Volume

1

NATURALIS BIODIVERSITY CENTER
BIOINFORMATICS

MapReduce Pruner V2.0 User Guide

BIOINFORMATICS

MapReduce Pruner V2.0 User Guide

Naturalis Biodiversity Center
Darwinweg 4 • Bioinformatics
Leiden, 2333 CR
Phone 071 568 7600 • Fax

Table of Contents

Chapter

1

Chapter I: MapReduce Pruner algorithm V2.0.

Origin and background information on the MapReduce Pruner algorithm. Introduction as to why to use this program.

Ongoing research ensures that more and more biological species are found over the years. In the study of phylogenetics, the relationships among groups of these organisms are explored. Inferred relationships between species are represented in so called phylogenetic trees. These trees capture the relationships between species based on differences and similarities in genetic and physical characteristics. In these trees, taxa that are joined together are implied to be descended from a common ancestor. A problem with some current phylogenetic trees is that they have become too large to handle conveniently. Loading the trees in software that has been made to view phylogenetic trees takes too long because of the large amount of complex data that has to be loaded. In addition, when a researcher wants only a small fragment of the entire tree, it is not a simple task to extract this from the entire tree, because of the very complex structure that these phylogenetic trees usually have. This means that scientists are sometimes practically constrained to use the entire tree to be able to analyse the data of those few organisms they actually need. This means that they would have to use far more computing power than necessary and it would take a lot longer to analyse the data.

The goal of the program that is to extract small subsets out of arbitrary-size phylogenetic trees. In order to extract the subsets,

the program uses a programming model called MapReduce. This model works on the principle of filtering and reducing, allowing the program to reduce the content of the input (the nodes and tips of the tree) to the desired subset.

MR-program introduction

The program consists of three chained MapReduce jobs, which reduces the input tree to the desired subtree. By filtering out the unnecessary nodes and the redundant internal nodes, the sub tree can be obtained. The tree is then converted to the Newick file format.

Chapter 2

Chapter II: User instructions MR-Pruner

Instructions on the documents, source code and the execution of the program, on the Hadoop cluster and local.

Clone the documents from Github

To download or clone the data from the Github repository, go to the directory the data has to be stored using the terminal. Then type the command to clone the data;

In the terminal move to the wanted directory with the 'cd' command.

```
~$ git clone
https://github.com/naturalis/tolomatic-java.git
```

The URL used in the command can be found on the main page of the tolomatic-java repository. It is also possible to download the entire repository as a zip file from the Github Naturalis/tolomatic-java main page.

Project structure

To be able to use the supplied (e.g.)config file, it is important to maintain the project structure as this has been used on the Github repository. It is, of course, possible to change the project structure. In this case note that the set-up settings in the config file will have to be changed accordingly. In this case it might be easier to start using the command line arguments.

Input and output data setup

Data folder

The data folder contains the de-normalized database. Creating this database is done in a pre-processing step, of the MapReduce Pruner program. The treesplitter.pl script, included on the Github repository, can be used to create this de-normalized database. The database has to include the data of all taxons of the original tree.

Note!!

The treesplitter.pl script is written specific for a certain input file, e.g. Newick or tree file. When using a different input file, the program shall have to be edited.

Input folder

The input folder contains the taxon file. This document is composed of a list of either taxon names or ID's. The taxons should be newline separated and in case of taxon names, these should all be capitalized. Concatenated taxon names are also allowed, in which case only the first name needs to be capitalized and all the names need to be whitespace separated. The content of this list, is what specifies the sub tree that will be extracted.

The temp folder

This folder will be used to store the intermediate output data. This is the data that is saved after all three of the Reduce steps of the program. The amount of files that are stored in this folder, is dependent on the amount of Reducers that are used. The folder does not have to be created before executing the program, since the program will create or overwrite this folder itself.

The output folder

The data from the third Reduce step is used to create the Newick tree and it then stored in the location indicated in the config file. This can just be the project directory, but if multiple trees are created it would be helpful to store them in an output folder. This folder, as opposed to the temp folder, will have to be created before executing the program.

Source code and test classes

As it is, the program consists of four packages. Of most classes of the program, Junit classes have been made in order to simplify testing the program. The test classes are included in the src folder within the 'test' package and are stored in packages corresponding to the packages that contain the source code.

mapreducepruner

Classes

All classes in the mapreducepruner package are involved in creating, emitting and runnen the Hadoop job, either local or on the server. In short, the MapReducePruner class, which is the main, calls the MrpRun class. The latter holds the large part of code to create and emit the MapReduce jobs. MrpPass1Mapper till MrpPass3Reducer hold the code for the actual Map and Reduce jobs. MrpResult is used to convert the output of the third result to a Newick tree format.

