Using the *cherry* R package

Jelle Goeman Aldo Solari Rosa Meijer

Package version 0.5-10
Date 2014-04-24

Contents

1 Citing *cherry* **2**

2 Introduction **2**
2.1 Exploratory inference 2
2.2 Intersection hypotheses and local tests 3

3 Methods based on p-values **3**
3.1 Fisher combinations 4
3.2 The Simes inequality 6

4 The general method **7**
4.1 Defining a local test 7
4.2 Performing closed testing 8
4.3 Defining rejections and the shortlist 10
4.4 Adjusted p-values 11

5 Region procedure **13**
5.1 Using the region procedure 13
5.2 Confidence sets for the number of false hypotheses in a given set . . . 17

6 DAG multiple testing procedure **17**
6.1 Using the DAG procedure 18
6.2 Confidence sets for the number of false hypotheses in a given set . . . 20

1 Citing *cherry*

If you use the *cherry* package, please cite the paper J. J. Goeman and A. Solari, Multiple testing for exploratory research, *Statistical Science*, 26 (4) 584–597.

2 Introduction

The *cherry* package is a package for multiple hypothesis testing. The approach used by *cherry* is specially designed for exploratory inference.

2.1 Exploratory inference

Suppose a researcher has performed an experiment, possibly a genomics experiment or some other experiment in which a large or small number of statistical hypotheses have been tested. From the results of the experiment the user wants to select a number of 'promising' results. Which results are considered promising may depend on any mixture of considerations, such as (unadjusted) significance, effect size, and domain knowledge. One question the researcher may ask is how many false positive findings are present in the selected list. This is the question the *cherry* package is designed to answer.

The suggested way of working with this package is as follows. Before the data are gathered, two choices have to be made. The first choice is what hypotheses are of potential interest. This is the working collection of hypotheses. The second choice is what statistical test is to be done for each hypothesis, and for each intersection (or combination) of hypotheses. Examples of such choices are given in the rest of this vignette. Nothing else has to be decided before data collection. After data collection, the researcher can study the data as much as he or she likes, before deciding on a collection of rejected hypotheses of interest. This choice, together with the working collection and the tests is fed into the *cherry* package, which will return a confidence statement on the maximum number of true null hypotheses, i.e. false rejections, among the selected set. On the basis of this assessment, the researcher may reevaluate and come up with a different selection of interesting hypotheses, for which *cherry* will again give a confidence statement. These confidence statements are not compromised by previous looks at the data, but remain valid however many times the researcher comes up with a new set.

In *cherry* the traditional roles of user and algorithm in multiple testing have been reversed. In classical multiple testing procedures the user's task is to set an error rate to be controlled, and the task of the multiple testing procedure is to decide what hypotheses to reject. In *cherry*, the user chooses what hypotheses to reject, and the multiple testing procedure calculates the error rate. This way of working allows the user much more freedom and control. Most importantly, the error rates that are calculated are not invalidated by multiple looks at the data, and the user is free to study the data in every possible way before finally deciding what rejections to make. This makes the method ideally suited for exploratory research.

Throughout the package and the vignette, the words "true hypothesis" and "false rejection" are used interchangeably, as rejection of a true hypothesis amounts to a false rejection. Similarly, "false hypothesis" and "correct rejection" often used as synonyms.

The theory behind the methods is explained in detail in the papers Goeman and Solari (2011a) and Goeman and Solari (2011b) and we refer readers to these papers for

details. In this document, we present some worked out examples to demonstrate how *cherry* can be used and how its results should be interpreted. We start in Section 3 with methods based on p-values that use either Fisher combinations or Simes inequality to combine p-values. These methods have the advantage that they are quick to use even if tens of thousands of hypotheses have been tested, but they do depend on assumptions and cannot always be used. Next, Section 4 presents the general method in its full flexibility. This general method is computationally quite intensive and should only be used if the total number of hypotheses in the multiple testing problem is not much greater than 20. More methods of different types will be added to the *cherry* package in the future.

For an explanation of the theory behind the methods, and for more explanation and examples, see the papers Goeman and Solari (2011a) and Goeman and Solari (2011b).

2.2 Intersection hypotheses and local tests

The *cherry* package assumes that before data collection the user is able to make a complete list of all hypotheses that are of potential interest. For each of these hypotheses, a statistical hypothesis test must be formulated. Moreover, statistical tests must also be formulated for all possible intersections of these chosen hypotheses. An intersection hypothesis of a collection of hypotheses is a hypothesis that is true if and only if every hypothesis in the collection of hypotheses is true. For example, if null hypothesis H_A asserts that the mean treatment effect of drug A is zero, and the hypothesis H_B asserts that the mean treatment effect of drug B is zero, then the intersection hypothesis $H_{AB} = H_A \cap H_B$ asserts that the treatment effects of both drugs are zero. A statistical test for an intersection hypothesis is known as a *global test* or a *local test*. Examples of frequently used global and local tests are F-tests in ANOVA models or regression models, or gene set tests in microarray data analysis.

The user of the *cherry* package is free to choose any local test that is valid for the data at hand. The methods in Section 4 allow the user to work with any self-defined local test. Unfortunately, these methods are computationally very expensive, and should only be used when the total number of hypotheses in the multiple testing problem is not much greater than 20. For specific choices of the local test, that use either Fisher combinations or Simes inequality, quicker algorithms are available. Special functions for those local tests are describe in Section 3

3 Methods based on p-values

In this section we present some simple methods that can be used when each hypothesis in the working collection has been tested and given a p-value, and when intersection hypotheses are tested with simple p-value-based global tests such as Simes inequality or Fisher combinations. We will illustrate these methods using a data set NAEP taken from Benjamini and Hochberg (2000)

```
> library(cherry)
> data(NAEP)
```

These are p-values for 34 null hypotheses for 34 American states. In each state, a test was performed for the hypothesis that there was no change in the average eighth-grade mathematics achievement scores between 1990 and 1992. We can assume the

data for different states, and therefore the p-values, to be independent. The p-values are sorted here, but that is not necessary for *cherry*.

