A C++ Implementation of the Bayesian Optimization Algorithm (BOA) with Decision Graphs

Martin Pelikan

IlliGAL Report No. 2000025
May 2000

To be used with the source code
of the *Bayesian optimization algorithm with decision graphs*, available at
`ftp://ftp-illigal.ge.uiuc.edu`

Illinois Genetic Algorithms Laboratory
University of Illinois at Urbana-Champaign
117 Transportation Building
104 S. Mathews Avenue Urbana, IL 61801
Office: (217) 333-0897
Fax: (217) 244-5705

A C++ Implementation of the Bayesian Optimization Algorithm (BOA) with Decision Graphs

Martin Pelikan
Illinois Genetic Algorithms Laboratory
104 S. Mathews Avenue, Urbana, IL 61801
University of Illinois at Urbana-Champaign
Phone/FAX: (217) 333-2346, (217) 244-5705
`pelikan@illigal.ge.uiuc.edu`

Abstract

The paper explains how to download, compile, and use the C++ implementation of the Bayesian optimization algorithm (BOA) with decision graphs (Pelikan, Goldberg, & Sastry, 2000). It provides the instructions for creating input files for the BOA to solve various problems with various parameter settings and for adding new test functions into the existing code. Outputs of an example experiment are discussed.

1 Introduction

The purpose of this paper is to give basic instructions for downloading, compiling, and using the implementation of the Bayesian optimization algorithm with decision graphs in C++ that is publicly available at the IlliGAL anonymous ftp-site. The installation instructions are designed for UNIX operating systems. However, we suppose no major modifications are necessary and the source codes should be compiled under most operating systems with a number of different compilers without major problems.

We have tried to keep the implementation as simple as possible and yet sufficiently powerful to demonstrate the basic principle of the algorithm and either to reproduce the results of recent experiments or to produce new ones. Therefore, the code does not include many features discussed in the papers discussing the algorithm as the incorporation of prior information or other than a simple greedy search for constructing the network modeling the data. However, the code contains all necessary features required to replicate our experiments in Pelikan et al. (2000). Although some low-level constructs are written in object-oriented C++, on a higher level we have avoided the use of object-oriented features so that the code is tractable even for the users used to structural or functional languages.

The paper starts by providing the instructions for downloading and extracting the package including the source code and a few example input files. Section 3 explains how to compile the extracted source code. Section 5 discusses the features of the implementation or what has actually been implemented. In section 6, the format of input files and the description of parameters that it can specify are presented. Section 7 discusses the format of output files for a simple example experiment. A short description of the source files, the list of implemented test functions, and the instructions for plugging in new test functions can be found in Section 8.

2 How to Download and Extract the Source Code

The package with the source code and few example input files is available at the IlliGAL anonymous ftp site in `ftp://ftp-illigal.ge.uiuc.edu/pub/src/decisionGraphBOA/C++/dBOA.tar.Z`.

After downloading the package (`dBOA.tar.Z`), the files can be extracted by typing the following:

```
uncompress dBOA.tar.Z
tar xvf dBOA.tar
```

A sub-directory `dBOA` will be created automatically as it is included in the file structure of the compressed files. After extracting the files correctly, the created sub-directory `dBOA` should contain the following files containing the source code and a directory with the example input files with their outputs:

```
COPYRIGHT    boa.h                      graph.h              operator.cc     select.h
Makefile     decisionGraph.cc           header.cc            operator.h      stack.cc
README       decisionGraph.h            header.h             population.cc   stack.h
args.cc      fitness.cc                 help.cc              population.h    startUp.cc
args.h       fitness.h                  help.h               profile         startUp.h
bayesian.cc  frequencyDecisionGraph.cc  labeledTreeNode.cc   random.cc       statistics.cc
bayesian.h   frequencyDecisionGraph.h   labeledTreeNode.h    random.h        statistics.h
binary.h     getFileArgs.cc             main.cc              replace.cc      utils.cc
boa          getFileArgs.h              memalloc.h           replace.h       utils.h
boa.aix      gmon.out                   mymath.cc            sBOA.tar
boa.cc       graph.cc                   mymath.h             select.cc
```

In the sub-directory `examples`, the files starting with `input` are example input files. The names of the example input files say what fitness function these files are intended to optimize and what the problem size is. Parameters that can be specified in the input files as well as the format of output files are explained in the remainder of this paper.

