Journal of Statistical Software

November 2015, Volume 68, Code Snippet 1. *doi: 10.18637/jss.v068.c01*

A Genetic Algorithm for Selection of Fixed-Size Subsets with Application to Design Problems

Mark A. Wolters
Shanghai Center for Mathematical Sciences

Abstract

The R function `kofnGA` conducts a genetic algorithm search for the best subset of k items from a set of n alternatives, given an objective function that measures the quality of a subset. The function fills a gap in the presently available subset selection software, which typically searches over a range of subset sizes, restricts the types of objective functions considered, or does not include freely available code. The new function is demonstrated on two types of problem where a fixed-size subset search is desirable: design of environmental monitoring networks, and D-optimal design of experiments. Additionally, the performance is evaluated on a class of constructed test problems with a novel design that is interesting in its own right.

Keywords: discrete optimization, heuristics, evolutionary computation, network design, D-optimal design, optimization test problems.

1. Introduction

This article introduces an R (R Core Team 2015) function for selecting a subset of fixed size k from a set of n alternatives, given an arbitrary objective function measuring the quality of any subset. The function implements a genetic algorithm (GA), and should be suitable for various subset selection problems where k is constant and n is large enough to make exhaustive search impractical. Two types of problems fitting this description are the design of environmental monitoring networks, as discussed by Le and Zidek (2006); and D-optimal experimental design.

The use of stochastic search methods (including GA) for subset selection is not new. Various algorithms for carrying out such a search have been described in the literature, and some freely available software packages exist. At present, however, there appears to be no publicly available code that both i) restricts the search to only subsets of size k, and ii) places no restrictions on the objective function. The function described here, called `kofnGA`, aims to fill this gap.

The remainder of this section introduces the environmental monitoring network design and the D-optimal design problems, and reviews some of the relevant work in the area. Section 2 describes the `kofnGA` function and its use. Section 3 demonstrates the function's performance on seven test problems. The first four of the test cases are design problems, while the last three are constructed subset selection problems with a novel design that allows control over the size and difficulty of the problem. Section 4 provides a brief summary of the results and considers how further improvements might be made.

Remark. Because GA constitute a well established optimization approach, this work contains only limited consideration of the new algorithm's sensitivity to its tuning parameters and its performance in comparison with other optimization algorithms. Rather, the test problems of Section 3 are used to demonstrate that the new function is a reasonable implementation of a GA, that can be used with confidence to get good approximate solutions to subset selection problems. As with any optimization heuristic, application specific tuning of the algorithm is important to achieve optimal performance (and good performance relative to competitors). Further comments on this point are given at the end of Section 2.2.

1.1. Environmental monitoring network design

Environmental monitoring networks consist of a set of monitoring stations placed at various locations in the region of interest. The need for efficient use of resources leads to the question of optimal placement of monitoring stations. Given a pre-existing network, the network design problem can be one of extension (adding new sites to existing ones), of reduction (removing sites), or of redesign (adding some sites and removing others). Each of these cases is discussed by Le and Zidek (2006), and all are expressed as a search for the set of sites that maximize the determinant of an appropriately specified (hyper-)covariance matrix.

Take the network extension problem as default. If k new monitoring stations are to be added to a network, and there are n potential sites to choose from, the goal is to select the best subset of k sites from the n candidates – that is, to choose one combination from the $\binom{n}{k}$ possibilities that is optimal in some sense. Because of the combinatorial nature of the problem, one would be satisfied with fast near-optimal solutions when n is large.

Le and Zidek (2006, Chap. 11) define optimality in terms of the determinant of the covariance matrix of the selected sites. Label the candidate sites by the integers $1 \dots n$, and let the $n \times n$ matrix $\mathbf{\Sigma}$ be the covariance matrix for all n candidate sites. Let $\mathbf{v}$ be an index vector holding the indices of the k chosen sites. Define the $k \times k$ covariance matrix of the chosen sites to be $\mathbf{\Sigma_v}$; it is formed by taking the intersection of the rows and columns of $\mathbf{\Sigma}$ that are indexed by $\mathbf{v}$ (such a matrix is called a *principal submatrix* of $\mathbf{\Sigma}$ indexed by $\mathbf{v}$). The goal of optimization is to choose $\mathbf{v}$ such that the determinant $|\mathbf{\Sigma_v}|$ is maximized. This criterion is based on the maximum entropy principle. Other factors, such as the cost of adding a station at each site, can also be included in the problem statement with suitable modifications to the objective function.

Recent work by Ruiz-Cárdenas, Ferreira, and Schmidt (2010) specifically addresses the need for fast approximate solutions to network design problems. They consider three different algorithms: a simulated annealing algorithm, a GA, and a GA incorporating a local search step to speed up convergence. Their GA bears much resemblance to the independently developed

GA described here, however their article does not include code for running the search.

For the R computing environment, two packages offer GA subset selection capability, but neither is well suited to the network design problem. The **genalg** package (Willighagen 2015) contains GA functions for general continuous and discrete optimization, but the package does not allow restriction to a single subset size. The package **subselect** (Orestes Cerdeira, Duarte Silva, Cadima, and Minhoto 2015) contains functions for size constrained subset selection using both GA and simulated annealing, but it is restricted to use only certain built-in objective functions suitable for variable selection.

1.2. D-optimal design

Another large body of work that involves selection of fixed-size subsets, and therefore has many relevant algorithms, is D-optimal experimental design. The goal of D-optimal design is to choose k of n possible experimental runs such that the resulting model matrix $\mathbf{X}$ maximizes $|\mathbf{X}^\top\mathbf{X}|$. The objective function is again the determinant of a matrix that is dependent on the chosen subset.

As with the network design problem, exhaustive search for D-optimal designs is impractical when the number of candidate runs is sufficiently large. An early approximate algorithm, based on repeated exchanges of rows, was given by Mitchell (1974a; 1974b). An exact method using branch-and-bound to improve search efficiency has also been used for problems of moderate size (Ko, Lee, and Queyranne 1995); the key to this method is an upper bound for the solution, found using the eigenvalue interlacing property of symmetric matrices.

Other papers have proposed GAs for finding D-optimal designs. Broudiscou, Leardi, and Phan-Tan-Luu (1996) specifically address the value of GA for this application, and use a GA implementation that searches designs of all sizes. They employ a normalized determinant criterion to prevent overly large designs. More recently, Hamada, Martz, Reese, and Wilson (2001), Hereida-Langner, Montgomery, Carlyle, and Borror (2004), and Mandal, Wu, and Johnson (2006) employ various forms of GA for choosing D-optimal designs in different settings. Importantly, none of these works provide readily available code implementing their algorithms.

