregatorUsage;
import org.apache.giraph.worker.WorkerContext;
import org.apache.hadoop.io.BooleanWritable;
import org.apache.hadoop.io.DoubleWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.log4j.Logger;

import java.io.IOException;
import java.util.Iterator;

/**
 * Basic abstract class for writing a BSP application for computation.
 *
 * During the superstep there can be several instances of this class, each doing
 * computation on one partition of the graph's vertices.
 *
 * Note that each thread will have its own {@link Computation}, so accessing any
 * data from this class is thread-safe. However, accessing global data (like
 * data from {@link WorkerContext}) is not thread-safe.
 *
 * Objects of this class only live for a single superstep.
 *
 * @param <I>
 *     Vertex id
 * @param <V>
 *     Vertex data
 * @param <E>
 *     Edge data
 * @param <M1>
 *     Incoming message type
 * @param <M2>
 *     Outgoing message type
 */
public abstract class Computation<I extends WritableComparable, V extends Writable, E extends Writable, M1 extends
Writable, M2 extends Writable>
    extends DefaultImmutableClassesGiraphConfigurable<I, V, E> implements
    TypesHolder<I, V, E, M1, M2>, WorkerAggregatorUsage {
    /** Logger */
    private static final Logger LOG = Logger.getLogger(Computation.class);

    /** Global graph state */
    private GraphState graphState;
    /** Handles requests */
    private WorkerClientRequestProcessor<I, V, E> workerClientRequestProcessor;
    /** Graph-wide BSP Mapper for this Computation */
    private GraphTaskManager<I, V, E> graphTaskManager;
    /** Worker aggregator usage */
    private WorkerAggregatorUsage workerAggregatorUsage;
    /** Worker context */
    private WorkerContext workerContext;

    /**
     * Modifications for Dynamic Graphs
     */
    /** True is the database is under update */
    protected BooleanWritable underUpdate = new BooleanWritable(false);
    /**
     * Injector vertex information
     */
    public static final LongWritable INJECTOR_VERTEX_ID = new LongWritable(-1);
    public static final DoubleWritable INJECTOR_VERTEX_VALUE = new DoubleWritable(
        -100);

    /**

```

```

/**
 * Must be defined by user to do computation on a single Vertex.
 *
 * @param vertex
 *       Vertex
 * @param messages
 *       Messages that were sent to this vertex in the previous
 *       superstep. Each message is only guaranteed to have a life
 *       expectancy as long as next() is not called.
 */
public abstract void compute(Vertex<I, V, E> vertex, Iterable<M1> messages)
    throws IOException;

/**
 * Prepare for computation. This method is executed exactly once prior to
 * {@link #compute(Vertex, Iterable)} being called for any of the vertices
 * in the partition.
 */
public void preSuperstep() {
}

/**
 * Finish computation. This method is executed exactly once after
 * computation for all vertices in the partition is complete.
 */
public void postSuperstep() {
}

/**
 * Injects new vertices into the computation, should be user defined.
 */
public void Inject() {
}

/**
 * Initialize, called by infrastructure before the superstep starts.
 * Shouldn't be called by user code.
 *
 * @param graphState
 *       Graph state
 * @param workerClientRequestProcessor
 *       Processor for handling requests
 * @param graphTaskManager
 *       Graph-wide BSP Mapper for this Vertex
 * @param workerAggregatorUsage
 *       Worker aggregator usage
 * @param workerContext
 *       Worker context
 */
public void initialize(GraphState graphState,
    WorkerClientRequestProcessor<I, V, E> workerClientRequestProcessor,
    GraphTaskManager<I, V, E> graphTaskManager,
    WorkerAggregatorUsage workerAggregatorUsage,
    WorkerContext workerContext) {
    this.graphState = graphState;
    this.workerClientRequestProcessor = workerClientRequestProcessor;
    this.graphTaskManager = graphTaskManager;
    this.workerAggregatorUsage = workerAggregatorUsage;
    this.workerContext = workerContext;
}

/**
 * Retrieves the current superstep.
 *
 * @return Current superstep
 */
public long getSuperstep() {
    return graphState.getSuperstep();
}

