DiGGer User Book

Neil Coombes

2020-04-29

Contents

Chapter 1

Introduction

`DiGGer` is a flexible tool for finding experimental designs that are efficient for specified blocking and correlation patterns. The package `DiGGer` ("NSW Dpi Biometrics Software Download Page" 2018) is an add-on for the statistical computing language and environment `R` (R Core Team 2015). `DiGGer` provides object oriented methods through two additional required `R` packages: `R.methodsS3` and `R.oo`. `DiGGer` is available for Windows 32/64, Linux and MacOSX. The versions for Mac and Linux require `gfortran` to be installed.

Constructor functions are used to create the instructions required for a search. Constructor functions begin with capitals while `camelCase` is used for function and variable names.

To create a design search `DiGGer` requires

- a design layout specified as
  - `rowsInDesign`
  - `columnsInDesign`
  - optionally `rowsInReplicate`
  - optionally `columnsInReplicate`
- treatment information
  - `numberOfTreatments`
  - optionally `treatRepPerRep`
  - optionally `treatGroup`
- objectives to be optimised with blocks, linear covariates and correlation between plots specified

- search control parameters for each search phase, giving the number of interchanges to test, optimality measure, swap restrictions and correlation block size.

From an initial design a `DiGGer` search when run interchanges 2 treatments and calculates a new objective measure of design efficiency. The new design is either accepted or rejected based on rules in the Reactive Tabu Search (RTS) algorithm (Battiti and Tecchiolli (1994), Coombes (2002)) . The search continues until stopping rules are met. The objective measure is an A value, related to the average variance of pairwise comparisons of treatments of interest. Lower values are better for the postulated correlation pattern. After the search the `DiGGer` object has a component, `$dlist`, containing the field book listing of the design.

The constructor functions: `DiGGer`, `Block`, `Correlation`, `Objective` and `Phase` may be used to construct a design search (see Section 4). Users however, will generally choose to use wrapper functions which create searches for particular design types. The wrapper functions in this version are

- `corDiGGer` for general block designs with correlated errors and single treatment factors
- `ibDiGGer` for incomplete block designs
- `rcDiGGer` for row-column designs
- `prDiGGer` for partially replicated designs which must have a number of unreplicated treatments
- `facDiGGer` for designs with factorial treatments
- `r2dDiGGer` for strict two-directional replicates
- `nurseryDiGGer` for replicated designs with additional high frequency check treatments

`DiGGer` was developed for field designs in rectangular arrays of plots so all design searches use specifications of rectangular layouts and block structures. For some of the design types listed these rectangular specifications may be nominal and may not correspond to actual physical layouts.

This version of `DiGGer` differs from previous versions with more uniform syntax and random number generator seeds that allow the recovery of particular designs. The random number generator is initialised in R and the current state of the random number generator is passed to the Fortran program when the search is run. The Fortran program uses a version of RTS to find near

optimal designs for the specified objective.

Section 2 gives some preliminary concepts concerning design searches. Section 3 details the constructor functions, Section 4 illustrates the use of constructor functions in creating a block design. These two sections provide background information on the internal workings of the wrapper functions and may be skipped in an initial reading. Section 5 presents the wrapper functions with some examples and Section 6 gives examples of the flexibility that `DiGGer` provides.

The R Help gives details of the full range of functions in the `DiGGer` package.

Chapter 2

Preliminary concepts

2.1 What is a design?

In `DiGGer` a design is a matrix of design numbers. The design numbers are consecutive, 1 to `numberOfTreatments`, at the appropriate replication. Each number represents a treatment applied to an experimental unit, or plot, in the trial or experiment. The matrix has `rowsInDesign` rows and `columnsInDesign` columns. Field experiments are usually physically arranged in rows and columns but many experiments can be notionally described in terms of rows and columns.

The design may be arranged in replicates in a Randomised Complete Block (RCB) design. Each replicate defined by `rowsInReplicate` and `columnsInReplicate` has treatments repeated according to `treatRepPerRep`.

`treatRepPerRep` is a vector `numberOfTreatments` long with the relative replication levels for each treatment in a replicate.

`k*sum(treatRepPerRep)=rowsInReplicate*columnsInReplicate`, `k` an integer. `DiGGer` will randomise `k` repeats of the replication scheme within each replicate.

The blocks with template [`rowsInReplicate` × `columnsInReplicate`] must tile evenly within the design (Figure 2.1). All blocks tile from Row 1, Column (or Range) 1.

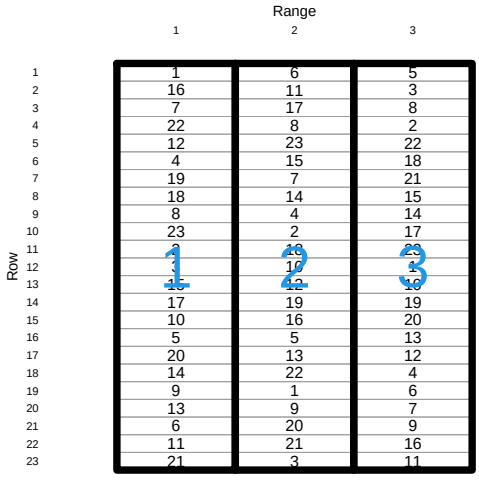

Figure 2.1: Tiling of replicates within a design

If the design is completely randomised (CR) then `rowsInReplicate = rowsInDesign` and `columnsInReplicate = columnsInDesign`.

2.2 Design properties

2.2.1 Blocks

The design may be divided into blocks. Blocks specify sections of the design with more similar experimental material. `Block` objects in `DiGGer` are defined with three parameters:

- `rowsInBlock`
- `columnsInBlock`
- `blockVarianceRatio`.

If the `blockVarianceRatio > 0` then blocks are random, otherwise blocks are fixed. Blocks tile down and across the design from (Row 1, Column 1)

using the block template, [`rowsInDesign` × `columnsInDesign`] (Figure 2.2). Blocks may tile unevenly:

- the final row of blocks may have fewer than `rowsInBlock` rows
- the final column of blocks may have fewer than `columnsInBlock` columns.

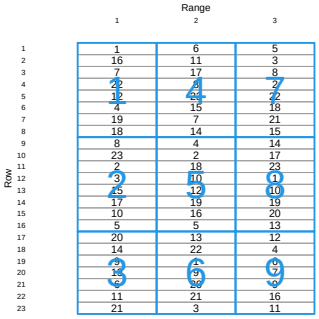

Figure 2.2: Tiling of blocks within a design. The final row of blocks (3,6,9) has one fewer row than the other blocks.

2.2.2 Blocking sequence

The blocking sequence is a list of dimension pairs

`blockSequence = list(c(r1, c1), c(r2, c2), ...)`.

Each dimension pair in the list is used to create up to three sets of blocks for a search phase, the pair `c(r1, c1)` produces:

- Row blocks across the design with `r1` rows: [`r1` × `columnsInDesign`]
- Column blocks down the design with `c1` columns: [`rowsInDesign` × `c1`]
- Intersection blocks: [`r1` × `c1`]

Redundant blocks created by the above procedure should be automatically dropped from the search specifications. Figure 2.3 shows the three sets of blocks.

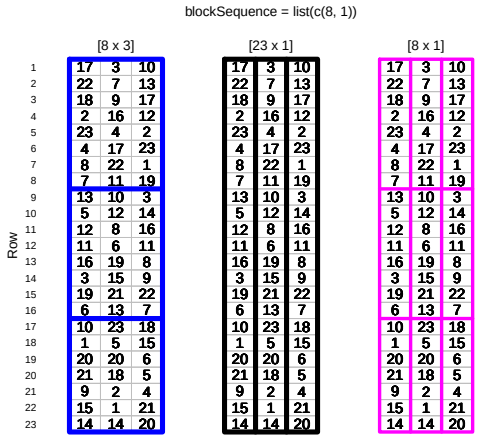

Figure 2.3: Blocking induced by `blockSequence = list(c(8, 1))`: $[8 \times 3]$, $[23 \times 1]$ and $[8 \times 1]$.

Note that the set of intersection blocks is nested, or completely contained, within the other blocking structures. The relative importance of these blocking structures is controlled through `objectiveWeight` or `blockVarianceRatio` specifications.

2.2.3 Correlation

Correlation may be defined across the whole of the design or within independent blocks which must tile evenly into the design. There are three models for the correlation that may occur between plots within blocks: Identity (`ID`), AutoRegressive (`AR`) and Moving Average `MA`.

- `ID`, equal correlation between plots within a block
- `AR`, the correlation ($\rho : -1.0 < \rho < 1.0$) between plots decreases with the separation of plots $\rho^{|i-j|}$, with i and j either row numbers or column numbers
- `MA`, adjacent experimental units are correlated ($\rho : -0.5 < \rho < 0.5$) but others are not correlated.

The correlation model assumes that the row correlation model and the column correlation model represent separable processes and the overall correlation is formed by the kronecker product of these correlations. For the `AR` $\otimes$ `AR` model treatment neighbours will tend to be balanced across rows and columns. The model ensures that the same treatment is strongly discouraged from occurring next to itself in rows or columns but it encourages a treatment to occur diagonally adjacent to itself as a self-diagonal pair. One reason to introduce blocking into the design is to reduce the possibility of self-diagonals.

Although it is not possible to know the true correlation between units in an experimental design the default AR model has been shown to be appropriate in many field designs. The designs produced are robust to misspecification of the correlation parameters. The designs produced for particular AR models are only marginally less efficient for AR models with different parameters.

2.2.4 Efficiency of the design

There are a number of ways of defining the efficiency of a design. The efficiency depends on the placement of treatments in the design, the blocks in the design and the assumed correlation across the design. `DiGGer` uses an A efficiency measure to compare candidate designs in a search. The A-measure is the average variance of comparisons between pairs of treatments in the design. By minimising the A-measure `DiGGer` finds near optimal designs for given blocking and correlation structures.

Mathematically the information matrix, $\mathbf{C}$, of a design adjusted for nuisance fixed effects is

$$\begin{aligned}\mathbf{C} &= \mathbf{X}_1'\mathbf{\Lambda}^*\mathbf{X}_1 \\ \mathbf{\Lambda}^* &= \mathbf{\Lambda}^{-1} - \mathbf{\Lambda}^{-1}\mathbf{X}_2(\mathbf{X}_2'\mathbf{\Lambda}^{-1}\mathbf{X}_2)^{-1}\mathbf{X}_2'\mathbf{\Lambda}^{-1} \\ \mathbf{\Lambda} &= \sum \gamma_{b_i}\mathbf{Z}_{b_i}\mathbf{Z}_{b_i}' + \gamma_s\mathbf{\Sigma}_c \otimes \mathbf{\Sigma}_r\end{aligned}$$

where

$\mathbf{X}_1$ is the design matrix for the treatments

$\mathbf{X}_2$ is the design matrix for nuisance fixed effects such as the overall mean, fixed blocks and linear trend in rows or columns

$\mathbf{\Lambda}$ is the covariance matrix between plots of the design

$\mathbf{\Lambda}^*$ is the covariance matrix adjusted for nuisance fixed effects

$\mathbf{Z}_{b_i}$ is the design matrix for a random block factor with variance ratio γ_{b_i}

$\mathbf{\Sigma}_r$ is the spatial process between the rows

$\mathbf{\Sigma}_c$ is the spatial process between the columns

γ_s is the variance ratio associated with the spatial process.

The A value for comparing all treatments is proportional to the sum of the diagonal elements of the generalised inverse of the information matrix $\mathbf{C}^-$. The efficiency of the design is clearly dependent on the postulated blocking and correlation pattern in the design.

The efficiency of a design may also be calculated relative to particular treatment comparisons. Treatments may be divided into different groups of treatments and the efficiency may be based on within treatment group comparisons or between treatment group comparisons.

2.3 Searching for improvements

`DiGGer` is a search algorithm and requires

- an initial design
  - The initial design is a matrix of design numbers and is either generated from treatment replication and layout information or is supplied as `initialDesign` by the user.
- a method for changing the design
  - A design is modified by interchanging treatments that are applied to two experimental units.
  - Swap restrictions are required to preserve replicates and to preserve blocking structures that are created as part of the search. Swap restrictions may be specified by a swap block dimension pair or

may be given in matrix form using `initialSwap` which restricts interchanges to experimental units with the same swap code.