2. Algorithm description

The new subset selection function is called `kofnGA(k, n, OF, ...)`. Its required inputs are the subset size `k`, the number of candidates `n`, and the objective function `OF`. In keeping with common optimization practice, the function attempts to minimize `OF`. The function `OF` takes as its first argument a k-vector of indices representing a particular subset.

2.1. GA design

This section discusses the main design decisions implemented in `kofnGA`. A basic understanding of GAs is assumed; see, for example, Michalewicz and Fogel (2004), Talbi (2009), or Holland (1992) for more background on GA design and implementation.

Population

The population is a set of candidate solutions (each solution is an index vector of length

k). The fitness of each solution is determined by its objective function value. Since we are minimizing, fitness is based on reverse `OF` ranking, e.g., for a population of size P, the solution with the ith lowest objective value gets fitness score $P - i$. Larger sized populations will tend to improve the diversity of the search, especially in the early stages, but with a corresponding computational cost. The default population size in `kofnGA` is 200.

Selection

At each generation enough offspring are created to completely replace the previous generation. Mating pairs of solutions are chosen by tournament selection. For each new offspring, one parent is chosen by first selecting t solutions at random to form a tournament. The parent is chosen by sampling from this group, with probabilities proportional to within-tournament fitness ranks. The other parent is chosen similarly using another tournament. This form of selection ensures that poorer solutions can occasionally become parents, and gives easy control over the selection pressure. Increasing t puts more emphasis on good solutions and speeds up convergence. By default, the tournament size is set to be one tenth of the population size.

Crossover and mutation

Each new offspring is formed by first pooling the unique indices of the two parent vectors, and then sampling a set of k unique indices from this pool, uniformly at random. After this crossover step, each element of every new offspring has a fixed probability of undergoing mutation. Elements selected for mutation are assigned a new value at random from all of the indices not currently in the vector. Higher mutation rates will improve exploration of the search space, while reducing the convergence rate of the population. The default mutation probability is 0.01.

Elitism

Elitism refers to carrying forward the best solutions from the previous generation into the next generation's population. If the population size is P, let P_e be the number of elites to retain. After all mutated offspring are created, their fitness is evaluated. The new population is formed by combining the P_e most fit solutions from the old generation with the $P - P_e$ most fit offspring. Increasing the proportion of elites retained in the population will accelerate convergence; by default, one tenth of the population is reserved for elites.

Stopping

No stopping rule has been included in the function. The user must supply a number of generations to run (or accept the default value of 500). Setting the number of generations involves a trade-off between run time (which is highly dependent on the complexity of the objective function) and the quality of the eventual solution. In practice a series of short trial runs is usually adequate to determine an appropriate number of generations.

The usual evolution of the population is as follows. An initial sequence of generations sees the best `OF` value decrease rapidly, with a simultaneous decrease in the diversity of the population. During this period the crossover step is producing a variety of solutions and better regions of the search space are being found. Eventually this period ceases, and the population is quite homogeneous. The best `OF` value will continue to decrease occasionally, but improvements on the solution become smaller and less frequent. During this latter stage the improvements in the solution are largely due to mutations.

2.2. Using the function

The function `kofnGA` is available in the R package **kofnGA** (Wolters 2015), which is available from the Comprehensive R Archive Network (CRAN) at http://CRAN.R-project.org/package=kofnGA. The function's source and help files include detailed comments describing all optional arguments, outputs, and steps of the algorithm. The following trivial example demonstrates how the function is used. Consider the problem of choosing the $k = 4$ smallest numbers from a list of $n = 100$ values generated from a standard uniform distribution. The following code generates an instance of this problem.

```
R> Numbers <- sort(runif(100))
R> Numbers[1:6]

[1] 0.01188266 0.03572477 0.04114588 0.04510093 0.05307582 0.05897293
```

The vector of generated values in `Numbers` has been sorted, so that the first four elements of the vector contain the optimal solution. In other words, the indices $(1, 2, 3, 4)$ encode the optimal solution.

The following lines create the objective function for the search, call `kofnGA` with default settings, and display the results of the search.

```
R> ObjFun <- function(v, some_numbers) sum(some_numbers[v])
R> out <- kofnGA(n = 100, k = 4, OF = ObjFun, some_numbers = Numbers)
R> summary(out)


Genetic algorithm search, 500 generations
Number of unique solutions in the final population: 1

Objective function values:
                     average   minimum
Initial population 1.8049168 0.5970088
Final population   0.1338542 0.1338542

Best solution (found at generation 10):
1 2 3 4
```

The package includes a `summary` method (used above), as well as `print` and `plot` methods for inspecting the results. From the summary we can see that the globally minimal solution was found early on in the search, and that the final population is completely composed of this best solution (i.e., the search has converged). The default run of 500 generations is overly long for this problem.

The `kofnGA` function imposes no particular structure on the objective function, other than that its first argument (here, `v`) be an index vector of length k that encodes a solution. In the above example, the extra argument `some_numbers` is passed through to `ObjFun` to allow it to compute the desired sum.

The output of the function is a list with member `bestsol` holding the indices of the best solution found and `bestobj` holding the corresponding objective function value. Additional members of this list record details of the search history, and are described in the function's comments and its help file. Optional arguments `popsize`, `keepbest`, `ngen`, `tourneysize`, and `mutprob` allow user control of, respectively, the population size, the number of elites, the number of generations, the size of the selection tournament, and the mutation probability. An additional argument `initpop` can also be used to specify the initial population (which is otherwise randomly generated). This input is useful for continuing a search that is deemed to have terminated too early.

The optional arguments' default values have been chosen to work well with problems of moderate size (for example, $n < 300$ and $k < 20$). When the objective function is reasonably efficient to compute, good solutions to problems of this scale can be obtained in seconds or minutes. If further adjustment of the values is desired, the following insights may be helpful. They are based on experience with a variety of problems.

- Changing the value of `popsize` is the simplest way to modify the behavior of the search. Note that in addition to increasing the number of generations needed to obtain good solutions, increasing the population size will also increase the computation time per generation (because the objective function must be evaluated once for each solution in the population).

- Adjusting `mutprob` is the next most straightforward way to control the search. It is helpful to remember that if the mutation rate is ρ, the number of mutations in an offspring solution is binomially distributed with mean $k\rho$. As most mutations will have a deleterious effect on fitness, generally one should choose `mutprob` to ensure that mutations occur only rarely (typically, the expected number of mutations per solution will be less than one, so that part of each new generation receives no mutations at all).

- The values of `tourneysize` and `keepbest` can also be changed to modify the genetic selection pressure from one generation to the next. The effects of these parameters are similar to those of `popsize` and `mutprob`. However, ordinarily it should not be necessary to change them from their default values. The exception to this is when the objective function is costly to compute. In this case it might not be feasible to increase the population size, and one could instead modify the tournament size or the number of elites to influence overall run time.