The fact that we have p-values means that the choice of a statistical test for the individual hypotheses has already been made. The only remaining choice is the test for intersection hypotheses. Two options to test these intersection hypotheses are presented in this section.

3.1 Fisher combinations

Fisher combinations are based on the fact that if a p-value p_i is under the null hypothesis, it is uniformly distributed (or stochastically smaller than that). Consequently, $-\log(p_i)$ is exponentially distributed with parameter $\lambda = 1$, and $-2\log(p_i)$ is χ^2 distributed with 2 degrees of freedom. If a number of r p-values is independent, then $-2\sum_{i=1}^{r}\log(p_i)$ is χ^2 distributed with $2r$ degrees of freedom. This suggests the Fisher combination test, a test for intersection hypotheses based on negative sums of logarithms of p-values that uses critical values from a χ^2-distribution. This test is valid only when p-values for hypotheses that are under the null are independent. Fisher combination tests are powerful for detecting the presence of many small effects, but not so powerful for detecting few larger ones. Functions in *cherry* that use this test are the `pickFisher` and `curveFisher` functions.

Suppose the researcher has chosen Fisher combinations and, after looking at the NAEP data, picks the hypotheses HI, MN and IA, and wants to know how many correct rejections he or she would make when rejecting these null hypotheses. This can be found with

```
> pickFisher(NAEP, c("HI","MN","IA"))

Rejected 3 hypotheses at confidence level 0.95.
Correct rejections >= 2; False rejections <= 1.
```

The hypotheses are referred to by name in this function call, but they can be referred to by any other selection method method, such as a logical vector, index or negative index. We can conclude at the default 95% confidence that among the hypotheses HI, MN and IA there are at least 2 false hypotheses and at most one true one.

Leaving out the second argument, *select*, means rejecting all 34 hypotheses. This gives us an upper confidence bound to the number of true hypotheses in the complete working collection

```
> pickFisher(NAEP)

Rejected 34 hypotheses at confidence level 0.95.
Correct rejections >= 19; False rejections <= 15.
```

There are at least 19 correct rejections, i.e. false null hypotheses, and at most 15 false rejections, i.e. true null hypotheses among the 34 hypotheses. The 95% confidence set for the number of true null hypotheses among the 34 goes from 0 (lower bound) to 15 (upper bound), and consequently the same confidence set for the number of false hypothese from 19 to 34. For the selected set HI, MN and IA, the 95% confidence set for the number of true hypotheses goes from 0 to 1. It is important to know (and central to the method underlying the *cherry* package) that all these confidence intervals are simultaneous. There is no need to correct for multiple testing when selecting the most interesting one from all these confidence intervals.

Several options can be set in `pickFisher`. Setting the type I error rate *alpha* (default 0.05) changes the confidence level of the statements made. Setting *silent* to `TRUE` suppresses printing to the screen of the result. The `pickFisher` function's return value is the lower bound of the number of false null hypotheses, i.e. correct rejections.

```
> res <- pickFisher(NAEP, silent=TRUE)
> res

[1] 19
```

The `curveFisher` function can give some additional information over `pickFisher`. Called without further arguments, the function returns lower bounds for the number of false null hypotheses, like `pickFisher`, but simultaneously for selecting the hypotheses with the smallest 1,2,3,..., p-values. The results are displayed in a graph unless the *plot* argument is set to `FALSE`. From these results, we see with 95% confidence that 19 false null hypotheses are are present among all 34 hypotheses, as we saw before, but also that these 19 false null hypotheses must be among the 25 hypotheses with smallest p-values.

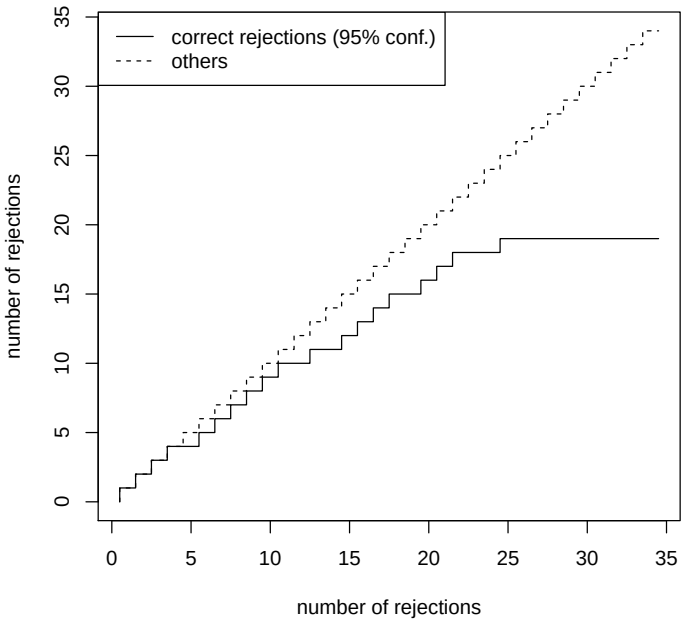
```
> res <- curveFisher(NAEP)
> res

RI NC HI MN NH IA CO TX ID AZ KY OK CT NM WY FL PA NY OH CA
 1  2  3  4  4  5  6  7  8  9 10 10 11 11 12 13 14 15 15 16
MD WV VA WI IN LA MI DE ND NE NJ AL AR GA
17 18 18 18 19 19 19 19 19 19 19 19 19 19
```

