Linear edit manipulation and error localization with the `editrules` package

Package version 0.7.1

Edwin de Jonge and Mark van der Loo

June 29, 2011

Abstract

This vignette is not completely finished. Version 1.0 fo the package will have the full vignette. This paper is the first of two papers describing the editrules package. The current paper is concerned with the treatment of numerical data under linear constraints, while the accompanying paper is concerned with constrained categorical and mixed data. The `editrules` package is designed to offer user-friendly interface for edit definition and manipulation. The package offers funtionality for edit checking, error localization based on the paradigm of Fellegi and Holt, and a flexible interface to binary programming based on the choice point paradigm. Lower-level functions include echelon transformation of linear systems, variable subtitution and a fast Fourier-Motzkin elimination routine. We describe theory, implementation and give examples of package usage.

Contents

List of Algorithms

1 Introduction

The value domain of real numerical data records with n variables is often restricted to a subdomain of $\mathbb{R}^n$ due to linear equality and inequality relations which the variables in the record have to obey. Examples include equality restrictions imposed by financial balance accounts, positivity demands on certain variables or limits on the ratio of variables.

Any such restriction is of the form

$$\mathbf{a} \cdot \mathbf{x} \odot b \text{ with } \odot \in \{<, \leq, =\}, \tag{1}$$

where $\mathbf{x}$ is a numerical data record, $\mathbf{a}$, $\mathbf{x} \in \mathbb{R}^n$ and $b \in \mathbb{R}$. In data editing literature, data restriction rules are refered to as *edits*, or *edit rules*. We will call edits, written in the form of Eq. (1), edits in *normal form*.

Large complex surveys are often endowed with dozens or even hundreds of edit rules. For example, the Dutch Structural Business Survey, which aims to report on the financial structure of companies in the Netherlands, contains on the order of 100 variables, endowed with a similar number of linear equality and inequality restrictions.

Defining and manipulating large edit sets can be a daunting task when implemented directly as matrix representations. Also, edit violations give rise to the error localization problem, which can quite simply be stated as *which variables contain the errors that cause a record to violate certain edits rules?*.

The `editrules` package for the R statistical computing environment (R Development Core Team, 2011) aims to provide an environment to conveniently define, parse and check linear (in)equality restrictions, perform common edit manipulations and offer error localization functionality based on the (generalized) paradigm of Fellegi and Holt (1976). This paradigm is based on the assumption that errors are distributed randomly over the variables, and there is no detectable cause of error. The paradigm also decouples the detection from correction of corrupt variables. Certain causes of error, such as sign flips, typing errors or rounding errors can be detected and are closely related to their resolution. The reader is referred to the `deducorrect` package (Van der Loo et al., 2011; Scholtus, 2008, 2009) for treating such errors.

The following chapters demonstrates the functionality of the `editrules` package with coded examples as well a description of of the underlying theory and algorithms. For a detailed per-function description the reader is referred to the reference manual accompanying the package. Unless mentioned otherwise, all code shown in this paper can be executed from the R commandline after loading the `editrules` package.

2 Defining and checking numerical restrictions

2.1 The `editmatrix` object

For computational processing, a set of edits of the form

$$\mathbf{a} \cdot \mathbf{x} \odot b \text{ with } \odot \in \{<, \leq, =, \geq, >\}. \tag{2}$$

is most conveniently represented as a matrix. In the `editrules` package, a set of linear edits is stored as an `editmatrix` object. This object stores the linear

relations as an augmented matrix $[\mathbf{A}, \mathbf{b}]$, where $\mathbf{A}$ is the matrix obtained by combining the $\mathbf{a}$ vecors of Eq. (2) in rows of $\mathbf{A}$ and constants b in $\mathbf{b}$. A second attribute holds the comparison operators as a `character` vector. Formally, we denote that every `editmatrix` E is defined by

$$E = \langle [\mathbf{A}, \mathbf{b}], \odot \rangle \text{ with } [\mathbf{A}, \mathbf{b}] \in \mathbb{R}^{m \times n+1},\ \odot \in \{<, \leq, =, \geq, >\}^m, \tag{3}$$

where n is the number of variables, m the number of edit rules and the notation $\langle\ ,\ \rangle$ denotes a combination of objects. Retrieval functions for various parts of an `editmatrix` are available, see Table 1 (p. 7) for an overview. Defining augmented matrices by hand is tedious and prone to error, which is why the `editmatrix` function derives edit matrices from a textual representation of edit rules. Since most functions of the `editrules` package expect an `editmatrix` in normal form (that is $\odot \in \{<, \leq, =\}$), the `editmatrix` function by default transforms all linear edits to normal form.

As an example, consider the set of variables

turnover	t
personell cost	c_p
housing cost	c_h
total cost	c_t
profit	p,

subject to the rules

$$\begin{aligned}
t &= c_t + p && (4)\\
c_t &= c_h + c_p && (5)\\
p &\leq 0.6t && (6)\\
c_t &\leq 0.3t && (7)\\
c_p &\leq 0.3t && (8)\\
t &> 0 && (9)\\
c_h &> 0 && (10)\\
c_p &> 0 && (11)\\
c_t &> 0. && (12)
\end{aligned}$$

Clearly, these can be written in the form of Eq. (1). Here, the equality restrictions correspond to balance accounts, the 3rd, 4th and 5th restrictions are sanity checks and the last four edits demand positivity. Figure 1 shows how these edit rules can be transformed from a textual representation to a matrix representation with the `editmatrix` function.

As Figure 1 shows, the `editmatrix` object is shown on the console as a matrix, as well as a set of textual edit rules. The `editrules` package is capable of coercing a set of R expressions to an `editmatrix` and *vice versa*. To coerce text to a matrix, the `editmatrix` function processes the R language parsetree of the textual R expressions as provided by the R internal `parse` function. To coerce the matrix representation to textual representation, an R character string is derived from the matrix which can be parsed to a language object.

In the example, the edits were automatically named `e1`, `e2`, ..., `e9`. It is possible to name and comment edits by reading them from a `data.frame`.

```
> E <- editmatrix(c(
+ "t  == ct + p" ,
+ "ct == ch + cp",
+ "p  <= 0.6*t",
+ "cp <= 0.3*t",
+ "ch <= 0.3*t",
+ "t  >  0",
+ "ch >  0",
+ "cp >  0",
+ "ct >  0"), normalize=TRUE)
> E

Edit matrix:
   ct  p    t ch cp Ops CONSTANT
e1 -1 -1  1.0  0  0  ==        0
e2  1  0  0.0 -1 -1  ==        0
e3  0  1 -0.6  0  0  <=        0
e4  0  0 -0.3  0  1  <=        0
e5  0  0 -0.3  1  0  <=        0
e6  0  0 -1.0  0  0   <        0
e7  0  0  0.0 -1  0   <        0
e8  0  0  0.0  0 -1   <        0
e9 -1  0  0.0  0  0   <        0

Edit rules:
e1 : t == ct + p
e2 : ct == ch + cp
e3 : p <= 0.6*t
e4 : cp <= 0.3*t
e5 : ch <= 0.3*t
e6 : 0 < t
e7 : 0 < ch
e8 : 0 < cp
e9 : 0 < ct
```

Figure 1: Defining an `editmatrix` from a `character` vector containing verbose edit statements. The option `normalize=TRUE` ensures that all comparison operators are either $<$, $\leq$ or $==$.

The ability to read edit sets from a `data.frame` facilitates defining and maintaining the rules outside of the R environment by storing them in a user-filled database or textfile. Manipulating and combining edits, for example through variable elimination methods will cause `editrules` to drop or change the names and drop the comments, as they become meaningless after certain manipulations.