/**
 * Get the total (all workers) number of vertices that existed in the
 * previous superstep.
 *
 * @return Total number of vertices (-1 if first superstep)

```

```

*/
public long getTotalNumVertices() {
    return graphState.getTotalNumVertices();
}

/**
 * Get the total (all workers) number of edges that existed in the previous
 * superstep.
 *
 * @return Total number of edges (-1 if first superstep)
 */
public long getTotalNumEdges() {
    return graphState.getTotalNumEdges();
}

/**
 * Send a message to a vertex id.
 *
 * @param id
 *         Vertex id to send the message to
 * @param message
 *         Message data to send
 */
public void sendMessage(I id, M2 message) {
    workerClientRequestProcessor.sendMessageRequest(id, message);
}

/**
 * Send a message to all edges.
 *
 * @param vertex
 *         Vertex whose edges to send the message to.
 * @param message
 *         Message sent to all edges.
 */
public void sendMessageToAllEdges(Vertex<I, V, E> vertex, M2 message) {
    workerClientRequestProcessor.sendMessageToAllRequest(vertex, message);
}

/**
 * Send a message to multiple target vertex ids in the iterator.
 *
 * @param vertexIdIterator
 *         An iterator to multiple target vertex ids.
 * @param message
 *         Message sent to all targets in the iterator.
 */
public void sendMessageToMultipleEdges(Iterator<I> vertexIdIterator,
    M2 message) {
    workerClientRequestProcessor.sendMessageToAllRequest(vertexIdIterator,
        message);
}

/**
 * Sends a request to create a vertex that will be available during the next
 * superstep.
 *
 * @param id
 *         Vertex id
 * @param value
 *         Vertex value
 * @param edges
 *         Initial edges
 */
public void addVertexRequest(I id, V value, OutEdges<I, E> edges)
    throws IOException {
    Vertex<I, V, E> vertex = getConf().createVertex();
    vertex.initialize(id, value, edges);
    workerClientRequestProcessor.addVertexRequest(vertex);
}

/**
 * Sends a request to create a vertex that will be available during the next
 * superstep.
 *

```

```

* @param id
*     Vertex id
* @param value
*     Vertex value
*/
public void addVertexRequest(I id, V value) throws IOException {
    addVertexRequest(id, value, getConf().createAndInitializeOutEdges());
}

/**
 * Request to remove a vertex from the graph (applied just prior to the next
 * superstep).
 *
 * @param vertexId
 *     Id of the vertex to be removed.
 */
public void removeVertexRequest(I vertexId) throws IOException {
    workerClientRequestProcessor.removeVertexRequest(vertexId);
}

/**
 * Request to add an edge of a vertex in the graph (processed just prior to
 * the next superstep)
 *
 * @param sourceVertexId
 *     Source vertex id of edge
 * @param edge
 *     Edge to add
 */
public void addEdgeRequest(I sourceVertexId, Edge<I, E> edge)
    throws IOException {
    workerClientRequestProcessor.addEdgeRequest(sourceVertexId, edge);
}

/**
 * Request to remove all edges from a given source vertex to a given target
 * vertex (processed just prior to the next superstep).
 *
 * @param sourceVertexId
 *     Source vertex id
 * @param targetVertexId
 *     Target vertex id
 */
public void removeEdgesRequest(I sourceVertexId, I targetVertexId)
    throws IOException {
    workerClientRequestProcessor.removeEdgesRequest(sourceVertexId,
        targetVertexId);
}

/**
 * Get the mapper context
 *
 * @return Mapper context
 */
@SuppressWarnings("rawtypes")
public Mapper.Context getContext() {
    return graphState.getContext();
}

public void setUnderUpdate(BooleanWritable status) {
    this.underUpdate = status;
}

public boolean isUnderUpdate() {
    return this.underUpdate.get();
}

public Writable getInjectorId() {
    return INJECTOR_VERTEX_ID;
}

/**
 * Get the worker context
 *
 * @param <W>

```

```

*      WorkerContext class
* @return WorkerContext context
*/
@SuppressWarnings("unchecked")
public <W extends WorkerContext> W getWorkerContext() {
    return (W) workerContext;
}