A characteristic of GA searches is rapid initial convergence to the neighborhood of local optima. This characteristic is observed over a wide range of parameter settings, for a wide range of problems (which is one of the reasons for the popularity of GAs). After this initial rapid descent, however, further improvements can be slow. A significant fraction of a problem's computational budget can be spent waiting for favorable mutations to bring down the objective function value.

This latter portion of the GA search is difficult to influence through the initial settings of control parameters like `mutprob`, `tourneysize`, and `keepbest`. To eliminate late-stage stagnation of the search, adaptive or iterative approaches can be considered. For example, one could periodically alternate the mutation probability, tournament size, or elitism to ensure that the search iterates through "exploration phases" and "improvement phases." In such a scheme, `kofnGA` can be called repeatedly, passing the search results of one phase on to the next using the `initpop` argument. The examples in the **kofnGA** package include a simple demonstration of this approach.

For additional guidance on the topic of tuning GAs to specific problems, see Michalewicz and Fogel (2004, Chap. 10) or Bartz-Beielstein, Chiarandini, Paquete, and Preuss (2010, Part III).

3. Test problems

The proposed GA was tested on four realistic design problems and three artificial problems. Among the design problems, only the first is small enough that the true globally optimal solution is known; the size of the other three problems makes exhaustive search impractical. The three artificial problems are constructed in such a way that the problem has both a very large search space and a known best solution.

Except where noted in the text, all of these examples were run with the same GA parameters: population size 200, tournament size 50, elite group size 50, mutation probability 0.01. The duration of the search was chosen on a problem-specific basis, starting with 500 generations and increasing as necessary for the larger problems.

3.1. Environmental monitoring, small problems (solution known)

Chapter 14 in Le and Zidek (2006) contains an example related to ozone concentrations in the state of New York. Among other things, this example calculates a 100 × 100 hypercovariance matrix $\mathbf{\Lambda_0}$ that is subsequently used for design of a network extension. This matrix was used to test `kofnGA` for different problem sizes. The negative of the logarithm of the determinant was used to convert the problem into a minimization.

Considering only a subset of the 100 candidate points, and keeping k small, it is possible to find the true optimal solution by exhaustive search, e.g., using the `ldet.eval` function supplied with the R package **EnviroStat** (Le, Zidek, White, and Cubranic 2015) accompanying the aforementioned book. This was done for two cases: choosing $k = 4$ out of $n = 64$ sites (635,376 combinations), and choosing $k = 5$ out of $n = 49$ sites (1,906,884 combinations). The GA repeatedly found the optimal solution for both of these cases. The best solutions are shown on a map in Figure 1.

For the larger of these two problems, the exhaustive search took approximately 30 seconds. The GA took about 40 seconds to run 500 generations, but the optimal solution was found very early (after tens of generations, or about four seconds).

3.2. Environmental monitoring, larger problem (solution unknown)

The ozone example can also be used to examine the performance of `kofnGA` for larger problems, by using the full covariance matrix and increasing k. Unfortunately the true optimal solution is not available in this case, but repeated runs of the GA can be used to see whether different solutions result.

Setting $n = 100$ and $k = 15$ (about 2.5×10^{17} combinations), `kofnGA` returned the same solution in each of 10 runs. This solution is shown in Figure 2. The right half of the figure

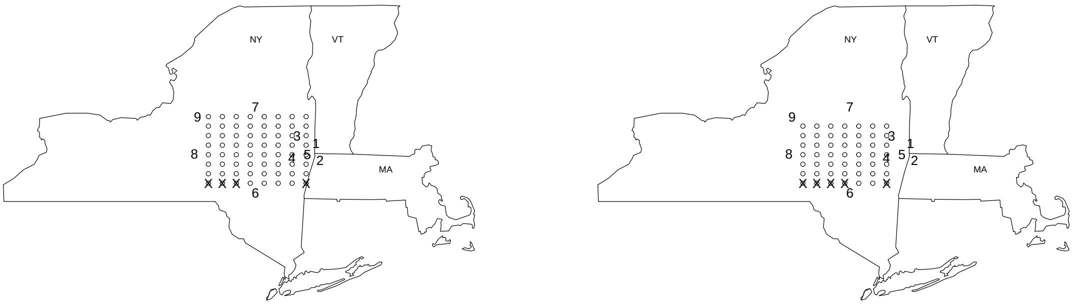

Figure 1: Results for small problems using the ozone example. On each map, numbers indicate the pre-existing monitoring sites, circles indicate candidate extension sites, and X's denote the optimally selected sites. Left: $(n, k) = (64, 4)$. Right: $(n, k) = (49, 5)$.

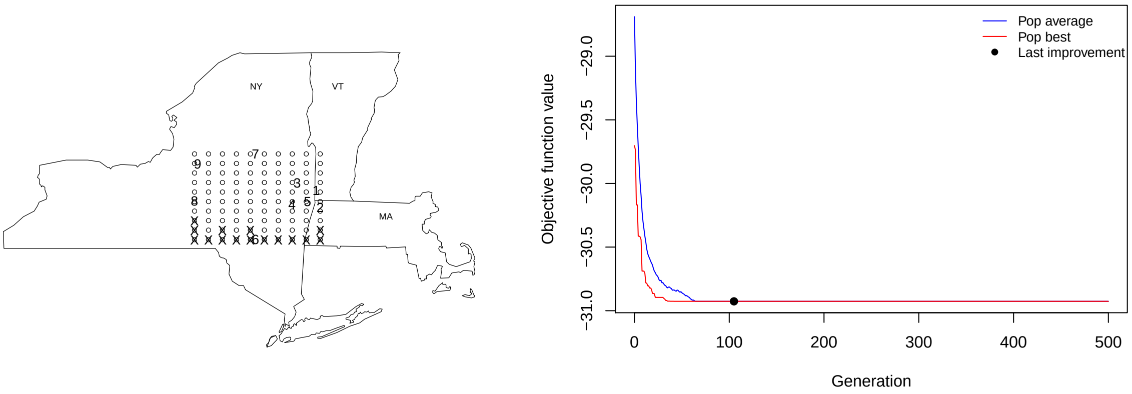

Figure 2: Results for a larger problem using the ozone example. Left: the best solution found in each of the 10 GA repetitions (compare with Figure 1). Right: population average objective value and best objective value versus generation. The dot on the plot shows the generation at which the last reduction of the objective function was made.

shows the progress of the GA solution over generations, for a single run. Note that the best solution very quickly decreases to be close to its final value. The algorithm was run with the same parameters as for the previous small examples, and again took approximately 40 seconds to complete 500 generations.