2.2 Basic manipulations and edit checking

Table 1 shows simple manipulation functions available for an `editmatrix`. Basic manipulations include retrieval functions for the augmented matrix, coefficient matrix, constant vector and operators of an `editmatrix`. There are functions to test for and transform to normality. The function `violatedEdits` expects

```
> data(edits)
> edits

  name          edit              description
1   b1   t == ct + p            total balance
2   b2 ct == ch + cp             cost balance
3   s1    p <= 0.6*t            profit sanity
4   s2   cp <= 0.3*t    personell cost sanity
5   s3   ch <= 0.3*t      housing cost sanity
6   p1          t >0      turnover positivity
7   p2        ch > 0  housing cost positivity
8   p3        cp > 0 personel cost positivity
9   p4        ct > 0    total cost positivity

> editmatrix(edits)

Edit matrix:
   ct  p    t ch cp Ops CONSTANT
b1 -1 -1  1.0  0  0  ==        0
b2  1  0  0.0 -1 -1  ==        0
s1  0  1 -0.6  0  0  <=        0
s2  0  0 -0.3  0  1  <=        0
s3  0  0 -0.3  1  0  <=        0
p1  0  0 -1.0  0  0   <        0
p2  0  0  0.0 -1  0   <        0
p3  0  0  0.0  0 -1   <        0
p4 -1  0  0.0  0  0   <        0

Edit rules:
b1 : t == ct + p [ total balance ]
b2 : ct == ch + cp [ cost balance ]
s1 : p <= 0.6*t [ profit sanity ]
s2 : cp <= 0.3*t [ personell cost sanity ]
s3 : ch <= 0.3*t [ housing cost sanity ]
p1 : 0 < t [  turnover positivity ]
p2 : 0 < ch [ housing cost positivity ]
p3 : 0 < cp [ personel cost positivity ]
p4 : 0 < ct [ total cost positivity ]
```

Figure 2: Declaring an editmatrix with a `data.frame`. The input `data.frame` is required to have three columns named `name`, `edit` (textual representation of the edit rule) and `description` (a comment stating the intent of the rule). All must be of type `character`.

an `editmatrix` and a `data.frame` or a named numeric vector. It returns a `logical array` where every row indicates which edits are violated (`TRUE`) by records in the `data.frame`. Figure 3 demonstrates the result of checking two records against the editrules defined in Eqs. (4)–(12). Indexing of edits with the `[` operator is restricted to selection only. No assignment can be made to indexed `editmatrix` objects. In stead, `as.editmatrix` should be used.

Table 1: Simple manipulation functions for objects of class `editmatrix`. Only the mandatory arguments are shown, refer to the built-in documentation for optional arguments.

function	description
`getA(E)`	Get matrix $\mathbf{A}$
`getb(E)`	Get constant vector $\mathbf{b}$
`getAb(E)`	Get augmented matrix $[\mathbf{A}, \mathbf{b}]$
`getOps(E)`	Get comparison operators
`E[i,]`	Select edit(s)
`as.editmatrix(A,b,ops)`	Create an `editmatrix` from its attributes
`normalize(E)`	Transform `E` to normal form
`isNormalized(E)`	Check whether `E` is in normal form
`violatedEdits(E, x)`	Check which edits are violated by $\mathbf{x}$
`duplicated(E)`	Check for duplicates in rows of `E`
`isObviouslyRedundant(E)`	Check for tautologies and duplicates in `E`
`isObviouslyUnfeasible(E)`	Check for contradictions in rows of `E`
`isFeasible`	Complete feasibility check for `E`

```
> # define two records in a data.frame
> dat <- data.frame(
+   t = c(1000, 1200),
+  ct = c(400,  200),
+  ch = c(100,  350),
+  cp = c(500,  575),
+  p =  c(500,  652 ))
> # check for violated edits
> violatedEdits(E,dat)

      e1   e2    e3   e4    e5    e6    e7    e8    e9
[1,] TRUE TRUE FALSE TRUE FALSE FALSE FALSE FALSE FALSE
[2,] TRUE TRUE FALSE TRUE FALSE FALSE FALSE FALSE FALSE
```

Figure 3: Checking which edits are violated for every record in a `data.frame`. The first record violates `e1` and `e2`, the second record violates `e1`,`e2`, and `e4`.

2.3 Obvious redundancy and infeasibility

When manipulating linear edit sets by value substitution and/or variable elimination, the edit set can become polluted with redundant edits or, when variable values are substituted, become infeasible. The `editrules` package has two methods available which check for easily detectable redundancies or infeasibility. The Fourier-Motzkin elimination method has auxilary built-in redundancy removal, which is described in Section 3.3.

A system of inequalities $\mathbf{Ax} \leq \mathbf{b}$ is called infeasible when there is no real vecor $\mathbf{x}$ satisfying it. It is a consequence of Farkas' lemma (Farkas (1902), but see Schrijver (1998) and/or Kuhn (1956)) on feasibility of sytems of linear equalities, that a system is infeasible if and only if $0 \leq -1$ can be derived by taking positive linear combinations of the rows of the augmented matrix $[\mathbf{A}, \mathbf{b}]$. The function `isObiouslyinfeasible` returns a `logical` indicating whether such a contradiction is present. Subsitution of values may also lead to equalities of the

Algorithm 1 isObviouslyInfeasible(E)

Input: a normalized `editmatrix` E

for $\mathbf{a} \cdot \mathbf{x} \odot b \in E$ **do**
  if $\mathbf{a} = \mathbf{0}$ **then**
    if $(\odot \in \{=\} \wedge b \neq 0) \vee (\odot \in \{\leq, <\} \wedge b < 0)$ **then**
      return TRUE
return FALSE

Output: ▷ `logical` indicating if E is obviously infeasible.

Algorithm 2 isObviouslyRedundant(E, duplicates, ε)

Input: a normalized `editmatrix` E, with m edits, a boolean "duplicates", and a tolerance ε.

$\mathbf{v} \leftarrow (\text{FALSE})^{\times m}$
for $\mathbf{a}_i \cdot \mathbf{x} \odot b_i \in E$ **do**
  if $\mathbf{a} = \mathbf{0}$ **then**
    if $(\odot \in \{=\} \wedge b = 0) \vee (\odot \in \{\leq, <\} \wedge b > 0)$ **then**
      $v_i \leftarrow$TRUE
if duplicates **then**
  for $\{(\mathbf{a}_i \cdot \mathbf{x} \odot_i b_i, \mathbf{a}_j \cdot \mathbf{x} \odot_j b_j) \in E \times E : j > i\}$ **do**
    if $|(\mathbf{a}_i, b_i) - (\mathbf{a}_j, b_j)| \leq \varepsilon$ elementwise $\wedge \odot_i = \odot_j$ **then**
      $v_j \leftarrow$TRUE

Output: $\mathbf{v}$ ▷ `logical` vector indicating which rows of E are obviously redundant.

form $0 = 1$, which also indicate that the system has become infeasible. Being obviously infeasible is sufficient for an `editmatrix` to be infeasible, but not necessary. Algorithm 1 gives the pseudocode for reference purposes.

The function `isFeasible` eliminates variables one by one using Fourier-Motzkin elimination (Section 3.3), and checks for obvious infeasibilities. If no obvious inconsistencies are found after the last variable has been eliminated, the system is consistent.

When new edits are derived, either by value substitution or by variable elimination, redundant rules of the form $0 \leq 1$ or $0 = 0$ can be generated. The function `isObviouslyRedundant` detects such rules and returns a `logical` vector indicating which rows of an `editmatrix` are redundant. By default, the function detects row duplicates (within an adjustable tolerance), but this may be swithched of by providing the option `duplicates=FALSE`. Pseudocode is given in Algorithm 2. The actual implementation avoids explicit loops and makes use of R's built-in `duplicated` function, which is also overloaded for `editmatrix` (see Table 1).

