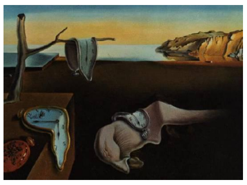
Journal of Statistical Software

July 2006, Volume 16, Issue 6. *http://www.jstatsoft.org/*



An Accept-and-Reject Algorithm to Sample from a Set of Permutations Restricted by a Time Constraint

Johannes Hüsing
Koordinierungszentrum für Klinische Studien Heidelberg

Abstract

A modification of an accept-and-reject algorithm to sample from a set of restricted permutations is proposed. By concentrating on a special class of matrices obtained by restriction of the permutation in time, assuming the objects to be permuted to be events in time, the modified algorithm's running time can be shown to be linear instead of geometric in the number of elements. The implementation of the algorithm in the language R is presented in a Literate Programming style.

Keywords: accept-and-reject, permutation test, dynamic allocation.

1. Introduction

The problem of uniform sampling from a finite set of size N can always be solved by enumerating the elements of the set and then sample the nth element, where n is sampled from $\{1, \ldots, N\}$ with equal probability. If the set is large, however, this brute force method is infeasible. Instead, one looks for stepwise methods where the size of each subset chosen by a step is known. One example of this strategy is to sample from the set of permutations of $|v|$ elements, where one samples element by element without replacement with probability $|v| - k + 1$ in the kth step. Another obvious example is sampling from the list of all samples of a given size with replacement.

One example where it is computationally expensive to calculate the size of the set, let alone to construct a list of its elements, can be formulated as follows. Let $E \in V \times V$ be a relation that denotes a restriction on the set of permutations of V, the set of the components of vector v. A permutation $\Phi(v)$, where $\phi(i)$ denotes the position of the ith element of $\Phi(v)$ is only feasible according to E if $(i, \phi(i)) \in E$. The number of possible permutations that obey this restriction is also named the *permanent* Π (Minc 1978) of the adjacency matrix A of the graph formed by the elements of v as nodes and E as the set of edges (and having 1s in the main diagonal). A permanent can be developed like a determinant of a matrix, with the rule $\Pi(A) = \sum_{j=1}^{I} \Pi(A_{\check{i}\check{j}}) a_{ij}$, where $A_{\check{i}\check{j}}$ is the minor obtained by deleting the ith row and jth column from A. If the permanent were calculated in acceptable time, one could pick a row i of A and sample the jth row with sampling weight $\Pi(A_{\check{i}\check{j}}) a_{ij} / \Pi(A)$. The problem of calculating $\Pi(A)$, however, has been shown to be of the #P class (Valiant 1979). Descending through sampling steps knowing the exact probability is therefore as infeasible as building the whole list.

2. Accept-and-reject

Generating the whole list, however, is not a necessary condition for sampling from it. One possibility of sampling in a stepwise approach is by the accept-and-reject algorithm. One needs a bound from above for the permanent, $M(A)$ with $\Pi(A) \leq M(A)$, where $M(A) = \Pi(A) = a_{11}$ if A is of dimension 1×1.

An important requirement for M to qualify for accept-and-reject is the inequality:

$$M(A) \geq \sum_{j=1}^{I} M(A_{\check{i}\check{j}}) \quad \forall \quad i \in \{1, \dots, I\} \tag{1}$$

Then an accept-and-reject algorithm can be formulated as follows:

1. Pick the ith row from the 0-1 matrix.
2. Sample the (i, j)th element from all elements in the row with probability $M(A_{\check{i}\check{j}})/M(A)$.
3. If no element is selected (which happens with probability $1 - \frac{1}{M(A)} \sum_{j=1}^{I} M(A_{\check{i}\check{j}})$), the algorithm terminates with failure and has to be restarted to obtain a permutation.
4. If A is of dimension 1×1, and $A = (1)$, then the algorithm terminates with success.
5. Otherwise, mark a_{ij}, set $A \leftarrow A_{\check{i}\check{j}}$, and reiterate the algorithm.

At the end, all marked elements denote the permutation. It is straightforward to show that all elements are sampled with equal probability, considering all envelope probabilities occur in the nominator as well as in the denominator of the product of dependent probabilities.