@Override
public <A extends Writable> void aggregate(String name, A value) {
    workerAggregatorUsage.aggregate(name, value);
}

@Override
public <A extends Writable> A getAggregatedValue(String name) {
    return workerAggregatorUsage.<A> getAggregatedValue(name);
}
}

```

Table 26. Modified computation class

```

/*
 * Licensed to the Apache Software Foundation (ASF) under one
 * or more contributor license agreements. See the NOTICE file
 * distributed with this work for additional information
 * regarding copyright ownership. The ASF licenses this file
 * to you under the Apache License, Version 2.0 (the
 * "License"); you may not use this file except in compliance
 * with the License. You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
package org.apache.giraph.graph;

import org.apache.giraph.bsp.CentralizedServiceWorker;
import org.apache.giraph.comm.WorkerClientRequestProcessor;
import org.apache.giraph.comm.messages.MessageStore;
import org.apache.giraph.comm.netty.NettyWorkerClientRequestProcessor;
import org.apache.giraph.conf.ImmutableClassesGiraphConfiguration;
import org.apache.giraph.io.SimpleVertexWriter;
import org.apache.giraph.metrics.GiraphMetrics;
import org.apache.giraph.metrics.MetricNames;
import org.apache.giraph.metrics.SuperstepMetricsRegistry;
import org.apache.giraph.metrics.TimerDesc;
import org.apache.giraph.partition.Partition;
import org.apache.giraph.partition.PartitionStats;
import org.apache.giraph.time.SystemTime;
import org.apache.giraph.time.Time;
import org.apache.giraph.time.Times;
import org.apache.giraph.utils.MemoryUtils;
import org.apache.giraph.utils.TimedLogger;
import org.apache.giraph.worker.WorkerContext;
import org.apache.giraph.worker.WorkerThreadAggregatorUsage;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.log4j.Logger;

import com.google.common.collect.Iterables;
import com.google.common.collect.Lists;
import com.yammer.metrics.core.Counter;
import com.yammer.metrics.core.Timer;
import com.yammer.metrics.core.TimerContext;

import java.io.IOException;
import java.util.Collection;
import java.util.List;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.Callable;

```

```

/**
 * Compute as many vertex partitions as possible. Every thread will have its
 * own instance of WorkerClientRequestProcessor to send requests. Note that
 * the partition ids are used in the partitionIdQueue rather than the actual
 * partitions since that would cause the partitions to be loaded into memory
 * when using the out-of-core graph partition store. We should only load on
 * demand.
 *
 * @param <I> Vertex index value
 * @param <V> Vertex value
 * @param <E> Edge value
 * @param <M1> Incoming message type
 * @param <M2> Outgoing message type
 */
public class ComputeCallable<I extends WritableComparable, V extends Writable,
    E extends Writable, M1 extends Writable, M2 extends Writable>
    implements Callable<Collection<PartitionStats>> {
    /** Class logger */
    private static final Logger LOG = Logger.getLogger(ComputeCallable.class);
    /** Class time object */
    private static final Time TIME = SystemTime.get();
    /** Context */
    private final Mapper<?, ?, ?, ?>.Context context;
    /** Graph state */
    private final GraphState graphState;
    /** Thread-safe queue of all partition ids */
    private final BlockingQueue<Integer> partitionIdQueue;
    /** Message store */
    private final MessageStore<I, M1> messageStore;
    /** Configuration */
    private final ImmutableClassesGiraphConfiguration<I, V, E> configuration;
    /** Worker (for NettyWorkerClientRequestProcessor) */
    private final CentralizedServiceWorker<I, V, E> serviceWorker;
    /** Dump some progress every 30 seconds */
    private final TimedLogger timedLogger = new TimedLogger(30 * 1000, LOG);
    /** VertexWriter for this ComputeCallable */
    private SimpleVertexWriter<I, V, E> vertexWriter;
    /** Get the start time in nanos */
    private final long startNanos = TIME.getNanoseconds();

    // Per-Superstep Metrics
    /** Messages sent */
    private final Counter messagesSentCounter;
    /** Message bytes sent */
    private final Counter messageBytesSentCounter;
    /** Timer for single compute() call */
    private final Timer computeOneTimer;