3.3. A constructed covariance matrix (solution unknown)

To get a hypothetical network design test problem of arbitrary size, it is desirable to be able to generate symmetric positive definite matrices at random. The generated matrix should not have any very small eigenvalues, otherwise numerical problems can result during the search. Ko *et al.* (1995) tackle this by generating diagonally dominant matrices; here, a method based on QR and eigenvalue decompositions is used.

An $n \times n$ covariance matrix $\mathbf{\Sigma}$ is generated as $\mathbf{\Sigma} = \mathbf{QLQ}^{\top}$, where $\mathbf{Q}$ is orthogonal and $\mathbf{L}$ is a diagonal matrix with diagonal elements equal to the desired eigenvalues of $\mathbf{\Sigma}$. The random orthogonal matrix $\mathbf{Q}$ is generated through QR decomposition of an $n \times n$ matrix populated with standard normal random variables.

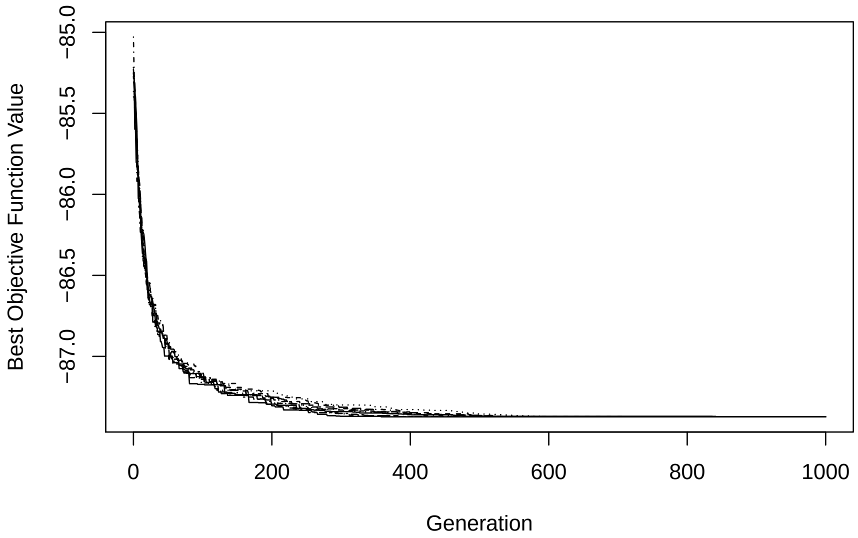

Figure 3: Progress of the best solution versus generation, for a constructed covariance matrix with $n = 500$ and $k = 50$. Each of the ten lines on the plot represents a replicate GA search from a different random starting point.

For the present test problem, values of $n = 500$ and $k = 50$ were chosen, and the eigenvalues were set to be evenly spaced numbers from 1 to 10. This problem is substantially larger than the ozone examples, with approximately 10^{69} subsets to choose from. A matrix was generated, and the GA search was repeated ten times with a randomly initialized population at each run. The number of generations was increased to 1000 for these runs, but other control parameters were unchanged compared to the ozone problems.

The progress of the best solution over generations is shown for all ten runs in Figure 3. The runs show an unexpected level of consistency, all decreasing to be near their final solution by about 400 generations. Nine of the ten runs returned the same solution, the best one found. The other solution differed from this best solution by only one element, and had the same objective function value to four significant digits. Each run took less than two minutes to complete.

3.4. A D-optimal design problem (solution unknown)

Goos and Jones (2011, Sec. 4.2) consider a D-optimal design approach to an experiment involving three quantitative factors (temperature, pressure, and speed) and one categorical factor (supplier). All four factors have three levels, leading to a total of 81 possible experimental runs. An experiment consisting of 24 runs is desired. Restricting attention to designs that do not include replicates, the objective is to choose one design from the $\binom{81}{24} \approx 10^{20}$ possibilities, such that the D-optimality criterion is maximized (henceforth we use the equivalent criterion of minimizing $-\log|\mathbf{X}^\top\mathbf{X}|$).

The analysis in the original source used the software product **JMP** (SAS Institute Inc. 2012b) to find candidate D-optimal designs using a coordinate exchange algorithm (SAS Institute Inc. 2012a). The optimal criterion value quoted in the text is −47.728 on the negative log scale, although the design matrix given in the text does not actually achieve this value. To better

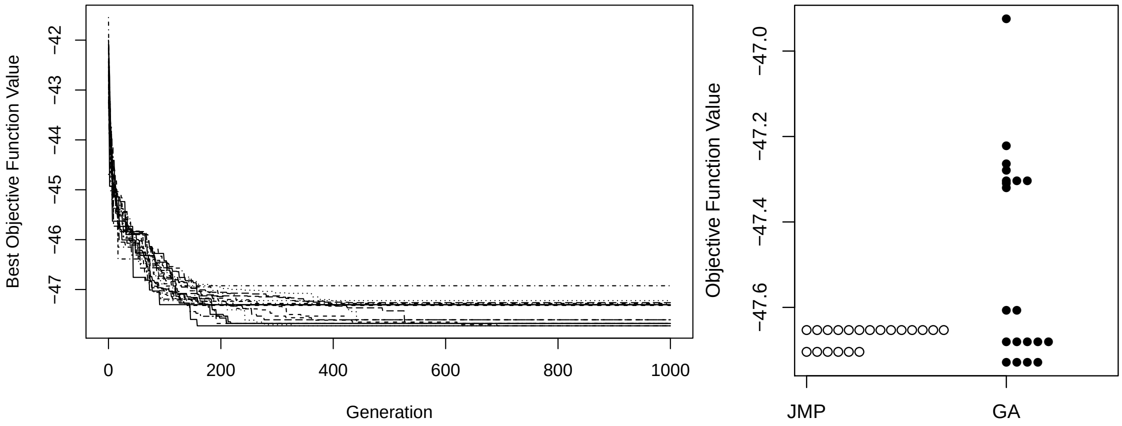

Figure 4: Results for the D-optimal design example. Left: solution progress for the 20 repeated GA runs. Right: dot plot showing the distribution of objective function values for solutions found by **JMP** (open circles) and by `kofnGA` (filled circles).

assess the typical results from the coordinate exchange algorithm, the design generation step was repeated 20 times using the **JMP** software, with default settings (which allow the search to re-start ten times from different origins). None of these repetitions reached the quoted objective value. Rather, six of the designs had objective value -47.704, while the remaining 14 designs had objective value -47.653.

Twenty runs of `kofnGA` were performed, at the default parameter settings, to compare with the **JMP** results. The progress of the best objective function value over 1000 generations, and the corresponding best objective function values found, are shown in Figure 4. As with the previous network design examples, the objective value of the best GA solution drops rapidly in the first few hundred generations, and subsequently levels off or decreases only slightly due to favorable mutations.