A relatively tight bound M for the permanent was found by Bregman (1973). In Huber (2003) a modification of this bound, $M(A)$ was presented that satisfies the requirement of inequality (Equation 1).

There are several suggestions to use Markov Chain Monte Carlo (MCMC) methods to sample from a random walk through the set of permutations (Efron and Petrosian 1999) or "near-perfect matches" (permutations with one element left out) (Jerrum, Sinclair, and Vigoda 2001). Also, the bootstrap has been suggested as a time-saving method which sacrifices precision on the sampling probabilities (Efron and Petrosian 1999). The relative merit of the accept-and-reject algorithm compared to the MCMC methods is appraised in Huber (2003).

3. Subclass of adjacency matrices

3.1. Description of subclass

The algorithm leaves the question open how to select i. The suggestion in Huber (2003) is to select the i in their natural row by row order. For some subclasses of 0-1 matrices, there are possibilities to speed up the algorithm by judiciously selecting a row. This is the main topic of the approach presented here. The subclass of matrices considered here is the class of symmetric 0-1 matrices with a "tube stucture":

$$i < j \Leftrightarrow a_{i\,j+1} \leq a_{i\,j} \leq a_{i+1\,j}. \tag{2}$$

This class of matrices is obtained by a crude similarity matrix on ordered elements in one-dimensional space: $a_{i\,j}$ is equal to 1 iff $|i - j| < \Delta$.

3.2. Motivation of subclass of matrices

The problem at hand is motivated by the problem of finding appropriate tests for dynamic sequential allocation designs (Taves 1974). In an experiment with sequential inclusion of the observational units, as is typical in a clinical trial, the idea of dynamic allocation is to determine the treatment group by minimizing the imbalance between the treatment groups with respect to predefined covariates. The price to pay for optimal balance is the lack of a random element in the allocation process, and therefore of a straightforward and interpretable basis for statistical inference. One suggestion to overcome this difficulty was made in Simon (1979): For a permutation test, the order of arrival of observational units can be permuted. Applying the same allocation procedure to the permuted sequences will most probably lead to alternative allocations and therefore to a null distribution of a test statistic. Although the idea of permuting the sequence was rejected in the same article as based on a "questionable assumption that the sequence of patient arrivals is random", it has been employed with apparent success (Ohashi 1990). While the rank in which the patient arrives in a trial is not informative for the treatment allocated, it may not, however, be independent of the success of the treatment. Temporal effects on the treatment outcome are not uncommon, for instance seasonality, experience with a new treatment scheme, or even cohort effects in a long trial. A restricted randomization based on the time lag between the accrual of different observational units can control the most obvious problems of sequence permutation.

The strategy of sampling is named "separating accept-and-reject". It employs a variation of the strategy of "clotting" which is described in detail in Hüsing (2006). If a matrix has a tube structure, the strategy aims at obtaining minors that have a block-diagonal structure. The criteria by which the optimal row to separate the matrix is selected are described in Section 4.4.

A very similar structure of admissible permutations is described in Efron and Petrosian (1999), where the application is a permutation test on data that can only be observed if they fall into an interval $[u_i; v_i]$ specific to the observation. A transposition of elements i and j is permissible only if the realization y_i lies in the interval $[u_j; v_j]$ and vice versa. It has been shown (Diaconis, Graham, and Holmes 1999) that the indicator matrix for permissible elements also has a tube structure, though not necessarily symmetric.

4. The code

4.1. Introductory remarks

The code has been generated along the lines of the Literate Programming paradigm Knuth (1992), using Edward Ream's Literate Editor with Outlines (http://leo.sourceforge.net/) and woven with Norman Ramsey's Noweb (Ramsey 1994).

The code is provided by the package **resper**, written in the R system for statistical computing (R Development Core Team 2006) and available from the Comprehensive R Archive Network at http://CRAN.R-project.org/.

