Data Mining Algorithms In R/Frequent Pattern Mining/The FP-Growth Algorithm

In Data Mining the task of finding frequent pattern in large databases is very important and has been studied in large scale in the past few years. Unfortunately, this task is computationally expensive, especially when a large number of patterns exist.

The FP-Growth Algorithm, proposed by Han in [1], is an efficient and scalable method for mining the complete set of frequent patterns by pattern fragment growth, using an extended prefix-tree structure for storing compressed and crucial information about frequent patterns named frequent-pattern tree (FP-tree). In his study, Han proved that his method outperforms other popular methods for mining frequent patterns, e.g. the Apriori Algorithm [2] and the TreeProjection [3]. In some later works [4] [5] [6] it was proved that FP-Growth has better performance than other methods, including Eclat [7] and Relim [8]. The popularity and efficiency of FP-Growth Algorithm contributes with many studies that propose variations to improve his performance [5] [6] [9] [10] [11] [12] [13] [14] [15] [16].

This chapter describes the algorithm and some variations and discuss features of the R language and strategies to implement the algorithm to be used in the R. Next a briefly conclusion and future works are proposed.

Contents

The algorithm

The FP-Growth Algorithm is an alternative way to find frequent itemsets without using candidate generations, thus improving performance. For so much it uses a divide-and-conquer strategy [17]. The core of this method is the usage of a special data structure named frequent-pattern tree (FP-tree), which retains the itemset association information.

In simple words, this algorithm works as follows: first it compresses the input database creating an FP-tree instance to represent frequent items. After this first step it divides the compressed database into a set of conditional databases, each one associated with one frequent pattern. Finally, each such database is mined separately. Using this strategy, the FP-Growth reduces the search costs looking for short patterns recursively and then concatenating them in the long frequent patterns, offering good selectivity.

In large databases, it's not possible to hold the FP-tree in the main memory. A strategy to cope with this problem is to firstly partition the database into a set of smaller databases (called projected databases), and then construct an FP-tree from each of these smaller databases.

The next subsections describe the FP-tree structure and FP-Growth Algorithm, finally an example is presented to make it easier to understand these concepts.

FP-Tree structure

The frequent-pattern tree (FP-tree) is a compact structure that stores quantitative information about frequent patterns in a database [4].

Han defines the FP-tree as the tree structure defined below [1]:

1. One root labeled as "null" with a set of item-prefix subtrees as children, and a frequent-item-header table (presented in the left side of Figure 1);
2. Each node in the item-prefix subtree consists of three fields:
    1. Item-name: registers which item is represented by the node;
    2. Count: the number of transactions represented by the portion of the path reaching the node;
    3. Node-link: links to the next node in the FP-tree carrying the same item-name, or null if there is none.

1. Each entry in the frequent-item-header table consists of two fields:
    1. Item-name: as the same to the node;
    2. Head of node-link: a pointer to the first node in the FP-tree carrying the item-name.

Additionally the frequent-item-header table can have the count support for an item. The Figure 1 below show an example of a FP-tree.

Figure 1: An example of an FP-tree from [17].

The original algorithm to construct the FP-Tree defined by Han in [1] is presented below in Algorithm 1.

Algorithm 1: FP-tree construction

Input: A transaction database DB and a minimum support threshold ?.

Output: FP-tree, the frequent-pattern tree of DB.

Method: The FP-tree is constructed as follows.

1. Scan the transaction database DB once. Collect F, the set of frequent items, and the support of each frequent item. Sort F in support-descending order as FList, the list of frequent items.
2. Create the root of an FP-tree, T, and label it as "null". For each transaction Trans in DB do the following:

    - Select the frequent items in Trans and sort them according to the order of FList. Let the sorted frequent-item list in Trans be [p | P], where p is the first element and P is the remaining list. Call insert tree([p | P], T).
    - The function insert tree([p | P], T) is performed as follows. If T has a child N such that N.item-name = p.item-name, then increment N 's count by 1; else create a new node N , with its count initialized to 1, its parent link linked to T , and its node-link linked to the nodes with the same item-name via the node-link structure. If P is nonempty, call insert tree(P, N) recursively.

By using this algorithm, the FP-tree is constructed in two scans of the database. The first scan collects and sort the set of frequent items, and the second constructs the FP-Tree.

FP-Growth Algorithm

After constructing the FP-Tree it's possible to mine it to find the complete set of frequent patterns. To accomplish this job, Han in [1] presents a group of lemmas and properties, and thereafter describes the FP-Growth Algorithm as presented below in Algorithm 2.

Algorithm 2: FP-Growth

Input: A database DB, represented by FP-tree constructed according to Algorithm 1, and a minimum

support threshold ?.

Output: The complete set of frequent patterns.

Method: call FP-growth(FP-tree, null).