3 Manipulation of linear restrictions

There are two fundamental operations possible on edit sets, both of which (possibly) reduce the number of variables involved in the edit set. The first, most simple one is when a value is substituted into an edit. The second possibility is variable elimination. For a set of linear equalities, one can apply Gaussian

Algorithm 3 substValue(E, j, x)

Input: $E = \langle [\mathbf{A} = [\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_j, \ldots, \mathbf{a}_n] | \mathbf{b}], \odot \rangle$, $x \in \mathbb{R}$, $j \in \{1, 2, \ldots n\}$

▷ Note that here, the subscripts of $\mathbf{a}$ denote the column index of $\mathbf{A}$

Output: $\langle [\mathbf{A} = [\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_{j-1}, \mathbf{0}, \mathbf{a}_{j+1}, \ldots \mathbf{a}_n] \, | \mathbf{b} - \mathbf{a}_j x], \odot \rangle$

```
> substValue(E, "t", 10)

Edit matrix:
   ct  p t ch cp Ops CONSTANT
e1 -1 -1 0  0  0  ==      -10
e2  1  0 0 -1 -1  ==        0
e3  0  1 0  0  0  <=        6
e4  0  0 0  0  1  <=        3
e5  0  0 0  1  0  <=        3
e7  0  0 0 -1  0   <        0
e8  0  0 0  0 -1   <        0
e9 -1  0 0  0  0   <        0

Edit rules:
e1 : 10 == ct + p
e2 : ct == ch + cp
e3 : p <= 6
e4 : cp <= 3
e5 : ch <= 3
e7 : 0 < ch
e8 : 0 < cp
e9 : 0 < ct
```

Figure 4: Substituting the value 10 for the turnover variable using the `substValue` function.

elimination, while for sets of inequalities or mixed sets of equalities and inequalities Fourier-Motzkin elimination is applied. While variable substitution and Gaussian elimination guarantee that the eliminated variable is not involved in the derived edit set anymore, this is not necessarily the case for Fourier-Motzkin elimination.

3.1 Value substitution

Given a set of m linear edits as defined in Eq. (3). For any record $\mathbf{x}$ it must hold that

$$\mathbf{A}\mathbf{x} \odot \mathbf{b}, \quad \odot \in \{<, \leq, =, \geq, >\}^m. \tag{13}$$

Substituting one of the unknowns x_j by a certain value x amounts to replacing the j^{th} column of $\mathbf{A}$ with $\mathbf{0}$ and $\mathbf{b}$ with $\mathbf{b} - \mathbf{a}'_j x$. After this, the reduced record of unknowns, with x_j replaced by x has to obey the adapted system (13). For reference purposes, Algorithm 3 spells out the substitution routine. The function was named `substValue` since `substitute` is already defined in the R-base. Figure 4 shows how `substValue` can be called from the R environment.

Algorithm 4 ECHELON(E)

Input: An `editmatrix` of the form $\langle [\mathbf{A}|\mathbf{b}], = \rangle$, $[\mathbf{A}|\mathbf{b}] \in \mathbb{R}^{m \times n+1}$, $m \leq n+1$.

$I \leftarrow \{1, 2, \ldots, m\}$
$J \leftarrow \{1, 2, \ldots, n+1\}$
for $j \in I$ **do** ▷ eliminate variables
  $i \leftarrow \arg\max_{i' \,:\, j \leq i' \leq m} |A_{i'j}|$
  if $|A_{ij}| > 0$ **then**
    if $i > j$ **then**
      Swap rows i and j of $[\mathbf{A}|\mathbf{b}]$.
    $[\mathbf{A}|\mathbf{b}]_{I \setminus j, J} \leftarrow [\mathbf{A}|\mathbf{b}]_{I \setminus j, J} - [\mathbf{A}|\mathbf{b}]_{I \setminus j, j} \otimes [\mathbf{A}|\mathbf{b}]_{j, J} A_{jj}^{-1}$
Divide each row $[\mathbf{A}|\mathbf{b}]_{i,J}$ by A_{ii} when $A_{ii} \neq 0$
Move rows of $[\mathbf{A}|\mathbf{b}]$ with all zeros to bottom.

Output: E, transformed to reduced row echelon form.

3.2 Gaussian elimination

The well-known Gaussian elimination routine has been implemented here as a utility function, enabeling users to reduce the equality part of their edit matrices to reduced row echelon form. The `echelon` function has been overloaded to take either an R `matrix` or an `editmatrix` as argument. In the latter case, the equalities are transformed to reduced row echelon form, while inequalities are left untreated. Gaussian elimination is explained in many textbooks. Algorithm 4 is written in a notation which is close to our R implementation in the sence that it involves just one explicit loop. Figure 5 demonstrates a call to the R function.

3.3 Fourier-Motzkin elimination

Fourier-Motzkin elimination [Fourier (1826); Motzkin (1936), but see Williams (1986) for an elaborate or Schrijver (1998) for a consice description] is an extension of Gaussian elimination to solving systems of linear inequalities. While Gaussian elimination is based on the reversible operations of row permutation and linear combination, Fourier-Motzkin elimination is based on the irreversible action of taking positive combinations of rows.

A full Fourier-Motzkin operation on a system of inequalities involves eliminating variables (where possible) one by one from the augmented matrix $[\mathbf{A}|\mathbf{b}]$. Eliminating a single variable is an important step in the error localization algorithms elaborated in Section 4.

Consider a system of inequalities $\mathbf{Ax} \leq \mathbf{b}$. The j^{th} variable is eliminated by generating a positive combination of every row of $[\mathbf{A}|\mathbf{b}]$ where $A_{ij} > 0$ with every row of $[\mathbf{A}|\mathbf{b}]$ where $A_{ij} < 0$ such that for the resulting row the j^{th} coefficient equals zero. Rows of $[\mathbf{A}|\mathbf{b}]$ for which $A_{ij} = 0$ are copied to the resulting system. If the system does not contain rows for which $A_{ij} > 0$ and rows for which $A_{ij} < 0$, an elimination operation leaves the system unchanged.

Mixed systems with linear restrictions of the form $\mathbf{a} \cdot \mathbf{x} \odot b$ with $\odot \in \{<, \leq, =\}$ can in principle be transformed to a form where every $\odot \in \{\leq\}$. However, it is more efficient to take the comparison operators into account when combining rows. In that case, new rules are derived by first solving the the j^{th} from

```
> echelon(E)

Edit matrix:
   ct p    t ch cp Ops CONSTANT
e1  1 0  0.0 -1 -1  ==        0
e2  0 1 -1.0  1  1  ==        0
e3  0 1 -0.6  0  0  <=        0
e4  0 0 -0.3  0  1  <=        0
e5  0 0 -0.3  1  0  <=        0
e6  0 0 -1.0  0  0   <        0
e7  0 0  0.0 -1  0   <        0
e8  0 0  0.0  0 -1   <        0
e9 -1 0  0.0  0  0   <        0

Edit rules:
e1 : ct == ch + cp
e2 : p + ch + cp == t
e3 : p <= 0.6*t
e4 : cp <= 0.3*t
e5 : ch <= 0.3*t
e6 : 0 < t
e7 : 0 < ch
e8 : 0 < cp
e9 : 0 < ct
```

Figure 5: Transforming linear equalities of an editmatrix to reduced row echelon form. See Figure 1 for the original definition of `E`.

each equality and substituting them in each inequality. Next, inequalities are treated as stated before. When inequalities are combined where one comparison operator is $<$ and the other is $\leq$, it is not difficult to show that $<$ becomes the operator for the resulting inequality.

