1 Introduction

There are several approaches for including constraints into heuristics; see Chapter 12 of [Gilli et al. 2011]. The notes in this vignette give examples for simple repair mechanisms. These can be called in DEopt, GAopt and PSopt through the repair function; in LSopt/TAopt, they could be included in the neighbourhood function.

```r
> set.seed(112233)
> options(digits = 3)
```

2 Upper and lower limits

Suppose the solution \( x \) is to satisfy \( \text{all}(x \geq lo) \) and \( \text{all}(x \leq up) \), with \( lo \) and \( up \) being vectors of \( \text{length}(x) \).

2.1 Setting values to the boundaries

One strategy is to replace elements of \( x \) that violate a constraint with the boundary value. Such a repair function can be implemented very concisely. An example:

```r
> up <- rep(1, 4L)
> lo <- rep(0, 4L)
> x <- rnorm(4L)
> x
[1] 2.127 -0.380 0.167 1.600

Three of the elements of \( x \) actually violate the constraints.

```r
> repair1a <- function(x, up, lo)
  pmin(up, pmax(lo, x))
> x
[1] 2.127 -0.380 0.167 1.600
```
\texttt{x}
}

\texttt{> repair1c <- function(x, up, lo) {}
  xadjU <- x - up
  xadjU <- xadjU + abs(xadjU)
  xadjL <- lo - x
  xadjL <- xadjL + abs(xadjL)
  x - (xadjU - xadjL)/2
}}

The function \texttt{repair1b} uses comparisons to replace only the relevant elements in \texttt{x}. The function \texttt{repair1c} uses the ‘trick’ that
\[
p\max(x,y) = \frac{x+y}{2} + \frac{|x-y|}{2},
\]
\[
p\min(x,y) = \frac{x+y}{2} - \frac{|x-y|}{2}.
\]

\texttt{> repair1a(x, up, lo)}
\begin{verbatim}
[1] 1.000 0.000 0.167 1.000
\end{verbatim}

\texttt{> repair1b(x, up, lo)}
\begin{verbatim}
[1] 1.000 0.000 0.167 1.000
\end{verbatim}

\texttt{> repair1c(x, up, lo)}
\begin{verbatim}
[1] 1.000 0.000 0.167 1.000
\end{verbatim}

\texttt{> trials <- 5000L}
\texttt{> strials <- seq_len(trials)}
\texttt{> system.time(for(i in strials) y1 <- repair1a(x, up, lo))}
\begin{verbatim}
user  system elapsed
0.047 0.000 0.048
\end{verbatim}

\texttt{> system.time(for(i in strials) y2 <- repair1b(x, up, lo))}
\begin{verbatim}
user  system elapsed
0.012 0.000 0.012
\end{verbatim}

\texttt{> system.time(for(i in strials) y3 <- repair1c(x, up, lo))}
\begin{verbatim}
user  system elapsed
0.011 0.000 0.011
\end{verbatim}

The third of these functions would also work on matrices if \texttt{up} or \texttt{lo} were scalars.

\texttt{> X <- array(rnorm(25L), dim = c(5L, 5L))}
\texttt{> X}
\begin{verbatim}
[1,] 0.1962   0.4340  -2.1550  -1.5881  -1.0290
[2,] 0.2284   1.2310   0.9750   0.0682   1.8180
[3,] -1.1492   0.5800  -0.7110  -0.4457  -1.3150
[4,] -0.0712   0.2460   0.6280  1.4662   0.5110
[5,] -0.5619   0.3880  -0.1360  -0.8412  1.3370
\end{verbatim}
The speedup comes at a price, of course, since there is no checking (e.g., for NA values) in `repair1b` and `repair1c`. We could also define new functions `pmin2` and `pmax2`.

```r
> pmax2 <- function(x1, x2) ((x1 + x2) + abs(x1 - x2)) / 2
> pmin2 <- function(x1, x2) ((x1 + x2) - abs(x1 - x2)) / 2
```

A test follows.