3 How to Compile the Source Code

The compilation is very simple. In a file `Makefile` that can be found in the directory with the uncompressed source code, the following two changes should be performed:

Line 36
In the statement `CC = gcc`, the `gcc` should be changed to the name of a preferred C++ compiler on your machine (if it is not `gcc`). With an SGI `CC` compiler, for instance, the line should be changed to `CC = CC`.

Line 42
In the statement `OPTIMIZE = -ffloat-store -O3`, the required optimization level should be set. For no code optimization, use only `OPTIMIZE = `. We have used a switch `-ffloat-store` to make sure that the floating point operations are the same in both the optimized as well as not optimized version. All modules are compiled at once since some compilers (as SGI `CC`) use

inter-modular optimization that does not allow them to compile each source file separately and link them together afterwards. Feel free to modify the makefile to make it fit your own needs.

After making the above modifications to the `Makefile`, you can compile the simple BOA by typing the following line:

```
make all
```

We have tested the code for various operating systems and compilers. On some compilers the results of the optimized code slightly differ from the results of the unoptimized one. However, the differences are insignificant. The platforms and compilers we have tried are listed in the following table:

Operating System	**Compiler**
IBM AIX 4.1	xlC 3.1
IBM AIX 4.2	gcc 2.7.2
Linux 2.2.13	egcs-2.91.66
Linux 2.2.9	pgcc-2.91.60
Sun OS 5.7	gcc 2.95.1
ULTRIX 4.4	gcc 2.7.2

After correctly compiling the source code there should be an executable file `boa` in your directory.

4 Command Line Parameters

Without any command line parameters, the `boa` runs with all parameters set to their default values. The `boa` can be called with either of the following parameters (for examples see Table 1):

`<filename>`
Runs the BOA with input parameters specified in the file <filename>.

`-h`
Prints the description of command line parameters.

`-paramDescription`
Prints the description, the type, and the default value of all input file parameters.

Table 1: Examples on command line parameters

Command line	**Description**
`boa`	runs the boa with all input parameters set to their default values
`boa myInputFile`	runs the boa with input parameters specified in the file `myInputFile`
`boa -paramDescription`	prints the description of all input file parameters
`boa -h`	prints the help on the command line parameters

5 What Has Been Actually Implemented?

The following list briefly summarizes what can be found the implementation:

Representation of Solutions
Solutions are represented by binary strings of fixed length.

Test Functions
Few decomposable test functions with and without overlapped building blocks have been implemented. The user can add his own test functions easily. Each test fitness function can (but need not) include the initialization and done methods. This feature can be useful for more sophisticated functions that need to read input parameters or allocate some memory before their first evaluation and perform certain actions in order to clean the used memory or the like after their last evaluation in the run).

Problem Size
The problem size can be set by the user. Very big problem sizes are allowed (up to 32767). The bigger the problem, the bigger the population, and the longer the run takes.

Population Size
The population size can be specified by the user. Very big populations can be used (up to $2,147,483,647$). However, the bigger the population size, the slower it takes to process one generation and the more memory the algorithm uses, and therefore very big populations are not very useful. We have used populations up to $50,000$.

Selection method
Tournament selection with replacement was implemented. To select each new individual a set of s individuals is first picked and the best solution out of the picked solutions is inserted into the set of selected solutions. The user can control the selection pressure by choosing the size s of the tournaments (by default set to 4).

Replacement method
Replacement of the worst has been implemented. With replacement of the worst, the worst solutions in the original population are replaced by offspring. The user can specify the number of offspring (in percent of population). When all original solutions are to be replaced, the size of the offspring is set to 100 (i.e., 100%).

Scoring Metric
The Bayesian Dirichlet metric for Bayesian networks with decision graphs with a pressure to simple models as described in Pelikan, Goldberg, and Sastry (2000) has been implemented. For the sake of keeping the implementation simple, no prior information in form of the prior network or the set of high-quality solutions can incorporated into the metric in our implementation.

Prior Information
No prior information but the maximal number of incoming edges into each node can be used. This number corresponds to the maximal order of interactions to be considered in the distribution estimate. However, since the metric biases model construction to simpler models, a quite high value (e.g., 20) can be used for all problems.

Network Construction Method

A greedy algorithm which updates the decision graphs for each variable by splitting and merging the nodes has been implemented as described in Pelikan, Goldberg, and Sastry (2000).