Procedure FP-growth(Tree, a) {

(01) if Tree contains a single prefix path then // Mining single prefix-path FP-tree {

(02) let P be the single prefix-path part of Tree;

(03) let Q be the multipath part with the top branching node replaced by a null root;

(04) for each combination (denoted as ß) of the nodes in the path P do

(05) generate pattern ß ∪ a with support = minimum support of nodes in ß;

(06) let freq pattern set(P) be the set of patterns so generated;

}

(07) else let Q be Tree;

(08) for each item ai in Q do { // Mining multipath FP-tree

(09) generate pattern ß = ai ∪ a with support = ai .support;

(10) construct ß's conditional pattern-base and then ß's conditional FP-tree Tree ß;

(11) if Tree ß ≠ Ø then

(12) call FP-growth(Tree ß , ß);

(13) let freq pattern set(Q) be the set of patterns so generated;

}

(14) return(freq pattern set(P) ∪ freq pattern set(Q) ∪ (freq pattern set(P) × freq pattern set(Q)))

}

When the FP-tree contains a single prefix-path, the complete set of frequent patterns can be generated in three parts: the single prefix-path P, the multipath Q, and their combinations (lines 01 to 03 and 14). The resulting patterns for a single prefix path are the enumerations of its subpaths that have the minimum support (lines 04 to 06). Thereafter, the multipath Q is defined (line 03 or 07) and the resulting patterns from it are processed (lines 08 to 13). Finally, in line 14 the combined results are returned as the frequent patterns found.

An example

This section presents a simple example to illustrate how the previous algorithm works. The original example can be viewed in [18].

Consider the transactions below and the minimum support as 3:

	i(t)
1	ABDE
2	BCE
3	ABDE
4	ABCE
5	ABCDE
6	BCD

To build the FP-Tree, frequent items support are first calculated and sorted in decreasing order resulting in the following list: { B(6), E(5), A(4), C(4), D(4) }. Thereafter, the FP-Tree is iteratively constructed for each transaction, using the sorted list of items as shown in Figure 2.

(a) Transaction 1: BEAD (b) Transaction 2: BEC (c) Transaction 3: BEAD (d) Transaction 4: BEAC

(e) Transaction 5: BEACD (f) Transaction 6: BCD

Figure 2: Constructing the FP-Tree iteratively.

As presented in Figure 3, the initial call to FP-Growth uses the FP-Tree obtained from the Algorithm 1, presented in Figure 2 (f), to process the projected trees in recursive calls to get the frequent patterns in the transactions presented before.

Using a depth-first strategy the projected trees are determined to items D, C, A, E and B, respectively. First the projected tree for D is recursively processed, projecting trees for DA, DE and DB. In a similar manner the remaining items are processed. At the end of process the frequent itemset is: { DAE, DAEB, DAB, DEB, CE, CEB, CB, AE, AEB, AB, EB }.

Figure 3: Projected trees and frequent patterns founded by the recursively calls to FP-Growth Algorithm.

FP-Growth Algorithm Variations

As mentioned before, the popularity and efficiency of FP-Growth Algorithm contributes with many studies that propose variations to improve its performance [5] [6] [9] [12] [19] [12] [13] [14] [15] [16]. In this section some of them are briefly described.

DynFP-Growth Algorithm

The DynFP-Growth [13] [5], has focused in improving the FP-Tree algorithm construction based on two observed problems:

1. The resulting FP-tree is not unique for the same “logical” database;
2. The process needs two complete scans of the database.

To solve the first problem Gyorödi C., et al. [13] proposes the usage of a support descending order together with a lexicographic order, ensuring in this way the uniqueness of the resulting FP-tree for different “logically equivalent” databases. To solve the second problem they proposed devising a dynamic FP-tree reordering algorithm, and employing this algorithm whenever a “promotion” to a higher order of at least one item is detected.

An important feature in this approach is that it’s not necessary to rebuild the FP-Tree when the actual database is updated. It’s only needed to execute the algorithm again taking into consideration the new transactions and the stored FP-Tree.

Another adaptation proposed, because of the dynamic reordering process, is a modification in the original structures, by replacing the single linked list with a doubly linked list for linking the tree nodes to the header and adding a master-table to the same header. See [13] for more details.

FP-Bonsai Algorithm

The FP-Bonsai [6] improve the FP-Growth performance by reducing (pruning) the FP-Tree using the ExAnte data-reduction technique [14]. The pruned FP-Tree was called FP-Bonsai. See [6] for more details.