Of the twenty `kofnGA` runs, four were able to improve on the **JMP** results, achieving the quoted optimal value of -47.728, which we may surmise is the global minimum. Five others found solutions with objective values of -47.681, which is between the objective values of the two groups of **JMP** solutions. The remaining nine runs returned solutions of lower quality, having objective values from -46.93 to -47.61. That the GA results showed greater spread than the exchange algorithm results is perhaps not surprising, since the exchange algorithm had the benefit of multiple restarts. Nevertheless, all of the solutions found are practically useful, as every solution had a relative D-efficiency (a comparative measure of solution quality in the D-optimality sense; see Goos and Jones 2011, p. 86) greater than 95%.

3.5. Sparse subset problem, version 0 (solution known)

Whitley, Beveridge, Guerra-Salcedo, and Graves (1998) construct a subset selection problem they refer to as the “120-bit sparse subset problem.” They describe the problem as a search for the best subset of any size, but here we will aim at finding the best subset of size $k = 20$ from $n = 120$ alternatives. Potential solutions are represented as strings of 120 digits, containing 100 zeros and 20 ones. Bits with value 1 represent selected elements. From this binary representation, we may recover the corresponding index vector representation by reporting the locations of the nonzero bits.

Let a 120-bit string be $\mathbf{b} = [\mathbf{b}_1|\mathbf{b}_2|\cdots|\mathbf{b}_{20}]$, where $\mathbf{b}_i, i = 1, 2, \ldots, 20$ are six-bit substrings referred to as *blocks*. Define the sequence 000001 to be the *target configuration* for a block, and denote this configuration by $\mathbf{b}_t$. The objective function is defined such that the optimal solution is $\mathbf{b}^* = [\mathbf{b}_t|\mathbf{b}_t|\cdots|\mathbf{b}_t]$, that is, the string with all 20 blocks having the target configuration. This solution can be expressed as the index vector $\mathbf{v}^* = [6, 12, 18, \ldots, 120]$.

The objective function will be denoted by $\mathcal{SS}_0(\mathbf{b})$. It is described in detail here since the next two test problems are more general variants of it. The problem is described as a maximization, i.e., the sign will be changed when it is to be solved by `kofnGA`.

For ease of exposition, let bits taking value 1 be referred to as *1-bits*. For a given solution $\mathbf{b}$, let s_L be the number of 1-bits in the left half of the bit string; that is, in positions 1 through 60. Similarly take s_R to be the number of 1-bits in positions 61 through 120, the right half. Computation of the objective proceeds as follows:

1. Set $\mathcal{SS}_0 = 0$.

2. For each block $\mathbf{b}_i$,

   (a) Add the *block score*: set $\mathcal{SS}_0 = \mathcal{SS}_0 + \text{score}(\mathbf{b}_i)$, where $\text{score}(\mathbf{b}_i)$ is determined using:

number of 1-bits in $\mathbf{b}_i$:	0	1	2	3	4	5	6
$\text{score}(\mathbf{b}_i)$:	0	3	6	9	12	15	20

   (b) Add the *target bonus*: if $\mathbf{b}_i = \mathbf{b}_t$, set $\mathcal{SS}_0 = \mathcal{SS}_0 + 9$.

3. Subtract the *left penalty*: if $s_L > 13$, set $\mathcal{SS}_0 = \mathcal{SS}_0 - 2s_L$.

4. Subtract the *right penalty*: if $s_R > 13$, set $\mathcal{SS}_0 = \mathcal{SS}_0 - 2s_R$.

The value of $\mathcal{SS}_0(\mathbf{b})$ is determined by the cumulative block score, plus any target bonuses, minus the left and right penalties. To make this more concrete, consider the following candidate solution:

$$\begin{aligned}\mathbf{b} = \;& 000100\ 001010\ 000010\ 111111\ 000100\ 101000\ 000010\ 000000\ \underline{000001}\ 000000 \\ & 000010\ \underline{000001}\ 001000\ 000000\ 000000\ 010000\ 000000\ 000010\ 000000\ 000000\end{aligned}\ .$$

The left half of $\mathbf{b}$ is shown on the first line, and the right half is shown on the second line. White space has been inserted to make the six-bit blocks visually more distinct. Two of the blocks, $\mathbf{b}_9$ and $\mathbf{b}_{12}$, have the target configuration; these have been underlined. For this example, the total block score accrued by the 20 blocks is 62; the total target bonus is 18; and the left half of the string, having 15 1-bits, incurs a penalty of 30. Thus the final fitness value is $\mathcal{SS}_0(\mathbf{b}) = 50$. By comparison, the optimal solution, with 000001 in each block, has $\mathcal{SS}_0(\mathbf{b}^*) = 240$.

To see why $\mathbf{b}^*$ is optimal, first define a block with value 000000 to be *empty*, and a block with two or more 1-bits as a *surplus block*. Then note that, because there are a total of 20 1-bits and 20 blocks, there must be at least as many empty blocks as surplus blocks in any given solution. For any given solution, either of the following two operations will always improve the fitness value:

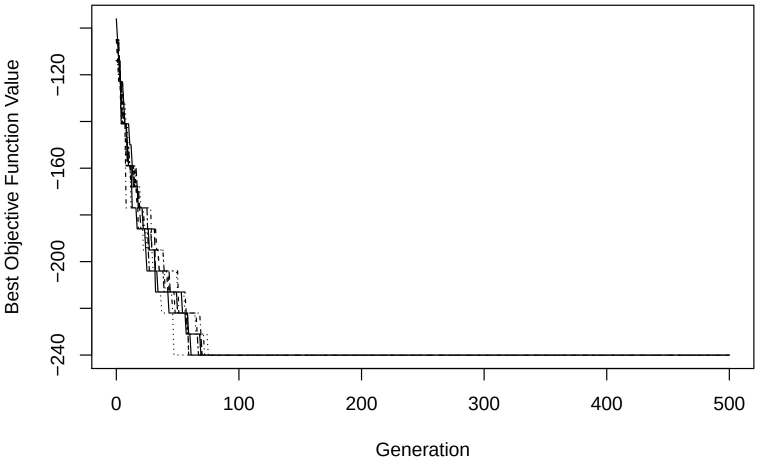

Figure 5: Progress of the best solution versus generation, for ten runs of the 120-bit sparse subset problem ($n = 120, k = 20$).

1. If there are any surplus blocks, move one of the 1-bits in that block to the rightmost position of an empty block, producing a target block.

2. If there are any blocks with a single 1-bit not in the rightmost position, replace this block with a target block.

Repeatedly performing these operations until no more moves are possible will produce the solution $\mathbf{b}^*$, with maximal fitness.

The `kofnGA` search was replicated ten times using $-\mathcal{SS}_0(\mathbf{b})$ as the objective function. Figure 5 illustrates the solution progress over 500 generations. All of the runs found the true optimum, and did so in less than 100 generations. Each run took about 53 seconds for the full 500 generations.