```r
> x1 <- rnorm(100L)
> x2 <- rnorm(100L)
> t1 <- system.time(for (i in S1) z1 <- pmax(x1, x2) )
> t2 <- system.time(for (i in S1) z2 <- pmax2(x1, x2))
> t1[[3L]]/t2[[3L]] ## speedup
[1] 2.78
> all.equal(z1, z2)
[1] TRUE
```

```r
> t1 <- system.time(for (i in S1) z1 <- pmin(x1, x2) )
> t2 <- system.time(for (i in S1) z2 <- pmin2(x1, x2))
> t1[[3L]]/t2[[3L]] ## speedup
[1] 2.89
> all.equal(z1, z2)
[1] TRUE
```

One downside of this repair mechanism is that a solution may quickly become stuck at the boundaries (but of course, in some cases this is exactly what we want).

### 2.2 Reflecting values into the feasible range

The function `repair2` reflects a value that is too large or too small around the boundary. It restricts the change in a variable `x[i]` to the range `up[i] - lo[i]`.

```r
> repair2 <- function(x, up, lo) {
    done <- TRUE
    e <- sum(x - up + abs(x - up) + lo - x + abs(lo - x))
    if (e > 1e-12) done <- FALSE
    r <- up - lo
    while (!done) {
```
\[ \text{adjU} <- x - \text{up} \]
\[ \text{adjU} <- \text{adjU} + \text{abs(adjU)} \]
\[ \text{adjU} <- \text{adjU} + r - \text{abs(adjU - r)} \]
\[ \text{adjL} <- \text{lo} - x \]
\[ \text{adjL} <- \text{adjL} + \text{abs(adjL)} \]
\[ \text{adjL} <- \text{adjL} + r - \text{abs(adjL - r)} \]
\[ x <- x - (\text{adjU} - \text{adjL})/2 \]
\[ e <- \text{sum}(x - \text{up} + \text{abs(x - up)} + \text{lo} - x + \text{abs(lo - x)}) \]
\[ \text{if (e < 1e-12)} \]
\[ \text{done} <- \text{TRUE} \]
\}\]
\[ x \]
\}
\>
\> x
\>
\>[1] 2.127 -0.380 0.167 1.600
\>
\> repair2(x, up, lo)
\>
\>[1] 0.873 0.380 0.167 0.600
\>
\> system.time(for (i in strials) y4 <- repair2(x,up,lo))
\>
\> user system elapsed
\>[0.030 0.000 0.031]

### 2.3 Adjusting a cardinality limit

Let \( x \) be a logical vector.

\>
\> T <- 20L
\> x <- logical(T)
\> x[runif(T) < 0.4] <- TRUE
\>
\> x
\>

Suppose we want to impose a minimum and maximum cardinality, \( k_{\text{min}} \) and \( k_{\text{max}} \).

\>
\> kmax <- 5L
\> kmin <- 3L

We could use an approach like the following (for the definition of resample, see ?sample):

\>
\> resample <- function(x, ...) x[sample.int(length(x), ...)]
\> repairK <- function(x, kmax, kmin) {
\> \hspace{1em} sx <- \text{sum}(x)
\> \hspace{1em} if (sx > kmax) {
\> \hspace{2em} i <- resample(\text{which}(x), sx - kmax)
\> \hspace{2em} x[i] <- FALSE
\> \hspace{1em} } else if (sx < kmin) {
\> \hspace{2em} i <- resample(\text{which}(!x), kmin - sx)
\> \hspace{2em} x[i] <- FALSE
\> \}
\>
\> repairK(x, kmax, kmin)
> x[i] <- TRUE
>
}  
x
}

> printK <- function(x)
    cat(paste(ifelse(x, "o", "."), collapse = ""),
        "-- cardinality", sum(x), "\n")

For kmax:
> for (i in 1:10) {
    if (i==1L) printK(x)
    x1 <- repairK(x, kmax, kmin)
    printK(x1)
}

.00.00......00...0... -- cardinality 8
.00.00......00...0... -- cardinality 5
.0.00....00...0... -- cardinality 5
.00.00......00...0... -- cardinality 5
.00.00......00...0... -- cardinality 5
.00.00......00...0... -- cardinality 5
.00.00......00...0... -- cardinality 5
.00.00......00...0... -- cardinality 5
.00.00......00...0... -- cardinality 5
.00.00......00...0... -- cardinality 5

For kmin:
> x <- logical(T); x[10L] <- TRUE
> for (i in 1:10) {
    if (i==1L) printK(x)
    x1 <- repairK(x, kmax, kmin)
    printK(x1)
}

.........0.......... -- cardinality 1
.........0.......... -- cardinality 3
.........0.......... -- cardinality 3
.........0.......... -- cardinality 3
.........0.......... -- cardinality 3
.........0.......... -- cardinality 3
.........0.......... -- cardinality 3
.........0.......... -- cardinality 3
.........0.......... -- cardinality 3
.........0.......... -- cardinality 3

References


5