AFOPT Algorithm

Investigating the FP-Growth algorithm performance Liu proposed the AFOPT algorithm in [12]. This algorithm aims at improving the FP-Growth performance in four perspectives:

- Item Search Order: when the search space is divided, all items are sorted in some order. The number of the conditional databases constructed can differ very much using different items search orders;
- Conditional Database Representation: the traversal and construction cost of a conditional database heavily depends on its representation;
- Conditional Database Construction Strategy: constructing every conditional database physically can be expensive affecting the mining cost of each individual conditional database;
- Tree Traversal Strategy: the traversal cost of a tree is minimal using top-down traversal strategy.

See [12] for more details.

NONORDFP Algorithm

The Nonordfp algorithm [15] [5] was motivated by the running time and the space required for the FP-Growth algorithm. The theoretical difference is the main data structure (FP-Tree), which is more compact and which is not needed to rebuild it for each conditional step. A compact, memory efficient representation of an FP-tree by using Trie data structure, with memory layout that allows faster traversal, faster allocation, and optionally projection was introduced. See [15] for more details.

FP-Growth* Algorithm

This algorithm was proposed by Grahne et al [16] [19], and is based in his conclusion about the usage of CPU time to compute frequent item sets using FP-Growth. They observed that 80% of CPU time was used for traversing FP-Trees [9]. Therefore, they used an array-based data structure combined with the FP-Tree data structure to reduce the traversal time, and incorporates several optimization techniques. See [16] [19] for more details.

PPV, PrePost, and FIN Algorithm

These three algorithms were propsed by Deng et al [20] [21] [22], and are based on three novel data structures called Node-list [20], N-list [21], and Nodeset [22] respectively for facilitating the mining process of frequent itemsets. They are based on a FP-tree with each node encoding with pre-order traversal and post-order traversal. Compared with Node-lists, N-lists and Nodesets are more efficient. This causes the efficiency of PrePost [21] and FIN [22] is higher than that of PPV [20]. See [20] [21] [22] for more details.

Data Visualization in R

Normally the data used to mine frequent item sets are stored in text files. The first step to visualize data is load it into a data-frame (an object to represent the data in R).

The function read.table could be used in the following way:

```
var <- read.table(fileName, fileEncoding=value, header = value, sep = value)

Where:
  • var: the R variable to receive the loaded data.
  • fileName: is a string value with the name of the file to be loaded.
  • fileEncoding: to be used when the file has no-ASCII characters.
  • header: indicates that the file has headers (T or TRUE) or not (F or FALSE).
  • sep: defines the field separator character (“,”, “;” or “\t” for example)
  • Only the filename is a mandatory parameter.
```

Another function in R to load data is called scan. See the R Data Import/Export Manual (http://cran.r-project.org/doc/manuals/R-data.html) for details.

The visualization of the data can be done in two ways:

- Using the variable name (var), to list the data in a tabular presentation.
- And summary(var), to list a summary of the data.

Example:

```
> data <- read.table("boolean.data", sep=",", header=T)

> data
      A    B     C     D     E
1  TRUE TRUE FALSE  TRUE  TRUE
2 FALSE TRUE  TRUE FALSE  TRUE
3  TRUE TRUE FALSE  TRUE  TRUE
4  TRUE TRUE  TRUE FALSE  TRUE
5  TRUE TRUE  TRUE  TRUE  TRUE
6 FALSE TRUE  TRUE  TRUE FALSE

> summary(data)
     A              B               C              D              E
 Mode :logical   Mode:logical   Mode :logical   Mode :logical   Mode :logical
 FALSE:2         TRUE:6         FALSE:2         FALSE:2         FALSE:1
 TRUE :4         NA's:0         TRUE :4         TRUE :4         TRUE :5
 NA's :0                        NA's :0         NA's :0         NA's :0
```

In the example above the data in "boolean.data", that have a simple binary database, was loaded in the data-frame variable data. Typing the name of the variable in the command line, its content is printed, and typing the summary command the frequency occurrence of each item is printed. The summary function works differently. It depends on the type of data in the variable, see [23] [24] [25] for more details.

The functions presented previously can be useful, but to frequent item set datasets use an specific package called arules (https://r-forge.r-project.org/projects/arules/) [26] which is better to visualize the data.

Using arules, several functions are made available:

- read.transactions: used to load the database file into a variable.
- inspect: used to list the transactions.
- length: returns the number of transactions.
- image: plots an image with all transactions in a matrix format.
- itemFrequencyPlot: calculates the frequency of each item and plots it in a bar graphic.

Example:

```
> data <- read.transactions("basket.data", format="basket", sep = ",")

> data
transactions in sparse format with
 6 transactions (rows) and
 5 items (columns)

> inspect(data)
  items
1 {A,
   B,
   D,
   E}
2 {B,
   C,
   E}
3 {A,
   B,
   D,
   E}
4 {A,
   B,
   C,
   E}
5 {A,
   B,
   C,
   D,
   E}
6 {B,
   C,
   D}

> length(data)
[1] 6

> image(data)

> itemFrequencyPlot(data, support=0.1)
```

In this example we can see the difference in the usage of the variable name in the command line. From transactions, only the number of rows (transactions) and cols (items) are printed. The result of image(data) and itemFrequencyPlot(data, support = 0.1) are presented in the figures 4 and 5 below.