3.6. Sparse subset problem, version 1 (solution known)

A simplified version of the sparse subset problem can be used to devise problems of various sizes, where n is a multiple of k. As with the original version, the solution is viewed as a binary string for the purpose of function evaluation. This time, however, the string consists of k blocks, each of size r (so $n = rk$). As before, the target configuration $\mathbf{b}_t$ is a block with only a single 1-bit, in the rightmost position; the optimal solution $\mathbf{b}^*$ has all blocks in the target configuration.

In this version of the problem the left and right penalties are eliminated and the objective function depends only on a block score and a target bonus. To write the objective function in a general way, let s_i to be the number of 1-bits in block $\mathbf{b}_i$, and let N_t be the number of blocks matching the target configuration. Then the objective function is

$$\mathcal{SS}_1(\mathbf{b}) = \sum_{i=1}^{k} \varphi_1(s_i) + \frac{1}{k} N_t, \tag{1}$$

where $\varphi_1(s_i)$ is the block score for $\mathbf{b}_i$, computed as follows:

$$\varphi_1(s_i) = \begin{cases} 0 & \text{if } s_i = 0 \\ r - s_i & \text{if } s_i > 0 \end{cases} . \tag{2}$$

A block with no 1-bits in it ($s_i = 0$) contributes a score of zero, and any block with $s_i > 0$ contributes a score of $r - s_i$. This means that the maximum possible block score is $r - 1$, achieved by any block containing a single 1-bit. To create a unique solution, a bonus of $1/k$ is added for each block matching the target configuration. Thus the optimal solution, consisting of k target blocks, receives a fitness of $\mathcal{SS}_1(\mathbf{b}^*) = k(r-1)+1$, while any solution with exactly one 1-bit per block receives a fitness of at least $k(r-1)$. As an index vector, the best solution is $\mathbf{v}^* = [r, 2r, 3r, \ldots, kr]$. Once again, the negative of this objective function is used in the actual optimization.

To illustrate this construction, consider a problem with five three-bit blocks ($k = 5$ and $r = 3$). In this case we are searching for best sets of size 5 taken from 15 options. The target configuration is 001. A solution can be encoded as a five-element vector or as a 15-bit string. Consider, for example the solution $\mathbf{v}^\dagger = [2, 4, 5, 8, 12]$ and the optimal solution $\mathbf{v}^* = [3, 6, 9, 12, 15]$, with corresponding binary representations

$$\begin{aligned} \mathbf{b}^\dagger &= \ 010\ 110\ 010\ \underline{001}\ 000, \\ \mathbf{b}^* &= \ \underline{001}\ \underline{001}\ \underline{001}\ \underline{001}\ \underline{001}, \end{aligned}$$

where blocks matching the target configuration are underlined. The fitness values of these two solutions are $\mathcal{SS}_1(\mathbf{b}^\dagger) = 7.2$ and $\mathcal{SS}_1(\mathbf{b}^*) = 11$. Equations 1 and 2 imply that non-target blocks containing (0, 1, 2, 3) 1-bits contribute scores of (0, 2, 1, 0) to the objective, while the specific block 001 contributes a score of 2.2. The optimal solution's function value is 11, and any other solution with a single 1-bit per block achieves a value of at least ten.

The `kofnGA` function was tested on a much larger instance of this problem, with $k = 100$ and $r = 10$, that is, taking subsets of 100 from 1000. The best solution has (negative) objective function value -901, and any other solution with a single 1-bit in each block has a function value below -900. Incidentally, note that the GA bases its selection on fitness ranks, not on actual fitness scores, so the magnitude of differences between solutions does not matter.

The number of generations was expanded to 2500 for this trial, since the solution space is now extremely large (approximately 10^{140} combinations). This caused each run's execution time to increase to about 11 minutes. Figure 6 shows the results. The left plot in the figure shows the best solutions for ten runs over all generations, and the right plot focuses in on generations 1000 to 2500. By 1000 generations, all of the runs had settled on the solutions having only a single 1-bit per block. For the subsequent generations, improvements were still being made, with small drops of magnitude $1/k$ whenever the number of optimally-configured blocks was increased.

By the end of the 2500th generation, none of the runs had found the true optimum. Note that once the population has settled down to the set of best sub-optimal solutions, the GA is performing much like a random exchange algorithm. The solutions can be expected to improve with longer runs until eventually the true best solution is found. This behavior was observed in other trials for somewhat smaller versions of this problem, say with $k = 50$ and $r = 20$.

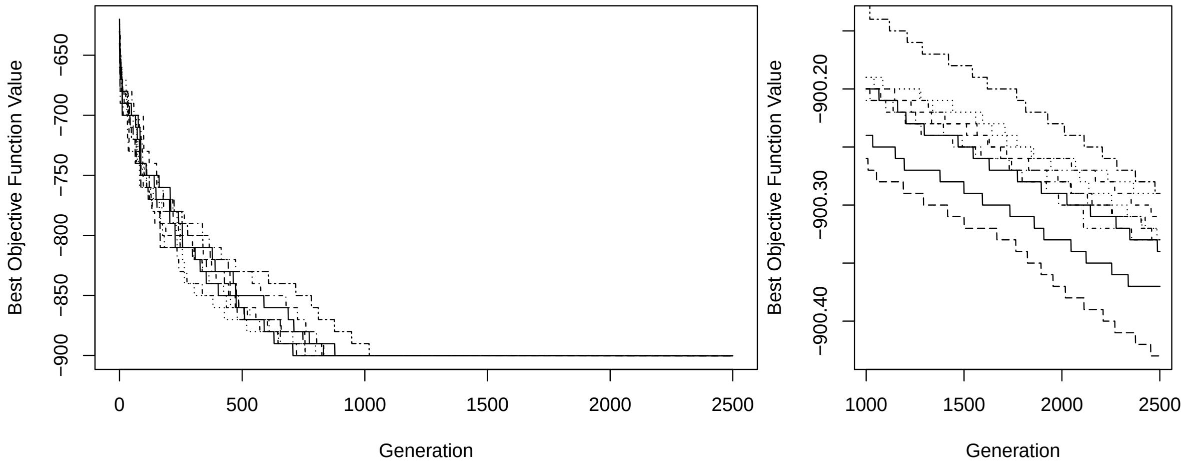

Figure 6: Progress of the best solution versus generation, for ten runs of the modified sparse subset problem (version 1, with $n = 1000, k = 100$). Left: all 2500 generations. Right: focus on the last 1500 generations.