    /**
     * Constructor
     *
     * @param context Context
     * @param graphState Current graph state (use to create own graph state)
     * @param messageStore Message store
     * @param partitionIdQueue Queue of partition ids (thread-safe)
     * @param configuration Configuration
     * @param serviceWorker Service worker
     */
    public ComputeCallable(
        Mapper<?, ?, ?, ?>.Context context, GraphState graphState,
        MessageStore<I, M1> messageStore,
        BlockingQueue<Integer> partitionIdQueue,
        ImmutableClassesGiraphConfiguration<I, V, E> configuration,
        CentralizedServiceWorker<I, V, E> serviceWorker) {
        this.context = context;
        this.configuration = configuration;
        this.partitionIdQueue = partitionIdQueue;
        this.messageStore = messageStore;
        this.serviceWorker = serviceWorker;
        this.graphState = graphState;

        SuperstepMetricsRegistry metrics = GiraphMetrics.get().perSuperstep();
        // Normally we would use ResetSuperstepMetricsObserver but this class is
        // not long-lived, so just instantiating in the constructor is good enough.

```

```

computeOneTimer = metrics.getTimer(TimerDesc.COMPUTE_ONE);
messagesSentCounter = metrics.getCounter(MetricNames.MESSAGES_SENT);
messageBytesSentCounter =
    metrics.getCounter(MetricNames.MESSAGE_BYTES_SENT);
}

@Override
public Collection<PartitionStats> call() {
    // Thread initialization (for locality)
    WorkerClientRequestProcessor<I, V, E> workerClientRequestProcessor =
        new NettyWorkerClientRequestProcessor<I, V, E>(
            context, configuration, serviceWorker);
    WorkerThreadAggregatorUsage aggregatorUsage =
        serviceWorker.getAggregatorHandler().newThreadAggregatorUsage();
    WorkerContext workerContext = serviceWorker.getWorkerContext();

    vertexWriter = serviceWorker.getSuperstepOutput().getVertexWriter();

    List<PartitionStats> partitionStatsList = Lists.newArrayList();
    while (!partitionIdQueue.isEmpty()) {
        Integer partitionId = partitionIdQueue.poll();
        if (partitionId == null) {
            break;
        }

        Partition<I, V, E> partition =
            serviceWorker.getPartitionStore().getPartition(partitionId);

        Computation<I, V, E, M1, M2> computation =
            (Computation<I, V, E, M1, M2>) configuration.createComputation();
        computation.initialize(graphState, workerClientRequestProcessor,
            serviceWorker.getGraphTaskManager(), aggregatorUsage, workerContext);
        computation.preSuperstep();

        try {
            PartitionStats partitionStats =
                computePartition(computation, partition);
            partitionStatsList.add(partitionStats);
            long partitionMsgs = workerClientRequestProcessor.resetMessageCount();
            partitionStats.addMessagesSentCount(partitionMsgs);
            messagesSentCounter.inc(partitionMsgs);
            long partitionMsgBytes =
                workerClientRequestProcessor.resetMessageBytesCount();
            partitionStats.addMessageBytesSentCount(partitionMsgBytes);
            messageBytesSentCounter.inc(partitionMsgBytes);
            timedLogger.info("call: Completed " +
                partitionStatsList.size() + " partitions, " +
                partitionIdQueue.size() + " remaining " +
                MemoryUtils.getRuntimeMemoryStats());
        } catch (IOException e) {
            throw new IllegalStateException("call: Caught unexpected IOException," +
                " failing.", e);
        } catch (InterruptedException e) {
            throw new IllegalStateException("call: Caught unexpected " +
                "InterruptedException, failing.", e);
        } finally {
            serviceWorker.getPartitionStore().putPartition(partition);
        }

        computation.postSuperstep();
    }

    // Return VertexWriter after the usage
    serviceWorker.getSuperstepOutput().returnVertexWriter(vertexWriter);

    if (LOG.isInfoEnabled()) {
        float seconds = Times.getNanosSince(TIME, startNanos) /
            Time.NS_PER_SECOND_AS_FLOAT;
        LOG.info("call: Computation took " + seconds + " secs for " +
            partitionStatsList.size() + " partitions on superstep " +
            graphState.getSuperstep() + ". Flushing started");
    }
}
try {
    workerClientRequestProcessor.flush();
    // The messages flushed out from the cache is

```