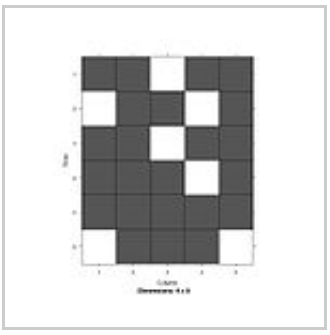

Figure 4: Result of the image(data) call.

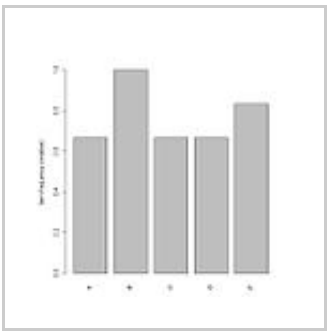

Figure 5: Result of the itemFrequencyPlot(data, support = 0.1) call.

Implementation in R

The R [23] [24] [25] [27] [28] provides several facilities for data manipulation, calculation and graphical display very useful for data analysis and mining. It can be used as both a statistical library and a programming language.

As a statistical library, it provides a set of functions to summary data, matrix facilities, probability distributions, statistical models and graphical procedures.

As a programming language, it provides a set of functions, commands and methods to instantiate and manage values of different type of objects (including lists, vectors and matrices), user interaction (input and output from console), control statements (conditional and loop statements), creation of functions, calls to external resources and create packages.

This chapter isn't accomplished to present details about R resources and will focus on the challenges to implement an algorithm using R or to be used in R. However, to better understanding the R power, some basic examples based in [28] are presented in Appendix A.

To implement an algorithm using R, normally it would be necessary to create complex objects to represent the data structures to be processed. Also, it would be necessary to implement complex functions to process this data structures. Thinking in the specific case of implementing the FP-Growth algorithm could be very hard to represent and process an FPTree using only the R resources. Moreover, for performance reasons it could be interesting to implement the algorithm using other languages and integrate it with R. Other reasons for using other languages are to get better memory management and to use existing packages[29].

Two ways to integrate R with other languages are available and will be briefly presented below: creating a package and making an external call using interface functions [24]. Next it is presented the FP-Growth implementation used in this work and the efforts to integrate it with R. For both would be necessary to install the RTools (http://www.murdoch-sutherland.com/Rtools/).

Creating a Package

Package is a mechanism for loading optional code implemented in other languages in R [24]. The R distribution itself includes about 25 packages, and some extra packages used in this WikiBook can be listed:

- aRules (http://r-forge.r-project.org/projects/arules/)
- arulesNBMiner (http://cran.fiocruz.br/web/packages/arulesNBMiner/index.html)
- arulesSequences (http://cran.r-project.org/web/packages/arulesSequences/index.html)
- cluster (http://cran.r-project.org/web/packages/cluster/index.html)

To create a package it's necessary to follow some specifications. The sources of an R package consist in a directory structure described below:

- **Root**: the root directory containing a DESCRIPTION file and some optional files (INDEX, NAMESPACE, configure, cleanup, LICENCE, COPYING and NEWS).
- **R**: contains only R code files that could be executed by the R command source(filename) to create R objects used by users. Alternatively, this directory can have a file sysdata.rda. This file has a saved image of R objects created in an execution of R console.
- **data**: aimed to have data files, either to be made available via lazy-loading or for loading using function data(). These data files could be from three different types: plain R code (.r or .R), tables (.tab, .txt, or .csv) or saved data from R console (.RData or .rda). Some additional compressed file can be used to table's files.
- **demo**: contains scripts in pain R code (for running using function demo()) that demonstrate some of the functionality of the package
- **exec**: could contain additional executables the package needs, typically scripts for interpreters such as the shell, Perl, or Tcl.
- **inst**: its content will be copied to the installation directory after it is built and its makefile can create files to be installed. May contain all information files that intended to be viewed by end users.
- **man**: should contain only documentation files for the objects in the package (using an specific R documentation format). An empty man directory causes an installation error.
- **po**: used for files related to internalization, in other words, to translate errors and warning messages.
- **src**: contains the sources, headers, makevars and makefiles. The supported languages are: C, C++, FORTRAN 77, Fortran 9x, Objective C and Objective C++. It's not possible to mix all these languages in a single package, but mix C and FORTRAN 77 or C and C++ seems to be

successful. However, there ways to make usage from other packages.
- **tests**: used for additional package-specific test code.