- an optimality/efficiency measure which depends on
  - the randomisation of the treatments
  - the correlation and blocking in the design
  - the treatments that are being compared. The default optimality is the average variance of comparisons between all pairs of treatments and is given the code `A++` in `DiGGer`. It is $2/(\texttt{numberOfTreatments} - 1)$ times the A-value of the design under the specified correlation and blocking. `DiGGer` aims to minimise the A-measure.
- rules to accept changes to the design
  - The algorithm in `DiGGer` is based on Reactive Tabu Search (RTS). It accepts any change which improves the best design in the search but allows the search to move to a poorer intermediate design when no improvement is possible from a single interchange.
  - A local minimum A-measure is reached when no treatment interchange improves the design. The search continues by choosing the best allowable interchange from a percentage of the possible interchanges as specified by the `searchIntensity` property of a search phase. With the default `searchIntensity=100`, the search for an improvement is concentrated close to the local minimum design. The lower the value of `searchIntensity` the more randomness is introduced into the search.
- a method to stop the search
  - a predefined maximum number of interchanges has been tested
  - a target A-measure has been bettered.

The search itself may consist of a number of component searches, or search phases.

Chapter 3

Constructor functions

The constructor functions in this version are:

- `DiGGer` which creates a `DiGGer` object with an initial design, initial swap matrix and treatment information. There is an empty slot created ready for the search specifications and results.
- `Correlation` defines the plot to plot correlation which may be identity, auto-regressive or moving average.
- `Block` defines a rectangular block factor with row, column and variance ratio properties.
- `Objective` defines a search objective used in the optimisation process combining correlation and block objects.
- `Phase` defines a search phase with swap and correlation specifications, type of optimisation measure and the objectives to be optimised.

3.1 `DiGGer`

The `DiGGer` constructor function is used to initialise a `DiGGer` object. It will usually be hidden within the specific design type wrapper functions but shares many function arguments with them. The arguments to `DiGGer` are:

- `numberOfTreatments`, the number of distinct treatments applied to the experimental units
- Layout information

  - `rowsInDesign`
  - `columnsInDesign`
  - `rowsInReplicate`, rows in each replicate template block. If not given `rowsInDesign` is used.
  - `columnsInReplicate`, columns in each replicate template block. If not given `columnsInDesign` is used.
- Treatment information
  - `treatName`, optional treatment label name.
  - `treatNumber`, optional treatment label number.
  - `treatRepPerRep`, vector giving the repeats of each treatment in each replicate. If not given, treatments are equally replicated.
  - `treatGroup`, vector of numeric codes which may be used to define comparisons of interest in the search. If not given treatments are given the same group code.
- `initialDesign`, an optional matrix of design numbers, `1` to `numberOfTreatments`, giving an initial treatment allocation. Missing plots in the matrix can be coded as a `0` design number.
- `initialSwap`, an optional matrix of swap codes to control allowable swaps. Treatment swaps are only possible between experimental units with the same swap code.
- `rngSeeds`, an optional pair of seeds for the random number generator (RNG) $c(i, j)$ where $0 \leq i \leq 31328$ and $0 \leq j \leq 30081$. New seeds must be used to produce different designs. Best practice is to allow the `DiGGer` functions to create the seeds.
- `rngState`, the current state of the random number generator. This is useful if the search consists of a series of calls to `DiGGer` and the random number generator has been initialised.

After the call to `DiGGer` the design object will be a list with

- `$treatment` a data frame with columns `Name`, `Number`, `Repeats` and `Group` and a row for each treatment.
- `$idsgn` an initial design matrix.
- `$iswap` an initial swap matrix.
- `$ddphase` a list slot ready to be populated with search parameter information.

3.2 Block

The `Block` constructor function has three parameters:

- `rowsInBlock`, the number of rows in a template block.
- `columnsInBlock`, the number of columns in a template block.
- `blockVarianceRatio`, the variance component ratio. If greater than zero the blocks are treated as random, otherwise blocks are treated as fixed. The use of fixed blocks with small numbers of experimental units can cause a search to crash.

The template blocks are tiled down rows and across columns of the design starting from Row 1 and Column (or Range) 1. Blocks may tile unevenly:

- the final row of blocks may have fewer than `rowsInBlock` rows
- the final column of blocks may have fewer than `columnsInBlock` columns.

3.3 Correlation

The `Correlation` constructor function has parameters:

- `rowModel`, may be `"AR"` for autoregressive, `"MA"` for moving average or `"ID"` for identity.
- `rowParameter`, for autoregressive models the parameter range is $(-1, 1)$, for moving average the parameter range is $(-0.5, 0.5)$ and the identity parameter is 0.0.
- `columnModel` as for the row models.
- `columnParameter` as for the row parameters.
- `spatialVarianceRatio`, the variance parameter associated with the spatial process.

3.4 Objective

The `Objective` constructor function combines blocks and correlation to form an objective to be optimised. The parameters to this function are:

- `weight`, a multiplier to give the relative importance of this objective when there are multiple objectives.
- `linearCovariate`, with options `"ROWS"`, `"COLUMNS"` or `"BOTH"` to model linear trends associated with rows and columns.
- `numberOfBlocks` gives the number of blocking factors in the current objective.
- `blocks` is a list of block objects `numberOfBlocks` long.
- `corr` give a correlation object associated with the objective.

3.5 Phase

The `Phase` constructor combines objectives for a search and controls the search settings. The parameters of the function are:

- `rowsInSwapBlock` defines the number of rows in a template block in which treatment swaps are possible.
- `columnsInSwapBlock` defines the column dimension of the swap block.
- `rowsInCorrelationBlock` defines the extent of blocks with correlated experimental units. The block must tile evenly within the design layout.
- `columnsInCorrelationBlock` defines the column dimension of blocks with correlated experimental units.
- `aType` is the optimality measure used in comparing designs during a search. The default `"A++"` calculates an A measure using all treatment comparisons. A measures relate to the average variance of pairwise treatment comparisons. When there is more than one treatment group alternative A measures are possible. `"Agg"` calculates an A measure for comparisons between treatments with different group codes. Other codes: `"A22"` for comparisons between group 2 treatments, `"A11"` for comparisons between group 1 treatments, `"A1+"` for comparisons of group 1 treatments with others, `"Aa2"` for comparisons between the average of group 1 treatments with group 2 treatments and `"Aaa"` for the comparison between the average of group 1 and the average of group 2 treatments.
- `targetAValue` the objective value below which the search will stop. As the A-measures are always positive a value of 0 will not come into play as a stopping rule.

- `maxInterchanges` the maximum number of treatment interchanges to test in the phase.
- `searchIntensity` the $[0, 100]$ percentage of swaps considered when making a non-improving treatment exchange. Zero causes a random interchange if no improvement when all possible interchanges have been considered.
- `numberOfObjectives` gives the number of objectives in the phase.
- `objectives` is a list of the objective objects.

Chapter 4

Using constructor functions

To illustrate the steps to perform a search with these functions, consider the block design in (Cochran and Cox 1957), plan 11.21, for 13 treatments in 26 blocks of 3, where the 13 by 3 blocks each contain a randomisation of three repeats of the 13 treatments. After initialising with a call to `DiGGer` the R object has an initial design, an initial swap matrix and the Random Number Generator (RNG) has been initialised. The block structure to be optimised is created using the `Block` function. There is no correlation in this design so the `Correlation` function creates an identity correlation object. The block and correlation objects are used in a call to the `Objective` function to create a search objective. The search phase is created using the `Phase` function. Swaps are resticted to $[13 \times 3]$ blocks. The correlation block is reduced to $[1 \times 3]$ corresponding to the size of the blocks of interest.

Besides constructor functions there are a number of modifier functions to change search settings. See the package information in the R Help for the full list. A phase may be added to the `DiGGer` object using one such function, `addPhase()`. The search may be run using the `run()` function.

```
b13_26x3 <- DiGGer(numberOfTreatments=13,
                   rowsInDes=26, columnsInDes=3,
                   rowsInRep=13, columnsInRep=3,
                   rngSeeds = c(2351, 8342))
b111 <- Block(rowsInBlock=1, columnsInBlock=3,
              blockVarianceRatio=1.0)
```

```
c0 <- Correlation(rowModel="ID", rowParameter=0,
                  columnModel="ID", columnPar=0,
                  spatialVarianceRatio=1.0)
o11 <- Objective(weight=1.0, linearCovariate="NONE",
                 numberOfBlocks=1, blocks=list(b111),
                 corr=c0)
p1 <- Phase(rowsInSwapBlock=13, columnsInSwapBlock=3,
            rowsInCorrelationBlock=1,
            columnsInCorrelationBlock=3, aType="A++",
            targetAValue=0, maxInterchanges=100000,
            searchIntensity=100, numberOfObjectives=1,
            objectives=list(o11))
addPhase(b13_26x3, p1)
b13_26x3 <- run(b13_26x3)
```

The `getConcurrence` function tabulates the number of times that pairs of treatments occur together in blocks. For a balanced design each pair of treatments will occur together an equal number of times in blocks.

```
getConcurrence(b13_26x3)
```

```
## B RANDOM BLOCK: 1 Row by 3 Columns, VarianceRatio 1.000000
##
## Treatment-Block

##
## Number of Concurrences    0  1 2
## Number of Treatment Pairs 2 74 2
```

After 100,000 interchanges the optimal resolvable block design has not been found. Concurrences of pairs of treatments within the blocks of this design should be 0 or 1. The `run()` function has options for re-running or continuing the current search:

- `continue=TRUE` restarts the search at the current best design and repeats the final phase of the search. The tabu history is restarted.
- `newpath=TRUE` re-runs the search using the initial design from the original search but re-initialising the RNG so that the search path is different.
- `newstart=TRUE` runs the search again with a new initial design.

```
b13_26x3 <- run(b13_26x3, continue=TRUE)
```

```
getConcurrence(b13_26x3)
```

```
## B RANDOM BLOCK: 1 Row by 3 Columns, VarianceRatio 1.000000
##
## Treatment-Block
```

```
##
## Number of Concurrences     1
## Number of Treatment Pairs 78
```

The wrapper function `ibDiGGer()` can be used to repeat the search.

```
ib13_26x3 <- ibDiGGer(
    numberOfTreatments = 13,
    rowsInDesign = 26, columnsInDesign = 3,
    rowsInRep = 13, columnsInRep = 3,
    rowsInBlock = 1, columnsInBlock = 3,
    rngSeeds = c(2351, 8342))
getConcurrence(ib13_26x3)
ib13_26x3 <- run(ib13_26x3, continue=TRUE)
```

```
getConcurrence(ib13_26x3)
```

```
## B RANDOM BLOCK: 1 Row by 3 Columns, VarianceRatio 1.000000
##
## Treatment-Block
```

```
##
## Number of Concurrences     1
## Number of Treatment Pairs 78
```

```
table(getDesign(b13_26x3) - getDesign(ib13_26x3))
```

```
##
##  0
## 78
```

The designs obtained using both methods and the same RNG seeds are identical.

Chapter 5

The wrapper functions

Wrapper functions use the constructor functions to perform searches for different design types. The wrapper functions in this version are:

- `corDiGGer` which can perform searches for spatially adjusted incomplete block designs in the manner of the `DiGGer()` function in version 0.2-3.
- `ibDiGGer` for searches for incomplete block designs.
- `rcDiGGer` for row-column design searches.
- `prDiGGer` for partially replicated design searches.
- `facDiGGer` for searches for factorial and split plot designs.
- `r2dDiGGer` for searches with strict replication in two directions where replicates in the second set are not rectangular.
- `nurseryDiGGer` for replicated designs where higher frequency control treatments need to be spread regularly across the design.

5.1 General correlated designs, `corDiGGer`

`corDiGGer` is a wrapper function to perform the standard searches of the previous `DiGGer` versions. Standard searches consisted of a blocking phase in the absence of correlation followed by a search phase with random row and column blocks and autoregressive correlation of 0.5 between rows and between columns.

The arguments to `corDiGGer` are:

- `numberOfTreatments`
- `rowsInDesign`
- `columnsInDesign`
- `rowsInReplicate`
- `columnsInReplicate`
- `blockSequence`, a list of dimension pairs defining block search phases
- `independentBlocks`, a dimension pair defining blocks of correlated plots that must tile evenly across the design
- `maxInterchanges`, default `100000`.
- `searchIntensity`, default `100`. Percentage of possible interchanges to monitor when a detrimental interchange is accepted.
- `aType`, default `"A++"`, compare all treatments.
- `targetAValue`, default `0`. Search stops when A-measure falls below this value.
- `runSearch`, default `TRUE`.
- `rngSeeds`, default `NULL`.
- `rngState`, current state of the random number generator.
- `treatName`
- `treatNumber`
- `treatRepPerRep`
- `treatGroup`
- `spatial`, default `TRUE`, defines separable autoregressive processes for rows and columns, (`"AR"`, `0.5`), applied after the blocking search phases.
- `rowColumn`, default `TRUE`, defines random row and column blocks with variance ratio `1.0`. Applied after blocking phases of the search.
- `nuggetVar`, default variance ratio `0.1` added for each plot in the final search phase.
- `initialDesign`, default `NULL`. Matrix of design numbers.
- `initialSwap`, default `NULL`. Dimension pair or full matrix of codes giving allowable swaps. Plots with the same code may have treatments interchanged.