```

// from the last partition processed
if (partitionStatsList.size() > 0) {
    long partitionMsgBytes =
        workerClientRequestProcessor.resetMessageBytesCount();
    partitionStatsList.get(partitionStatsList.size() - 1).
        addMessageBytesSentCount(partitionMsgBytes);
    messageBytesSentCounter.inc(partitionMsgBytes);
}
aggregatorUsage.finishThreadComputation();
} catch (IOException e) {
    throw new IllegalStateException("call: Flushing failed.", e);
}
}
return partitionStatsList;
}

/**
 * Compute a single partition
 *
 * @param computation Computation to use
 * @param partition Partition to compute
 * @return Partition stats for this computed partition
 */
private PartitionStats computePartition(
    Computation<I, V, E, M1, M2> computation,
    Partition<I, V, E> partition) throws IOException,
    InterruptedException {
    PartitionStats partitionStats = new PartitionStats(partition.getId(),
        0, 0, 0, 0);
    // Make sure this is thread-safe across runs
    synchronized (partition) {
        for (Vertex<I, V, E> vertex : partition) {
            Iterable<M1> messages = messageStore.getVertexMessages(vertex
                .getId());

            // There is no computation in the case of an update.
            if ((computation.isUnderUpdate())
                && (vertex.getId() != computation.getInjectorId())) {
                computation.removeVertexRequest(vertex.getId());
                LOG.info("Superstep ["
                    + graphState.getSuperstep() + "]: Removing vertex");

                // Need to unwrap the mutated edges (possibly)
                vertex.unwrapMutableEdges();
                // Write vertex to superstep output (no-op if it is not
                // used)
                vertexWriter.writeVertex(vertex);
                // Need to save the vertex changes (possibly)
                partition.saveVertex(vertex);
            } else {
                // Injector vertex should run compute
                if (vertex.getId() == computation.getInjectorId()) {
                    if (computation.isUnderUpdate()) {
                        LOG.info("Update detected, waking injector");
                        vertex.wakeUp();
                        computation.Inject();
                    } else {
                        //The injector will be inactive while there are no updates.
                        vertex.voteToHalt();
                    }
                }

                // Need to unwrap the mutated edges (possibly)
                vertex.unwrapMutableEdges();
                // Write vertex to superstep output (no-op if it is not
                // used)
                vertexWriter.writeVertex(vertex);
                // Need to save the vertex changes (possibly)
                partition.saveVertex(vertex);
            } else {
                if (vertex.isHalted() && !Iterables.isEmpty(messages)) {
                    vertex.wakeUp();
                }
                if (!vertex.isHalted()) {
                    context.progress();
                    TimerContext computeOneTimerContext =
computeOneTimer

```



```

import org.json.JSONArray;
import org.json.JSONException;

import uk.co.qmul.giraph.dynamicgraph.PathAggregator;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.List;
import com.google.common.collect.Lists;

/**
 * Demonstrates a basic structure in order to allow Pregel to make computations
 * over Dynamic Graphs.
 *
 * @author Marco Aurelio Lotz
 */

@Algorithm(name = "Dynamic Graph Computation", description = "Makes computation on dynamic graphs")
public class DynamicGraphComputation
    extends
        BasicComputation<LongWritable, DoubleWritable, FloatWritable, DoubleWritable> {

    /**
     * Path Aggregator name. This Aggregator is used to communicate the
     * injection Path from the Master to the Injector vertex.
     */
    private static String PATH_AGG = "PathAgg";

    /**
     * Used to indicate the master that the update is finished
     */
    private static final BooleanWritable UpdateFinish = new BooleanWritable(false);

    /**
     * Serves as a flag of communication between vertices and the master.
     * Its value controls the update behaviour.
     */
    private static String U_UP_AGG = "UnderUpgradeAgg";