It is a basic result of the theory of linear inequalities that the system resulting from a single variable elimination is equivalent to the original system (that is, they have the same solution set $\{\mathbf{x}\}$). In fact, k elimination steps can generate up to $(\frac{1}{2}m)^{2k}$ new rows (m being the original number of rows), of which many are redundant. Since the number of redundant rows increases fast during elimination, removing (most of) them is highly desirable. In our implementation, we use the property that if k variables have been eliminated, any row derived from more than $k+1$ rows of the original system is redundant. This result was originally stated by Černikov (1963) and rediscovered by Kohler (1967). A proof can also be found in Williams (1986). For the implementation in R, an `editmatrix` is augmented with an integer h, recording the number of eliminations and a `logical` array $\mathbf{H}$, which records for each edit from which original edit it was derived. Obviously, $\mathbf{H}$ is TRUE only on the diagonal when $h = 0$. It is worth mentioning that by using R's vectorized indices and recycling properties, it is possible to avoid any explicit looping in the elimination process. Algorithm 5 gives an overview of the algorithm where explicit loops are included for readability. Figure 6 shows how one or more variables can be eliminated from an editmatrix with the `eliminateFM` function. Note that when multiple variables are eliminated, the `editmatrix` must be overwritten to at every iteration to ensure that the history $\mathbf{H}$ is updated accordingly.

Algorithm 5 ELIMINATEFM(E, j). In the actual implementation all explicit loops are avoided by making use of R's recycling properties and vectorized indices.

Input: A normalized `editmatrix` $E = \langle [\mathbf{A}|\mathbf{b}], \odot, \mathbf{H}, h \rangle$, and a variable index j.

if $\mathbf{H} = \varnothing$ **then**
  $\mathbf{H} \leftarrow \text{diag}(\text{TRUE})^m$
  $h \leftarrow 0$
$J \leftarrow \{1, 2, \ldots, n+1\}$
$I_0 \leftarrow \{i : A_{ij} = 0\}$
$I_= \leftarrow \{i : \odot_i \in \{=\}\} \backslash I_0$
$I_+ \leftarrow \{i : A_{ij} > 0\} \backslash I_=$
$I_- \leftarrow \{i : A_{ij} < 0\} \backslash I_=$
for $i \in \{1, 2, \ldots, m\} \backslash I_0$ **do** ▷ All rows get j^{th} coefficient in $\{-1, 0, 1\}$
  if $\odot_i \in \{<, \leq\}$ **then**
    $[\mathbf{A}|\mathbf{b}]_{i,J} \leftarrow [\mathbf{A}|\mathbf{b}]_{i,J} |A_{ii}|^{-1}$
  else
    $[\mathbf{A}|\mathbf{b}]_{i,J} \leftarrow [\mathbf{A}|\mathbf{b}]_{i,J} A_{ii}^{-1}$

▷ Substitute equalities and inequalities with positive j^{th} coefficient in inequalities with negative j^{th} coefficient:

for $(i, j) \in (I_= \cup I_+) \times I_-$ **do**
  $k \leftarrow k + 1$
  $[\tilde{\mathbf{A}}|\tilde{\mathbf{b}}]_{k,J} \leftarrow [\mathbf{A}|\mathbf{b}]_{i,J} + [\mathbf{A}|\mathbf{b}]_{j,J}$
  $\tilde{\mathbf{H}}_{k,J} \leftarrow \mathbf{H}_{i,J} \vee \mathbf{H}_{j,J}$
  if $\odot_i \in \{<\}$ **then** $\tilde{\odot}_k \leftarrow \odot_i$ **else** $\tilde{\odot}_k \leftarrow \odot_j$

▷ Substitute equalities in inequalities with positive j^{th} coefficient

for $(i, j) \in I_+ \times I_=$ **do**
  $k \leftarrow k + 1$
  $[\tilde{\mathbf{A}}|\tilde{\mathbf{b}}]_{k,J} \leftarrow [\mathbf{A}|\mathbf{b}]_{i,J} - [\mathbf{A}|\mathbf{b}]_{j,J}$
  $\tilde{\mathbf{H}}_{k,J} \leftarrow \mathbf{H}_{i,J} \vee \mathbf{H}_{j,J}$
  $\tilde{\odot}_k \leftarrow \odot_i$

for $\{(i, j) \in I_=^{\times 2} : j > i\}$ **do** ▷ Substitute equalities in equalities
  $k \leftarrow k + 1$
  $[\tilde{\mathbf{A}}|\tilde{\mathbf{b}}]_{k,J} \leftarrow [\mathbf{A}|\mathbf{b}]_{i,J} - [\mathbf{A}|\mathbf{b}]_{j,J}$
  $\tilde{\mathbf{H}}_{k,J} \leftarrow \mathbf{H}_{i,J} \vee \mathbf{H}_{j,J}$
  $\tilde{\odot}_k \leftarrow \odot_i$

$$\tilde{E} \leftarrow \left\langle \left[\tilde{\mathbf{A}}|\tilde{\mathbf{b}}]', [\mathbf{A}|\mathbf{b}]'_{I_0,J}\right]', (\tilde{\odot}, \odot_{I_0}), \tilde{\mathbf{H}}, h+1 \right\rangle$$

Remove edit rules of $\tilde{E}$ which have more than $h+1$ elements of $\mathbf{H}_{i,J}$ TRUE
Remove edit rules of $\tilde{E}$ for which ISOBVIOUSLYREDUNDANT$(\tilde{E})$ is TRUE

Output: `editmatrix` $\tilde{E}$ with variable j eliminated and updated history

```
> eliminateFM(E, "t")

Edit matrix:
   ct          p t        ch        cp Ops CONSTANT
e1 -1  0.6666667 0  0.000000  0.000000  <=        0
e2 -1 -1.0000000 0  0.000000  3.333333  <=        0
e3 -1 -1.0000000 0  3.333333  0.000000  <=        0
e4 -1 -1.0000000 0  0.000000  0.000000   <        0
e5  1  0.0000000 0 -1.000000 -1.000000  ==        0
e6  0  0.0000000 0 -1.000000  0.000000   <        0
e7  0  0.0000000 0  0.000000 -1.000000   <        0
e8 -1  0.0000000 0  0.000000  0.000000   <        0

Edit rules:
e1 : 0.666666666666667*p <= ct
e2 : 3.33333333333333*cp <= ct + p
e3 : 3.33333333333333*ch <= ct + p
e4 : 0 < ct + p
e5 : ct == ch + cp
e6 : 0 < ch
e7 : 0 < cp
e8 : 0 < ct

> F <- E
> for (var in c("t", "cp", "p")) F <- eliminateFM(F, var)
> F

Edit matrix:
           ct p t        ch cp Ops CONSTANT
e1 -2.5000000 0 0  0.000000  0   <        0
e2  0.8333333 0 0 -3.333333  0  <=        0
e3 -2.5000000 0 0  3.333333  0  <=        0
e4 -1.0000000 0 0  1.000000  0   <        0
e5  0.0000000 0 0 -1.000000  0   <        0
e6 -1.0000000 0 0  0.000000  0   <        0

Edit rules:
e1 : 0 < 2.5*ct
e2 : 0.833333333333334*ct <= 3.33333333333333*ch
e3 : 3.33333333333333*ch <= 2.5*ct
e4 : ch < ct
e5 : 0 < ch
e6 : 0 < ct
```

Figure 6: Above: eliminating `t` from the editmatrix with the `eliminateFM` function. Below: to eliminate multiple variables, the original editmatrix must be overwritten at each iteration to ensure that the derivation history is updated at every step.