The blocking phases are defined using a block sequence list, `blockSequence`. Each element of the list is a dimension pair, `c(nr, nc)` defining the shape of an individual block. Each dimension pair is combined with the overall design dimensions to define two additional block shapes:

- `c(nr, nc)`
  - [`nr` × `columnsInDesign`]

- [rowsInDesign × nc]
- [nr × nc]

This ensures that the strips of nr rows and the strips of nc columns can be included in the optimisation. For smaller designs these two strips define one objective and the [nr × nc] blocks define a second objective. The default weights for the objective with the larger blocks is 0.8 and the weight for the smaller block is 0.2. These weights are designed to give more emphasis to the larger blocks when replicates in two directions is the aim. The weights may need to be adjusted in cases where the block size disparity is very large or where there is unequal treatment replication. The setObjectiveWeight() function can be used to adjust weights prior to running the search. Set runSearch = FALSE in the call to corDiGGer then use the run function to run the job.

```
d18 <- corDiGGer(
  numberOfTreatments = 18,
  rowsInDesign = 18,
  columnsInDesign = 3,
  blockSequence = list(c(6,1)),
  rngSeeds = c(111, 222))
## plot(d18)
```

The desTab function calculates the frequencies of treatments occurring in blocks and shows that replicates have been set in two directions from this completely randomised initial design.

```
desTab(getDesign(d18), 18, 1)
```

```
##        B1 B2 B3
## freq_1 18 18 18
```

```
desTab(getDesign(d18), 6, 3)
```

```
##        B1 B2 B3
## freq_1 18 18 18
```

corDiGGer has a blockSequence = "default" option which sets up blocks that cut across replicates. The following example of 20 treatments arranged in a [20 × 3] layout with [20 × 1] replicates shows the action when replication in two directions is not possible. Four equal blocks of [5 × 3] can be formed,

equivalent to setting `blockSequence = list(c(5, 1))`.

```
d20 <- corDiGGer(
    numberOfTreatments = 20,
    rowsInDesign = 20, columnsInDesign = 3,
    rowsInRep = 20, columnsInRep = 1,
    blockSequence = "default",
    rngSeeds = c(111, 333))
```

```
getBlock(d20, phaseNo = 1, objectiveNo = 1,
         blockNo = 1)
```

```
## B RANDOM BLOCK: 5 Rows by 3 Columns, VarianceRatio 1.000000
```

```
getBlock(d20, 1,2,1)
```

```
## B RANDOM BLOCK: 5 Rows by 1 Column, VarianceRatio 1.000000
```

```
desTab(getDesign(d20), 5, 3)
```

```
##        B1 B2 B3 B4
## freq_1 15 15 15 15
```

The `treatRepPerRep` parameter allows for specification of unequal replication in replicate blocks. Consider a design for 33 entries where the first treatment is replicated 4 times in each $[36 \times 1]$ replicate block in an overall $[36 \times 4]$ layout. Setting the block sequence with $[9 \times 1]$ blocks will allow for the possibility of $[9 \times 4]$ replicates with efficient $[9 \times 1]$ sub-blocks.

```
d33 <- corDiGGer(
    numberOfTreatments = 33,
    rowsInDesign = 36, columnsInDesign = 4,
    rowsInRep = 36, columnsInRep = 1,
    blockSeq = list(c(9,1)),
    treatRepPerRep= c(4,rep(1,32)),
    rngSeeds = c(111, 222))
## plot(d33)
## plot(d33, trts=1, col=2, new=FALSE)
```

```
desTab(getDesign(d33), 9, 4)
```

```
##        B1 B2 B3 B4
```

```
## freq_1 32 32 32 32
## freq_4  1  1  1  1
```

```
desTab(getDesign(d33), 9, 1)
```

```
##        B1 B2 B3 B4 B5 B6 B7 B8 B9 B10 B11 B12 B13 B14 B15 B16
## freq_1  9  9  9  9  9  9  9  9  9   9   9   9   9   9   9   9
```

The search has found replicates in two directions with each of the sixteen $[9 \times 1]$ blocks having no repeats of the higher frequency treatment.

5.2 Incomplete Block Designs, ibDiGGer

The arguments to ibDiGGer are:

- numberOfTreatments
- rowsInDesign
- columnsInDesign
- rowsInReplicate
- columnsInReplicate
- rowsInBlock
- columnsInBlock
- fixedBlocks
- blockVarianceRatio
- aType
- targetAValue
- maxInterchanges
- searchIntensity
- runSearch
- rngSeeds
- rngState
- treatName
- treatNumber
- treatRepPerRep
- treatGroup
- initialDesign
- initialSwap

Blocks to be optimised are specified by `rowsInBlock` and `columnsInBlock`. If `rowsInReplicate` and `columnsInReplicate` are supplied the design will be constrained to have replicates. Here is an example of a balanced incomplete block design. The RNG seeds have been chosen to illustrate a search which requires a continuation to find the optimum. The optimal design for random blocks with variance ratio of 1.0 was found to be 0.2857143 in a separate search. A slightly larger target A value of 0.2857145 may be specified to cause the search to stop once the A value falls below this value. The maximum interchanges to test is increased from the 100000 default to 1000000.

```
ib19b <- ibDiGGer(numberOfTreatments = 19,
                  rowsInDesign = 57, columnsInDesign = 3,
                  rowsInBlock = 1, columnsInBlock = 3,
                  maxInterchanges = 1000000,
                  targetAValue = 0.2857145,
                  rngSeeds = c(9342, 18602),
                  runSearch = FALSE)
ib19b
ib19b <- run(ib19b)
```

```
getConcurrence(ib19b)
```

```
## B RANDOM BLOCK: 1 Row by 3 Columns, VarianceRatio 1.000000
##
## Treatment-Block

##
## Number of Concurrences    0   1 2
## Number of Treatment Pairs 2 167 2
```

The `getConcurrence()` function tabulates the number of times that pairs of treatments occur together in blocks. If no block is specified the first block of the first objective of the first search phase is used. After this call to `ibDiGGer` there are 2 pairs of treatments that do not occur together in blocks and 2 pairs of treatments that occur together twice in blocks. There is an option in the `run()` function to continue the search with the current best design. The search history records are cleared before continuing but the current settings of the RNG, `rngState`, are retained for repeatability of the search.

```
ib19b <- run(ib19b, continue = TRUE)
```

```
getConcurrence(ib19b)
```

```
## B RANDOM BLOCK: 1 Row by 3 Columns, VarianceRatio 1.000000
##
## Treatment-Block
```

```
##
## Number of Concurrences      1
## Number of Treatment Pairs 171
```

```
ib19b$ddphase[[1]]$lastImprovement
```

```
## [1] 610171
```

After the continuation all 171 pairs of treatments occur together once in blocks. This needed 1610171 interchanges in total.

A balanced lattice design has resolvable blocks. Again the known optimum from a previous search with random blocks can be used as a stopping rule.

```
ib25s <- ibDiGGer(numberOfTreatments = 25,
                  rowsInDesign = 30, columnsInDesign = 5,
                  rowsInRep = 5, columnsInRep = 5,
                  rowsInBlock = 1, columnsInBlock = 5,
                  maxInterchanges = 10000000,
                  targetAValue = 0.387097,
                  rngSeeds = c(11301, 29798))
```

```
##      Phase,    Search%,    A-measure
## [1] 1.0000000 0.0000000 0.3966814
## [1] 1.0000000 5.0000000 0.3870968
##  [1] 0.3870968 0.3870968 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
##  [8] 0.0000000 0.0000000 0.0000000
```

```
getConcurrence(ib25s)
```

```
## B RANDOM BLOCK: 1 Row by 5 Columns, VarianceRatio 1.000000
##
## Treatment-Block
```

```
##
## Number of Concurrences      1
## Number of Treatment Pairs 300
```

```
ib25s$ddphase[[1]]$lastImprovement
```

```
## [1] 520760
```

5.3 Row-Column Designs, `rcDiGGer`

The arguments to the `rcDiGGer` function are:

- `numberOfTreatments`
- `rowsInDesign`
- `columnsInDesign`
- `rowsInReplicate`
- `columnsInReplicate`
- `twoPhase`, default `TRUE`, causes larger of the blocks to be optimised in the first search phase and the smaller blocks to be optimised in a second phase. The order of the optimisation can be specified as `"rowThenCol"` or `"colThenRow"`. If `FALSE` both sets of blocks are optimised in the one search phase.
- `nested`, default `TRUE`. If `TRUE` block sizes are restricted to within replicates, otherwise blocks extend across the full design.
- `fixedBlocks`, default `FALSE`.
- `blockGammas`, default `list(rGamma = 1, cGamma = 1)`.
- `aType`
- `targetAValue`
- `maxInterchanges`
- `searchIntensity`
- `runSearch`
- `rngSeeds`
- `rngState`
- `treatName`
- `treatNumber`
- `treatRepPerRep`
- `treatGroup`

- initialDesign
- initialSwap

The rows and columns may extend across the full design or they may be nested within replicates, `nested = TRUE`.

Row-column searches can be undertaken as two-phase searches or single phase searches. The default search action is for two-phase searches, `twoPhase = TRUE`. In a default two-phase search the first phase optimises the blocks, rows or columns, that are smaller. In the second phase swaps are restricted within these blocks while the other blocks are optimised. The order of optimisation can be controlled using `twoPhase = "rowThenCol"` or `twoPhase = "colThenRow"`. With `twoPhase = FALSE` both sets of blocks are optimised in the one search phase.

These approaches are illustrated in a trial for 12 treatments in a $[3 \times 12]$ design with replicates $[3 \times 4]$. The default settings result in $[3 \times 1]$ column blocks being optimised before nested $[1 \times 4]$ row blocks. To help in efficiency calculations, `fixedBlocks = TRUE`.

```
test1 <- rcDiGGer(numberOfTreatments = 12,
                  rowsInDesign = 3, columnsInDesign = 12,
                  rowsInReplicate = 3, columnsInReplicate = 4,
                  maxInterchanges = 500000, fixedBlocks = TRUE,
                  rngSeeds = c(111, 222))
```

```
getConcurrence(test1, 2,1,1)
```

```
## B FIXED BLOCK: 1 Row by 4 Columns
##
## Treatment-Block

##
## Number of Concurrences     0  1  2
## Number of Treatment Pairs 24 30 12
```

```
getConcurrence(test1, 2,1,2)
```

```
## B FIXED BLOCK: 3 Rows by 1 Column
##
## Treatment-Block
```

```
##
## Number of Concurrences     0  1
## Number of Treatment Pairs 30 36
```

```
2/3/test1$ddphase[[2]]$aMeasures[1]
```

```
## [1] 0.5075503
```

The design efficiency is calculated from the fixed block A-measure as $2/r/A$ where r is the number of replicates and A is the `DiGGer` fixed effect `aMeasure`. For the `test1` design it is 0.508. When row blocks are optimised before column blocks the the overall efficiency is reduced with row block concurrences better and column block concurrences worse.

```
test2 <- rcDiGGer(numberOfTreatments = 12,
                  rowsInDesign = 3, columnsInDesign = 12,
                  rowsInReplicate = 3, columnsInReplicate = 4,
                  fixedBlocks = TRUE, twoPhase = "rowThenCol",
                  rngSeeds = c(111, 222))
```

```
getConcurrence(test2, 2,1,1)
```

```
## B FIXED BLOCK: 1 Row by 4 Columns
##
## Treatment-Block
```

```
##
## Number of Concurrences     0  1 2
## Number of Treatment Pairs 21 36 9
```

```
getConcurrence(test2, 2,1,2)
```

```
## B FIXED BLOCK: 3 Rows by 1 Column
##
## Treatment-Block
```

```
##
## Number of Concurrences     0  1 2
## Number of Treatment Pairs 32 32 2
```

```
2/3/test2$ddphase[[2]]$aMeasures[1]
```

```
## [1] 0.5021073
```