3.7. Sparse subset problem, version 2 (solution known)

A more difficult variant of the previous problem can be obtained by modifying the block score formula and the target bonus. In this version of the problem solutions still consist of k blocks of size r, and the target block and optimal solution are unchanged. However, the objective function is

$$\mathcal{SS}_2(\mathbf{b}) = \sum_{i=1}^{k} \varphi_2(s_i) + \left(\frac{\mathcal{M}+1}{k} - 1\right) N_t, \tag{3}$$

where

$$\varphi_2(s_i) = \begin{cases} 0 & \text{if } s_i = 0 \\ 2s_i - 1 & \text{if } s_i > 0 \end{cases} \tag{4}$$

and

$$\mathcal{M} = (2r-1)\left\lfloor \frac{k}{r} \right\rfloor + \varphi_2\left(\mathrm{mod}(k, r)\right), \tag{5}$$

with $\lfloor\cdot\rfloor$ and $\mathrm{mod}(\cdot,\cdot)$ denoting the floor and modulo functions, respectively.

Consider the block score $\varphi_2(\cdot)$ first. While $\mathcal{SS}_1$ gave the largest block score to blocks having a single 1-bit, $\mathcal{SS}_2$ assigns greater scores to blocks having more ones. If the block size is $r = 5$, for example, blocks with (0, 1, 2, 3, 4, 5) 1-bits get scores of (0, 1, 3, 5, 7, 9), respectively. Furthermore, we see that for any p and q nonzero such that $p + q \leq r$, the inequality $\varphi_2(p) + \varphi_2(q) < \varphi_2(p + q)$ holds. This implies that if a solution includes two non-empty blocks $\mathbf{b}_i$ and $\mathbf{b}_j$ with r or fewer 1-bits between them, it is always advantageous to move all of the 1-bits into a single block and leave the other block empty.

The consequence of this is that the largest possible value for the total block score is achieved whenever the maximum possible number of blocks are full of 1-bits, with the remaining 1-bits also all together in the same block. Without loss of generality we may think of this as the situation where the first k bits in the string take value 1, with the rest zeros. This will result in the first $\lfloor k/r \rfloor$ blocks being completely full, and the next block containing $\mathrm{mod}(k, r)$ 1-bits. The combined block score for this configuration is $\mathcal{M}$, given in Equation 5.

The target bonus $(\mathcal{M}+1)/k-1$ was chosen to force the same sparse solution $\mathbf{b}^*$ to be optimal. Note that when every block has only a single 1-bit in its last position, $\sum_{i=1}^{k}\varphi_2(s_i) = \sum_{i=1}^{k}\varphi_2(1) = k$ and $N_t = k$. Then, referring to Equation 3,

$$\mathcal{SS}_2(\mathbf{b}^*) = k + \left(\frac{\mathcal{M}+1}{k} - 1\right) k = \mathcal{M} + 1.$$

So by construction the fitness of solution $\mathbf{b}^*$ exceeds $\mathcal{M}$ by one, where $\mathcal{M}$ is the largest objective value achievable without any target bonuses.

This version of the sparse subset problem is harder than either of the other two because the optimal solution is sparse, with one 1-bit per block, but most of the near-optimal solutions are dense, with many full blocks and many empty blocks. This makes it difficult for local modification operators like the GA crossover and mutation operators to produce a jump from a near-optimal solution to the neighborhood of the true optimum (the concept of a neighborhood has not been formalized here but one can, for example, use the Hamming distance to measure similarity of solutions).

The `kofnGA` function was run on this version of the sparse subset problem with the same problem size as the previous section, $k = 100$ and $r = 10$. Again, the number of generations was set to 2500, but this time the population size was increased from 200 to 300. This change further increased the run time (to about 17 minutes per trial). Ten repetitions were run as in the previous examples.

Figure 7 shows the results. As with previous examples, the best solutions dropped quickly to be close to the optimal value. In this case, however, the algorithm was not able to reach the sparse solutions. Rather, the best solutions had 1-bits predominantly in clumps, just as expected from consideration of the objective function. The lower plot in Figure 7 shows all of the best solutions found, using a matrix of dark and white pixels to represent the 1000-element bit strings. Each row of this image represents one run's best solution, and the selected locations are plotted as dark pixels. The grouping of the selected locations is evident. The optimal objective function value for this case is -191. Any solution having all its ones in blocks of ten has an objective value of $\mathcal{M} = -190$. Of the ten solutions returned, three had objective values of -189, six had -188, and one had -187.

4. Final remarks

The `kofnGA` function performed well in the test problems. For the environmental network design problems and the original sparse subset problem, either the optimal solution was found, or the GA detected the same solution repeatedly when started from different initial conditions.

For both the D-optimal design problem and version 1 of the sparse subset problem, near-optimal solutions were found easily, and it appears that the true optimum could be found given sufficient computing time. That said, the algorithm becomes less efficient once it gets near the optimum. At that point the population tends to be quite homogeneous, and improved solutions are found primarily through mutation. This suggests it would be advantageous to switch to using other local search methods (like a greedy exchange algorithm) once the rate of progress of the GA becomes slow.

The final problem (version 2 of the sparse subset problem) is very challenging, and we may conjecture that other combinatorial optimizers would also have difficulty with this problem.

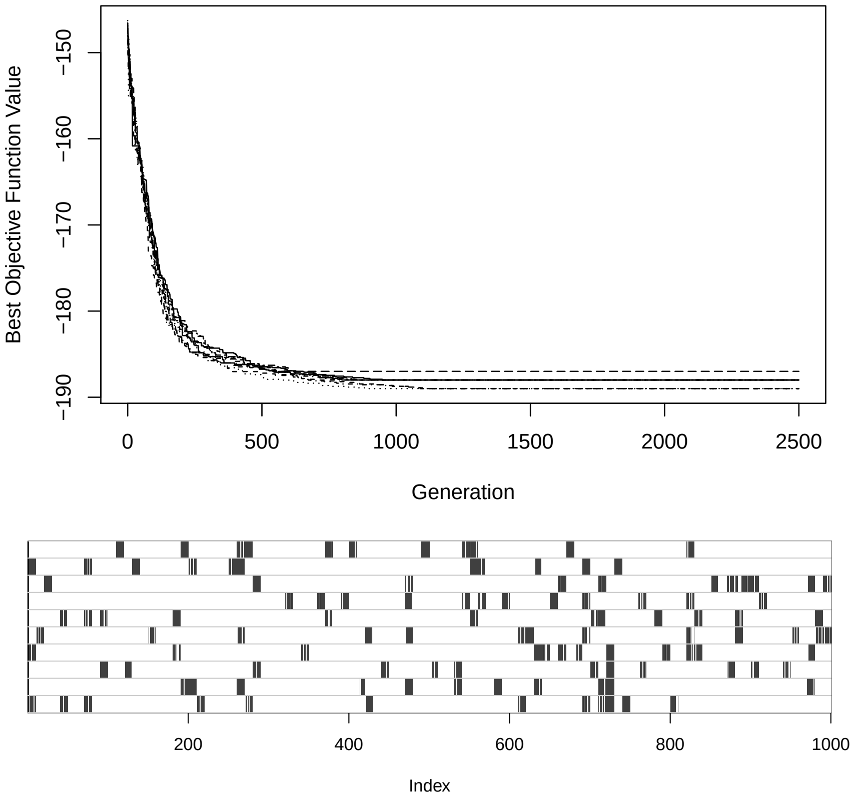

Figure 7: Results of ten runs of the modified sparse subset problem (version 2, with $n = 1000, k = 100$). Top: best solution versus generation. Bottom: binary string representation of the ten best solutions found. Each row of the bottom figure has a dark pixel in the location of a selected index.