4 Error localization for numerical data

While checking whether a numerical record violates any imposed restrictions (within a certain limit) is easy, finding out which variable(s) of the record cause the violation(s) can be far from trivial. When possible, the cause of the violation, should be sought out, since it leads immediately to repair suggestions. The `deducorrect` package (Van der Loo et al., 2011) mentioned above offers functionality to detect and repair common errors like typing errors, rounding errors and sign errors. Although not directly available in R, methods for detecting repairing unit measure errors or other systematic errors have been described in literature and may readily be implemented in R (see De Waal et al. (2011) Chapter 2 for an overview).

After systematic errors with detectable causes in a data set have been resolved, one may assume that remaining errors are distributed randomly (but not necessarily uniformly) over one or more of the variables. In that case, error localization based on the (generalized) principle of Fellegi and Holt can be applied.

4.1 The generalized Fellegi-Holt paradigm

In line with the good practice of altering source data as little as possible, the paradigm of Fellegi and Holt (1976) advises to edit an as small amount of variables as possible, under the condition that after editing, every edit rule can be obeyed. A generalization of this principle says that a weighted number of variables should be minimized. More formally, the principle yields the following problem. Given a record $\mathbf{x}$, violating a number of edits in an edit matrix E (see Eqn. (3)) with m rules and n variables, find G such that

$$\begin{aligned} G = & \underset{g \subset \{1,2,\ldots,n\}}{\operatorname{argmin}} \sum_{j \in g} w_j \delta(x_j, \tilde{x}_j), \\ & \text{such that a solution } \tilde{\mathbf{x}} \in \mathbb{R}^{|G|} \text{ exists for} \\ & \sum_{j \in G} A_{ij} \tilde{x}_j \odot_i b_i - \sum_{j \notin G} A_{ij} x_j, \quad i \in \{1, 2, \ldots, m\}. \end{aligned} \tag{14}$$

In other words, for every variable in $\mathbf{x}$, we have to deceide wheter to adapt it or not. Variables which are not adapted will be replaced with their value x_j while variables that will be adapted will have to bereplaced by a value $\tilde{x}_j$, which has to to be determined. The solution to (14) need not be unique, but there is always at least one solution unless the edit rules in E are contradictory.

The minimization (14) amounts to a binary search problem, of which the search space increases as 2^n (n the number of variables). De Waal (2003) and De Waal et al. (2011) describe a branch-and-bound binary search algorithm which generates all minimal weight solutions. It works by generating the following binary tree: the root node contains E and $\mathbf{x}$ and weight $w = 0$. Both left and right nodes receive a copy of the objects in their parents. In the left child node, x_1 is assumed correct and its value is substituted in E. In the right child node, x_1 is assumed to contain an error and it is eliminated from E by Fourier-Motzkin elimination. The weight w in the right node is increased by w_1. Each child node gets a left and right child node where x_2 is substituted or eliminated, and so on untill every variable has been treated. Every path from root to leave

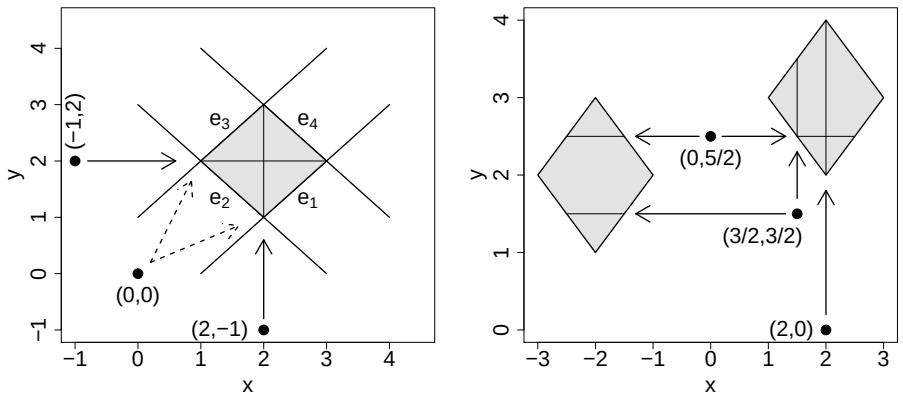

Figure 1: Graphic representation of editrules and the allowed area. Left panel: a convex case, as defined by Eq. (15). Right panel: the nonconvex nonconnected case, as defined by Eq. (23). Grey areas indicate the valid record domain, black dots indicate erroneous records and black arrows indicate the solution of the error localization problem, while the thin black lines show the range of solutions. The dotted arrows in the left panel indicate the range of directions in which the record (0,0) can move to reach the valid area.

represents one element of the search space. A branch is pruned when E contains obvious inconsistencies, so no combinations not satisfying the condition in (14) are generated. If a solution, with certain weight w is found, branches developed later, receiving a higher weight are pruned as well.

To clarify the above, in the next subsection we give two worked examples. Subsection (4.4) describes a flexible binary search algorithm, which we implememted to support general binary search problems. Subsection 4.3 describes its application to the branch-and-bound algorithm mentioned above.

4.2 Two examples

To illustrate the binary search algorithm outlined above we will consider a simpe two-dimensional example. The reader is encouraged to follow the reasoning below by checking the calculations using the R-functions mentioned in the previous sections.

Consider a 2-variable record (x, y) subject to the set of constaints E:

$$E = \begin{cases} e_1: & y > x - 1 \\ e_2: & y > -x + 3 \\ e_3: & y < x + 1 \\ e_4: & y < -x + 5. \end{cases} \tag{15}$$

Each separate inequality yields a half-plane of which the borders is determined by the line obtained by replacing $<$ or $>$ by $=$. The intersection of the four half-planes is the region of allowed records. In this example, the region is a diamond, depicted as the grey area in Figure 1. The borders are labed with the editrules in Eq. (15). Consider the record $(y = 2, x = -1)$, depicted as

the bottom black dot in Figure 1. It is easy to confirm either graphically or by substitution that $(2, -1)$ violates edits e_1 and e_2, and that the record can be made consistent by altering only y and leaving x constant (indicated by the black arrow). It is also clear from the graph that the allowed values for y are between 1 and 3 (indicated by the thin black vertical line in the diamond). The case $(x = 0, y = 0)$ also violates e_1 and e_2 and can only be replaired by altering both x and y, while the record $(x = -1, y = 2)$ can be repaired by changing x only.

In the following we show that the binary search algorithm described in the previous subsection indeed solves the error localization problem for $(x = 2, y = -1)$. To find the unweighted, least number of variables to adapt, so that E can be fulfilled, consider the triple

$$T_0 = \langle E, (2, -1), w = 0) \rangle \,, \tag{16}$$

This is the root node of the binary search tree described in the previous subsection, with w the initial solution weight. The left child is generated by assuming that the first value in the record is correct. We therefore replace the variable x in E by its value in the record, which yields after removing redundancies,

$$T_{0l} = \left\langle \begin{array}{l} y > 1 \\ y < 3 \end{array} , (2, -1), 0 \right\rangle . \tag{17}$$

In this notation, each time a left (right) node is added, the subscribt of T is augmented with an l (r). Substituting one of the values further restricts the possible values for variables that have not been treated yet. In fact, after the error localization problem has been solved, subsitituting all unaltered values into E yields a set of equations which determine the range of the variable vector which have to be altered or imputed.