By searching with `twoPhase = FALSE` the row concurrences are worse than the `"rowThenCol"` option. However continuing the search quickly finds an equivalent to the `test1` design.

```
test3 <- rcDiGGer(numberOfTreatments = 12,
                  rowsInDesign = 3, columnsInDesign = 12,
                  rowsInReplicate = 3, columnsInReplicate = 4,
                  fixedBlocks = TRUE,
                  twoPhase = "FALSE", rngSeeds = c(111, 222))
```

```
getConcurrence(test3, 1,1,1)
```

```
## B FIXED BLOCK: 1 Row by 4 Columns
##
## Treatment-Block
```

```
##
## Number of Concurrences     0  1  2
## Number of Treatment Pairs 24 30 12
```

```
getConcurrence(test3, 1,1,2)
```

```
## B FIXED BLOCK: 3 Rows by 1 Column
##
## Treatment-Block
```

```
##
## Number of Concurrences     0  1 2
## Number of Treatment Pairs 32 32 2
```

```
2/3/test3$ddphase[[1]]$aMeasures[1]
```

```
## [1] 0.5072034
```

```
test3 <- run(test3, continue = TRUE)
```

```
getConcurrence(test3, 1,1,1)
```

```
## B FIXED BLOCK: 1 Row by 4 Columns
##
## Treatment-Block
```

```
##
## Number of Concurrences     0  1  2
## Number of Treatment Pairs 24 30 12
```

```
getConcurrence(test3, 1,1,2)
```

```
## B FIXED BLOCK: 3 Rows by 1 Column
##
## Treatment-Block
```

```
##
## Number of Concurrences     0  1
## Number of Treatment Pairs 30 36
```

```
2/3/test3$ddphase[[1]]$aMeasures[1]
```

```
## [1] 0.5075503
```

An example of a non-resolvable row-column design for 80 entries in a $[12 \times 20]$ design is given in Venables and Eccleston (1993) with an efficiency bound of 0.8645.

```
rc80 <- rcDiGGer(numberOfTreatments = 80,
                 rowsInDesign = 12, columnsInDesign = 20,
                 nested = FALSE, fixedBlock = TRUE,
                 rngSeeds = c(16867, 12675),
                 maxInterchanges = 500000, twoPhase = FALSE)
```

```
2/3/rc80$ddphase[[1]]$aMeasures[1]
```

```
## [1] 0.8632279
```

5.4 Partially replicated designs, prDiGGer.

Partially replicated designs have some treatments that are unreplicated and rely on replicated treatments to make the trial analysable. Partially replicated design were described in Cullis, Smith, and Coombes (2006). It is recommended that at least 20% of the experimental units are occupied by replicated treatments. The aim of these experiments is usually to select promising

treatments from a set of replicated and unreplicated test treatments, with check and quality standard treatments providing the necessary replication overall to give a valid experiment.

The arguments to `prDiGGer` are:

- `numberOfTreatments`
- `rowsInDesign`
- `columnsInDesign`
- `blockSequence`
- `betweenRowCorrModel`, default `"AR"`.
- `betweenRowCorr`, default `0.5`.
- `betweenColumnCorrModel`, default `"AR"`.
- `betweenColumnCorr`, default `0.5`.
- `treatName`
- `treatNumber`
- `treatRepPerRep`
- `treatGroup`
- `maxInterchanges`, default `20000`.
- `targetGroup`, default `1` causing `"A11"` optimisation in the correlation phase of the search.
- `runSearch`, default `FALSE`. Refers to the final search which spatially optimises placement of replicated treatments within blocks.
- `splitOpt`, default `NULL`. A dimension pair can be specified, `c(r, c)` to define blocks for sub-optimal spatial distribution of replicated entries in over-sized designs.
- `rngSeeds`

Group codes are used to distinguish between the different treatment types, standards and tests, with the default expectation that test treatments are coded as group 1.

`prDiGGer` proceeds by optimising a reduced design of replicated treatments using a scaled version of the overall block sequence. Once these blocks are established they are expanded to the size of the full design and augmented with unreplicated treatments as described in Herzberg and Jarrett (2007). The design has not been spatially optimised at this point.

The final step is to `run` the `DiGGer` object to spatially optimise the location of replicated treatments within the smallest blocks of the blocking sequence.

Deciding on the blocking sequence is a major consideration in these designs. The general recommendation is to divide the designs into blocks that correspond to the replication levels in the design.

pr293 is an unreplicated design with 288 unreplicated treatments and 5 standards each replicated 18 times. Approximately 24% of the experimental units will have replicated treatments. The block sequence has been chosen to match the replication level in the design. The $[7 \times 3]$ blocks result in 18 blocks spread across the design and should allow for each block to contain a set of the 5 standards. The sub-blocks of $[7 \times 1]$ will encourage replicated entries to be evenly placed within columns of the design.

```
pr293 <- prDiGGer(numberOfTreatments = 293, rowsInDesign = 21,
                  columnsInDesign = 18,
                  blockSequence = list(c(7,3), c(7,1)),
                  treatRepPerRep = rep(c(1,18), c(288, 5)),
                  treatGroup = rep(c(1, 2), c(288, 5)),
                     rngSeeds = c(156, 444))
## pr293
pr293 <- run(pr293)
## plot(pr293)
## plot(pr293, trts = 289:293, col = 2, new = FALSE)
```

```
desTab(getDesign(pr293), 7, 3)
```

```
##        B1 B2 B3 B4 B5 B6 B7 B8 B9 B10 B11 B12 B13 B14 B15 B16 B17 B18
## freq_1 21 21 21 21 21 21 21 21 21  21  21  21  21  21  21  21  21  21
```

```
desTab(getDesign(pr293), 21, 1)
```

```
##        B1 B2 B3 B4 B5 B6 B7 B8 B9 B10 B11 B12 B13 B14 B15 B16 B17 B18
## freq_1 21 21 21 21 21 21 21 21 21  21  21  21  21  21  21  21  21  21
```

pr118 is a partially replicated design with 56 unreplicated test treatments, 56 test treatments with two replicates and 6 standards with 4 replicates. The layout $[16 \times 12]$ suggests a nested blocking sequence:

- $[16 \times 6]$ to give 2 blocks
- $[8 \times 6]$ to give 4 blocks
- $[8 \times 1]$ to optimise column blocks.

With `runSearch = TRUE` the final spatial optimisation is run immediately after the blocking phase augmentation.

```
pr118 <- prDiGGer(118, 16, 12,
                  blockSeq = list(c(16,6), c(8,6), c(8,1)),
                  treatRepPerRep = rep(c(1,2,4), c(56,56,6)),
                  treatGroup = rep(c(1,1,2), c(56,56,6)),
                  rngSeeds = c(8842, 9004),
                  runSearch = TRUE)
```

`desTab` is used to give a quick check of replication levels in each block. Notice that both the high replication treatments and the unreplicated treatments are distributed equally to the $[8 \times 6]$ blocks.

```
desTab(getDesign(pr118), 16,6)
```

```
##        B1 B2
## freq_1 84 84
## freq_2  6  6
```

```
desTab(getDesign(pr118), 8,6)
```

```
##        B1 B2 B3 B4
## freq_1 48 48 48 48
```

```
desTab(getDesign(pr118), 8,1)
```

```
##        B1 B2 B3 B4 B5 B6 B7 B8 B9 B10 B11 B12 B13 B14 B15 B16 B17 B18 B19 B20
## freq_1  8  8  8  8  8  8  8  8  8   8   8   8   8   8   8   8   8   8   8   8
##        B21 B22 B23 B24
## freq_1   8   8   8   8
```

```
plot(pr118, trts=57:112, col=5, new=TRUE)
plot(pr118, trts=113:118, col=2, new=FALSE)
plot(pr118, trts=1:56, col=8, new=FALSE, label=FALSE)
```

Figure 5.1 shows the design. The unreplicated treatments are not labelled and can be seen to cluster close to the higher replicated treatments 113 to 118.

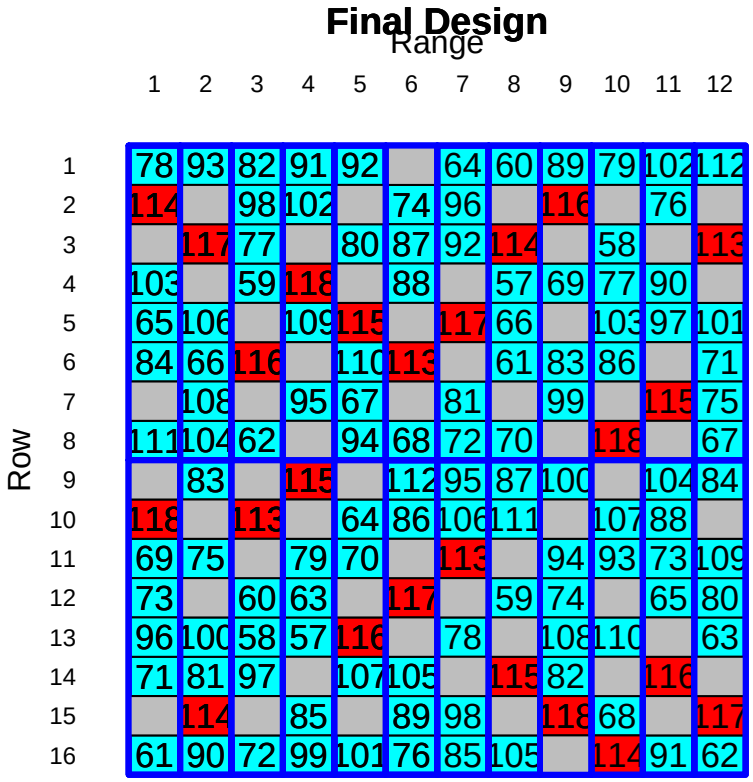

Figure 5.1: Partially replicated design for 118 treatments.

5.5 Factorial designs, facDiGGer

facDiGGer is a workaround to produce factorial designs using DiGGer. The function allocates treatment factors one at a time to the design, forming the interaction treatments at each step and using the last factor added as a group code in the search. corDiGGer is called once for each factor added. For factorial designs in general compromises are made regarding the importance of main effects and interactions in the design optimality measure. The approach used in facDiGGer is to optimise "F1", "F1:F2", ..., "F1:...:Fn" with restrictions on optimisation applied at each phase.

The arguments of facDiGGer are:

- factorNames, a text vector of factor names.
- rowsInDesign
- columnsInDesign
- rowsInReplicate
- columnsInReplicate
- mainPlotSizes, default NULL. If specified it is a sequence of block sizes defining main plot and sub-plot sizes in a split-plot design. The final pair in the list must be c(1, 1).
- treatDataFrame, gives the name of the dataframe with the factorial treatment information, see below.
- treatRepColumn, default column name in data frame "Repeats" with treatment replications per replicate.
- blockSequence
- objectiveWeight, default c(0.8, 0.2) relative weights for multiple objectives.
- searchIntensity
- chequerboard, default FALSE. Used with 2-level factors to break up the default chequerboard pattern that occurs when autoregressive correlation is present.
- spatial
- rowColumn
- designLayoutTemplate, default FALSE. An optional matrix showing replicate numbers with missing values.
- maxInterchanges
- rngSeeds

- rngState

The function expects a treatment dataframe to be supplied. The function createFactorialDF creates a dataframe ready for use by facDiGGer.

The arguments to createFactorialDF are:

- nlevelsVector
- treatName
- treatRepPerRep

After the first factor is allocated, the design for the levels of this factor is fixed. The levels of the second factor are allocated to each level of the first factor within a replicate. Treatment interchanges within replicates are restricted to interchanges between plots having the same first factor level. Before adding subsequent factors the design for the current interaction is fixed, with levels of the next factor allocated to each of the current interaction levels.

The function requires a treatment data frame with a column for each factor and the replication level of each factor combination.

A design template file may be used where there are missing plots in the rectangular design. The design template may have replicate codes which will be used to allocate replicates of the treatments from the treatment data frame. Missing plots are coded as 0.

createFactorialDF is a utility function to create treatment data frames suitable for facDiGGer.

```
DF45 <- createFactorialDF(c(3,3,5))
head(DF45)
```

```
##   TRT F1 Fname1 F2 Fname2 F3 Fname3 Repeats
## 1   1  1     A1  1     B1  1     C1       1
## 2   2  1     A1  1     B1  2     C2       1
## 3   3  1     A1  1     B1  3     C3       1
## 4   4  1     A1  1     B1  4     C4       1
## 5   5  1     A1  1     B1  5     C5       1
## 6   6  1     A1  2     B2  1     C1       1
```

The factorNames order is the order in which factors are allocated. The allocation has better flexibility if the factors are allocated in the order of

increasing number of levels.

```
d3x3x5 <- facDiGGer(
    factorNames = c("F1", "F2", "F3"),
    rowsInDesign = 45, columnsInDesign = 3,
    rowsInRep = 45, columnsInRep = 1,
    blockSequence = list(c(15,1)),
    treatDataFrame = DF45,
    treatRepColumn = "Repeats",
    maxInt=500000, rngSeeds = c(11283, 23951))
```

```
desTab(getDesign(d3x3x5),15,3)
```

```
##        B1 B2 B3
## freq_1 45 45 45
```

```
desTab(matrix(d3x3x5$dlist$F1,45),15,1)
```

```
##        B1 B2 B3 B4 B5 B6 B7 B8 B9
## freq_5  3  3  3  3  3  3  3  3  3
```

```
desTab(matrix(d3x3x5$dlist$F2,45),15,1)
```

```
##        B1 B2 B3 B4 B5 B6 B7 B8 B9
## freq_5  3  3  3  3  3  3  3  3  3
```

```
desTab(matrix(d3x3x5$dlist$F3,45),15,1)
```

```
##        B1 B2 B3 B4 B5 B6 B7 B8 B9
## freq_3  5  5  5  5  5  5  5  5  5
```

Tabulation shows replicates in the second direction and the $[15 \times 1]$ blocks have balanced allocation of main effect levels. If the factor order is changed so that the 5 level factor is not last the third factor allocated is not balanced within 15 unit blocks (not shown).