Test classes

Due to some difficulties, no test classes have been created for the MapReducePruner classes. Junit did not seem to work because of the 'context' MapReduce works with. And the open source MRUnit had not been updated in over a year, which caused version difficulties. However as Hadoop the definitive guide proscribes (White n.d.)), it is possible to create MapReduce test classes by making mock objects of the individual Map and Reduce methods. For now this will be for future work.

There is one testclass available within the mapreduceprune package. This test class contains a number of 'dirty' methods to test a number of methods from the first mapper. If further developments using mock objects have been made, this class better be left out.

mrpoption

Classes

This package contains all the classes that do the 'boring' work. For instance, all the checks concerning whether or not files and paths exist, with in addition. And also the code for the command line options using commons-cli and the int extension.

Test classes

Every method of every class in the mrpoption package has been tested using JUnit. The test classes can all be executed separately from each other, or with use of the test suite class that has been written, all together.

mrppath

Classes

The three classes included within the mrppath package are the only few classes that have stayed more or less the same to the prototype version. The Pathnode class is a helper class for the MapReduce Pruner program. Pathnode objects help to store and convert nodes in the paths from the taxons to their root nodes. The PathNodeInternal class extends the PathNode class with the integer TipCount. This count expresses how many (external) tip nodes have the particular (internal) node in their path. The PathNodeSet class maintains a set of PathNodes in a (jave.util.) TreeSet and contains some helper methods for it.

Test classes

Every method of every class in the mrptree package has been tested using JUnit. The test classes can all be executed separately from each other, or with use of the test suite class that has been written, all together.

mrptree

Classes

The mrptree package contains the two classes that are used for constructing a tree structure which can be used to construct a tree in Newick format.

Test classes

Every method of every class in the mrptree package has been tested using JUnit. The test classes can all be executed separately from each other, or with use of the test suite class that has been written, all together.

Additional documents

Maven pom.xml file

The Maven POM file is used for two purposes. First, the file is used for importing of the third-party libraries that are necessary to run the program

within the IDE. In addition the same file is used to create the jar necessary to run the program on the Hadoop cluster or local via the terminal. In the latter, the pom file will also ensure that all necessary third party libraries are included within the jar. When creating the jar via the pom file, a number of plugins are needed, these are included within the pom file. The included pom.xml file contains all the necessary libraries, plugins and further settings to support the current MapReduce Pruner program. When any additional packages are needed, these can be found on the online Maven repository (or a third party repository) and added to the pom.xml file within the dependencies section. If the necessary package is not included in the Maven repository, any other repository can be used as long as these support maven. When third-party repositories are used, a reference to this repository has to be made in the pom file under the repositories section.

Config file

The config file is a basic .ini file, which is used for specifications of the locations of files and folders indicated in the chapter *Input and output data setup*, page 5. This file contains three sections. The Main section contains the paths to the input, output and temp folders. It also contains a variable 'root', which indicates what the folder is that contains the de-normalized database. In the third section, the actual de-normalized database folder can be indicated using the 'dataDir' variable. This location has been included in a separate section to enable the possibility of using multiple trees. The second section is indicated with the section name 'Tree', the values included in this section, are used as default values in case the user has not specified a folder for the tree in the third section.
The user can also choose to not use the config file to indicate all the necessary files and folders. In that case, the files and folders can be specified using command line arguments, see chapter *Additional options; Command line arguments*.

Log4j.properties file

The program uses a logger to return certain values in the program (mostly used for debugging), instead of using standard print statements. The log4j.properties file makes that the logger statements can be returned to a log file, opposed to returning it to the console. This could be convenient to store potential errors of the program. The logger can be used on certain levels (e.g. debugging, info, error). Which level to use is dependent on the information that is wanted. The level of logging can be changed in the command on the first line of the config file. The default is set to Info.

Additional options

Command line arguments

In case the user decides to not use the config file for specification of the necessary files and folders, it is also possible to use command line arguments for specification. . In that case any command line options are used in addition to the command for execution of the program. All possible command line arguments are included in Table 1**.** The syntax needs to be like following examples;

```
~$ <yarn/java command> -c "<path to>/config.ini"

~$ <yarn/java command> -c "<path to>/config.ini"  -i "<path
to>/input.txt"
```

Table 1: *List of possible command line arguments, that can be used for the Tree Pruning program.*

Command line argument	**Short description**	**Long description**
n	Environment var	Boolean option: don't use environment variables yes/no
c	Config filename	path to config file
i	Input filename	path to taxon input file
t	Tempdir dirname	path to temp directory
o	Output file name	path to newick output file
r	Dataroot dirname	path to dataroot directory
u	Treeurl url	path url to treedata
d	Datadir dirname	name of data directory

Program preparation

Creating a jar using Maven

Creating a jar can be performed in many ways, but in this case it turned out best to use create a jar with use of Maven. When maven has been installed, by following the instructions in the readme.md file, the jar can be created by typing the following command;

In the terminal move to the project directory with the 'cd' command.

```
~$  mvn package
```

This command will create the jar and save it in the 'target' folder that is now created in the project directory. Two .jar files stored here, both named after the project name that has been specified in the pom.xml file. The .jar that, does not have the prefix 'original' is the one that will be used. Important is that building the jar this way has to be re-done every time the program has been changed.