The curveFisher function may be further tailored. The *select* argument can be used to consider only a subset of the hypotheses, and the function will return the number of false null hypotheses among the 1,2,... smallest p-values in the selected set. Alternatively, the *order* argument can be used to set the order in which hypotheses should be rejected, rather than taking the order of increasing p-values. Compare

```
> curveFisher(NAEP, select=c(8,3,4,2), plot=FALSE)

NC HI MN TX
 1  2  3  3

> curveFisher(NAEP, order=c(8,3,4,2), plot=FALSE)

TX HI MN NC
 0  1  2  3
```

Here, the first call uses the rejection order NC, HI, MN, TX, determined by the p-values; the second uses the rejection order TX, HI, MN, NC, given by the input. We interpret the second result as follows. Choosing TX only we have a result of 0, which means that we do not have 95% confidence that TX corresponds to a false null hypothesis. The second result is 1, which means that choosing both TX and HI, we have detected at least one false null hypothesis with 95% confidence. The third result, 2, refers too rejection of TX, HI and MN, the fourth result to the collection of all four hypotheses.

```
> curveFisher(NAEP)
```



3.2 The Simes inequality

The Simes inequality says that for a sequence of ordered p-values $p_{(1)}, \ldots, p_{(r)}$, under the null hypothesis we have $p_{(i)} \geq i\alpha/r$ simultaneously for all i with probability at least $1 - \alpha$. This inequality holds if p-values are independent, as shown by Simes, but also under some forms of positive correlation, as shown by Sarkar and Chang (1997). Simes' inequality suggests Simes' test, a test that rejects an intersection hypothesis of r hypotheses if either the smallest p-value is smaller than α/r or the second smallest is smaller than $2\alpha/r$, or ..., or the largest p-value is smaller than $r\alpha/r = \alpha$. This test can be used as a local test in the *cherry* package. It is valid if p-values of true null hypotheses are always either independent or positively correlated. An alternative, more conservative test was formulated by Hommel; this test is valid whatever the distribution of the p-values.

The functions `pickSimes` and `curveSimes` work in exactly the same way as `pickFisher` and `curveFisher` above, and they use the same options and arguments.

```
> pickSimes(NAEP, c("HI","MN","IA"))

Rejected 3 hypotheses. At confidence level 0.95:
Correct rejections >= 2; False rejections <= 1.

> curveSimes(NAEP, plot=FALSE)

RI NC HI MN NH IA CO TX ID AZ KY OK CT NM WY FL PA NY OH CA
 1  2  3  4  4  5  6  6  6  6  6  6  6  6  6  6  6  6  6
```

```
MD WV VA WI IN LA MI DE ND NE NJ AL AR GA
 6  6  6  6  6  6  6  6  6  6  6  6  6  6
```

Comparing these results with the ones obtained for the Fisher combinations above, we see that the Simes test allows fewer rejections. This is not generally true, and Simes may be more powerful in other data sets. In general, Fisher combinations can be said to have more power if there are many small effect sizes, whereas Simes has more power in the presence of a few stronger effects.

The more conservative Hommel variant that makes no assumptions on the p-value distribution can be obtained by setting *hommel=TRUE*. This variant is more conservative than one based on the regular Simes test.

```
> pickSimes(NAEP, c("HI","MN","IA"), hommel=TRUE)

Rejected 3 hypotheses. At confidence level 0.95:
Correct rejections >= 2; False rejections <= 1.

> curveSimes(NAEP, plot=FALSE, hommel=TRUE)

RI NC HI MN NH IA CO TX ID AZ KY OK CT NM WY FL PA NY OH CA
 1  2  3  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4
MD WV VA WI IN LA MI DE ND NE NJ AL AR GA
 4  4  4  4  4  4  4  4  4  4  4  4  4  4
```

4 The general method

There are many more possibilities than Fisher combinations or Simes inequality to make local tests. The `closed` function and its derivatives in *cherry* allow users to work with any type of local test. When working with such a user-defined local test, the output possibilities are greater than with the simple `pickFisher` and `curveFisher` functions above.

4.1 Defining a local test

We first illustrate the general method with the same local test as was used there: the Fisher combinations.

To illustrate these data we cannot take the NAEP data, as for 34 hypotheses the calculations will take too long. In fact, the code will not run for a collection of more than 31 hypotheses. We shall illustrate the general approach with a very small data set of 4 p-values, taken from Huang and Hsu (2007), but we take them out of the context they were presented in in that paper.

```
> ps <- c(A = 0.051, B = 0.064, C = 0.097, D = 0.108)
```

To define a local test, we must create an `R` function, such as the following

```
> mytest <- function(hypotheses) {
+   p.vals <- ps[hypotheses]
+   m <- length(p.vals)
+   statistic <- -2 * sum(log(p.vals))
+   p.out <- pchisq(statistic, df=2*m, lower.tail=FALSE)
+   return(p.out)
+ }
```

This function takes as input the names of the hypotheses that the intersection hypothesis is an intersection of, and returns a p-value, as calculated by the Fisher combinations test. Note that the function references the `ps` data we've just created. We can now call our test with

```
> mytest("A")

[1] 0.051

> mytest(c("B","C","D"))

[1] 0.02347135

> mytest(names(ps))

[1] 0.008391265
```

Note that calling `mytest` on a single hypothesis name just returns the p-value of that hypothesis.

A user can define their own favorite local test in exactly this way, by creating an R function that takes a vector of hypothesis names as input and gives a valid p-value for the corresponding intersection null hypothesis as output.