The example in Section 3.2 of Cochran and Cox (1957) is a factorial with added control with unequally replicated treatments.

```
DF9 <- createFactorialDF(
    c(2,3,5),
    treatName=list(Control = c("Control", "Treated"),
```

Table 5.1: Data frame DF9.

	TRT	F1	Control	F2	Level	F3	Fumigant	Repeats
1	1	1	Control	1	0	1	Nil	4
22	22	2	Treated	2	1	2	CN	1
23	23	2	Treated	2	1	3	CS	1
24	24	2	Treated	2	1	4	CM	1
25	25	2	Treated	2	1	5	CK	1
27	27	2	Treated	3	2	2	CN	1
28	28	2	Treated	3	2	3	CS	1
29	29	2	Treated	3	2	4	CM	1
30	30	2	Treated	3	2	5	CK	1

```
        Level = c(0,1,2),
        Fumigant = c("Nil", "CN", "CS", "CM", "CK")))
DF9 <- DF9[-c(2:15,16:20,21,26),]
DF9$Repeats <- rep(c(4,1), c(1,8))
```

Table 5.1 shows the factor order and replication levels. The argument `chequerboard` is used with two-level factors. When set `TRUE` the standard spatial model is used with two-level factors resulting in chequerboard self-diagonals of the levels. Setting `blockSequence = c(1,1)` is a special case which fits row and column blocks in one objective and spatial correlation in second objective.

```
test9x <- facDiGGer(
    factorNames = c("Control","Level","Fumigant"),
    rowsInDesign = 6, columnsInDesign = 8,
    rowsInRepl = 3, columnsInRep = 4,
    blockSequence = list(c(1,1)),
    chequerboard = TRUE,
    treatDataFrame=DF9, rngSeeds = c(111,444))
```

The resulting design sequence is shown in Figure 5.2. The two-level factor is allocated initially respecting replicate blocks. The second three-level factor is allocated respecting the allocated control treatments within replicates. The final treatment allocation has less flexibility as the replicates and previously

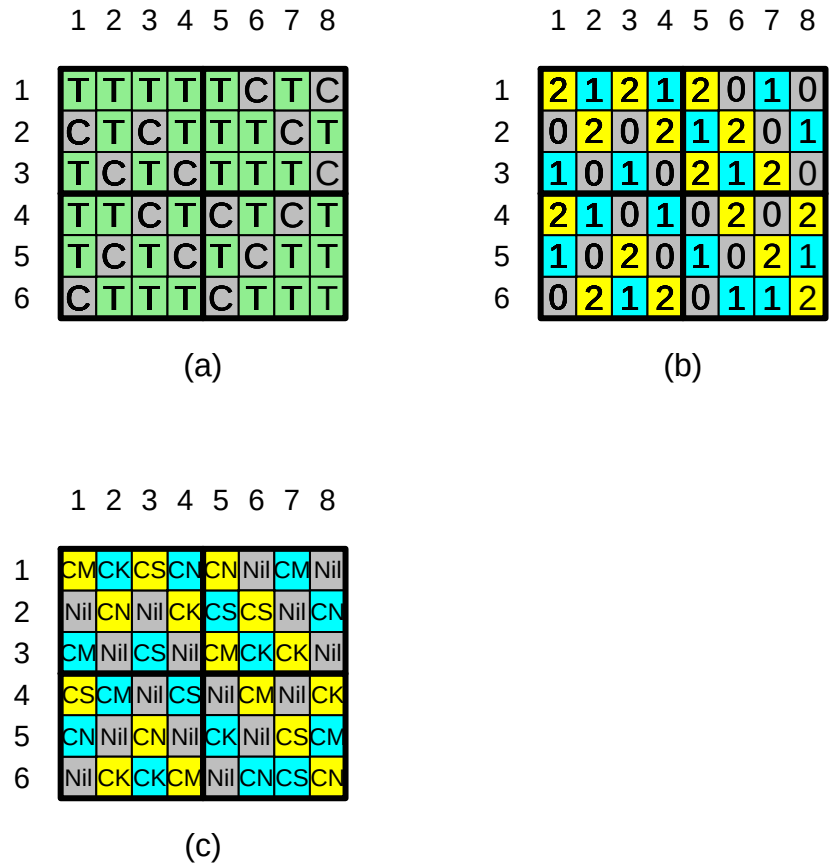

Figure 5.2: Sequential allocation: (a) `Control`, (b) `Level` within `Control` and (c) `Fumigant` within `Control:Level`.

allocated treatments must be respected. At each phase the added treatment is used to define a group code to use with `aType = "Agg"` (Table 5.2. This optimises on comparisons between treatments with different levels of the last factor added.

Phase	Treatments in design	Levels in design	Group code basis
1	`Control`	2	N/A
2	`Control:Level`	3	`Level`
3	`Control:Level:Fumigant`	9	`Fumigant`

Table 5.2: Sequential allocation of treatments

Split-plot designs rely on settings of `mainPlotSizes` to define the nested main plot sizes in split-plot designs. After the split-plot search phases `mainPlotSizes` should be `c(1,1)`. Consider a trial with a 3-level factor in main plots and a 5-level factor on experimental units. The block structure of this experiment is:

- Design: $[15 \times 3]$
- Replicate: $[15 \times 1]$
- Main plot: $[5 \times 1]$
- Sub-plot: $[1 \times 1]$

```
DF15 <- createFactorialDF(c(3,5))
sp3x5 <- facDiGGer(
  factorNames = c("F1", "F2"),
  rowsInDesign = 15, columnsInDesign = 3,
  rowsInRep = 15, columnsInRep = 1,
  mainPlotSizes = list(c(5,1), c(1,1)),
  treatDataFrame = DF15,
  treatRepColumn = "Repeats",
  maxInt=100000)
```

```
plot(sp3x5, trts=1:5, col=5, new=TRUE, label=FALSE)
plot(sp3x5, trts=6:10, col=7, new=FALSE, label=FALSE)
plot(sp3x5, trts=11:15, col=8, new=FALSE, label=FALSE)
desPlot(matrix(sp3x5$dlist$F2,15), new=FALSE)
```

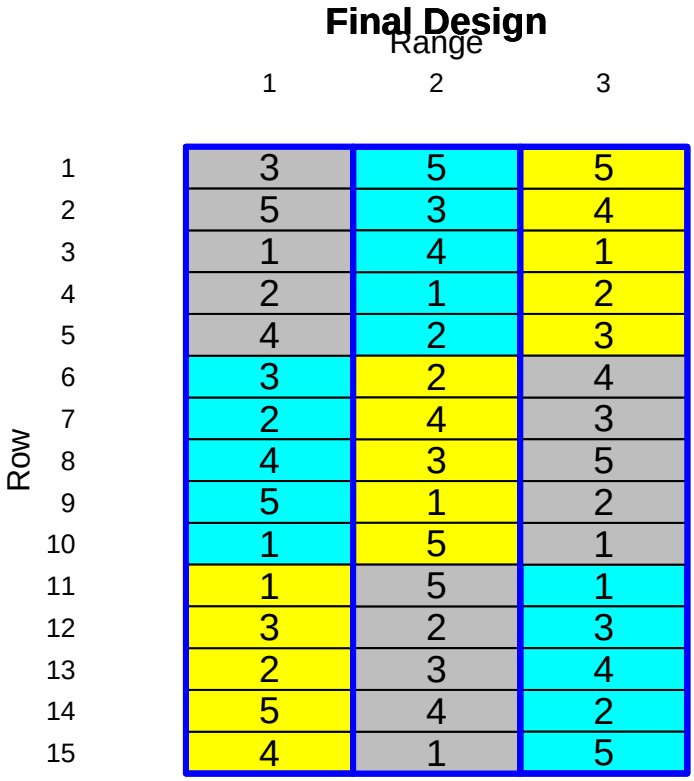

Figure 5.3: Split plot design for 3 main plot and 5 sub-plot treatments.

5.6 Strict 2D designs, `r2dDiGGer`

`r2dDiGGer` forces strict replication in two directions even when the number of replicates do not evenly divide the design dimensions.

The arguments to `r2dDiGGer` are:

- `numberOfTreatments`
- `rowsInDesign`
- `columnsInDesign`
- `rowsInReplicate`
- `columnsInReplicate`
- `maxInterchanges`, default 100000.
- `searchIntensity`, default 100. Percentage of possible interchanges to monitor when a detrimental interchange is accepted.
- `aType`, default `"A++"`, compare all treatments.
- `targetAValue`, default `0`. Search stops when A measure falls below this value.
- `runSearch`, default `TRUE`.
- `rngSeeds`, default `NULL`.
- `rngState`, current state of the random number generator.
- `treatName`
- `treatNumber`
- `treatRepPerRep`
- `treatGroup`
- `spatial`, default `TRUE`, defines separable autoregressive processes for rows and columns, (`"AR"`, `0.5`), applied after the blocking search phases.
- `rowColumn`, default `TRUE`, defines random row and column blocks with variance ratio `1.0`. Applied after blocking phases of the search.
- `nuggetVar`, default variance ratio `0.1` added for each plot in the final search phase.
- `initialDesign`, default `NULL`. Matrix of design numbers.
- `initialSwap`, default `NULL`. Dimension pair or full matrix of codes giving allowable swaps. Plots with the same code may have treatments interchanged.

Strict replication in two dimensions was requested to ensure that any single treatment was spread in well separated zones where rectangular replicates were not possible. The restrictions on randomisation imposed by replication

in two directions are that treatments are divided into sets that:

- always occur together in blocks
- occur with other sets in one block
- never occur together in blocks.

The 20 treatment example looked at in Section 5.1 can be replicated in two directions using `r2dDiGGer`.

```
d20a <- r2dDiGGer(numberOfTreatments = 20,
                  rowsInDesign = 20, columnsInDesign = 3,
                  rowsInRep = 20, columnsInRep = 1,
                  rngSeeds = c(111, 333))
```

(d20aT) Non-rectangular replicates in two directions, `d20a`.

5.7 High frequency control replicated designs, nurseryDiGGer

The `nurseryDiGGer` wrapper function was written to distribute high frequency check treatments evenly across a design where all treatments are replicated. The initial case was a disease nursery where varieties of known susceptibility needed to be spread across the design to monitor the disease pressure across a trial. The principle used in creating the design was to begin with an unreplicated design using only the high frequency controls as replicated treatments. The high frequency controls would be spread across the design to maximise the information about the remaining plots. The second phase of the search keeps the controls in fixed positions and distributes the remaining treatments around them. In some cases there are complete block replicates which need to be respected. Blocking sequences can be used in distributing the extra treatments across the design.