Program execution

Execution on the Hadoop cluster

Executing the program on the Hadoop cluster works in a slightly different way from a normal server. First a connection with the server has to be established. When all the installations for the Kerberos Client are done, a connection can be established by requesting a kerberos ticket.

Requesting a Kerberos ticket

To request such a ticket, enter the below commands in the terminal.

```
~$ cd hathi-client && . conf/settings.linux; cd -
```

```
~$ kinit <username>@CUA.SURFSARA.NL
```

When the connections has been established, the user has full access to the cluster as well as its own computer. The server can be approached by using the 'hdfs dfs' command. For instance, if the user would want to get an overview of all the folder and files of its account, the below command can be used;

```
~$ hdfs dfs -ls /user/<username>
```

A large amount of the standard bash commands can be used in the same way, together with the hdfs command. Noted should be that every command should start with a minus sign, since they are used as options of the hdfs command. So when the user would want to make a new folder on the cluster, "`-mkdir`" can be used, and to remove a folder "`-rm -r`".

Setting up the environment of the program

Between the program and the cluster, a certain setup of the files is necessary. Both the taxon file as the de-normalized database should be located at the Hadoop cluster. Using the option 'put' the files can be pushed to the cluster. The commands to put the input file and database on to the cluster should look like below;

```
~$ hdfs dfs -put <path to> /InputFile.txt /user/<username>/InputData
```

```
~$ hdfs dfs -put <path to> /data /user/<username>
```

All the other documents have to be located at the local computer, whereby three conditions have to be met. A jar has to be made of the entire source code (see … create jar), including the pom.xml file and the log4j.properties file. The config file has to be stored in a separate folder, which will be the work directory. When all the necessary requirements have been met, the program can be executed. To execute the program, a yarn command has to be used. Within the terminal, move to the folder that contains the config file, then execute the program using the following command;

```
~$ yarn jar <path to>/createdJar.jar
```

The program uses the config file that is stored in the folder of which the program is executed as default. When another config file should be used, then add " –config "<path to>/configFile.ini" to the yarn command above. The same goes if any other command line options are wanted. These can be included at the end of the yarn command as well. The available command line options are specified in chapter Additional options; Command line arguments, 10.

Obtaining the data

Afterwards, the data can be obtained using the hdfs command. Here, use the '-get' argument to move the data from the cluster to the local computer.

```
~$ hdfs dfs -get <path to> /LocalDir
/user/<username>/InputData/InputFile.txt


~$ hdfs dfs -get <path to> /localDir /user/<username>/data/*
```

Local execution

Although the program has been written for execution on the cluster, it is fully functional for local execution. In some cases the latter might be easier. For instance when using a very small dataset or when doing a simple program run for the purpose of testing for instance. Performing a local run, can be done in two ways, via the IDE or the terminal.

IDE : IntelliJ IDEA

When the installation has been performed according to the installation instructions in the readme.md file on Github (https://github.com/naturalis/tolomatic-java/blob/master/README.md). IntelliJ should have been installed. In addition a project has been made in which the pom.xml has been included as maven project. The next step is to include all the necessary files within the project directory at the right location. To do this, the files can be copied from the tolomatic-java Github clone (see chapter: Clone the documents from Github) directly into the project directory. The folders that have to be included at the very least, are the input and data folder, the conf folder and the src folder, although the last one cannot just be copied. When pasting the src folder, IntelliJ will argue that there already is such a folder in the project directory. To solve this problem simply, copy the content and not src itself. As last, the log4j.properties file has to be stored in the project directory as a separate file. Last, the command line arguments will have to be put in place in case necessary. These arguments can be set in the projects 'edit configuration', which can be found within the projects settings. When all of these steps have been conducted, the program can be executed.

Terminal

Executing the program local via the terminal is quite similar to executing it on the cluster. The environment is setup in the order as explained in chapter

Setting up the environment of the program, with the exception of both the input files. The taxon file and the de-normalized database are in this case stored in the work directory. The command that is used will use the java argument instead of the yarn argument;

```
~$  java jar <path to>/createdJar.jar
```

Here again any necessary command line arguments have to be inserted ad the end of the command. For the syntax see chapter Additional options; Command line arguments, 10.

Index