Since no variables were eliminated, the weight in T_{0l} is 0, and the record has not changed. In the right child of the root, x is assumed to be wrong, and therefore eliminated using Fourier-Motzkin elimination:

$$T_{0r} = \left\langle \begin{array}{l} y > 1 \\ y < 3 \end{array} , (x, -1), 1 \right\rangle . \tag{18}$$

The system of equations left after elimination of x illustrates the geometrical interpretation of Fourier-Motzkin elimination. The range of y corresponds to the projection of the diamond in the left pane of Figure 1 onto the y-axis. (The fact that T_{0l} yields the same system is mere coincidence and depends on the fact that the x-coordinate in the record at hand equals 2). Calculating the left child of T_{0l} means substiting y by -1 in the edits of T_{0l}. This yields

$$T_{0ll} = \left\langle \begin{array}{l} -1 > 1 \\ -1 < 3 \end{array} , (2, -1), 0 \right\rangle , \tag{19}$$

where the contradiction $-1 > 1$ which indicates that T_{0ll} is not a solution (which is obvious since none of the values in the records are assumed incorrect). The right child of T_{0l} is obtained by eliminating y:

$$T_{0lr} = \langle \varnothing, (2, y), 1 \rangle \,, \tag{20}$$

where the tautology $0 < 2$ was removed. This endnode does represent a solution, since no conflicting rules have been generated. To see if any other solutions exist, continue to calculate the left child node of T_{0r}

$$T_{0rl} = \left\langle \begin{array}{l} -1 > 1 \\ -1 < 3 \end{array} , (x, -1), 1 \right\rangle, \tag{21}$$

which is no solution since its edits hold a contradiction. The final, right child node of T_{0r} reads

$$T_{0rr} = \langle \varnothing, (x, y), 2 \rangle, \tag{22}$$

which also is a solution, but since both x and y have to be adapted, it has a higher weight than the solution T_{0lr} found earlier.

The edit sets described so far involved a single set of (in)equalities, yielding a convex record domain in $\mathbb{R}^n$. However, in practical cases the sets of allowed values for a record need not be convex, or even connected. As an example consider the space of allowed records, indicated by the grey areas in the right panel of Figure 1. Such a range can be defined by a conditional edit of the form

$$\textbf{if } e_0 : x < 0 \textbf{ then } \left\{ \begin{array}{ll} e_1 : & y > x + 3 \\ e_2 : & y > -x + 1 \\ e_3 : & y < x + 5 \\ e_4 : & y < -x + 1 \end{array} \right. \textbf{ else } \left\{ \begin{array}{ll} e'_1 : & y > x \\ e'_2 : & y > -x + 4 \\ e'_3 : & y < x + 2 \\ e'_4 : & y < -x + 6. \end{array} \right. \tag{23}$$

The error localization problem fo this can be handled by solving the partial localization problems for $\{e_0, e_1, \ldots, e_4\}$ and $\{\overline{e}_0, e'_1, \ldots, e'_4\}$ separately, where $\overline{e}_0$ stands for the complement $\overline{e}_0 : x \geq 0$. The partial solution with the lowest weight solves the complete optimization problem. As an illustration consider the record $(x = 2, y = 0)$ in the right panel of Figure 1. The error localization problem corresponding to $x < 0$ yields a solution where both x and y have to be altered, while the localization problem corresponding to $x \geq 0$ implies that only y needs to be altered.

To generalize this example, note that a conditional edit sets of the form

$$\textbf{if } E_0 \textbf{ then } E_1 \textbf{ else } E_2, \tag{24}$$

can be written as

$$(E_0 \wedge E_1) \vee (\overline{E}_0 \wedge E_2) \tag{25}$$

which may be treated by finding the minimum weight solution between the solutions generated by $E_0 \wedge E_1$ and $\overline{E}_0 \wedge E_2$. Taking the complement can cause the number of partial localization problems to grow quickly. As an illustration, consider the following case where taking the complement yields three cases to be treated by the error localization routine.

$$\begin{aligned} &\textbf{if } (x = 0) \textbf{ then } E_1 \textbf{ else } E_2 \\ &\quad \Leftrightarrow \quad ((x = 0) \wedge E_1) \vee ((x \neq 0) \wedge E_2) \\ &\quad \Leftrightarrow \quad ((x = 0) \wedge E_1) \vee ((x < 0) \wedge E_2) \vee ((x > 0) \wedge E_2). \end{aligned} \tag{26}$$

The number of partial error localization problems to be treated grows as $2n_{\text{eq}} + n_{\text{ineq}}$, where n_{eq} is the number of equalities and n_{ineq} the number of inequalities in E_0. This is easily derived from Eq. (25) since by De Morgan's rule

$$\overline{E}_0 = \overline{e_1 \wedge e_2 \wedge \ldots \wedge e_k} = \overline{e}_1 \vee \overline{e}_2 \vee \ldots \vee \overline{e}_k. \tag{27}$$

Here, each negated inequality translates to a single inequality, while each negated equality yields two inequalities (as in Eq. (26)).

We will have more to say on conditional edits in the accompanying paper where the error localization problem for categorical and mixed data are treated.

4.3 Error localization with `errorLocalizer`

The error localization problem detailed in the previous subsections can be automated with `errorLocalizer`. This function expects an `editmatrix`, a named `numerical` record and optionally a vector of reliability weights with the same length as the record. The return value is not the solution to the error localization problem but an object of class `choicepoint`. With a choicepoint object the branch-and-bound tree can be generated to find solutions one by one. As an example, consider the edits of Eqn. (15), and the record $(x = 1, y = -1)$. Figure 4.3 shows how the error localization problem can be solved with the `choicepoint` object returned by `errorLocalizer`. The internal machinery of `choicepoint` objects is detailed in the next subsection, in this section we show how to use such objects to solve error localization problems. In Figure 4.3, the edits of Eqn. (15) and the record $(2, -1)$ are offered to `errorLocalizer`. By calling the built-in `searchNext` function, the choicpoint object traverses the binary search tree depth-first, untill the first solution is found. When a solution is found, is is returned to the user as a list, containing the solution weight `w`, the edit matrix left after all substitutions and eliminations, and the logical vector `adapt` which is `TRUE` for variables which need to be changed, and `FALSE` for variables which can retain their original values. As expected, y is pointed out as the variable to change. Another call to `searchNext` will search for the next solution in the tree, with lower weight. However, since in this example there is only one solution, `searchNext` returns `NULL`.

The method `searchNext` is not the only method of the `choicepoint` object returned by `cp.editmarix`. The avaliable methods are

- `$searchNext` Searches for the next solution with a lower weight than the previously found solution.
- `$searchAll` Returns all solutions, regardless of the weight.
- `$searchBest` Returns the last solution. Note that although this is a lowest-weight solution, it does not need to be unique.

In fact, any `choicepoint` object is equipped with the `searchNext` and `searchAll` methods. The `searchBest` method is specific for `errorLocalizer`.

The choicepoint method offers a flexible interface for error localization. To understand what happens when there are multiple solutions, consider the case of a simple balance account for profit (p), loss (l) and turnover (t):

```
> E <- editmatrix(c("p + c == t"))
> r <- c(p=755, c=125, t=200)
> cp <- errorLocalizer(E, r)
```

The record obviously violates the edit in `E`. Since there is only a single edit rule, there are three solutions, all of which can be found by calling `cp$searchNext`

```
> cp$searchNext()$adapt
```

```
> E1 = editmatrix(c(
+     "y >  x + 1",
+     "y > -x + 3",
+     "y <  x + 1",
+     "y < -x + 5"))
> cp <- errorLocalizer(E1, c(x=2,y=-1))
> cp$searchNext()

$w
[1] 1

$E
Edit matrix:
   x y Ops CONSTANT
e1 0 0   <        0

Edit rules:
e1 : 0 < 0

$adapt
    x     y
FALSE  TRUE

> cp$searchNext()

NULL
```

Figure 7: Localizing errors with the `choicepoint` object generated by `errorLocalizer`

```
    p     c     t
FALSE FALSE  TRUE

> cp$searchNext()$adapt

    p     c     t
FALSE  TRUE FALSE

> cp$searchNext()$adapt

    p     c     t
 TRUE FALSE FALSE
```

Each solution has weight 1. Suppose that the turnover value is trusted more, for example because it comes from a more reliable source. We may increase its reliability weight by providing a weight vector:

```
> cp <- errorLocalizer(E, r, weight=c(1,1,2))
> cp$searchNext()$adapt

    p     c     t
FALSE  TRUE FALSE

> cp$searchNext()$adapt
```

```
    p     c     t
 TRUE FALSE FALSE

> cp$searchNext()$adapt

NULL
```

The solution where turnover must be adapted is not even found here. The reason is that `errorLocalizer` makes sure that during the search for solutions, variables with the highest reliability weight are the last ones to be assumed incorrect.