The arguments to `nurseryDiGGer` are:

- `numberOfTreatments`
- `rowsInDesign`
- `columnsInDesign`
- `rowsInReplicate`

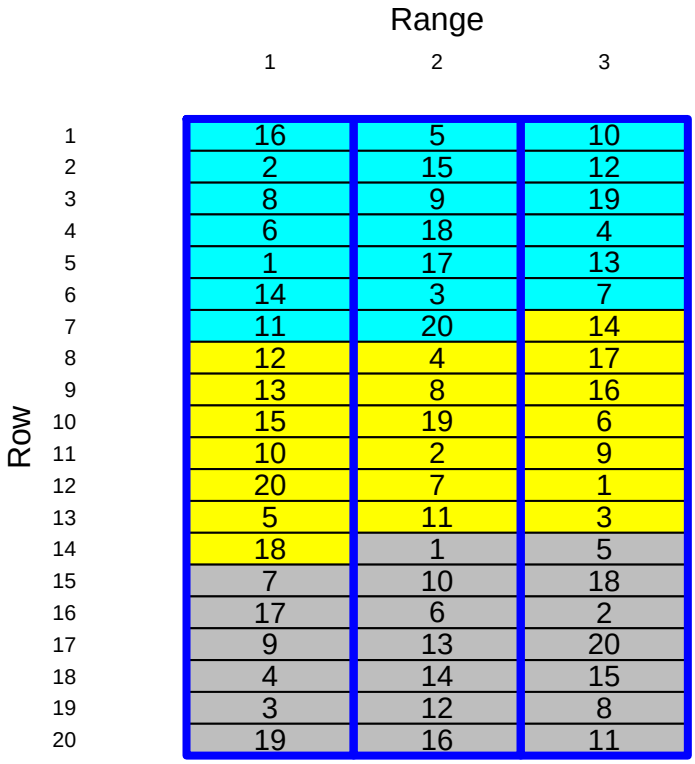

Figure 5.4: (d20aT)

- columnsInReplicate
- blockSequence
- betweenRowCorrModel
- betweenRowCorr
- betweenColumnCorrModel
- betweenColumnCorr
- treatName
- treatNumber
- treatRepPerRep
- treatGroup
- checkGroup
- maxInterchanges
- rngSeeds

A small design for 10 treatments in a $[6 \times 9]$ design with $[6 \times 3]$ replicates illustrates the method. In each replicate two treatments have five replicates and the remaining 8 treatments have no repeats in a replicate.

```
ny1 <- nurseryDiGGer(
    numberOfTreatments = 10,
    rowsInDesign = 6, columnsInDesign = 9,
    rowsInRep = 6, columnsInRep = 3,
    blockSeq = list(c(6,1), c(3,1)),
    treatRep = rep(c(5,1),c(2,8)),
    treatGroup = rep(c(2,1), c(2,8)),
    checkGroup = 2, rngSeeds = c(4964, 17712))
```

The block sequence first blocks to $[6 \times 1]$ columns followed by $[3 \times 1]$ sub-columns which also induce $[3 \times 9]$ blocks.

```
desPlot(getDesign(ny1), 1, col = 2, new = TRUE)
desPlot(getDesign(ny1), 2, col = 3, new = FALSE)
desPlot(getDesign(ny1), 3:10, col = 8, new = FALSE,
        bdef = cbind(6,3), bwd = 3, bcol = 4)
```

Consider an example where the controls do not split neatly into replicates. A $[90 \times 12]$ design has 233 entries with 3 replicates, 4 entries with 76 replicates and one entry with 77 replicates. The blocking sequence should include $[30 \times 12]$ or $[90 \times 4]$ to hold replicates of the 3 replicate entries. By specifying $[30 \times 4]$ blocks both the 3 replicate blocks would be induced. Depending on

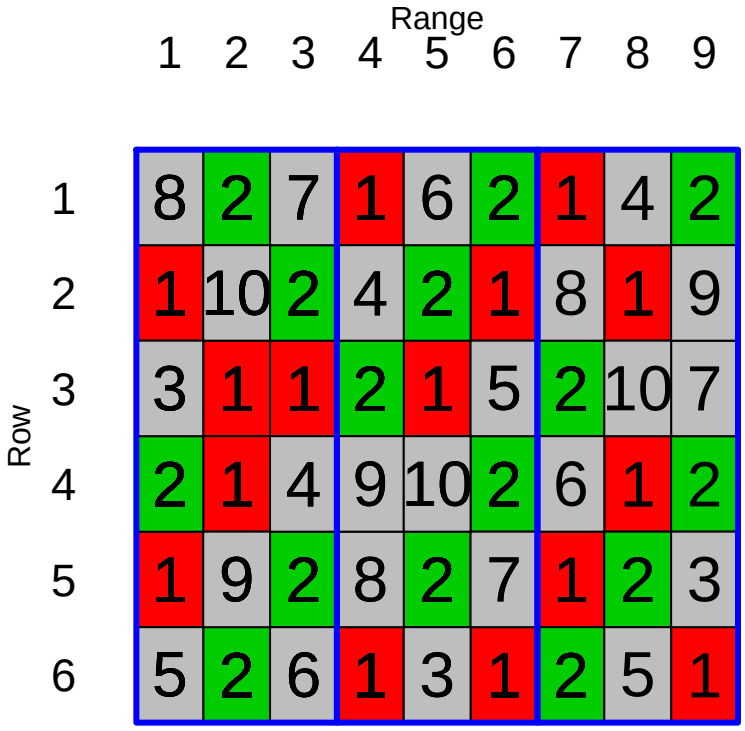

Figure 5.5: Nursery design with defined replicates.

plot shape it may be sensible to include $[15 \times 1]$ blocks which give 72 small blocks nearly corresponding to the maximum replication levels.

```
ny238 <- nurseryDiGGer(
  numberOfTreatments = 238,
  rowsInDesign = 90, columnsInDesign = 12,
  rowsInRep = 90, columnsInRep = 12,
  treatRep = rep(c(3,76,77),c(233,4,1)),
  treatGroup = rep(c(1,2),c(233,5)),
  blockSequence = list(c(30,4), c(15,1)),
  checkGroup = 2, rngSeeds = c(4660, 21436))
##  plot(ny238, trts=1:5 + 233, col = 2)
```

```
desTab(getDesign(ny238), 30,12)
```

```
##          B1  B2  B3
## freq_1  139 142 141
## freq_2   44  41  43
## freq_3    2   3   2
## freq_25   3   3   3
## freq_26   2   2   2
```

```
desTab(getDesign(ny238), 90,4)
```

```
##          B1  B2  B3
## freq_1  133 134 133
## freq_2   50  48  50
## freq_3    0   1   0
## freq_25   3   3   3
## freq_26   2   2   2
```

The tables show that the higher replication treatments have been evenly distributed to the proposed replicate blocks in each direction. The three replicate treatments have not. The search can be continued with a call to `corDiGGer`. The final search when `nurseryDiGGer` is run begins with the high replicate treatments in fixed positions. `ny238$iswap` has swap codes that keep these treatments fixed but has no restriction on swaps between the three replicate entries. The relative importance of blocks in an objective can be controlled with the `blockVarianceRatio` parameters. `setBlock` is used to change the default interaction block variance component from 0.10 to 0.01.

The first phase maximum interchanges is changed to 200000.

```
ny238a <- corDiGGer(
  238, 90, 12, 90, 12,
  treatRep = rep(c(3, 76, 77), c(233, 4, 1)),
  treatGroup = rep(c(1, 2), c(233, 5)),
  blockSequence = list(c(30, 4), c(15, 1)),
  rngState = ny238$.rng,
  initialDesign = getDesign(ny238),
  initialSwap = ny238$iswap, runSearch = FALSE)
setBlock(ny238a, phaseNo = 1, objectiveNo = 1, blockNo = 3,
         rowsInBlock = 30, columnsInBlock = 4,
         blockVarianceRatio = 0.01)
setMaxInterchanges(ny238a, phaseNo = 1, maxInt = 200000)
ny238a <- run(ny238a)
```

```
desTab(getDesign(ny238a),30,12)
```

```
##          B1  B2  B3
## freq_1  233 233 233
## freq_25   3   3   3
## freq_26   2   2   2
```

```
desTab(getDesign(ny238a),90,4)
```

```
##          B1  B2  B3
## freq_1  233 233 233
## freq_25   3   3   3
## freq_26   2   2   2
```

```
desTab(getDesign(ny238a),30,4)
```

```
##        B1 B2 B3 B4 B5 B6 B7 B8 B9
## freq_1 77 77 79 78 78 77 78 78 77
## freq_8  2  2  4  3  3  2  3  3  2
## freq_9  3  3  1  2  2  3  2  2  3
```

The 3 replicate treatments are now replicated in two directions.

Chapter 6

Miscellaneous examples

In this section example designs illustrate some flexible features of the `DiGGer` search program.

6.1 Missing plots

Designs with missing plots can be handled in `corDiGGer` using initial designs with the missing plots coded as `0`. Designs with missing plots must have:

- replicate dimensions equivalent to design dimensions
- correlation block dimensions equivalent to design dimensions.

6.1.1 Missing plots in a completely randomised design

Johnstone (2003) describes an experiment in a $[16 \times 11]$ design where the preparation of some internal plots failed and rows of the design had a gradation of dampness and shade which left some lower boundary plots unusable. The available plots are given as non-zero entries in an initial design matrix `id0`.

```
pl0 <- matrix(1,16,11)
pl0[c(2,10,15,22,44,48,64,80,92,93,94,95,96,104,111,112,
      120,127,128,132,135,142,143,144,146,158,159,160,
      169,174,175,176)] <- 0
```

```
# initial reps in row order
pl0 <- t(pl0)
id0 <- pl0
id0[pl0==1] <- c(sample(8),sample(8),sample(8),
                 sample(8),sample(8),sample(8),
                 sample(8),sample(8),sample(8),
                 sample(8),sample(8),sample(8),
                 sample(8),sample(8),sample(8),
                 sample(8),sample(8),sample(8))
pl0 <- t(pl0)
id0 <- t(id0)
```

```
zcol <- rainbow(8)
par(mfrow = c(1,2), mar = c(2,1,3,1)+0.1)
desPlot(pl0, 0, col = 8, new = TRUE,
        label = FALSE, border = FALSE,
        cstr = "", rstr = "")
desPlot(pl0, 1, col = 7, new = FALSE,
        label = FALSE, lwd = 3)
mtext('(a)', side = 1)
desPlot(id0,0,col=8,new=TRUE,
        label = FALSE, border=FALSE,cstr="",rstr="")
for (i in seq(8)){
  desPlot(id0,i,col=zcol[i],new=FALSE)
}
desPlot(id0,new=FALSE,label=FALSE,bdef=cbind(4,3),bwd=8)
desPlot(id0,new=FALSE,label=FALSE,bdef=cbind(4,3),bwd=4,bcol=7)
mtext('(b)', side = 1)
```

Figure 6.1 shows the available plots and the locations of missing plots. The allocation of initial treatments was done row-wise but swaps were allowed across the full design. Blocks of $[4 \times 3]$ were used to spread treatments across the design. Row and column blocks were specified by `c(1,1)` in the block sequence. This results in multiple objectives in the final search phase:

- random rows and columns with objective weight 0.8
- autoregressive processes with parameter 0.5 between rows and between columns and objective weight 0.2.

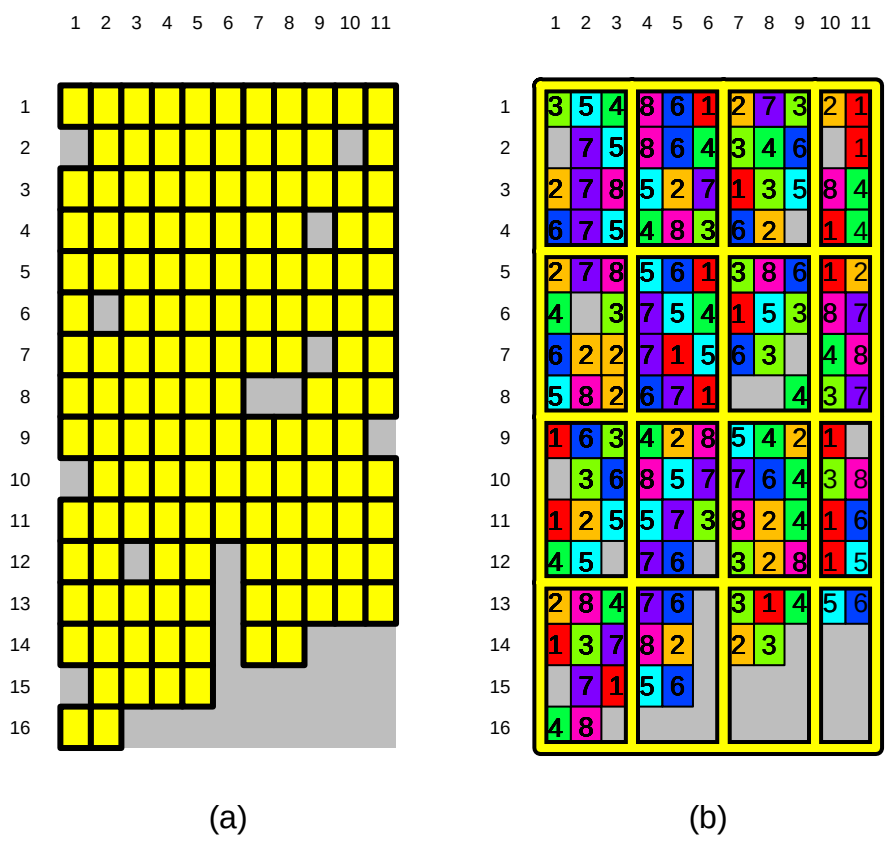

Figure 6.1: Design layout showing: (a) available plots and (b) an initial design showing proposed blocks.

```
mp1 <- corDiGGer(
    numberOfTreatments = 8,
    rowsInDesign = 16, columnsInDesign = 11,
    blockSeq = list(c(4, 3), c(1, 1)),
    initialDesign = id0,
    rngSeeds = c(4548, 9319))
```