Once a source package is created, it must be installed by the command line in the OS console:

```
R CMD INSTALL <parameters>
```

Alternatively, packages can be downloaded and installed from within R, using the command line in the R console:

```
> install.packages(<parameters>)
```

See the Installation and Administration manual (http://cran.r-project.org/doc/manuals/R-admin.html) [27], for details.

After installed, the package needs to be loaded to be used, using the command line in the R console:

```
> library(packageName)
```

Making external call using interface functions

Making external call using interface functions is a simple way to use external implementation without complies with all rules described before to create a package to R.

First the code needs to include R.h header file that comes with R installation.

To compile a source code is needs to use the compiler R at the OS command line:

```
> R CMD SHLIB <parameters>
```

Compiled code to be used in R needs to be loaded as a shared object in Unix-like OS, or as a DLL in Windows OS. To load or unload it can be used the commands in the R console:

```
> dyn.load(fileName)
> dyn.unload(fileName)
```

After the load, the external code can be called using some of these functions:

- .C
- .Call
- .Fortran
- .External

Two simple examples are presented below, using .C function:

Example 1: Hello World

```
## C code in file example1.c

#include <R.h>

Void do_stuff ()
{
   printf("\nHello, I'm in a C code!\n");
}


## R code in file example.R

dyn.load("example1.dll")

doStuff <-
function (){
    tmp <- .C("do_stuff")
        rm(tmp)
        return(0)
}

doStuff()


## Compiling code in OS command line

C:\R\examples>R CMD SHLIB example1.c
gcc -I"C:/PROGRA~1/R/R-212~1.0/include" -O3 -Wall -std=gnu99 -c example1.c -o example1.o
gcc -shared -s -static-libgcc -o example1.dll tmp.def example1.o -LC:/PROGRA~1/R/R-212~1.0/bin/i386 -lR


## Output in R console

> source("example1.R")

Hello, I'm in a C code!
>
```

Example 2: Calling C with an integer vector [29]

```
##C code in file example2.c

#include <R.h>

void doStuff(int *i) {
    i[0] = 11;
}


## Output in R console

> dyn.load("example2.dll")
> a <- 1:10
> a
 [1]  1  2  3  4  5  6  7  8  9 10
> out <- .C("doStuff", b = as.integer(a))
> a
 [1]  1  2  3  4  5  6  7  8  9 10
> out$b
 [1] 11  2  3  4  5  6  7  8  9 10
```

The FP-Growth Implementation

The FP-Growth implementation used in this work was maded by Christian Borgelt (http://www.borgelt.net//fpgrowth.html) [30] a principal researcher at European Centre for Soft Computing. He also implemented the code used in arules package (https://r-forge.r-project.org/projects/arules/) [26] for Eclat and Apriori algorithms. The source code can be downloaded in his personal site.

As described by Borgelt [30] implemented two variants of the core operation of computing a projection of an FP-tree. In addition, projected FP-trees are optionally pruned by removing items that has becoming in-frequent (using FP-Bonsai [6] approach).

The source code is divided into three main folders (packages):

- **fpgrowth**: contais the main file that implements the algorithm and manages the FP-Tree;
- **tract**: manages item sets, transactions and its reports;
- **util**: facilities to be used in fpgrowth and tract.

The syntax to call this implementation, from the OS command line, is:

```
> fpgrowth [options] infile [outfile [selfile]]
-t#      target type                              (default: s)
         (s: frequent, c: closed, m: maximal item sets)
-m#      minimum number of items per item set     (default: 1)
-n#      maximum number of items per item set     (default: no limit)
-s#      minimum support of an item set           (default: 10%)
         (positive: percentage, negative: absolute number)
-e#      additional evaluation measure            (default: none)
-d#      minimum value of add. evaluation measure (default: 10%)
-g       write output in scanable form (quote certain characters)
-H#      record header  for output                (default: "")
-k#      item separator for output                (default: " ")
-v#      output format for item set information   (default: "  (%1S)")
-q#      sort items w.r.t. their frequency        (default: 2)
         (1: ascending, -1: descending, 0: do not sort,
          2: ascending, -2: descending w.r.t. transaction size sum)
-j       use quicksort to sort the transactions   (default: heapsort)
```

```
-a#      variant of the fpgrowth algorithm to use (default: simple)
-x       do not prune with perfect extensions
-z       do not use head union tail (hut) pruning
         (only for maximal item sets, option -tm)
-b#      blank   characters                       (default: " \t\r")
-f#      field   separators                       (default: " \t,")
-r#      record  separators                       (default: "\n")
-C#      comment characters                       (default: "#")
-!       print additional option information
infile   file to read transactions from           [required]
outfile  file to write frequent item sets to      [optional]
selfile  file stating a selection of items        [optional]
```

There are options to choose the limits of items per set, the minimum support, evaluation measure, to configure the input and output format, and so on.