The `mytest` function used p-values as input data. This is not necessary or even typical. As a second example, we present a function for a local test based on the F-test in a multiple linear regression using the `LifeCycleSavings` example data. The null hypothesis of this test is the hypothesis that the covariates in its *hyps* argument all have regression coefficient zero in the multiple regression model fitted in `fullfit` below.

```
> hypotheses <- c("pop15", "pop75", "dpi", "ddpi")
> fullfit <- lm(sr~., data=LifeCycleSavings)
> myFtest <- function(hyps) {
+   others <- setdiff(hypotheses, hyps)
+   form <- formula(paste(c("sr~",  paste(c("1", others), collapse="+"))))
+   anov <- anova(lm(form, data=LifeCycleSavings), fullfit, test="F")
+   pvalue <- anov$"Pr("[2]        # NB replace Pr by P for for R < 2.14.0
+   return(pvalue)
+ }
> myFtest(c("pop15", "pop75"))

[1] 0.004834923
```

4.2 Performing closed testing

Next, we can perform the closed testing procedure using this definition of the local test. This is done using the function `closed`, as follows

```
> ct <- closed(mytest, names(ps))
```

Note that there is no need anymore to specify the data set that is used, because the reference to the data set is contained in the definition of `mytest`. The calculations can take a large amount of time, especially if the number of hypotheses is large.

By default, all tests are performed at level *alpha = 0.05* and only rejection or acceptance of hypotheses is stored, not adjusted p-values. It is possible to change the *alpha* argument to a different value, or to do the calculations using adjusted p-values so that the choice of the significance level may be postponed. This will be explained below in Section 4.4.

Using the `ct` object created by the call to `closed` we can now perform analysis by asking for the number of true and false hypotheses among certain sets of hypotheses of interest. Just displaying the object,

```
> ct

Closed testing result on 4 elementary hypotheses.
At confidence level 0.95: False hypotheses >= 2; True hypotheses <= 2.
```

gives a lower confidence bound on the number of false hypotheses among the collection of all four tested hypotheses. We conclude that there are at least two false null hypotheses among the four tested ones. If we are interested in a subset of hypotheses, we can use the `pick` function.

```
> pick(ct, c("A", "B"))

Rejected 2 hypotheses.
At confidence level 0.95: Correct rejections >= 1; False rejections <= 1.

> pick(ct, c("C", "D"))

Rejected 2 hypotheses.
At confidence level 0.95: Correct rejections >= 0; False rejections <= 2.
```

This gives us confidence limits for the number of false hypotheses in each chosen subset. The `pick` function returns the lower bound on the number of false hypotheses. It also displays the information on the screen, but this can be switched off if desired by setting the *silent* argument to `TRUE`. If the second argument to `pick` is left out, it is assumed that the set of all hypotheses is meant. To get back the names of the hypotheses, type

```
> hypotheses(ct)

[1] "A" "B" "C" "D"
```

From the results of `pick` above we see that there is evidence for one false hypothesis among A and B, but no such evidence among C and D. This seems to contradict the earlier statement that there was evidence for at least two false null hypotheses among the total set of A, B, C and D. However, this is only an apparent contradiction: the amount of evidence for a second false null hypothesis is not sufficient in the observed data of A and B alone, nor in the data of C and D alone, but the combined evidence of the data from all four hypotheses is sufficient.

Just like `pickSimes` and `pickFisher`, `pick` returns the lower confidence bound for the number of false null hypotheses as a number.

```
> res <- pick(ct, c("C", "D"), silent=TRUE)
> res

[1] 0
```

4.3 Defining rejections and the shortlist

The `pick` function allows users to check out any desired set of hypotheses, but gives no guidance as to what sets of hypotheses to probe. Two other functions can help to see structure in the results of the closed testing procedure.

The first of these is the `defining` function, which calculates the *defining rejections*. The defining rejections are a collection of sets of hypotheses with the property that for each set in the collection we can be confident that it contains at least one false null hypothesis. The collection of defining hypotheses is minimal in the sense that there are no smaller sets for which the same statement holds. In our `ct` object, the defining rejections are

```
> defining(ct)

[[1]]
[1] "A" "B"

[[2]]
[1] "A" "C"

[[3]]
[1] "B" "C"

[[4]]
[1] "A" "D"

[[5]]
[1] "B" "D"
```

For each of the listed defining sets, we can conclude that they contain at least one false hypothesis. For example, at least one of A and B must be false, but also at least one of A and C, and at least one of B and C. If any of the defining sets is a singleton, we can confidently conclude that the corresponding hypothesis is a false one.

The dual of the defining sets is the *shortlist*, introduced by Meinshausen (2011). Just like the defining sets, the shortlist is a collection of sets of hypotheses, but for the shortlist collection we can make the statement that at least one of the sets in the collection contains only false hypotheses. In the example,

```
> shortlist(ct)

[[1]]
[1] "A" "B"

[[2]]
[1] "B" "C" "D"

[[3]]
[1] "A" "C" "D"
```

the shortlist contains three sets. We conclude that either both hypotheses A and B are false, or all three hypotheses A and C and D or all three hypotheses B and C and D. Just as with the defining rejections, the shortlist only gives a minimum at the chosen

significance level. The possibility that all four hypotheses are false, for example, is equally compatible with the results of the closed testing procedure. However, the true set must contain at least one of the shortlist sets completely.

4.4 Adjusted p-values

Earlier calculations were all done at a significance level *alpha* of 0.05, and the results only used the rejection status of hypotheses. There is additional information is the p-values, however, and the `closed` function may also be used with adjusted p-values rather than a hard rejection yes/no. By definition, an adjusted p-value is the smallest alpha-level at which a certain hypothesis can be rejected in the multiple testing procedure. Using adjusted p-values is comparable to not setting the alpha level in advance, but simultaneously doing the same test procedure at all alpha levels. To use adjusted p-values, use

```
> cta <- closed(mytest, names(ps), alpha = NA)
```

Calculation times based on adjusted p-values can be substantially longer because more tests need to be calculated. If the user is not interested in adjusted p-values above a certain threshold, say 0.1, an alternative call is

```
> ctb <- closed(mytest, names(ps), alpha = 0.1, adjust = TRUE)
```

In this case, all adjusted p-values greater than the chosen threshold will be set to 1.

