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 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
```

We see that indeed all values greater than 1 are replaced with 1, and those smaller than 0 become 0. Two other possibilities that achieve the same result:

```r
> repair1b <- function(x, up, lo) {
  ii <- x > up
  x[ii] <- up[ii]
  ii <- x < lo
  x[ii] <- lo[ii]
}

> repair1a(x, up, lo)
[1] 1.000 0.000 0.167 1.000

> repair1b(x, up, lo)
x
}
> 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 repair1b uses comparisons to replace only the relevant elements in x. The function repair1c uses the `trick` that

\[
p_{\max}(x, y) = \frac{x+y}{2} + \frac{|x-y|}{2},
\]

\[
p_{\min}(x, y) = \frac{x+y}{2} - \frac{|x-y|}{2}.
\]

> repair1a(x, up, lo)
[1] 1.000 0.000 0.167 1.000
> repair1b(x, up, lo)
[1] 1.000 0.000 0.167 1.000
> repair1c(x, up, lo)
[1] 1.000 0.000 0.167 1.000

> trials <- 5000L
> strials <- seq_len(trials)
> system.time(for(i in strials) y1 <- repair1a(x, up, lo))
user  system  elapsed
 0.046   0.000   0.046

> system.time(for(i in strials) y2 <- repair1b(x, up, lo))
user  system  elapsed
 0.012   0.000   0.012

> system.time(for(i in strials) y3 <- repair1c(x, up, lo))
user  system  elapsed
 0.01   0.00   0.01

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

> X <- array(rnorm(25L), dim = c(5L, 5L))
> X
[1,]  0.1962  0.434 -2.155 -1.588 -1.029
[2,]  0.2284  1.231  0.975  0.068  1.818
[3,] -1.1492  0.580 -0.711 -0.445 -1.315
[4,] -0.0712  0.246  0.628  1.466  0.511
[5,] -0.5619  0.388 -0.136 -0.841  1.337

2
> repair1c(X, up = 0.5, lo = -0.5)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,]</td>
<td>0.1962</td>
<td>0.434</td>
<td>-0.500</td>
<td>-0.5000</td>
<td>-0.5</td>
</tr>
<tr>
<td>[2,]</td>
<td>0.2284</td>
<td>0.500</td>
<td>0.500</td>
<td>0.0682</td>
<td>0.5</td>
</tr>
<tr>
<td>[3,]</td>
<td>-0.5000</td>
<td>0.500</td>
<td>-0.5000</td>
<td>-0.4457</td>
<td>-0.5</td>
</tr>
<tr>
<td>[4,]</td>
<td>-0.0712</td>
<td>0.246</td>
<td>0.500</td>
<td>0.5000</td>
<td>0.5</td>
</tr>
<tr>
<td>[5,]</td>
<td>-0.5000</td>
<td>0.388</td>
<td>-0.136</td>
<td>-0.5000</td>
<td>0.5</td>
</tr>
</tbody>
</table>

The speedup comes at a price, of course, since there is no checking (eg, 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 strials) z1 <- pmax(x1,x2) )
> t2 <- system.time(for (i in strials) z2 <- pmax2(x1,x2))
> t1[[3L]]/t2[[3L]] ## speedup

[1] 3

> all.equal(z1, z2)

[1] TRUE

> t1 <- system.time(for (i in strials) z1 <- pmin(x1,x2) )
> t2 <- system.time(for (i in strials) z2 <- pmin2(x1,x2))
> t1[[3L]]/t2[[3L]] ## speedup

[1] 3

> 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) {
```
adjU <- x - up
adjU <- adjU + abs(adjU)
adjU <- adjU + r - abs(adjU - r)

adjL <- lo - x
adjL <- adjL + abs(adjL)
adjL <- adjL + r - abs(adjL - r)

x <- x - (adjU - adjL)/2
e <- sum(x - up + abs(x - up) + lo - x + abs(lo - x))
if (e < 1e-12)
  done <- 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
   0.029   0.000   0.029

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 \texttt{resample}, see \texttt{?sample}):

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

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)
}

References