4.2. Auxiliary functions

The recursive function

The novelty introduced by Huber is the variation of a bound of the permanent from above Bregman (1973) that works as a probability: The value of a function of a matrix is always at least as big as the sum of the values of the function of the minors, developed around a column of the matrix. It is the recursive function on the dimension n of the matrix:

$$G(n) := \begin{cases} \mathrm{e} & \text{for } n = 1 \\ G(n-1) + 1 + 0.5/G(n-1) + 0.6/G(n-1)^2 & \text{for } n > 1 \end{cases} \tag{3}$$

$\langle$*@others*$\rangle\equiv$

```
HuberGInner <- function(n) {
    if (n == 1)
        exp(1)
    else {
        gnm <- HubergGInner(n - 1)
        gnm + 1 + 0.5/gnm + 0.6/gnm/gnm
    }
}
```

R does not like recursion too much, so a non-recursive version is employed here:

$\langle$***$\rangle\equiv$

```
HuberGNonRecursive <- function(n) {
    for (i in 1:n) {
        if (i == 1)
            gnm <- exp(1)
        else gnm <- gnm + 1 + 0.5/gnm + 0.6/gnm/gnm
    }
    gnm
}
```

A wrapper for the function is used that checks for valid entries:

$\langle * \rangle {+}{\equiv}$

```
HuberGE <- function(n) {
    if (n > floor(n)) {
        stop("n must be of integer value")
    }
    else if (n < 0) {
        stop("n must be at least 1")
    }
    else if (n == 0)
        0
    else {
        HuberGNonRecursive(n)/exp(1)
    }
}
```

The selection probability

The actual probability is given by

$$M(A) \quad := \quad \prod_i G(c_i), \tag{4}$$

where c_i is the sum of the ith column (i.e. the number of ones in it).

$\langle * \rangle {+}{\equiv}$

```
### Formula (3)
PermBound <- function(mat) {
        prod(sapply(apply(mat, 2, sum), HuberGE))
}
```

As said, this is a probability because

$$M(A) \geq \sum_j a_{ij} M(A(\breve{\imath}, \breve{\jmath})) \quad \forall \quad i. \tag{5}$$

The `drop=FALSE` argument retains the array structure even if the dimension of the matrix is 1 (see Hornik (2006)). The vector of the above sum elements is needed to sample a row for a given column. The function pastes the difference of the sum of these elements from one to the end.

$\langle * \rangle {+}{\equiv}$

```
HuberProbs <- function(at, i) {
    n <- nrow(at)
    Mrt <- sapply(1:n, function(j) {
        if (at[i, j] > 0) {
            PermBound(at[-i, -j, drop=FALSE])
        }
        else {
            0
          }
        })
        c(Mrt, PermBound(at) - sum(Mrt))
      }
```

The most important application of a tube matrix in this context is the one that determines permutability between elements that are close to each other. Therefore, a function is desirable that returns such a matrix from an ordered sequence $(t_i)_{i\in\{1,\ldots,n\}}$ and a lag Δ.

The rows and columns of the matrix correspond to the elements in the ordered seqence, and an element a_{ij} is set to TRUE exactly when $|t_i - t_j| < \Delta$. The Boolean entries in the matrix are converted to 0s and 1s when arithmetic functions are applied to them.

⟨*⟩+≡
```
  WithinDeltaMat <- function(seq, delta) {
    outer(seq, seq+delta, '<') & outer(seq+delta, seq, '>')
  }
```

4.3. Original accept and reject algorithm

The following function runs the algorithm described in Huber (2003). It picks the first row of the matrix and randomly selects a column, where the sample elements are weighted by the probability formula given above. If the last element of the sampling vector is selected (the one that has fills up the probability sum to one), the whole sample is rejected and restarted. This is accomplished by a `repeat` loop encapsulated by another `repeat` loop, and the Boolean variable `accept`. The outermost loop is broken out of iff `accept==TRUE`.

⟨*⟩≡
```
  ###  Algorithm from Figure 2
  ResperByrow <- function(mat) {
    reject <- 0
    repeat { # the outer loop; exited after proper sample
      ⟨try and sample until proper permutation is chosen⟩
    }
    list(perm=perm, reject=reject)
  }
```

The accept flag is initialized to TRUE, the row sums and dimension are calculated, and the indices are initialized as a reference. These are needed because the matrix is reduced to its minors later, to maintain the reference from the column of a minor to the column of the matrix. Then the inner loop is entered. This `repeat` loop can be broken out of successfully (if a permutation is complete) or with a failure (if a permutation failed to be sampled). If it is exited successfully, exit this loop also, if not, increment the variable `reject`, re-initialize and re-enter the inner loop. The variable `reject` records the number of failures. It is returned with the permutation in a common data structure.