A simple calling to FP-Growth, and its results, using the test1.tab example file (that comes with source code) as input file, the test1.out, and minimum support as 30%, could be made as follow:

```
C:\R\exec\fpgrowth\src>fpgrowth -s30 test1.tab test1.out
fpgrowth - find frequent item sets with the fpgrowth algorithm
version 4.10 (2010.10.27)        (c) 2004-2010   Christian Borgelt
reading test1.tab ... [5 item(s), 10 transaction(s)] done [0.00s].
filtering, sorting and recoding items ... [5 item(s)] done [0.00s].
reducing transactions ... [8/10 transaction(s)] done [0.00s].
writing test1.out ... [15 set(s)] done [0.00s].
```

The presented result shows some information about Copyright and some execution data, as the number of items and transactions and the number of frequent set (21 in this example). The content of input and output files is presented below.

The input file content:

```
a b c
a d e
b c d
a b c d
b c
a b d
d e
a b c d
c d e
a b c
```

The output file content:

```
e d  (30.0)
e  (30.0)
a c b  (40.0)
```

```
a c  (40.0)
a d b  (30.0)
a d  (40.0)
a b  (50.0)
a  (60.0)
d b c  (30.0)
d b  (40.0)
d c  (40.0)
d  (70.0)
c b  (60.0)
c  (70.0)
b  (70.0)
```

Calling FP-Growth from R

As observed before, to create a package are imposed a several rules creating a standard directory structure and content to make it available an external source code. An alternative presented before is to creating a shared object, or a DLL, to be called using specific R functions (.C, .CALL, and so on).

To start a job of adapt an existing code to compose a package can be a hard job and spending too much time. An interesting approach is to iteratively create and adapt a shared object, or DLL, and make tests to validate it and after improve the adaptations in some iterations, when a satisfactory result has been done, start to work in a package version.

The intent iterations to make it available the C implementation in R are:

1. Create a simple command line call, without parameters making only two changes in the original source (the fpgrowth.c file):
    - Rename the main function to FP-Growth with the same signature;
    - Create a function to be called from R, creating the parameters from a configuration file (containing only a string with the same syntax of the command line, broken it in an array to be used as the argument array to FP-Growth function;
2. Compile the code project within the R compile command, including the R.h reader file and call it using R;
3. Implement the input parameters from the R call, eliminating the usage of a configuration file, including the change to define a input file name to data-frames in R;
4. Preparing the output in a R data-frame to be returned to R;
5. Create the R package.

The first iteration could be done easily, without any surprise.

Unfortunately, the second iteration, that sounds to be ease to be done either, in a practice proved to be very hard. The R compile command does not work with makefiles and the compile original code with it could not be done. After some experiments, the strategy was changed to build a library with the adapted code, without the function created to be called from R, and then create a new code containing this function and making use of the compiled library. Next, calling the new code, compiled as a DLL, from R raises execution errors. Debugging the execution, wasting several time, was detected that some compile configurations to create the library was wrong. To solving this problem, some tests are made creating an executable version to be run using OS command line until all execution errors are solved. However, solved this errors, another unexpected behavior was founded. Calling the version compiled using R command from

R console the incompatible cygwin version error was rised in loading DLL function. Several experiments, changing the compilation parameters, different versions of cygwin, and so on were tried, but have no success (these tests are made only under Windows OS). So, having no success in the second iteration, the next step was compromised.

The main expected challenge in third and fourth iterations is to interface the R data types and structure with its correspondents in the C language, either to dataset input and other input parameters to be converted and used internally than to output dataset needed to be created to be returned to R. An alternative is to adapt all the code to use the data received. However, it sounds to be more complex to be done.

The fifth iteration sounds to be a bureaucratic work. Once the code has been entirely adapted and validated, create the additional directory and required content should be an easy task.

Conclusion and Future Works

In this chapter an efficient and scalable algorithm to mine frequent patterns in databases was presented: the FP-Growth. This algorithm uses a useful data structure, the FP-Tree, to store information about frequent patterns. Also an implementation of the algorithm was presented. Additionally, some features of R language and experiments to adapt the algorithm source code to be used in R. We could observe that the job to make this adaptation is hard, and cannot be done in short time. Unfortunately, have no time yet to conclude this adaptation.

As a future work would be interesting to better understand the implementation of external resources on R and complete the job proposed in this work, and after comparing results with other algorithms to mining frequent itemsets available in R.

Appendix A: Examples of R statements

Some basic examples based in [28].