Furthermore, this problem might not be representative for the maximum-determinant search application. This is because covariance matrices tend to have some degree of "smoothness," in the sense that subsets that are nearly the same will not differ by too much in their determinant. Solutions near the global optimum should also have near-optimal objective function values.

This statement about the smoothness of the problem is actually a conjecture at this point, but it might be possible to prove (or find), for example, some bounds on the maximal difference in determinant between principal submatrices differing by only d columns. The tools used by Ko *et al.* (1995), i.e., eigenvalue interlacing property, submodularity of the objective function, and so on, should be applicable here. Establishing such results could provide some theoretical basis for the use of the GA or other algorithms involving local moves between nearby solutions.

The run time for executing `kofnGA` was not prohibitive for any of the example problems. Nevertheless, more rapid execution times are always desirable to expand the user-friendliness of the function and the range of problems it can handle. While at present the function is written wholly in R, in future releases the use of compiled code will be explored as a means to speed up the slowest parts of the algorithm. For all but the simplest cases, however, the objective function itself is by far the dominant factor in determining execution time. Therefore users in search of speed improvements are encouraged to optimize their objective functions as much as possible.

Acknowledgments

The author thanks Jim Zidek for stimulating this work through discussion of the environmental monitoring network problem, and W. John Braun for suggesting the matrix decomposition approach to constructing random covariance matrices.

References

Bartz-Beielstein T, Chiarandini M, Paquete L, Preuss M (eds.) (2010). *Experimental Methods for the Analysis of Optimization Algorithms*. Springer-Verlag, Berlin. doi:10.1007/978-3-642-02538-9.

Broudiscou A, Leardi R, Phan-Tan-Luu R (1996). "Genetic Algorithm as a Tool for Selection of D-Optimal Design." *Chemometrics and Intelligent Laboratory Systems*, **35**(1), 105–116. doi:10.1016/s0169-7439(96)00028-7.

Goos P, Jones B (2011). *Optimal Design of Experiments: A Case Study Approach*. John Wiley & Sons, West Sussex. doi:10.1002/9781119974017.

Hamada M, Martz H, Reese C, Wilson A (2001). "Finding Near-Optimal Bayesian Experimental Designs via Genetic Algorithms." *The American Statistician*, **55**(3), 175–181. doi:10.1198/000313001317098121.

Hereida-Langner A, Montgomery D, Carlyle W, Borror C (2004). "Model-Robust Optimal Designs: A Genetic Algorithm Approach." *Journal of Quality Technology*, **36**(3), 263–279.

Holland J (1992). *Adaptation in Natural and Artificial Systems*. MIT Press, Cambridge.

Ko CW, Lee J, Queyranne M (1995). "An Exact Algorithm for Maximum Entropy Sampling." *Operations Research*, **43**(4), 684–691. doi:10.1287/opre.43.4.684.

Le N, Zidek J (2006). *Statistical Analysis of Environmental Space-Time Processes*. Springer-Verlag, New York.

Le N, Zidek J, White R, Cubranic D (2015). ***EnviroStat**: Statistical Analysis of Environmental Space-Time Processes*. R package version 0.4-2; Fortran code for Sampson-Guttorp estimation by PD Sampson, P Guttorp, W Meiring and C Hurley and Runge-Kutta-Fehlberg method implemented by HA Watts and LF Shampine, URL http://CRAN.R-project.org/package=EnviroStat.

Mandal A, Wu C, Johnson K (2006). "SELC: Sequential Elimination of Level Combinations by Means of Modified Genetic Algorithms." *Technometrics*, **48**(2), 273–283. doi:10.1198/004017005000000526.

Michalewicz Z, Fogel D (2004). *How to Solve It: Modern Heuristics*. 2nd edition. Springer-Verlag, Berlin. doi:10.1007/978-3-662-07807-5.

Mitchell T (1974a). "An Algorithm for the Construction of "D-Optimal" Experimental Designs." *Technometrics*, **16**(2), 203–210. doi:10.1080/00401706.1974.10489175.

Mitchell T (1974b). "Computer Construction of "D-Optimal" First-Order Designs." *Technometrics*, **16**(2), 211–220. doi:10.1080/00401706.1974.10489176.

Orestes Cerdeira J, Duarte Silva P, Cadima J, Minhoto M (2015). ***subselect**: Selecting Variable Subsets*. R package version 0.12-5, URL http://CRAN.R-project.org/package=subselect.

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

Ruiz-Cárdenas R, Ferreira M, Schmidt A (2010). "Stochastic Search Algorithms for Optimal Design of Monitoring Networks." *Environmetrics*, **21**(1), 102–112.

SAS Institute Inc (2012a). ***JMP** 10 Design of Experiments Guide*. SAS Institute Inc., Cary, NC.

SAS Institute Inc (2012b). ***JMP**, Version 10*. SAS Institute Inc., Cary, NC.

Talbi EG (2009). *Metaheuristics: From Design to Implementation*. John Wiley & Sons, Hoboken.

Whitley D, Beveridge J, Guerra-Salcedo C, Graves C (1998). "Messy Genetic Algorithms for Subset Feature Selection." In *Proceedings of the International Conference on Genetic Algorithms*, pp. 568–575.

Willighagen E (2015). ***genalg**: R Based Genetic Algorithm*. R package version 0.2.0, URL http://CRAN.R-project.org/package=genalg.

Wolters M (2015). ***kofnGA**: A Genetic Algorithm for Fixed-Size Subset Selection*. R package version 1.2, URL http://CRAN.R-project.org/package=kofnGA.

Affiliation:

Mark A. Wolters
Shanghai Center for Mathematical Sciences
22nd Floor, East Guanghua Tower
Fudan Univeristy
Shanghai, China, 200433
E-mail: mwolters@fudan.edu.cn

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/

November 2015, Volume 68, Code Snippet 1 *Submitted:* 2013-03-22
doi:10.18637/jss.v068.c01 *Accepted:* 2014-11-25