Output Statistics

There are few different outputs from the algorithm. The evolution of the best, average, and worst fitness values, the best solution in a current generation, the bias of the population, and the model used to generate offspring during the run can be extracted. All outputs can be related to the current number of generation or a number of fitness evaluations performed so far.

Termination Criteria

The run can be terminated after a maximal number of generations, maximal number of fitness evaluations, or a maximal proportion of optima in a population are reached. The run can also be stopped when the population has almost converged and the bits on all positions are almost homogeneous or when the optimum has been found. Any of the criteria can be ignored and any combination of various criteria can be used. If a termination criterion that uses a proposition that decides whether the solutions is optimal or not, if this proposition is not defined (when the algorithm does not know what is the optimum and what is not), the criterion is ignored.

6 Input files

Input files can contain the statements of the following form:

```
<identifier> = <value>
```

where `<identifier>` is an identifier of a particular parameter and `<value>` is its new value. Empty lines and extra spaces that are not within the identifier or its value are ignored. The order of statements in the input file is not important. Each parameter can be defined at most once. If the value of a parameter is not specified in the input file, its default value is substituted. In the case of multiple definition of any parameter, the program ends up with an error message informing what parameter was multiply defined. If the identifier does not exist, the program ends up with an error message. The interpreter of input file is case sensitive.

The following list describes the values of parameters that can be specified in the input file, their types, and their default values. You can get a similar list by running `boa -paramDescription`.

`populationSize`

Description:	The size of a population.
Type:	`long`
Default:	`1200`

`offspringPercentage`

Description:	The number of offspring to generate (in percent of population).
Type:	`float`
Default:	`50`

`fitnessFunction`

Description:	The number of a fitness function to use. (See Section 8.2 for the list of test functions included in the implementation.
Type:	`int`
Default:	`2` (deceptive of order 3 without overlapping)

`problemSize`

Description:	The size of a problem (string length).
Type:	`int`
Default:	`30`

`tournamentSize`

Description:	The size of tournaments in tournament selection (¿=2). The higher the size of tournaments, the higher the selection pressure. This number roughly corresponds to the expected number of the best solution after selection.
Type:	`int`
Default:	`4`

`maxNumberOfGenerations`

Description:	Maximal number of generations to perform before terminating the run. -1 stands for unlimited.
Type:	`long`
Default:	`200`

`maxFitnessCalls`

Description:	Maximal number of fitness calls before terminating the run. -1 stands for unlimited.
Type:	`long`
Default:	`-1` (unlimited)

`epsilon`

Description:	A threshold for univariate frequencies for terminating the algorithm due to the so-called bit-convergence. If frequencies of all bits are closer than epsilon to either 0 or 1, the run is terminated. -1 stands for ignoring this criterion.
Type:	`float`
Default:	`0.01`

`stopWhenFoundOptimum`

Description:	Terminate the run when the optimum has been found (if the proposition identifying the optimum for the optimized function is defined)? A non-zero value stands for "Yes", zero stands for "No".
Type:	`char`
Default:	`0` (no)

`maxOptimal`

Description:	Terminate the run when the proportion of optimal solutions (in percent of a population) has reached this value. -1 stands for ignoring this criterion.
Type:	`float`
Default:	`-1` (ignore)

`maxIncoming`

Description:	Maximal number of incoming edges into any of the nodes in the considered networks. Corresponds to the maximal order of interactions that can be covered by the used class of models (it is equal to the order of interactions that can be covered minus 1). Since the pressure to simple models is considered in the used metric, this parameter can be set to a high value that ensures proper convergence. Default value should work fine for even very complex problems.
Type:	`int`
Default:	`20` (interactions of 21. order)

`allowMerge`

Description:	Allow the merge operator in the graph construction? A non-zero value stands for "Yes", zero stands for "No". Using merge operator doesn't usually help much, but we still allow this possibility. By default, this operator is disabled.
Type:	`char`
Default:	`0` (no)

`pause`

Description:	Wait for Enter key after each generation? A non-zero value stands for "Yes", zero stands for "No".
Type:	`char`
Default:	`0` (no)

`outputFile`

Description:	The base of output file names (will add the extensions to each output file name according to the type of the file).
Type:	`char*`
Default:	`NULL` (no output file)