⟨*try and sample until proper permutation is chosen*⟩≡
```
  accept <- TRUE
  at <- mat
  rt <- apply(at, 1, sum)
  n <- nrow(at)
  rownames(at) <- colnames(at) <- 1:n
  indizes <- 1:n
  perm <- NULL
  repeat { # the inner loop; exited on failed sample for retry
    ⟨see how far you get without failing⟩
  }
  if (accept) break else {
    reject <- reject+1
  }
```

Failures within this loop can occur if the matrix includes a zero row, or if the random sampling process selects the "column" beyond the matrix.

If a column of the matrix is sampled, the vector of permutation is extended by the selected column. Note that the column number has to denote the column number of the original matrix, not of the current, reduced, matrix. Therefore, the column numbers of the original matrix are passed as colnames and referenced when the permanent is extended.

⟨see how far you get without failing⟩≡

```
  ##  1:n doesn't change, even if n is changed
  ##  within the loop
  for (i in 1:n) {
    ##  reject if any row with only zeros
    if (prod(rt)==0) {
      accept <- FALSE
      break
    ##  trap special case of 1 by 1 matrix
    } else if (n==1) {
      elt <- 1
    } else {
      ⟨random sample column⟩
    }
    ##  update permutation vector
    perm <- c(perm, indizes[elt])
    if (n > 1) {
      ⟨reduce matrix to sampled minor⟩
      if (is.matrix(at)) {
        ##  re-calculate row sums and dimension
        rt <- apply(at, 1, sum)
        n <- nrow(at)
        ##  trap special case of 1 by 1 matrix
      } else {
        ##  don't reduce further, just
        ##  re-calculate row sums and dimension
        rt <- at
        n <- 1
      }
    }
  }
  break  #  proper permutation sampled
```

First, create the vector of sample weights (permanent bounds of all the minors), including the extra element that completes the sum so that it equals the permanent bound of the current matrix. Then, sample and reject if this extra element is selected.

⟨*random sample column*⟩≡

```
  Mrt <- HuberProbs(at, 1)
  elt <- sample(n+1, 1, prob=Mrt)
  if (elt==n+1) {
    accept <- FALSE
    break
  }
```

Both the matrix and the index vector have to have a column deleted.

⟨*reduce matrix to sampled minor*⟩≡

```

  at <- at[-1, -elt]
  indizes <- indizes[-elt]
```

4.4. The separating accept and reject algorithm

This function uses an accept-reject algorithm that tries to break up the matrix into a block-diagonal structure. It selects a row whose elimination will bring the matrix closer to a block-diagonal structure, samples an appropriate cell from the 1s in the row (denoting the position to which the element corresponding to the row will be shifted), eliminates the cell row and column from the matrix and reiterates the algorithm on the minor. The sampling weights are constructed by an envelope probability discovered by Huber (2003), which guarantees that the sum of weights over all minors is less than or equal to the weight of the matrix itself. If the algorithm samples the "less than" part, it will reject and restart the current attempt.

The modified algorithm does not take the rows in the given order, but picks a row by certain criteria.

The first criterion addresses possible separators, that is, sets of few columns that, when removed, leave the matrix with a block diagonal structure.

If one views the matrix as an adjacency matrix of a graph, the task is now to look for waists of the graph.

As one can rely on the matrix having a tube structure, a primitive algorithm is sufficient to look for waists in the corresponding graph. These columns are detected by counting the number of ones below the main diagonal. This selection pattern is suboptimal for non-symmetric tube matrices but should be sufficient for most purposes.

⟨*⟩≡

```
### selection criteria for separator
firstcrit <- function(mat) {
  n <- nrow(mat)
  lowerdiag <- outer(1:n, 1:n, ">=")
  ##  the most desirable is the column _after_ the one
  ##  with the least ones from the main diagonal
  apply(cbind(rep(1,n), mat[,1:n-1])*
        ##  do not choose columns with 1s to the bottom
        ##  therefore heavy weight to bottom row
        array(rep(c(rep(1, n-1), n), n), dim=c(n,n)) *
        lowerdiag, 2, sum)
}
```

If one aims to split a matrix into separate blocks, one would like the blocks to have the same size.

Therefore, the second criterion favors the middle rows and therefore is a convex, symmetric function of the row index.