Getting help about functions

```
> help(summary)
> ?summary
```

Creating an object

```
> perceptual <- 1.2
```

Numeric expressions

```
> z <- 5
> w <- z^2
> y <- (34 + 90) / 12.5
```

Printing an object value

```
> w
[1] 25
```

Creating a vector

```
> v <- c(4, 7, 23.5, 76.2, 80)
```

Vector operations

```
> x <- sqrt(v)
> v1<- c(4, 6, 87)
> v2 <- c(34, 32.4, 12)
> v1 + v2
[1] 38.0 38.4 99.0
```

Categhoric data

```
> s <- c("f", "m", "m", "f")
> s
[1] "f" "m" "m" "f"
> s <- factor(s)
> s
[1] f m m f
Levels: f m
> table(s)
s
f m
2 2
```

Sequences

```
> x <- 1:1000
> y <- 5:0
> z <- seq(-4, 1, 0.5) # create a sequence starting in -4, stopping in 1
                       # with an increment of 0.5
> w <- rnorm(10) # create a random sequence of 10 numeric values
> w <- rnorm(10, mean = 10, sd = 3) # create a normal distribution of
                                    # 10 numeric values with mean of 10
                                    # and standard deviation of 3
```

Matrices

```
> m1 <- matrix(c(45,  23,  66,  77,  33,  44),  2,  3)
> m1
        [,1]   [,2]   [,3]
[1,]     45     66     33
[2,]     23     77     44
> m2 <- matrix(c(12,  65,  32,  7,  4,  78),  2,  3)
> m2
        [,1]   [,2]   [,3]
[1,]     12     32      4
[2,]     65      7     78
> m1 + m2
        [,1]   [,2]   [,3]
[1,]     57     98     37
[2,]     88     84    122
```

Lists

```
> student <- list(nro = 34453, name = "Marie", scores = c(9.8, 5.7, 8.3))
> student[[1]]
[1] 34353
> student$nro
[1] 34353
```

Data Frames (represents database tables)

```
> scores.inform <- data.frame(nro = c(2355, 3456, 2334, 5456),
+ team = c("tp1", "tp1", "tp2", "tp3"),
+ score = c(10.3, 9.3, 14.2, 15))
> scores.inform
    nro    team   score
1  2355     tp1    10.3
2  3456     tp1     9.3
3  2334     tp2    14.2
4  5456     tp3    15.0
> scores.inform[score > 14,]
    nro    team   score
3  2334     tp2    14.2
4  5456     tp3    15.0
> team
```

```
[1]  tp1  tp1  tp2  tp3
Levels:  tp1  tp2  tp3
```

Conditional statement

```
> if (x > 0) y <- z / x else y <- z
> if (x > 0) {
+    cat('x is positive.\n')
+    y <- z / x
+} else {
+    cat('x isn't positive!\n')
+    y <- z
+}
```

Case statement

```
> sem <- "green"
> switch(sem, green = "continue", yellow = "attention", red = "stop")
[1] "continue"
```

Loop statements

```
> x <- rnorm(1)
> while (x < -0.3) {
+    cat("x=", x, "\t")
+    x <- rnorm(1)
+  }

> text  <-  c()
> repeat  {
+   cat('Type a phrase?  (empty to quit) ')
+   fr  <-  readLines(n=1)
+   if  (fr  ==  '')  break  else  texto  <-  c(texto,fr)
+}

> x <- rnorm(10)
> k <- 0
> for(v in x)  {
+   if(v > 0)
+     y <- v
+   else y <- 0
+     k <- k + y
+  }
```

Creating and calling functions

```
> cel2far <- function(cel) {
+   res  <-  9/5  *  cel  +  32
+   res
+ }
> cel2far(27.4)
[1]  81.32
> cel2far(c(0,  -34.2,  35.6,  43.2))
[1]  32.00  -29.56    96.08  109.76
```