    /**
     * Maximum number of Supersteps to be computed before halting.
     */
    public final int MAX_SUPERSTEPS = 33;

    /**
     * Flag used by the injector to wait a superstep after removal only Before
     * starting the injection
     */
    public static boolean WaitedRemoval = false;

    /** Class logger */
    private static final Logger LOG = Logger
        .getLogger(DynamicGraphComputation.class);

    @Override
    public void compute(
        Vertex<LongWritable, DoubleWritable, FloatWritable> vertex,
        Iterable<DoubleWritable> messages) throws IOException {
        if (getSuperstep() < MAX_SUPERSTEPS) {
            // -----
            // Remove code and insert any computation code here
            sendMessage(vertex.getId(), new DoubleWritable(1d));
            vertex.voteToHalt();
            // -----
        } else {
            vertex.voteToHalt();
        }
    }

    @Override
    public void Inject() {
        // checks for updateFlag, that is changed in the presuperstep
        if (true == isUnderUpdate()) {
            LOG.info("Superstep: "
                + getSuperstep())

```

```

+ " - The master has communicated a modification in the file system");

// Only injects in the second call of this method.
// In the first call the other vertices will be being removed
// One can also use the number of vertices to trigger this event.
if (true == WaitedRemoval) {
    LOG.info("Injector updateFile System on superstep: "
            + getSuperstep());
    try {
        UpdateFileSystem();
        // Tell the file system that the update is over.
        InformMasterCompute();
    } catch (IOException e) {
        LOG.info("Problem Updating vertex database.");
    }
    WaitedRemoval = false;
} else {
    WaitedRemoval = true;
}
}

/**
 * Gets the path from the aggregator and configures the file system in order
 * to retrieve the file data
 *
 * @throws IOException
 */
public void UpdateFileSystem() throws IOException {
    FileSystem fs;
    Configuration config = new Configuration();
    FileStatus fileStatus;

    // Gets the HDFS paths
    Text inputString = getAggregatedValue(PATH_AGG);
    Path inputPath = new Path(inputString.toString());

    LOG.info("Injector: the path is" + inputPath.getParent()
            + inputPath.getName());

    fs = FileSystem.get(config);

    LOG.info("Checking file in File System");
    fileStatus = fs.getFileStatus(inputPath);

    if (null != fileStatus) {
        // Do the injection.
        BeginInjection(fs, inputPath);
    } else {
        LOG.info("Problem looking for the file in File System");
    }
}

/**
 * injects new vertex from desired input.
 *
 * @param file
 *         system that is going to be used
 * @param path
 *         that is going to be read
 * @throws IOException
 */
public void BeginInjection(FileSystem fs, Path path) throws IOException {

    // Create JSON variables
    String line;
    Text inputLine;
    JSONArray preProcessedLine;
    JSONDynamicReader DynamicReader = new JSONDynamicReader();

    // Create injection variables
    LongWritable vertexId;
    DoubleWritable vertexValue;
    Iterable<Edge<LongWritable, FloatWritable>> vertexEdges;

```

```

// Creates a buffered reader to read the input file
BufferedReader br = new BufferedReader(new InputStreamReader(
    fs.open(path)));
try {
    line = br.readLine();
    while (line != null) {
        // Processes the line information
        inputLine = new Text(line);
        preProcessedLine = DynamicReader.preprocessLine(inputLine);
        vertexId = DynamicReader.getId(preProcessedLine);
        vertexValue = DynamicReader.getValue(preProcessedLine);
        vertexEdges = DynamicReader.getEdges(preProcessedLine);

        // Requests vertex add
        addVertexRequest(vertexId, vertexValue);
        LOG.info("Superstep: " + getSuperstep()
            + " Adding vertex [id,value]:" + vertexId + ","
            + vertexValue);

        // Add edges to that vertex
        for (Edge<LongWritable, FloatWritable> edge : vertexEdges) {
            addEdgeRequest(vertexId, edge);
        }
        line = br.readLine();
    }
} catch (JSONException e) {
    LOG.info("Problem in JSON reader");
    e.printStackTrace();
} finally {
    br.close();
}
}