Tables of the resulting design show that treatments have been spread across the design.

```
desTab(getDesign(mp1), 4, 11)
```

```
##         B1 B2 B3 B4
## freq_2   0  0  0  1
## freq_3   1  0  0  7
## freq_4   0  1  1  0
## freq_5   7  8  8  0
## freq_6   1  0  0  0
## freq_21  0  0  0  1
```

```
desTab(getDesign(mp1), 16, 3)
```

```
##         B1 B2 B3 B4
## freq_3   0  0  0  8
## freq_4   0  0  3  0
## freq_5   6  7  5  0
## freq_6   3  1  0  0
## freq_7   0  1  0  0
## freq_8   0  0  0  1
## freq_11  0  0  1  0
```

```
desTab(getDesign(mp1), 1, 11)
```

```
##        B1 B2 B3 B4 B5 B6 B7 B8 B9 B10 B11 B12 B13 B14 B15 B16
## freq_1  5  7  5  7  5  7  7  7  7   7   5   7   7   7   4   2
## freq_2  3  2  3  2  3  2  2  2  2   2   3   2   2   0   0   0
## freq_4  0  0  0  0  0  0  0  0  0   0   0   0   0   1   0   0
## freq_7  0  0  0  0  0  0  0  0  0   0   0   0   0   0   1   0
## freq_9  0  0  0  0  0  0  0  0  0   0   0   0   0   0   0   1
```

```
desTab(getDesign(mp1), 16, 1)
```

```
##        B1 B2 B3 B4 B5 B6 B7 B8 B9 B10 B11
## freq_1  3  2  2  2  2  5  3  3  5   4   4
## freq_2  5  7  7  7  7  3  5  5  3   4   4
## freq_3  1  0  0  0  0  0  1  1  0   0   0
## freq_4  0  0  0  0  0  0  0  0  0   1   1
## freq_5  0  0  0  0  0  1  0  0  1   0   0
```

```
desTab(getDesign(mp1), 4, 3)
```

```
##        B1 B2 B3 B4 B5 B6 B7 B8 B9 B10 B11 B12 B13 B14 B15 B16
## freq_1  6  6  6  6  4  4  6  6  6   7   4   5   8   8   8   2
## freq_2  3  3  3  3  4  4  3  0  3   1   4   0   0   0   0   0
## freq_3  0  0  0  0  0  0  0  0  0   1   0   0   0   0   0   0
## freq_6  0  0  0  0  0  0  0  1  0   0   0   0   0   0   0   1
## freq_7  0  0  0  0  0  0  0  0  0   0   0   1   0   0   0   0
```

The final design is shown in Figure 6.2.

6.1.2 Missing plots in a replicated complete block design

Another example that requires missing plots is the situation where 17 treatments with 3 replicates are to fit in a $[6 \times 9]$ design. Two options could be considered:

- three identical $[6 \times 3]$ blocks with 1 missing plot
- interlocking replicates with the last 3 plots missing

In the code that follows swap matrices are created and initial allocations of 17 treatments are formed from them.

```
s17a <- matrix(rep(c(1,0,2,0,3,0), c(17,1,17,1,17,1)), 6)
m17a <- s17a
s17b <- matrix(rep(c(1,2,3,0), c(17,17,17,3)),6)
m17b <- s17b
for (i in seq(3)){
    m17a[s17a == i] <- sample(17)
```

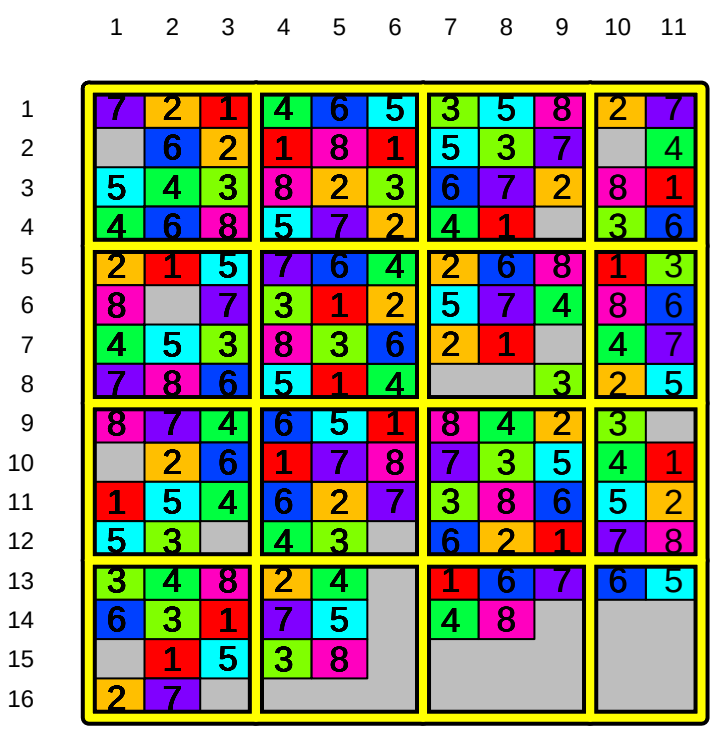

Figure 6.2: Final design with blocks and missing plots.

```
    m17b[s17b == i] <- sample(17)
}
```

Searches can now be run restricting swaps using `initialSwap` acting on initial designs provided by `initialDesign`.

```
mp2a <- corDiGGer(
    numberOfTreatments = 17,
    rowsInDesign = 6, columnsInDesign = 9,
    blockSequence = list(c(2,3)),
    initialDesign = m17a,
    initialSwap = s17a)
mp2b <- corDiGGer(
    numberOfTreatments = 17,
    rowsInDesign = 6, columnsInDesign = 9,
    blockSequence = list(c(2,3)),
    initialDesign = m17b,
    initialSwap = s17b)
```

The first search phase splits the design into $[2 \times 3]$ blocks which will have unequal numbers of treatments. The final search phase has the standard objective of spatial plus random rows and columns.

The final designs are shown in Figure 6.3.

6.1.3 Missing plots in a factorial design

A design for 41 treatments was required for 4 varieties by 10 treatments with added control. The control treatment was to be included 4 times in each replicate. The design was $[15 \times 12]$ with replicates $[15 \times 3]$ with one missing plot per replicate.

With factorial designs a design template can be used to specify the location of the missing plots. The template may also include replicate codes. Where replicate codes are used the repeats in the treatment dataframe are repeats per replicate.

```
DF31 <- createFactorialDF(c(2,5,11))
indf <- DF31$F1 == 1 | DF31$F2 == 1 | DF31$F3 == 1
```

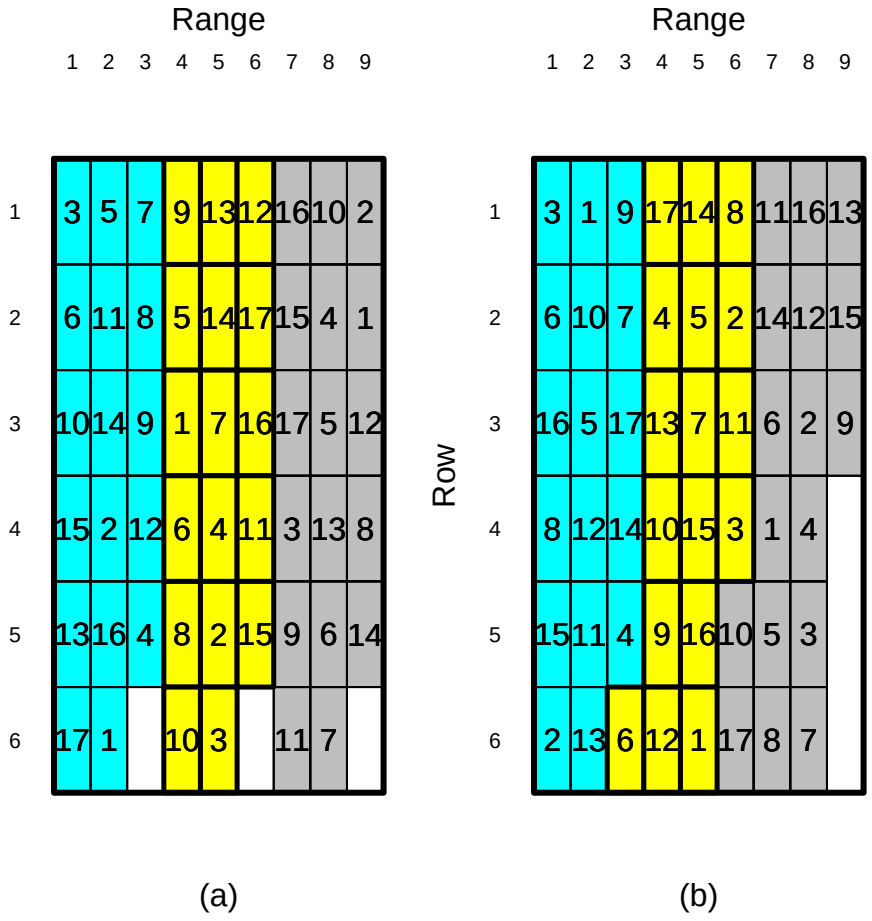

Figure 6.3: Alternative layouts with missing plots.

```
indf[1] <- FALSE
DF31 <- DF31[!indf,]
DF31$Repeats[1] <- 4
template44 <- findblks(15,12, 15,3)
template44[cbind(c(15,15,15,15),c(1,6,7,12))] <- 0
```

The template is shown in Figure 6.4 together with the design obtained with the following code.

```
f31 <- facDiGGer(
    factorNames = c("F1", "F2", "F3"),
    rowsInDesign =15, columnsInDesign =12,
    blockSeq = list(c(5, 3)),
    designLayoutTemplate = template44,
    treatDataFrame = DF31, rngSeeds = c(8871, 6732))
```

A quick indication of how overall treatments and factor levels are spread across $[5 \times 12]$ blocks can be found using `desTab` with `groupCode}` set for the factor of interest.

```
desTab(getDesign(f31), 5,12)
```

```
##        B1 B2 B3
## freq_1 25 26 29
## freq_2 15 14 11
## freq_4  0  0  1
## freq_5  1  0  1
## freq_6  0  1  0
```

```
desTab(getDesign(f31), 5,12, groupCode = DF31$F2)
```

```
##         B1 B2 B3
## freq_4   0  0  1
## freq_12  1  0  2
## freq_13  0  1  0
## freq_14  1  2  1
## freq_15  1  0  0
## freq_17  0  1  1
## freq_18  1  0  0
```

```
desTab(getDesign(f31), 5,12, groupCode = DF31$F3)
```

```
##          B1 B2 B3
## freq_4    0  0  1
## freq_5    4  6  8
## freq_6    5  3  1
## freq_9    1  0  1
## freq_10   0  1  0
```

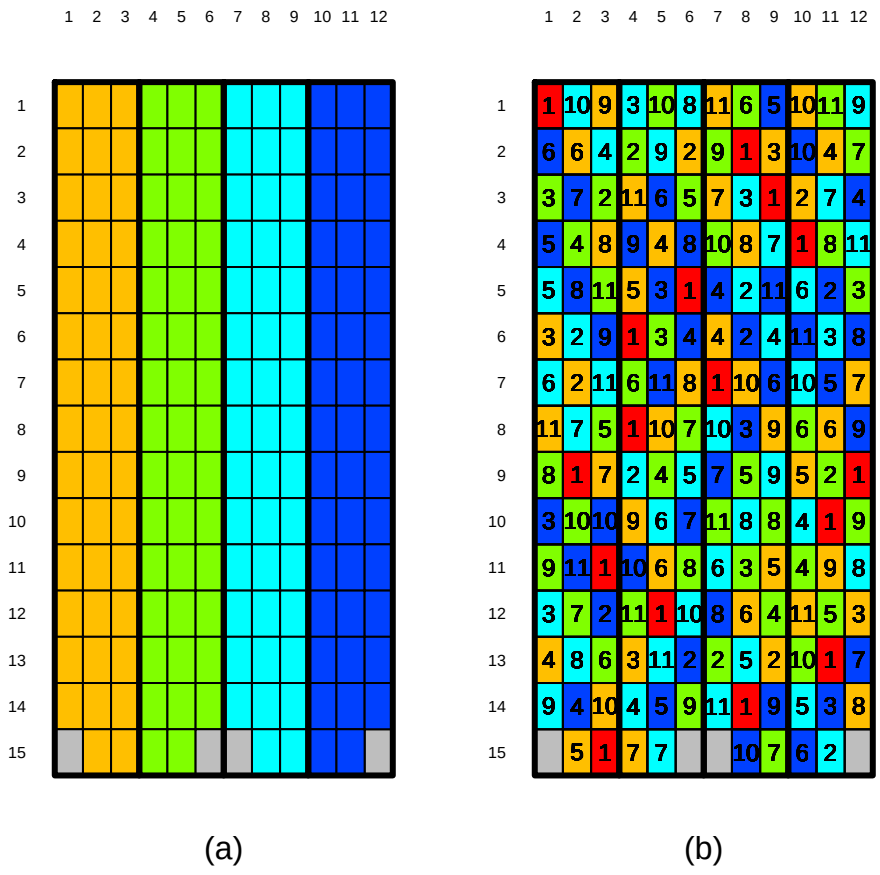

Figure 6.4: Design template (a) and finaldesign (b) for the replicated factorial experiment with missing plots.