References

1. J. Han, H. Pei, and Y. Yin. Mining Frequent Patterns without Candidate Generation. In: Proc. Conf. on the Management of Data (SIGMOD'00, Dallas, TX). ACM Press, New York, NY, USA 2000.
2. Agrawal, R. and Srikant, R. 1994. Fast algorithms for mining association rules. In Proc. 1994 Int. Conf. Very Large Data Bases (VLDB'94), Santiago, Chile, pp. 487–499.
3. Agarwal, R., Aggarwal, C., and Prasad, V.V.V. 2001. A tree projection algorithm for generation of frequent itemsets. Journal of Parallel and Distributed Computing, 61:350–371.
4. B.Santhosh Kumar and K.V.Rukmani. Implementation of Web Usage Mining Using APRIORI and FP Growth Algorithms. Int. J. of Advanced Networking and Applications, Volume: 01, Issue:06, Pages: 400-404 (2010).
5. Cornelia Gyorödi and Robert Gyorödi. A Comparative Study of Association Rules Mining Algorithms.
6. F. Bonchi and B. Goethals. FP-Bonsai: the Art of Growing and Pruning Small FP-trees. Proc. 8th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'04, Sydney, Australia), 155–160. Springer-Verlag, Heidelberg, Germany 2004.
7. M. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New Algorithms for Fast Discovery of Association Rules. Proc. 3rd Int. Conf. on Knowledge Discovery and Data Mining (KDD'97), 283–296. AAAI Press, Menlo Park, CA, USA 1997.
8. Christian Borgelt. Keeping Things Simple: Finding Frequent Item Sets by Recursive Elimination. Workshop Open Source Data Mining Software (OSDM'05, Chicago, IL), 66-70. ACM Press, New York, NY, USA 2005
9. Aiman Moyaid, Said and P.D.D., Dominic and Azween, Abdullah. A Comparative Study of FP-growth Variations. international journal of computer science and network security, 9 (5). pp. 266-272.
10. Liu,G. , Lu ,H. , Yu ,J. X., Wang, W., & Xiao, X.. AFOPT: An Efficient Implementation of Pattern Growth Approach, In Proc. IEEE ICDM'03 Workshop FIMI'03, 2003.
11. Grahne, G. , & Zhu, J. Fast Algorithm for frequent Itemset Mining Using FP-Trees. IEEE Transactions on Knowledge and Data Engineer,Vol.17,NO.10, 2005.
12. Gao, J. Realization of new Association Rule Mining Algorithm. Int. Conf. on Computational Intelligence and Security ,IEEE, 2007.
13. Cornelia Gyorödi, Robert Gyorödi, T. Cofeey & S. Holban. Mining association rules using Dynamic FP-trees. in Proceedings of The Irish Signal and Systems Conference, University of Limerick, Limerick, Ireland, 30th June 2nd July 2003, ISBN 0-9542973-1-8, pag. 76-82.
14. F. Bonchi, F. Giannotti, A. Mazzanti, and D. Pedreschi. Exante: Anticipated data reduction in constrained pattern mining. In Proc. of PKDD03.
15. Balázes Rácz. Nonordfp: An FP-Growth Variation without Rebuilding the FP-Tree. 2nd Int'l Workshop on Frequent Itemset Mining Implementations FIMI2004.
16. Grahne O. and Zhu J. Efficiently Using Prefix-trees in Mining Frequent Itemsets, In Proc. of the IEEE ICDM Workshop on Frequent Itemset Mining, 2004.
17. Jiawei Han and Micheline Kamber, Data Mining: Concepts and Techniques. 2nd edition, Morgan

Kaufmann, 2006.
18. M. Zaki and W. Meira Jr. Fundamentals of Data Mining Algorithms, Cambridge, 2010 (to be published)
19. Grahne, G. , & Zhu, J. Fast Algorithm for frequent Itemset Mining Using FP-Trees. IEEE Transactions on Knowledge and Data Engineer,Vol.17,NO.10, 2005.
20. Z. H. Deng and Z. Wang. A New Fast Vertical Method for Mining Frequent Patterns [1] (http://www.tandfonline.com/doi/abs/10.1080/18756891.2010.9727736). International Journal of Computational Intelligence Systems, 3(6): 733 - 744, 2010.
21. Z. H. Deng, Z. Wang，and J. Jiang. A New Algorithm for Fast Mining Frequent Itemsets Using N-Lists [2] (http://info.scichina.com:8084/sciFe/EN/abstract/abstract508369.shtml). SCIENCE CHINA Information Sciences, 55 (9): 2008 - 2030, 2012.
22. Z. H. Deng and S. L. Lv. Fast mining frequent itemsets using Nodesets [3] (http://www.sciencedirect.com/science/article/pii/S0957417414000463). Expert Systems with Applications, 41(10): 4505–4512, 2014.
23. W. N. Venables, D. M. Smith and the R Development Core Team. An Introduction to R: Notes on R: A Programming Environment for Data Analysis and Graphics. Version 2.11.1 (2010-05-31).
24. R Development Core Team. R Language Definition. Version 2.12.0 (2010-10-15) DRAFT.
25. R Development Core Team. Writing R Extensions. Version 2.12.0 (2010-10-15).
26. Michael Hahsler and Bettina G and Kurt Hornik and Christian Buchta. Introduction to arules – A computational environment for mining association rules and frequent item sets. March 2010.
27. R Development Core Team. R Installation and Administration. Version 2.12.0 (2010-10-15).
28. Luís Torgo. Introdução à Programação em R. Faculdade de Economia, Universidade do Porto, Outubro de 2006.
29. Sigal Blay. Calling C code from R an introduction. Dept. of Statistics and Actuarial Science Simon Fraser University, October 2004.
30. Christian Borgelt. An Implementation of the FP-growth Algorithm. Workshop Open Source Data Mining Software (OSDM'05, Chicago, IL), 1-5. ACM Press, New York, NY, USA 2005.