⟨*⟩+≡
```
secondcrit <- function(mat) {
    n <- nrow(mat)
    (1:n) * (n:1)/(n + 1)/(n + 1) * 8
}
```

The row with the minimal sum of first and second criterion is selected.

⟨*⟩+≡
```
selectrow <- function(mat) {
  which.min(firstcrit(mat)-secondcrit(mat))
}
```

The main idea of speeding up the process is to select the rows to reduce the matrix such as to obtain minors with a block-diagonal structure.

If a block-diagonal structure is detected, the algorithm calls itself recursively on the blocks and pastes the results together to get the whole permutation.

Again, this is not a function that works on general symmetric 0-1 matrices.

Instead of globally searching for a block structure, it looks if subsequent rows have at least one zero in either column.

⟨*⟩+≡
```
seps <- function(mat) {
    if (dim(mat)[1] < 3)
        NULL
    else {
        n <- nrow(mat)
        sumoff <- c(sum(mat[2:n, 1] + mat[1, 2:n]), sapply(2:(n -
            1), function(i) {
            sum(mat[1:i, (i + 1):n]) + sum(mat[(i + 1):n, 1:i])
        }))
        which(sumoff == 0)
    }
}
```

The next row is selected according to the two criteria, and the column is sampled according to the weights obtained by equation 3. If the element beyond the matrix rows is selected, the current sample is rejected and restarted.

⟨*⟩+≡
```
ResperClotInner <- function(mat) {
  reject <- 0
  repeat {
    ⟨try and sample until proper permutation⟩
  }
  list(perm=perm, reject=reject)
}
```

The permutation structure is initialized as a two-column matrix, the first column denoting the row indices and the second the column indices. The outer wrapper function converts these to a permutation vector.

The acceptance flag is initialized to `TRUE`, the row sums and dimension are calculated. The rownames and the colnames are initialized to the indices if they are not present. If there are rownames and colnames already, do not overwrite them as the function can be called recursively.

Then, sample, reject, and try again until a proper sample is selected.

⟨*try and sample until proper permutation*⟩≡
```
perm <- c(NULL, NULL)
accept <- TRUE
n <- nrow(mat)
if (is.null(colnames(mat))) {
  colnames(mat) <- 1:n
  rownames(mat) <- 1:n
}
at <- mat
rt <- apply(at, 1, sum)
repeat {
  ⟨see how far you get without failing⟩
}
if (accept) {
  break
  }
else {
  reject <- reject+1
}
```

First, trap the special case where the matrix has only one dimension. In this case, fill the permanent structure with the last row and column index, and return successfully.

```
⟨see how far you get without failing⟩≡
  if (n==1) {
    perm <- rbind(perm, as.numeric(c(rownames(at), colnames(at))))
    break
  }
```

If the matrix contains only ones, one can sample from the unrestricted set of permutations.

```
⟨see how far you get without failing⟩+≡
  else if (prod(at)==1) {
    ⟨sample from unrestricted set of permutations⟩
    ## exit successfully
    break
  }
```

If there is a block-diagonal structure, call function recursively on the blocks.

```
⟨see how far you get without failing⟩+≡
  else if (length(seps <- seps(at))>0) {
    ⟨call function recursively on the blocks⟩
    ##  exit successfully
    break
  }
```

The next test is on the main diagonal containing a 0. By virtue of the tube structure of the original matrix, if any of its minors has a 0 in the main diagonal, this minor necessarily contains a rectangular submatrix of only 0s that includes either both the first row and the last column or the last row and the first column. This is a sufficient condition for the permanent of this minor being 0 (a result cited in Minc (1978)), which in turn is reason enough to reject and restart.

```
⟨see how far you get without failing⟩+≡
  else if (prod(diag(at))==0) {
    accept <- FALSE
    break
  }
```

Now that we've handled the special cases, let's treat the normal case.

```
⟨see how far you get without failing⟩+≡
  else {
    ⟨select row and column to delete⟩
    ##  update permutation structure
    perm <- rbind(perm, as.numeric(c(rownames(at)[i],
                                   colnames(at)[j])))
    ⟨reduce matrix to minors⟩
    if (dim(at)[1]==0) break
  }
```

In the unrestricted case, one can simply use R's `sample()` algorithm. which is used for the column index column of the permutation structure, which is then updated row-wise by the permutation of the current matrix.