@Override
public void preSuperstep() {
    // Only creates the injector in the first superstep
    if (0 == getSuperstep()) {
        // One should make sure that only one injector is created
        // Or solve it through the vertex resolver.
        // This may cause race conditions in the execution of this code.
        try {
            addVertexRequest(INJECTOR_VERTEX_ID, INJECTOR_VERTEX_VALUE);
            LOG.info("Injector vertex created with success in superstep:"
                + getSuperstep());
        } catch (IOException e) {
            LOG.info("Problem creating the injector vertex.");
            e.printStackTrace();
        }
    }
    // Checks if Master indicated a modification in the FileSystem
    LOG.info("[PROMETHEUS] Checking update status" + getSuperstep());
    setUnderUpdate(((BooleanWritable) getAggregatedValue(U_UP_AGG)));
}

/**
 * Tells master compute that the vertex data-base is up-to-date
 */
public void InformMasterCompute() {
    LOG.info("Setting Under Update status to false");
    aggregate(U_UP_AGG, UpdateFinish);
}

/**
 * Master Compute associated with {@link DynamicGraphComputation}. It is the
 * first thing to run in each super step. It has an observer to track if
 * there is any modification in input
 */
public static class InjectorMasterCompute extends DefaultMasterCompute {
    /**
     * Insert the paths to be watched here. One can easily modify the
     * Aggregator to use an array of paths.
     */
    private String inputPath = "/user/hduser/dynamic/GoogleJSON.txt";
}

```

```

/**
 * Used by the master compute to avoid accessing the file system while
 * the workers are still processing a previous mutation
 */
boolean isUnderUpdate = false;

/**
 * Number of supersteps that it waited before
 * Before looking for another update.
 */
private int numberOfWaitedSupersteps = 0;

/** Class logger */
private final Logger LOG = Logger
    .getLogger(InjectorMasterCompute.class);

/**
 * Object that will track the given paths.
 */
FileObserver fileObserver;

@Override
public void initialize() throws InstantiationException,
    IllegalAccessException {

    // Register Aggregators
    registerPersistentAggregator(PATH_AGG, PathAggregator.class);
    registerPersistentAggregator(U_UP_AGG,
        BooleanOverwriteAggregator.class);

    // set Aggregators initial values
    setAggregatedValue(PATH_AGG, new Text(inputPath));
    setAggregatedValue(U_UP_AGG, new BooleanWritable(false));

    // Start the File Observer
    fileObserver = new FileObserver(inputPath);

    LOG.info("Dynamic Master Compute successfully initialized.");
}

@Override
public void compute() {
    LOG.info("MasterCompute - Superstep number: " + getSuperstep());
    isUnderUpdate = ((BooleanWritable) getAggregatedValue(U_UP_AGG))
        .get();

    // Waits two supersteps after an update in the files system
    // Then it assumes that the file system is up to date.
    // One can solve this by using aggregator.
    if (isUnderUpdate) {
        if (1 == numberOfWaitedSupersteps) {
            LOG.info("Changing is under update to FALSE");
            setAggregatedValue(U_UP_AGG, new BooleanWritable(false));
            numberOfWaitedSupersteps = 0;
        } else {
            numberOfWaitedSupersteps++;
        }
    }

    // If the framework already finished processing previous mutations
    if (!isUnderUpdate) {
        LOG.info("The framework is ready for mutations");
        // Reset aggregator value.
        setAggregatedValue(U_UP_AGG, new BooleanWritable(false));

        // Uncomment the line below in order to enable dynamic input
        // analysis
        // FileObserver.checkFileModification();

        // This next line is just for debugging the application. It will
        // indicate a file modification in determined supersteps.

        if (getSuperstep() % 10 == 9) {
            LOG.info("Creating framework update notification");
            setAggregatedValue(U_UP_AGG, new BooleanWritable(true));

```

```

    }

    // Inform injector if modification in FS
    if (true == fileObserver.getFileModified()) {
        LOG.info("Modification in file:" + inputPath);
        setAggregatedValue(U_UP_AGG, new BooleanWritable(true));
    }
}