The `pick` function works slightly differently if adjusted p-values were used. It simultaneously presents results for all levels of alpha.

```
> pick(cta)
```

```
         alpha confidence true<= false>=
1 0.008391265  0.9916087      3       1
2 0.023471346  0.9765287      2       2
3 0.058232610  0.9417674      1       3
4 0.108000000  0.8920000      0       4
```

The results should be read as follows. Up to $\alpha = 0.0084$ we only have the trivial result that the number of true hypotheses is ≤ 4. From $\alpha = 0.0084$ to $\alpha = 0.023$ we get the result that at least one hypothesis is false; from $\alpha = 0.023$ to $\alpha = 0.058$, we get at least false two hypotheses, etcetera. In terms of adjusted p-values, the adjusted p-value for the statement that there are ate least two false null hypotheses among the four can be read off as 0.02347. The confidence column is simply $1 - \alpha$.

To extract adjusted p-values directly, there is the `adjusted` function. Calling `adjusted` on a set of hypotheses without extra arguments returns the adjusted p-value of the corresponding intersection hypothesis. An additional third argument *n* can be given to get the adjusted p-value for making the statement that at least $n + 1$ false null hypotheses occur in the chosen set, i.e. corresponding to the null hypothesis that at most n false null hypotheses are present.

```
> adjusted(cta, c("A", "B", "C"))
```

```
[1] 0.01314629
```

```
> adjusted(cta, c("A", "B", "C"), n=1)

[1] 0.03775654
```

We conclude that the compound hypothesis that no hypothesis among A, B and C is false, and the compound hypothesis that at most one among A, B, and C is false are both rejected at $\alpha = 0.05$, with adjusted p-values 0.013 and 0.038, respectively.

The `pick` function has one other additional feature in that the number in the table can be visualized in a plot. Setting *plot = TRUE* in `pick` results in a plot such as in Figure 1. In this plot, the adjusted p-values can be read off as tail probabilities in the plotted "distribution". These values are displayed at the top of the plot. The *plot* argument is ignored if adjusted p-values were not calculated.

```
> pick(cta, names(ps), plot=TRUE)
```

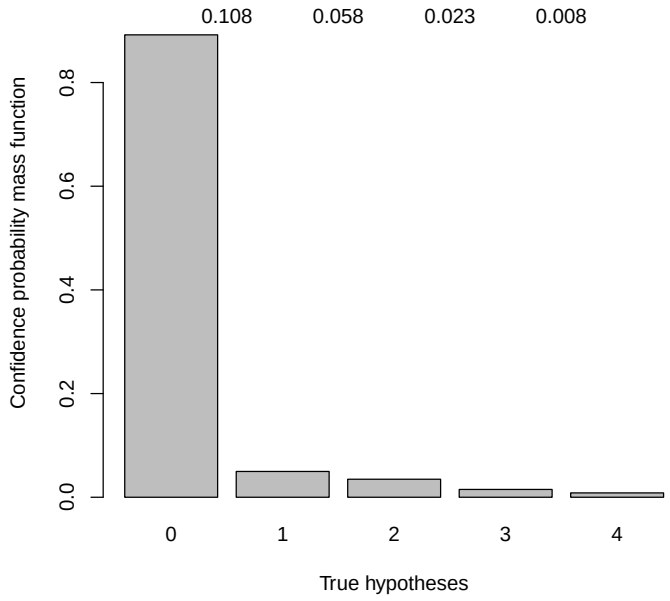


Figure 1: `pick(cta, names(ps), plot=TRUE)`

The `defining` and `shortlist` functions can be used for objects with adjusted p-values, but a specific value of alpha must be specified.

```
> shortlist(cta, alpha=0.05)

[[1]]
[1] "A" "B"

[[2]]
[1] "B" "C" "D"
```

```
[[3]]
[1] "A" "C" "D"
```

It is also possible to set the alpha level to be used in such cases in advance with the `alpha` function. Setting

```
> alpha(cta) <- 0.05
```

will cause functions such as `pick`, `defining`, `shortlist` to work as if alpha = 0.05 was set in advance. The object can be reset to adjusted p-values with

```
> alpha(cta) <- NA
```

5 Region procedure

In addition to the closed testing procedure, there is another multiple testing method implemented in the *cherry* package. The region procedure is a a multiple testing method for hypotheses that are ordered in space or time. Given such hypotheses, the elementary hypotheses as well as regions of consecutive hypotheses are of interest. Each region hypothesis will be denoted by H_{ij}, where i and j specify the left and right boundary of the corresponding region. The region procedure is a procedure that starts with testing the global null-hypothesis and when this hypothesis can be rejected it continues with further specifying the exact location/locations of the effect present. In every step of the procedure, only those hypotheses H_{ij} are tested for which all H_{kl} with $k \leq i \leq j \leq l$ are already rejected.

5.1 Using the region procedure

To perform the region procedure on a given dataset, we can use the function `regionmethod`. What is needed for the function to work is a local test that has a left and rightbound as input and returns a p-value for the corresponding region hypothesis, and a weight vector that indicates the weight each elementary hypothesis should receive. The length of this vector should equal the number of elementary hypotheses.

To illustrate the method, we first generate some data. Let us take a response vector Y and a matrix X of covariates that is generated in such a way that Y is associated with certain regions of the covariates.

```
> set.seed(1)
> n=100
> p=20
> X <- matrix(rnorm(n*p),n,p)
> beta <- c(rep(0,2),rep(1,4),rep(0,2),rep(1,4),rep(0,2),
+           rep(1,4),rep(0,2))
> Y <- X %*% beta + rnorm(n)
```

Y is a vector of length 100, whereas X is a matrix of dimension 100×20. Of the 20 covariates, only 12 covariates are associated with the response. They are organized in 3 groups of 4 covariates each. The intervals (3,6), (9,12) and (15,18) are associated with the response.