⟨*sample from unrestricted set of permutations*⟩≡
```
  unrestricted <- array(as.numeric(c(rownames(at),
                                   sample(colnames(at),
                                          size=ncol(at)))),
                        dim=c(nrow(at),2))
  perm <- rbind(perm, unrestricted)
```

⟨*call function recursively on the blocks*⟩≡
```
  bstart <- c(1, seps+1)
  bend <- c(seps, n)
  for (i in (1:length(bstart))) {
    ## recursively call the function on the blocks
    a <- ResperClotInner(at[bstart[i]:bend[i],
                            bstart[i]:bend[i], drop=FALSE])
    perm <- rbind(perm, a$perm)
    reject <- reject+a$reject
  }
```

The row is sampled according to the optimality criteria. The column is sampled at random. If the element beyond the matrix columns is selected, the accept-reject algorithm rejects.

⟨*select row and column to delete*⟩≡
```
  i <- selectrow(at)
  Mrt <- HuberProbs(at, i)
  j <- sample(n+1, 1, prob=Mrt)
  if (j==n+1) {
    accept <- FALSE
    break
  }
```

⟨*reduce matrix to minors*⟩≡
```
  at <- at[-i,-j, drop=FALSE]
  n <- n-1
```

Finally, a wrapper function is written that reduces the permanent data structure to a single permanent vector, as in the function for the Huber algorithm.

⟨*⟩+≡
```
  ResperClot <- function(mat) {
    a <- ResperClotInner(mat)
    a$perm <- a$perm[order(a$perm[,1]), 2]
    a
  }
```

5. Computation time considerations

The running time of the algorithm is considered on the class of k-diagonal 0-1 matrices. These matrices correspond to equidistant accrual time points. The original accept-and-reject algorithm by Huber, under these circumstances, has an expected running time of $O(n^{1.5}k^{.5n/k}5.3^{n/k})$.

The separating accept-and-reject operates recursively, therefore its running time can be obtained recursively as well. The result can eventually be given in closed form. Consider first that it takes at most k steps to cut a k-diagonal matrix into a block-diagonal structure. Within each of the steps, it takes k steps to compute $M(A_{\breve{i}\breve{j}})$, n steps for the criteria for which row to pick next, and the other steps within the loop, making for a total running time of $O(nk)$ for each successful step.

The number of rejections until a block-diagonal structure is reached can be assessed by the rejection probability P_R in each single step, which is

$$P_R = 1 - \frac{1}{M(A)} \sum_{j=1}^{I} M(A_{\breve{i}\breve{j}}). \tag{6}$$

In a k-diagonal matrix,

$$M(A) = \exp(-n) g(k)^{n-k+1} \prod_{j=\frac{k+1}{2}}^{k-1} g(j)^2 \tag{7}$$

whereas for the minor obtained by deleting "middle" rows and columns the bound for the permanent is equal to

$$M(A_{\breve{i}\breve{j}}) = \exp(-n+1) g(k)^{n-2k+2} g(k-1)^{k+1} \prod_{j=\frac{k+1}{2}}^{k-2} g(j)^2. \tag{8}$$

Then we have, as the row picked contains k ones,

$$\begin{aligned} \sum_{j=1}^{n} a_{ij} M(A_{\breve{i}\breve{j}})/M(A) &= kM(A_{\breve{i}\breve{j}})/M(A) && (9) \\ &= k\mathrm{e}\frac{g(k-1)^{k-1}}{g(k)^k}. && (10) \end{aligned}$$

With $g(k)/g(k-1) = 1 + k^{-1} + .5k^{-2} + .6k^{-3}$, the expression can be bounded from below for

$k \geq 2$ by

$$k\mathrm{e}\frac{g(k-1)^{k-1}}{g(k)^k} = \frac{k\mathrm{e}}{g(k)}\left(1+k^{-1}+.5k^{-2}+.6k^{-3}\right)^{1-k} \tag{11}$$

$$\geq \frac{k\mathrm{e}}{g(k)}\left(1+\frac{2.1}{k}\right)^{1-k} \tag{12}$$

$$\geq \frac{k\mathrm{e}}{g(k)}\exp(-2.1) \tag{13}$$

$$\geq \frac{k\mathrm{e}}{k+.5\log k+1.65}\exp(-2.1), \tag{14}$$

the last with help of a bound presented in Huber (2003).