If we add more restrictions, the number of solutions to the error localization problem decreases. Here, we demand that the cost to turnover ration does not exceed 0.6.

```
> E <- editmatrix(c(
+         "p + c == t",
+         "c - 0.6*t >= 0"))
> cp <- errorLocalizer(E, r)
> cp$searchNext()$adapt

    p     c     t
FALSE  TRUE  TRUE

> cp$searchNext()$adapt

    p     c     t
 TRUE FALSE FALSE

> cp$searchNext()$adapt

NULL
```

Here, first a solution of weight 2 is found, which may later be rejected in favor of the solution which demands only that the profit variable should be changed.

With `errorLocalizer` records with missing data can be handled as well. Since variables with missing values have to be replaced, they are eliminated from the edit matrix prior to further error localization. In the next example we add some extra variables and positivity demands on all variables.

```
> # An example with missing data.
> E <- editmatrix(c(
+     "p + c1 + c2 == t",
+     "c1 - 0.3*t >= 0",
+     "p > 0",
+     "c1 > 0",
+     "c2 > 0",
+     "t > 0"))
> cp <- errorLocalizer(E,x=c(p=755, c1=50, c2=NA,t=200))
> cp$searchNext()$adapt

    p    c1    c2     t
FALSE FALSE  TRUE  TRUE

> cp$searchNext()$adapt
```

```
    p    c1    c2     t
FALSE  TRUE  TRUE FALSE

> cp$searchNext()$adapt

NULL
```

There are two solutions, both of which include the field `c2` with the missing value.

4.4 General binary search with the `choicepoint` object

As stated in subsection 4.1, the error localization problem can be interpreted as a (pruned) binary programming problem. To facilitate implementation of error localization for numerical, categorical and mixed data, as well as to help further research in error localization algorithms, general-purpose binary search functionality was implemented in the form of binary choice point programming.

The term "choice point" stems from the field of nondeterministic programming. In nondeterministic programming, the control flow of a program is not determined explicitly by the programmer with standard branching statements. In stead, choice points may be created which store the full state of a program so that control flow can at any time return to a stored state and choose a new path from there. Choice point programming is supported by various niche programming environments, such as Alma-0 (Partington, 1997) and ELAN (Vittek, 1996). See Moreau (1998) for a clear introduction or Mart-Oliet and Mesguer (2002) for a bibliographic overview. The choice point paradigm offers an excellent environment for programming backtracking algorithms, of which the branch-and-bound algorithm of subsection 4.1 is just a specific example.

The R language is ideally suited to develop choice point-like systems because of its first-class environments. An R environment can be thought of as a list of R objects, forming the scope for expression evaluation. Expressions are a series of R statements which may create, manipulate and remove R objects within an environment. Having first-class environments means that expressions can also be used to create, manipulate and delete environments like any other R object. Moreover, expressions can be evaluated in any environment created by the programmer.

In our implementation, a sequence of connected nodes in a binary search tree is represented by a sequence of nested environments. Such a series of nested environments is equivalent to a stack, where a PUSH-operation corresponds to nesting a new environment and a POP-operation ensures that the next expression will be evaluated in the last-pushed environment. Since environments are nested, expressions evaluated in a child node have read access to information stored in the parent node. Pseudocode for the CHOICEPOINT object is given in Algorithm 6. Expressions are denoted with greek letters ψ or ϕ, environments are denoted as $\mathcal{E}$ and :: is the scope resolution operator. The symbol $\mathcal{S}$ denotes a formal stack. We denote the result of evaluating an expression ϕ in an environment $\mathcal{E}$ as $\phi(\mathcal{E})$. One can think of ϕ as a subroutine which alters the internal state of $\mathcal{E}$. It is also possible for ϕ to generate a return value (by issuing a `return` statement) which is pushed to the enveloping environment, similar to the action of a standard function.

Algorithm 6 Choicepoint object. ϕ_j and ψ are expressions, $\mathcal{E}$ and $\mathcal{E}'$ environments :: is the scope resolution operator and $\mathcal{S}$ a stack.

Struct CHOICEPOINT $(\phi_0, \phi_l, \phi_r, \psi)$
  $\mathcal{S} \leftarrow$ NEWSTACK
  $\mathcal{E} \leftarrow$ NEWENVIRONMENT
  $\mathcal{E}$ :: treatedleft $\leftarrow$ FALSE
  $\mathcal{E}$ :: treatedright $\leftarrow$ FALSE
  $\phi_0(\mathcal{E})$ ▷ ϕ_0 Initialize root node
  PUSH$(\mathcal{E}, \mathcal{S})$
  Method SEARCHNEXT
    $\mathcal{E} \leftarrow$ POP$(\mathcal{S})$ ▷ POP returns NULL if stack is empty
    while $\psi(\mathcal{E}) \in \{$FALSE, NULL$\} \wedge \mathcal{E} \neq$ NULL **do**
      if $\neg\mathcal{E}$ :: treatedleft **then**
        $\mathcal{E}' \leftarrow \mathcal{E}$ ▷ Create child node
        $\phi_l(\mathcal{E}')$ ▷ Treat child node
        $\mathcal{E}$ :: treatedleft $\leftarrow$ TRUE ▷ Mark parent node
        PUSH$(\mathcal{E}, \mathcal{S})$
        PUSH$(\mathcal{E}', \mathcal{S})$
      else if $\neg\mathcal{E}$ :: treatedright **then**
        $\mathcal{E}' \leftarrow \mathcal{E}$
        $\phi_r(\mathcal{E}')$
        $\mathcal{E}$ :: treatedright $\leftarrow$ TRUE
        PUSH$(\mathcal{E}, \mathcal{S})$
        PUSH$(\mathcal{E}', \mathcal{S})$
      $\mathcal{E} \leftarrow$ POP$(\mathcal{S})$
    return $\mathcal{E}$
  EndMethod
EndStruct

To construct a CHOICEPOINT object, the user provides an expression ϕ_0 to initialize the root node, expressions ϕ_l and ϕ_r to be evalated at left and right child nodes and an expression ψ to evaluate a node. The initialisation expression usually consists of a number of variable declarations. Expressions ϕ_l and ϕ_r alter the state of left or right child node, any returned values are ignored. The expression ψ serves two purposes. First of all, it judges a node $\mathcal{E}$and must return one of the following values:

$$\psi(\mathcal{E}) = \begin{cases} \text{TRUE} & \text{if environment } \mathcal{E} \text{ contains a solution} \\ \text{FALSE} & \text{if environment } \mathcal{E} \text{ cannot lead to a solution} \\ \text{NULL} & \text{if environment } \mathcal{E} \text{ contains no solution} \end{cases} \tag{28}$$

Secondly, ψ may be used to update weights and to prepare the variables in a node for output. The method SEARCHNEXT generates nodes in the binary tree, depth-first and returns the (contents of) the first environment corresponding to a solution. If CP is the instance of a CHOICEPOINT object, then each call to CP::SEARCHNEXT will return a new, and better solution, untill all solutions are found, in which case NULL is returned.