Furthermore, we have to specify a local test. We choose to work with an F-test, which can be defined in the following way:

```
> mytest <- function(left,right)
+ {
+   X <- X[,(left:right),drop=FALSE]
+   lm.out <- lm(Y ~ X)
+   x <- summary(lm.out)
+   return(pf(x$fstatistic[1],x$fstatistic[2],x$fstatistic[3],
+             lower.tail=FALSE))
+ }
```

The region procedure can now be used on the design matrix X to find which (regions of) covariates are associated with the response by using the following function call:

```
> reg <- regionmethod(rep(1,p), mytest, all_pvalues=TRUE, isadjusted=TRUE)
> summary(reg)

Region method result on 20 elementary hypotheses.
There are 179 region hypotheses rejected out of a total of 210 at an alpha-leve
```

Using the function `regionmethod` results in a `region` object. From this `region` object we can retrieve different information. By using the `show` or `summary` function, we can see how many of the region hypotheses could be rejected on the given α-level as specified by the *alpha_max* argument, which is 0.05 by default. In our example, we see that 179 region hypotheses could be rejected (out of a total number of 210). All these regions can be summarized by the regions that could be rejected themselves, but for which all smaller regions they contain could not get rejected. We will call those smallest rejectable region "implications".

To get the implications from a `region` object, we can use the `implications`. This function requires a `region` object as argument, and has an optional argument *alpha*. If *alpha* is not specified, the implications corresponding to the argument *alpha_max* of the `region` object are calculated. In our example, the implications look like this:

```
> implications(reg)

     left right  adj. pvalue
[1,]    4     4 0.0000931957
[2,]    5     5 0.0470085586
[3,]    9     9 0.0069271227
[4,]   10    10 0.0390512731
[5,]   12    15 0.0494077361
[6,]   16    16 0.0106803872
[7,]   17    17 0.0021602152
```

Of these 7 implications, 6 correspond to an elementary hypothesis. Only the fifth implication corresponds to a region hypothesis with length larger than 1, namely to the region hypothesis of region (12,15). If we would want to have the implications corresponding to a different α-level, we could either use the `regionmethod` function again and use a different value for *alpha_max*, or we could use our current `region` object and change the argument *alpha* in the `implications` function. However, the second approach will only work if the value of *alpha* is smaller than the value

alpha_max that was used to make the `region` method. Furthermore, the *all_pvalues* as well as the *isadjusted* argument in the call to the `regionmethod` function that was used to create the `region` object has to be set to `TRUE`. Only then, new implications can be computed. In our example, we set *all_pvalues*=`TRUE`, and we used the default value of 0.05 for *alpha_max*, so we can use the `implications` with *alpha* set to all values smaller than 0.05. For example, if we want to know what implications we would have had if we controlled the FWER on 0.01, we can do this:

```
> implications(reg, alpha=0.01)
```

```
     left right  adj. pvalue
[1,]    4     4 0.0000931957
[2,]    9     9 0.0069271227
[3,]   10    11 0.0065161207
[4,]   15    16 0.0046725078
[5,]   17    17 0.0021602152
```

On this lower α-level, only 3 elementary hypotheses could be rejected.

We see that implications come with a p-value, that can either be the true adjusted p-values (if the *isadjusted* argument of the `regionmethod` was set to `TRUE`) or just the α-level on which the FWER is controlled. To get p-values of rejected regions that do not correspond to an implication, we can use the `pvalue` method that is available for `region` objects. To be able to use this function, we again need to set the *all_pvalues* argument in the `regionmethod` function to `TRUE`. In our example, we used the right parameter settings, and we can thus ask for any adjusted p-value. We could for example get the adjusted p-value of the overall null-hypothesis, corresponding to region (1,20) as follows:

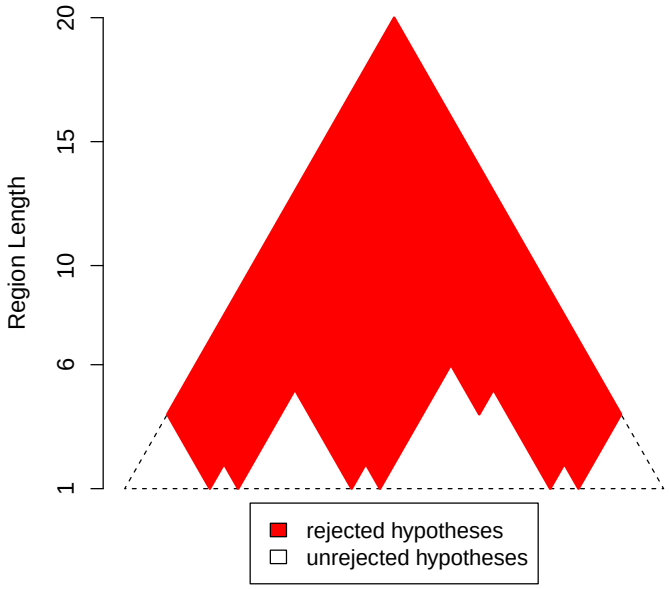
```
> pvalue(reg,1,20)
```

```
[1] 1.75461e-40
```

Instead of summarizing the results of the region procedure by looking at the implications, we can also visualize our results by representing the region hypotheses by a graph and drawing this graph. There are two functions that can be used in order to do so, namely `regionplot` and `regionplot2`. The first function does not display the nodes and edges of the region graph separately, but draws a polygon that follows the original graph structure. The `regionplot2` function visualizes the full graph with its nodes and edges. Although the information given by the list of all mplications and the plots are identical, at first sight the implications are more precise, but the plots give an immediate idea of which regions are interesting.