The probability of rejection until a block-diagonal structure is reached is roughly the kth power of the expression above, because at most k steps are needed until a block-diagonal structure is reached. The expected number of rejections is the reciprocal of this expression:

$$P_R = \left(\exp(2.1)\left(\frac{1}{\mathrm{e}}+\frac{.5\log k}{k\mathrm{e}}+\frac{1.65}{k\mathrm{e}}\right)\right)^k \tag{15}$$

$$\leq \left(\exp(1.1)\left(1.5-\frac{0.65}{k}\right)\right)^k \tag{16}$$

$$= \left(1.5\exp(1.1)\left(1-\frac{1.3}{3k}\right)\right)^k \tag{17}$$

$$\leq (1.5\exp(1.1))^k\exp(-1.3/3). \tag{18}$$

The bound can be gained by taking $\log(k) \leq k-1$ and, again, using the row expansion for the exponential function.

The two blocks obtained by separating will have the dimension of less than $n/2$. Therefore, the running time can be estimated to be

$$T(n,k) = O(4.51^k nk) + 2T(n/2,k) \tag{19}$$

which means that it is geometrical in k but linear in n. This behaviour is shown empirically by obtaining the computation times from 200 samples each in $(5, 9, 13)$-diagonal matrices of size $(20, 40, 60, 80)$ (see Figure 1 for a simulation result).

6. Discussion

The original accept-and-reject algorithm has been introduced with a running time polynomial in n for dense graphs only, where the row sums of the adjacency matrix k (i.e. the order of the graph) are proportional to n with increasing n. The current situation handles a special case of sparse matrices. In this situation, it is algorithmically simple to find separators of the graph. While the strategy of finding separators of the graphs and sample within the corresponding row first readily generalizes to all types of matrices, the problem to find all separators of a general graph is also NP-hard. For certain classes of graphs, especially sparse graphs, there are faster algorithms. Therefore, a generalization of the algorithm can be applied to classes of graphs where the problem of finding separators is tractable.