/**
 * Observes the HDFS for any modification in the input files
 */
public static class FileObserver {
    private long modificationTime;

    /**
     * Keeps track of the last modification time of the file.
     */
    private Path PATH;

    /**
     * True if the file was modified.
     */
    private boolean fileModified = false;

    /** Class logger */
    private final Logger LOG = Logger.getLogger(FileObserver.class);

    /**
     * Configuration file for the file system
     */
    Configuration config = new Configuration();

    private FileSystem fs;

    /**
     * Used to get the timestamp
     */
    FileStatus fileStatus;

    FileObserver(String inputPath) {
        PATH = new Path(inputPath);

        // Initialises the file system
        try {
            fs = FileSystem.get(config);
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Gets the designed file status
        try {
            fileStatus = fs.getFileStatus(PATH);
        } catch (IOException e) {
            e.printStackTrace();
        }

        modificationTime = fileStatus.getModificationTime();
        LOG.info("Original modification time (UTC) is: "
            + modificationTime);
        this.fileModified = false;
    }

    /**
     * Check if the tracked file was modified in the beginning of every
     * super step
     */
    public void checkFileModification() {

        // Check if one has to update the file status every single time.
        try {
            fileStatus = fs.getFileStatus(PATH);
        } catch (IOException e) {

```

```

        LOG.info("Error getting File status.");
    }

    long modtime = fileStatus.getModificationTime();
    if (modtime != modificationTime) {
        fileModified = true;
        this.modificationTime = modtime;
    } else {
        fileModified = false;
    }
}

public boolean getFileModified() {
    return this.fileModified;
}
}

/**
 * This is just a modification of the JSON Reader class available in the
 * Giraph original classes.
 */
public static class JSONDynamicReader {

    public JSONDynamicReader() {
    }

    public JSONArray preprocessLine(Text line) throws JSONException {
        return new JSONArray(line.toString());
    }

    public LongWritable getLd(JSONArray jsonVertex) throws JSONException,
        IOException {
        return new LongWritable(jsonVertex.getLong(0));
    }

    public DoubleWritable getValue(JSONArray jsonVertex)
        throws JSONException, IOException {
        return new DoubleWritable(jsonVertex.getDouble(1));
    }

    public Iterable<Edge<LongWritable, FloatWritable>> getEdges(
        JSONArray jsonVertex) throws JSONException, IOException {
        JSONArray jsonEdgeArray = jsonVertex.getJSONArray(2);
        List<Edge<LongWritable, FloatWritable>> edges = Lists
            .newArrayListWithCapacity(jsonEdgeArray.length());
        for (int i = 0; i < jsonEdgeArray.length(); ++i) {
            JSONArray jsonEdge = jsonEdgeArray.getJSONArray(i);
            edges.add(EdgeFactory.create(
                new LongWritable(jsonEdge.getLong(0)),
                new FloatWritable((float) jsonEdge.getDouble(1))));
        }
        return edges;
    }

    public Vertex<LongWritable, DoubleWritable, FloatWritable> handleException(
        Text line, JSONArray jsonVertex, JSONException e) {
        throw new IllegalArgumentException("Couldn't get vertex from line "
            + line, e);
    }
}
}

```

Table 28. Dynamic Graph Computation class

```

/**
 * Licensed to the Apache Software Foundation (ASF) under one
 * or more contributor license agreements. See the NOTICE file
 * distributed with this work for additional information
 * regarding copyright ownership. The ASF licenses this file
 * to you under the Apache License, Version 2.0 (the
 * "License"); you may not use this file except in compliance
 * with the License. You may obtain a copy of the License at

```

```

*
* http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*/

package uk.co.qmul.giraph.dynamicgraph;

import org.apache.giraph.aggregators.BasicAggregator;
import org.apache.hadoop.io.Text;

/**
 * Aggregator used for Text.
 * @author: Marco Aurelio Lotz
 */
public class PathAggregator extends BasicAggregator<Text> {

    @Override
    public void aggregate(Text value) {
        setAggregatedValue(value);
    }

    @Override
    public Text createInitialValue() {
        return new Text();
    }
}

```

Table 29. Path Aggregator class