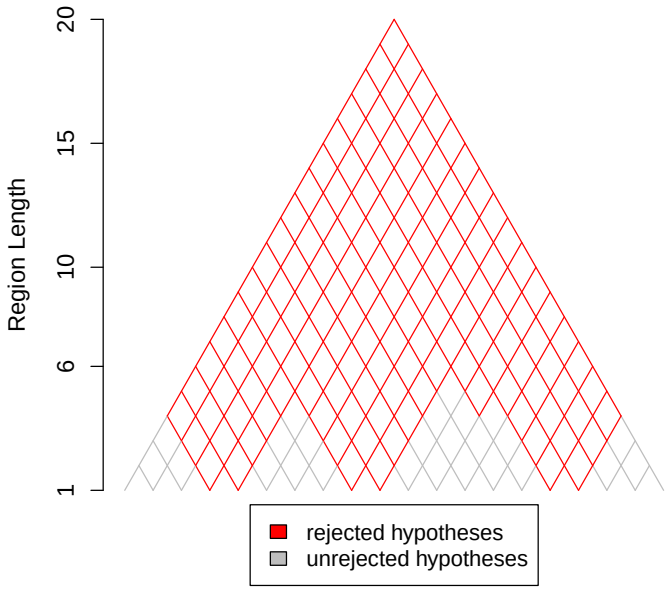
For our example, the plots can be made in the following way:

```
> regionplot(reg)
```



and

```
> regionplot2(reg)
```



The plot functions do also have an optional *alpha* argument, that can be used in the same way as described for the `implications` function.

5.2 Confidence sets for the number of false hypotheses in a given set

Just as in the case of a closed testing procedure, while using the region procedure we are able to make simultaneous $100(1-\alpha)\%$ confidence sets for the number of false (or true) rejection within a given set S of elementary hypotheses. Given a `region` object, we can calculate the minimal number of false elementary hypotheses (i.e. true findings) in any given set of elementary hypotheses. This set has to be specified by a list of regions. For example, if we want to know how many false hypotheses there should at least be in the set of the first and last 5 elementary hypotheses from our 20 elementary hypotheses, we can specify a list as follows: list(c(1,5),c(16,20)) and use the function `regionpick` to calculate the lower confidence bound for the number of false hypotheses in this set:

```
> regionpick(reg, list(c(1,5),c(16,20)))

At confidence level 0.95: False null-hypotheses (i.e. true findings) >= 4
```

We can also look at just one set, for example the total set of elementary hypotheses:

```
> regionpick(reg, list(c(1,20)))

At confidence level 0.95: False null-hypotheses (i.e. true findings) >= 7
```

We see that at least 7 out of our 20 elementary hypotheses have to be false with probability $(1-\alpha)$, where α is still the α on which the FWER is controlled in the region procedure, which is 0.05 in our example. This is in agreement with the earlier shown implications, which showed that 6 elementary hypothesis were rejected and one larger region, in which at least one of the elementary hypotheses had to be false as well.

Again, there is an optional argument *alpha* that can be used in the same way as in the other functions.

6 DAG multiple testing procedure

In addition to the closed testing and region procedure, there is another multiple testing method implemented in the *cherry* package. The DAG procedure is a multiple testing procedure that tests hypotheses that are structured in a Directed Acyclic Graph (DAG). Each node in the DAG will represent an hypothesis, whereas the edges represent the underlying subset relations.

Our proposed method is a top-down method. This means that we start testing at the root(s) of the graph, so at a node without any parents. If the null-hypothesis corresponding to this root node can be rejected, we test the children of the root node, and we continue to test only those nodes that have *at least one* of their parents or *all* their parent nodes rejected. The procedure that tests a node when at least one of its parents is rejected is called the *any*-parent variant, the procedure that tests a node only when all its parents are already rejected is called the *all*-parent variant.

Because the DAG is based on subset relations, each DAG has the so-called "oneway" property, which means that the falsehood of a node implies the falsehood of

its parents. Some DAGs have, in addition to this property, a second property: namely that the falsehood of a node implies the falsehood of at least one of its children. If a DAG has this property, we say that the DAG has a "twoway" property. An example of such a DAG would be the DAG consisting of several gene sets and all the individual genes. In that case, each node will be the union of its children, which is a sufficient condition to have the twoway property. In DAGs with twoway logical relations, more powerful methods are possible.

6.1 Using the DAG procedure

To perform the DAG procedure on a given dataset, we first have to create a DAG stucture. Given a collection of sets/hypotheses that we want to test, the corresponding DAG can be formed by using the function `construct`. This function needs a list of sets as input argument and returns a `DAGstructure` object. In the process, it is also checked whether all provided sets are unique, and if not, duplicates will be filtered out. A warning will be given in case this happens. To test the (unique) sets in the DAG structure, we can now use the function `DAGmethod`. This function needs the just created `DAGstructure` object as its first argument. Furthermore, what is needed for the function to work is a local test that has a set as input and returns a p-value for the corresponding hypothesis. Other options that can be specified are whether we want to have the any-parent or the all-parents variant, and whether we want to optimize our method further (in case we have a DAG with the twoway property). For the optimization step, we need a solver for (Integer) Linear Programs. A freely available R-package that does so is *lpSolve*. However, for large graphs it is often recommendable to choose some commercial software package. Our choice was to support the R-package *gurobi*. For academic use, this package is free, and can be found on the www.gurobi.com. If this package is not installed, we will automatically use *lpSolve* for all calculations.