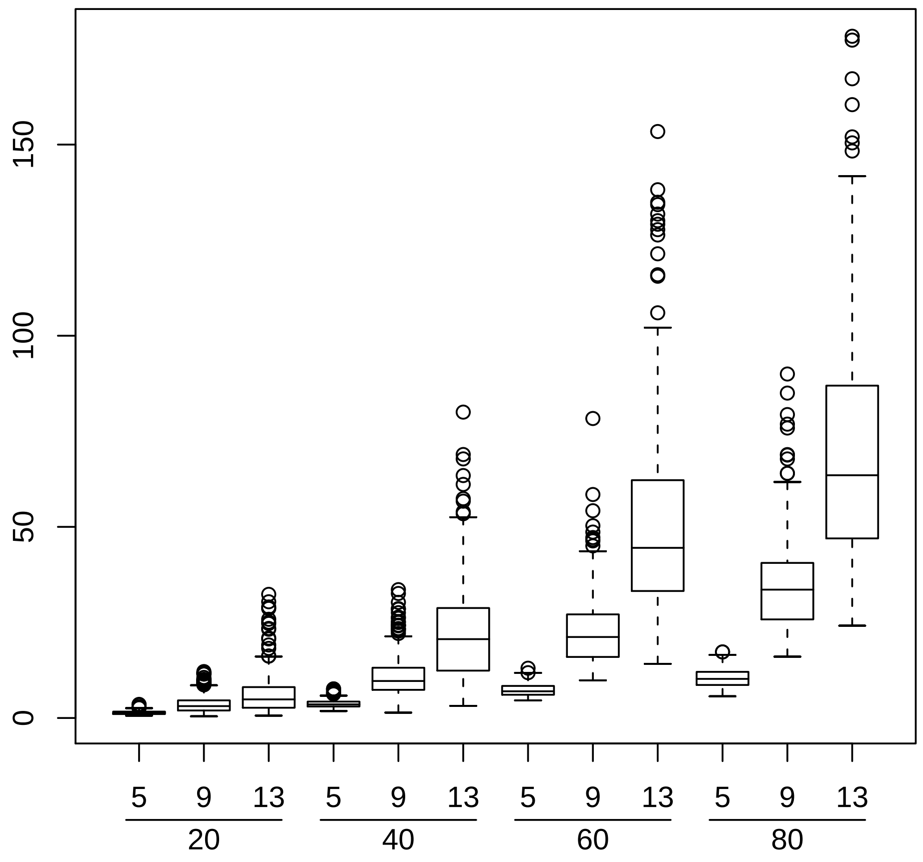


Figure 1: Empirical running times for sampling from a set of permutations restricted by $(5, 9, 13)$-diagonal matrices of size $(20, 40, 60, 80)$. The times were obtained using the code discussed here on a laptop with a 1200 MHz processor running Mac OS X 10.3.

If a matrix can be brought into a tube structure, it can also be represented by a clique matrix, where rows correspond to nodes and columns to maximal cliques in the graph.

It has been shown that the problem of sampling from a set of permutations can be solved by an algorithm with a computation time proportional to n. This is an encouraging result showing that one can feasibly generate randomization test under the assumption that only minor perturbations in the time domain make for realistic variations of the sample taken. In real-life applications, the thickness of the tube of 1s will vary considerably, leaving some very dense chunks of 1s after sampling from the thin rows. For dense matrices, however, the envelope probability provides a sharp bound for the one generated by the permanent, which leads to comparably few rejections and, again, short running times.

References

Bregman LM (1973). "Some Properties of Nonnegative Matrices and Their Permanent." *Sovietskyie Matematicheskyie Doklady*, **14**(4), 945–949.

Diaconis P, Graham R, Holmes SP (1999). *Statistical Problems Involving Permutations with Restricted Positions*, pp. 195–222.

Efron B, Petrosian V (1999). "Nonparametric Methods for Truncated Data." *Journal of the American Statistical Association*, **94**(447), 824–834.

Hornik K (2006). "The R FAQ." ISBN 3-900051-08-9, URL http://CRAN.R-project.org/doc/FAQ/R-FAQ.html.

Huber M (2003). "Exact Sampling from Perfect Matchings of Dense Nearly Regular Bipartite Graphs." arXiv:math.PR. 0310059 v1, URL http://arxiv.org/abs/math/0310059/.

Hüsing J (2006). "Sampling from a Set of Restricted Permutations to Obtain Null Distributions in Situations where Stimulus Allocations are Mainly Deterministic." Submitted.

Jerrum M, Sinclair A, Vigoda E (2001). "A Polynomial-Time Approximation Algorithm for the Permanent of a Matrix with Non-Negative Entries." In "ACM Symposium on Theory of Computing," pp. 712–721. URL http://citeseer.ist.psu.edu/jerrum01polynomialtime.html.

Knuth DE (1992). *Literate Programming*. CSLI Lecture Notes. Center for the Study of Language and Information, Stanford, California, 27th edition.

Minc H (1978). *Permanents*, volume 6 of *Encyclopedia of Mathematics and its Applications*. Addison-Wesley.

Ohashi Y (1990). "Randomization in Cancer Clinical Trials: Permutation Test and Development of a Computer Program." *Environmental Health Perspectives*, **87**, 13–17.

Ramsey N (1994). "Literate Programming Simplified." *IEEE Software*, **11**(5), 97–115.

R Development Core Team (2006). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-00-3, URL http://www.R-project.org/.

Simon R (1979). "Restricted Randomization Designs in Clinical Trials." *Biometrics*, **35**, 503–512.

Taves DR (1974). "Minimization: A New Method of Assigning Patients to Treatment and Control Groups." *Clinical Pharmacology and Therapy*, **15(5)**, 443–453.

Valiant LG (1979). "The Complexity of Computing the Permanent." *Theoretical Computer Science*, **8**, 189–201.

Affiliation:

Johannes Hüsing
Koordinierungszentrum für Klinische Studien
Voßstraße 2
D-69115 Heidelberg, Germany
E-mail: `johannes.huesing@med.uni-heidelberg.de`
URL: `http://www.klinikum.uni-heidelberg.de/Huesing.5379.0.html`

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume 16, Issue 6 *Submitted:* 2005-04-11
July 2006 *Accepted:* 2006-05-16