As an example, Figure 8 shows how to implement the branch-and-bound algorithm for error localization. The `choicepoint` function accepts the following arguments:

- `isSolution` : An R expression, corresponding to ψ of Eqn. (28).
- `choiceLeft` : An R expression, to be executed for left child nodes (ϕ_l) .
- `choiceRight` : An R expression, to be executed for left child nodes (ϕ_r).
- `...` : Named arguments, to initialize the root node (ϕ_0).

In Figure 8 the top environment (root node) receives an edit matrix `E`, a record `r`, a vector of variable names that have yet to be treated (`totreat`), a logical vector indicating whether a variable should be altered or not (`adapt`), a weight vector `weight` with reliability weights for each variable, and the weight `wsol` of the current solution is initialized to the maximum possible weight.

The expression `isSolution` first computes the weight of the current solution by adding all elements of `weight` for which `adapt==TRUE`. Next, it checks if the editmatrix is unfeasible, or if the current weight exceeds the weight of the last found solution. Since `wsol` is initialized on the maximum weight, the latter can only happen when at least one solution has been found. If either condition is met, the branch must be pruned, so `FALSE` is returned. Ortherwise, it is checked whether any variables are left to treat, if so, the expression ends, otherwise. The solution weight in the top environment is set (using the `<<-` operator) and `TRUE` is returned. Before returning, output is prepared by copying the variable `adapt` from the enveloping environment, and removing the empty vector `totreat`.

In `choiceLeft`, the first variable to be treated is chosen and its value replacen in the editmatrix. The value of `E` in the call to `substValue` is copied automagically from the enveloping environment which by construction holds the parent node of the node under treatment. For the same reason the assigning the indexed value of `adapt` the value `FALSE` works. The value corresponding to the variable under treatement in `adapt` is set to `FALSE` since a variable who's value is substituted in the editmatrix is assumed correct in the treated node. Finally, the vector of variables to be treated is updated.

In `choiceRight`, the same administrative chores are performed as in the `choiceLeft`. The only difference is that in the right node a variable is eliminated from the editmatrix, and therefore assumed incorrect.

The editmatrix used here corresponds to edit e_1 and e_2 of Eqn. (15), which are the edits violated by the record $(x = 2, y = -1)$. As expected, a single call to `cp$searchNext()` yields the correct solution.

```
> cp <- choicepoint(
+     isSolution = {  # check for solution or pruning
+         w <- sum(weight[adapt])
+         if ( isObviouslyInfeasible(E) || w > wsol ) return(FALSE)
+         if (length(totreat) == 0){
+             wsol <<- w
+             adapt <- adapt
+             rm(totreat)
+             return(TRUE)
+         }
+     },
+     choiceLeft = {  # things to do in the left node
+         .var <- totreat[1]
+         E <- substValue(E, .var , r[.var])
+         adapt[.var] <- FALSE
+         totreat <- totreat[-1]
+     },
+     choiceRight = { # things to do in the right node
+         .var <- totreat[1]
+         E <- eliminateFM(E, .var)
+         adapt[.var] <- TRUE
+         totreat <- totreat[-1]
+     },
+     # Initialize variables in root node
+     E = editmatrix(c("y > x-1 ","y > -x+3")),
+     r = c(x=2,y=-1),
+     totreat = c("x","y"),
+     adapt = c(x=FALSE, y=FALSE),
+     weight = c(1,1),
+     wsol = 2
+ )
> cp$searchNext()

$w
[1] 1

$E
Edit matrix:
   x  y Ops CONSTANT
e1 0 -1   <       -1

Edit rules:
e1 : 1 < y

$adapt
    x     y
FALSE  TRUE
```

Figure 8: Solving a simple error localization problem using the `choicepoint` object directly.

5 Conclusions

The `editrules` package offers a convenient interface to define and manipulate sets of linear (in)equality restrictions. Linear restrictions can be entered textually for for automated translation to matrix form or *vice versa*. Edit sets can be manipulated by value substitution or variable elimination, through a newly developed fast routine for Fourier-Motzkin elimination. The latter routine also allows the user to check sets of linear (in)equalities for internal consistency.

The package offers the ability to efficiently identify the edit rules violated by a set of records. Moreover, based on the Fellegi-Holt assumption, one can localize the erroneous fields in edit-violating records. The error localization routines are based on a choicepoint-programming paradigm which is exported to user space, providing users with a flexible and easy to use interface for solving binary programming problems.

References

Černikov, S. N. (1963). The solution of linear programming problem by elimination of unknowns. In *Soviet mathematics DOKLADY 2*.

De Waal, T. (2003). *Processing of erroneous and unsafe data*. Ph. D. thesis, Erasmus University Rotterdam.

De Waal, T., J. Pannekoek, and S. Scholtus (2011). *Handbook of statistical data editing*. Wiley handbooks in survey methodology. Hoboken, New Jersey: John Wiley & Sons.

Farkas, G. (1902). Über die theorie der einfachen ungleichungen. *Journal für die Reine und Angewandte Mathematik 124*, 1–27.

Fellegi, I. P. and D. Holt (1976). A systematic approach to automatic edit and imputation. *Journal of the Americal Statistical Association 71*, 17–35.

Fourier, J. (1826). Solution d'une question particulière du calcul des inégalités. *Ouevres II*, 317–328.

Kohler, D. (1967). Projections of convex polyhedral sets. Technical Report ORC 67-29, University of California, Berkely.

Kuhn, H. W. (1956). Solvability and consistency for linear equations and inequalities. *The American Mathematical Monthly 63*, 217–232.

Mart-Oliet, N. and J. Mesguer (2002). Rewriting logic: Roadmap and bibliography. *Theoretical Computer Science 285*, 121–154.

Moreau, P.-E. (1998, June). A choice-point library for backtrack programming. In *JICSLP'98 Post-Conference Workshop on Implementation Technologies for Programming Languages based on Logic*.

Motzkin, T. S. (1936). Beitrage zur Theorie der Linearen Ungleichungen. Inaugural Dissertation, Basel-Jerusalem.

Partington, V. (1997). Implementation of an imperative programming language with backtracking. Technical Report P9714, University of Amsterdam, Programming research group.

R Development Core Team (2011). *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. ISBN 3-900051-07-0.

Scholtus, S. (2008). Algorithms for correcting some obvious inconsistencies and rounding errors in business survey data. Technical Report 08015, Statistics Netherlands, Den Haag. The papers are available in the inst/doc directory of the R package or via the website of Statistics Netherlands.

Scholtus, S. (2009). Automatic correction of simple typing error in numerical data with balance edits. Technical Report 09046, Statistics Netherlands, Den Haag. The papers are available in the inst/doc directory of the R package or via the website of Statistics Netherlands.

Schrijver, A. (1998). *Theory of linear and integer programming.* Wiley-Interscience series in descrete mathematics and optimization. New York: John Wiley and Sons.

Van der Loo, M., E. de Jonge, and S. Scholtus (2011). *deducorrect: Deductive correction of simple rounding, typing and sign errors.* R package version 0.9-2.

Vittek, M. (1996). A compiler for nondeterministic term rewriting systems. In H. Ganziger (Ed.), *Proceedings of RTA'96*, Volume 1103 of *Lecture Notes in Computer Science*, New Brunswick (New Jersey), pp. 154–168. Springer-Verlag.

Williams, H. (1986). Fourier's method of linear programming and its dual. *The American mathematical monthly 93*, 681–695.

Index