To illustrate the method, we first generate some data. Let us take a response vector Y of length 100, and a matrix X of 4 covariates that is generated in such a way that Y is associated with only the second and the third covariate.

```
> set.seed(1)
> n=100
> p=4
> X <- matrix(rnorm(n*p),n,p)
> beta <- c(0,0.5,0.5,0)
> Y <- X %*% beta + rnorm(n)
```

Let us assume we are interested in testing all covariates separately, but also in the set of all covariates simultaneously (corresponding to the null-hypothesis that none of the covariates is associated with the response), the set of covariate 1 and 2 together, the set of covariate 2,3 and 4 togehter, and the set of covariate 2 and 3. This would result in the following list:

```
> sets <- list(c(1,2,3,4), c(1,2), c(2,3,4), c(2,3), 1, 2, 3, 4)
> names(sets) <- c(1234, 12, 234, 23, 1, 2, 3, 4)
```

Based on these sets, we can make the corresponding graph structure by using the function `construct`:

```
> struct <- construct(sets)
```

From this object we can check whether our desired DAG has twoway properties (which it should have, because all individual covariates are added):

```
> istwoway(struct)

[1] TRUE
```

For the actual DAG multiple testing method to work, we additionally have to specify a local test. We choose to work with an F-test, which can be defined in the following way:

```
> mytest <- function(set)
+ {
+   X <- X[,set,drop=FALSE]
+   lm.out <- lm(Y ~ X)
+   x <- summary(lm.out)
+   return(pf(x$fstatistic[1],x$fstatistic[2],
+             x$fstatistic[3],lower.tail=FALSE))
+ }
```

The DAG procedure can now be used on the design matrix X to find which (sets of) covariates are associated with the response by using the following function call:

```
> DAG <- DAGmethod(struct, mytest, isadjusted=TRUE)
> summary(DAG)

The allparents-method result on 8 hypotheses.
There are 6 hypotheses rejected out of a total of 8 at an alpha-level of 0.05.
```

By default, the all-parents procedure is chosen and the (possible) twoway structure of the DAG is not used to further optimize the method. To use the any-parents procedure or different optimization options, the arguments *method* and *optimization* can be set to different values than their default settings of respectively "all" and "none".

Using the function `DAGmethod` results in a `DAG` object. From this `DAG` object we can retrieve different information. By using the `show` or `summary` function, we can see how many of the hypotheses could be rejected on the given α-level as specified by the *alpha_max* argument, which is 0.05 by default. In our example, we see that 6 sets could be rejected (out of a total number of 8). All these sets can be summarized by the smallest sets that could be rejected themselves, but for which all smaller sets they contain (i.e. their offspring) could not get rejected. We will call those smallest rejectable sets "implications".

To get the implications from a `DAG` object, we can use the `implications`. This function only requires a `DAG` object as argument. In our example, the implications look like this:

```
> implications(DAG)

           2            3
3.834613e-05 4.362210e-04
```

We see that the 6 rejections can be summarized by the rejections of the set only consisting of covariate 2, and the set only consisting of covariate 3. The other 4 rejections are larger sets containing one or both of the covariates. We see that implications

come with a p-value, that can either be the true adjusted p-values (if the *isadjusted* argument of the `DAGmethod` was set to `TRUE`) or just the α-level on which the FWER is controlled. To get p-values of rejected sets that do not correspond to an implication, we can use the `pvalue` method that is available for `DAG` objects. The `pvalue` function expects a `DAG` object and an index or name as input arguments. The index must be the index of the set for which you want to know the p-value in the list of sets that is stored in the `DAGstructure` object. Normally, this will be the same index as the index found in the original list of sets, that is given as argument for the `construct` function, but if there were duplicate sets in the original list, these duplicates will be filtered by the `construct` function, and then the index can be different as well.

In our example all sets are unique, so the original list will be identical to the list stored in the `DAGstructure` object. To get the adjusted p-value of the null-hypothesis corresponding to the fourth set, which is the set consisting of covariate 2 and 3 we can thus use the following:

```
> pvalue(DAG,4)

           23
1.741771e-08
```

or (is we use the name of the set)

```
> pvalue(DAG, "23")

           23
1.741771e-08
```

6.2 Confidence sets for the number of false hypotheses in a given set

Just as in the case of a closed testing and the region procedure, while using the DAG procedure we are able to make simultaneous $100(1-\alpha)\%$ confidence sets for the number of false (or true) rejection within a given set S of elementary hypotheses. We can even do more this time, because we can make such a confidence set for the number of false rejection within *any* collection of hypotheses that we tested in our DAG procedure. In order to do so, we will again need an ILP solver. Given a `DAG` object, we can calculate the minimal number of false elementary hypotheses (i.e. true findings) in any given set of hypotheses. This set has to be specified by a vector of indices or names. For example, if we want to know how many false hypotheses there should at least be in the family of the first three sets we wanted to test, we can specify a vector 1:3, and use the function `DAGpick` to calculate the lower confidence bound for the number of false sets/hypotheses in this family:

```
> DAGpick(DAG, 1:3)

[1] 3
```

We see that the minimal number of sets that can be rejected with $(1-\alpha)\%$ confidence within these first 3 sets, is 3. An almost trivial observation, since all these sets were indeed rejected by our DAG method.

References

Benjamini, Y. and Hochberg, Y. (2000). On the adaptive control of the false discovery rate in multiple testing with independent statistics. *Journal of Educational and Behavioral Statistics*, 25(1):60–83.

Goeman, J. and Solari, A. (2011a). Multiple testing for exploratory research. *Statistical Science*, 26(4):584–597.

Goeman, J. and Solari, A. (2011b). Rejoinder. *Statistical Science*, 26(4):608–612.

Huang, Y. and Hsu, J. (2007). Hochberg's step-up method: cutting corners off holm's step-down method. *Biometrika*, 94(4):965–975.

Meinshausen, N. (2011). Discussion of multiple testing for exploratory research by J. J. Goeman and A. Solari. *Statistical Science*, 26(4):601–603.

Sarkar, S. and Chang, C. (1997). The simes method for multiple hypothesis testing with positively dependent test statistics. *Journal of the American Statistical Association*, pages 1601–1608.