6.2 Restricting randomisation, a design with conditions

An initial design and an intitial swap matrix may be used to restrict randomisations to meet a researchers requirements. Consider the case where there are: - all entries included in the first $[12 \times 10]\}$ block of a $[24 \times 10]$ design - 39 entries with 4 replicates - 32 entries with two replicates in each block - 7 entries with one occurrence in the first block and 3 replicates in the second block - 35 entries with 2 replicates, one in each block - 14 entries with 1 occurrence in the first block only.

The first 39 entries may be split $32:7$ randomly and the appropriate replications added to each $[12 \times 10]$ block.

```
t4 <- sample(39)
t2 <- sample(35)+39
t1 <- sample(14)+74
idsgn <- matrix(
    c(sample(c(t4[seq(7)], t4[seq(32)+7], t4[seq(32)+7],
               t2, t1)),
      sample(c(t4[seq(7)], t4[seq(7)], t4[seq(7)],
               t4[seq(32)+7], t4[seq(32)+7], t2))),
    ncol=10, byrow=TRUE)
```

With only 14 unreplicated entries and the restriction on randomisation `corDiGGer` can be used to produce an appropriate design. The initial design is supplied using `initialDesign` and swaps are restricted to $[12 \times 10]$ using `initialSwap`. Blocking using $[6 \times 5]$ blocks spread the unreplicated entries across the first major block. These are coded using `blockSequence`.

```
pr88 <- corDiGGer(
    numberOfTreatments = 88,
    rowsInDesign = 24, columnsInDesign = 10,
    blockSequence = list(c(6,5)),
    treatRepPerRep = c(table(idsgn)),
    initialDesign = idsgn, initialSwap=c(12,10))
```

```
desTab(getDesign(pr88), 24, 5)
```

```
##        B1 B2
## freq_1 41 44
## freq_2 38 38
## freq_3  1  0
```

```
desTab(getDesign(pr88), 12, 10)
```

```
##        B1 B2
## freq_1 56 35
## freq_2 32 32
## freq_3  0  7
```

```
desTab(getDesign(pr88), 6, 10)
```

```
##        B1 B2 B3 B4
## freq_1 60 60 52 54
## freq_2  0  0  4  3
```

6.3 Multi-site p-rep designs

Consider an example where entries from several families are to be allocated across multiple sites. `corDiGGer` may be used to allocate entries to sites before each site is optimised separately. An example in Feoktistov, Pietravalle, and Heslot (2017) describes a design where 3 replicates of 400 entries from 3 families are allocated across 5 sites. The families have 14, 187 and 189 entries respectively with 240 entries to be allocated to each site. The full layout is specified as $[240 \times 5]$ and columns represent the sites. Designating replicates as $[80 \times 5]$ avoids the problem of too many possible swaps if an initial completely randomised design is used. Column blocks are specified using `blockSequence` and the spatial plus row-column optimisation phase is not applied. `treatGroup` has codes for the families and `aType=Agg` optimises comparisons between entries from different families.

```
f400r3 <- corDiGGer(400,240, 5, 80, 5,
                    treatGroup = rep(c(1,2,3),c(14,187,199)),
                    blockSeq = list(c(240,1)),
                    spatial=FALSE, rowCol=FALSE,
                    aType='Agg',
```

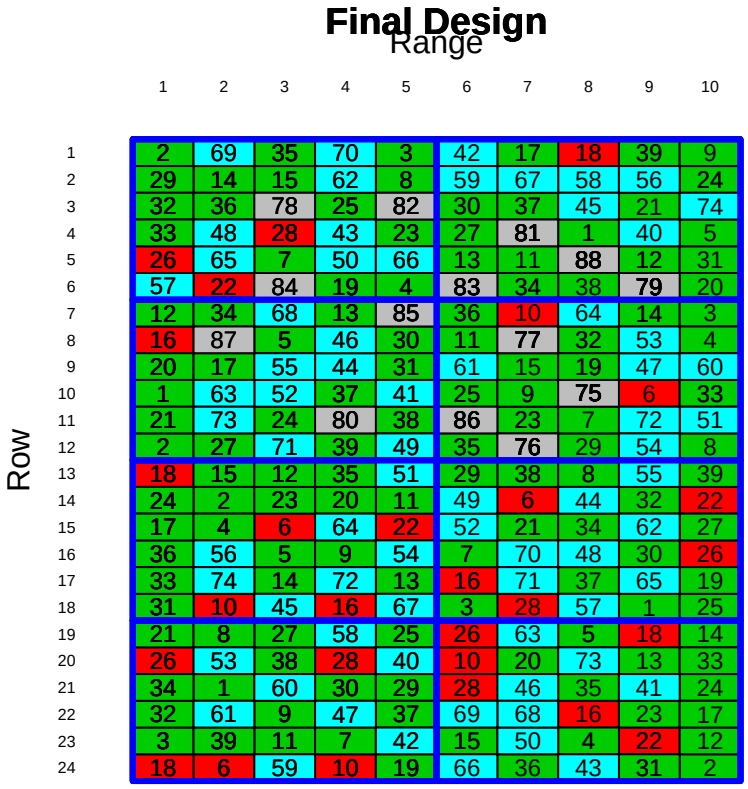

Figure 6.5: Partially replicated design with conditions.

```
                    maxInt=20000, rngSeeds = c(10207, 1084))
```

Tabulation is used to check family distribution to sites.

```
##         B1  B2  B3  B4  B5
## freq_1 240 240 240 240 240
```

```
##          B1 B2 B3 B4 B5
## freq_8    1  0  0  1  1
## freq_9    0  1  1  0  0
## freq_112  0  1  1  1  1
## freq_113  1  0  0  0  0
## freq_119  1  1  1  0  0
## freq_120  0  0  0  1  1
```

The occurrences of each line at each site split for families is tabulated.

```
##
##             F1 F2 F3
##   0.0.1.1.1  0 19 21
##   0.1.0.1.1  2 20 18
##   0.1.1.0.1  1 18 21
##   0.1.1.1.0  3 17 20
##   1.0.0.1.1  2 20 18
##   1.0.1.0.1  2 18 20
##   1.0.1.1.0  1 18 21
##   1.1.0.0.1  1 17 22
##   1.1.0.1.0  0 18 22
##   1.1.1.0.0  2 22 16
```

The optimisation of families within a single site is illustrated with an example from Feoktistov, Pietravalle, and Heslot (2017). Family 1 has a check variety with 9 replicates and 39 unreplicated lines. Family 2 has a check variety with 8 replicates and 39 unreplicated lines. Family 3 has a check variety with 8 replicates and 41 unreplicated lines. The $[12 \times 12]$ design can be split into 9 $[4 \times 4]$ blocks nearly corresponding to the replication levels of the check varieties. The AR1 correlation parameters are reduced to 0.1 to allow for correlation between plots but to discourage self-diagonals in the checks.

```
pr122 <- prDiGGer(122, 12, 12,
                  treatRep = rep(c(9,8,8,1), c(1,1,1,119)),
                  treatGroup = rep(c(2,2,2,1), c(1,1,1,119)),
                  blockSeq = list(c(4,4), c(4,1)),
                  betweenRowCorr = 0.1, betweenColumnCorr = 0.1,
                  rngSeeds = c(19066, 12912))
pr122 <- run(pr122)
```

The pr122 design does not use family information to distribute the unreplicated across the design. The check entries may be held fixed while the family group codes of unreplicated lines are used in an optimisation.

```
iswap122s <- getDesign(pr122)
iswap122s[iswap122s > 3] <- 4
d122g3 <- corDiGGer(
    122, 12, 12,
    treatRep = rep(c(9,8,8,1), c(1,1,1,119)),
    treatGroup = rep(c(1,2,3,1,2,3), c(1,1,1,39,39,41)),
    blockSeq = list(c(4,4), c(4,1)),
    initialDesign = getDesign(pr122),
    initialSwap = iswap122s, rngState = pr122$.rng)
desPlot(getDesign(d122g3), trts=4:42, new=TRUE, label=FALSE,
        col='lightcyan1')
desPlot(getDesign(d122g3), trts=43:81, new=FALSE, label=FALSE,
        col='orchid1')
desPlot(getDesign(d122g3), trts=82:122, new=FALSE, label=FALSE,
        col='khaki1')
desPlot(getDesign(d122g3), trts=1, new=FALSE, label=TRUE, col=5)
desPlot(getDesign(d122g3), trts=2, new=FALSE, label=TRUE, col=6)
desPlot(getDesign(d122g3), trts=3, new=FALSE, label=TRUE, col=7,
        bdef=cbind(12,12), bcol=4, bwd=4)
```

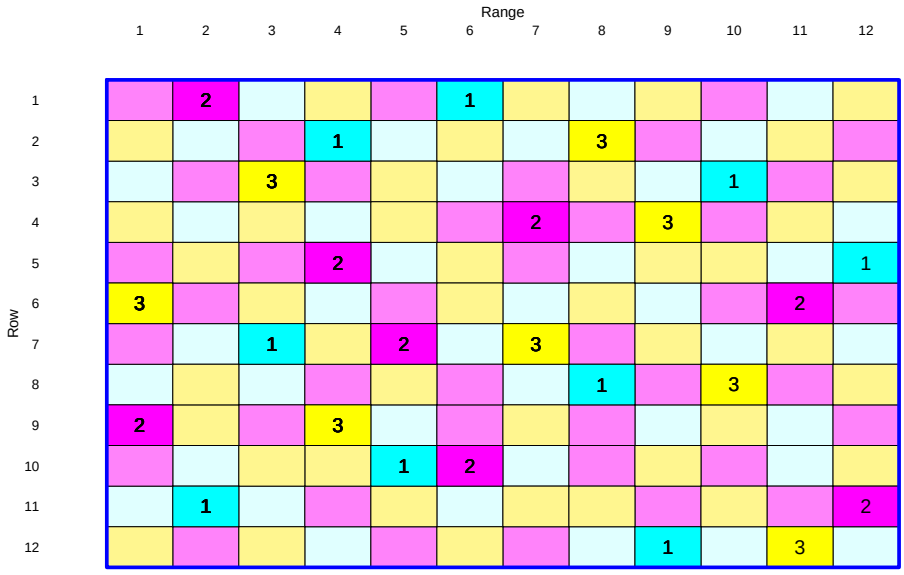

Figure 6.6: Partially replicated design showing family groups.

Chapter 7

References

Battiti, R., and G. Tecchiolli. 1994. "The Reactive Tabu Search." *ORSA Journal on Computing* 6 (2): 126–40.

Cochran, W.G., and G.M. Cox. 1957. *Experimental Designs*. 2nd ed. New York: Wiley International Edition.

Coombes, N.E. 2002. "The Reactive Tabu Search for Efficient Correlated Experimental Designs." PhD thesis, Liverpool John Moores University.

Cullis, B.R., A.B. Smith, and N.E. Coombes. 2006. "On the Design of Early Generation Variety Trials with Correlated Data." *Journal of Agricultural, Biological and Environmental Statistics* 11: 381–93.

Feoktistov, V., S. Pietravalle, and N. Heslot. 2017. "Optimal Experimental Design of Field Trials Using Differential Evolution." In *2017 IEEE Congress on Evolutionary Computation, CEC 2017, Donostia, San Sebastián, Spain, June 5-8, 2017*, 1690–6. https://doi.org/10.1109/CEC.2017.7969505.

Herzberg, A.M., and R.G. Jarrett. 2007. "A-Optimal Block Designs with Additional Singly Replicated Treatments." *Journal of Applied Statistics* 34: 61–70.

Johnstone, P. 2003. "Random Generation and Selection of One- and Two-Dimensional Designs for Experiments on Blocks of Natural Size." *Journal of Agricultural, Biological and Environmental Statistics* 8: 67–74.

"NSW Dpi Biometrics Software Download Page." 2018. http://nswdpibiom.org/austatgen/software.

R Core Team. 2015. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. https://www.R-project.org.

Venables, W.N., and J.A. Eccleston. 1993. "Randomized Search Strategies for Finding Near-Optimal Block or Row-Column Designs." *Australian Journal of Statistics* 35: 371–82.