`guidanceThreshold`

Description:	A threshold for a population bias displayed each generation. As soon as the frequency of a bit gets closer than this parameter to either 0 or 1, the bit is said to be biased to the corresponding value.
Type:	`float`
Default:	`0.3`

`randSeed`

Description:	A random seed.
Type:	`long`
Default:	`time` (current time)

An example of input file is presented in Figure 6. With this input file the deceptive function of order 3 of the size (number of bits) 30 will be optimized with the population size of 1200. Truncation selection that selects the best half of the solutions will be used. Each generation, the half of the original population is replaced by the offspring. The networks are to have two incoming edges into each node at maximum. The run will be terminated after either 300 generations are performed or the frequencies of all bits are closer than 0.01 to either 0 or 1. Outputs will be written to output files with the base `output.3deceptive.30` and additional extensions corresponding to their type. Random seed is set to 123. Other parameters are set to their default values. The presented input file is included in the package along with the output files.

```
populationSize  = 1000
problemSize     = 30
fitnessFunction = 2

offspringPercentage = 50

tournamentSize = 4
allowMerge = 0

maxNumberOfGenerations = 40
epsilon                = 0.01

outputFile = output.3deceptive.30

randSeed = 123
```

Figure 1: Example input file (included in the package as `input.3deceptive.30`).

7 Outputs

In this section, we will briefly describe the outputs of the `boa` for the input file `input.3deceptive.30` shown in Figure 6 included in the package. We divide the section into two parts. The first one discusses the outputs to the standard output (which is mostly the screen). The second one discusses the outputs that can be found in produced output files (if any).

7.1 What Can You See on the Screen?

At first, the header with the name of the program, the author, the date of its release, and the name of input file is printed on the screen. It is followed by the list of all parameters and their values (not only those specified in the input file). For each parameter, its description, identifier, type, and the current value is displayed. Since this initial output described above is very simple and easy to understand, we do not present an example here.

After printing the information about the product and the parameters, the information about the generation number, the number of fitness evaluations performed so far, the best, average, and the worst fitness values in a current population, the proportion of optima in a current population (if the proposition checking for optima is defined for the used test function), the population bias (the guidance vector), and the best solutions in the current population is displayed. This information is printed out each generation. An example of the output information written each generation is shown in Figure 7.1. The example was produced with the `boa` with input parameters specified in the input file `input.3deceptive.30`, included in the package. It shows, that the number of current generation is 8, 5000 fitness evaluations were performed so far from the beginning of the run, the best fitness in the current population is 10, the average fitness in the population is 9.5265, and the worst fitness in the population is 8.3. There are 0.3% of global optima in the population. The bits on positions $1-6$, $9-12$, $16-18$, 28, and 30 are biased to 1 (the frequencies of 1 on these positions is closer than the parameter `guidanceThreshold` to 1). All other positions are unbiased (the "." is displayed on unbiased positions). The best solution in a population has 1's on all positions.

```
Generation                    : 8
Fitness evaluations           : 5000
Fitness (max/avg/min)         : (10.000000 9.526500 8.300000)
Percentage of optima in pop.  : 0.30
Population bias               : 111111..1111...111.........1.1
Best solution in the pop.     : 111111111111111111111111111111
```

Figure 2: Example information produced by the `boa` each generation to standard output.

After the run is terminated, the similar information is produced for the last generation. In addition to this, the information on the reasons for terminating the run is provided (in case of meeting multiple criteria for terminating the run, the first identified reason is displayed). Again, when the proposition for checking the optima is not available, the items that use this are excluded. An example of the final information closing the run is shown in Figure 7.1. The example was produced with the `boa` with input parameters specified in the input file `input.3deceptive.30`, included in the package.

```
FINAL STATISTICS
Termination reason            : Bit convergence (with threshold epsilon)
Generations performed         : 15
Fitness evaluations           : 8500
Fitness (max/avg/min)         : (10.000000 10.000000 10.000000)
Percentage of optima in pop.  : 100.00
Population bias               : 111111111111111111111111111111
Best solution in the pop.     : 111111111111111111111111111111
```

Figure 3: Example information produced by the `boa` at the end of the run.

After performing the whole run, a string "The End." should be printed out, as the last line sent to the standard output.

7.2 What Can You Find in the Output Files?

If you specify the parameter `outputFile` in the input file, the `boa` produces three output files, each starting with the base given by the `outputFile` parameter with an extension depending on the file type.

The following list displays the file names and the short description of the content corresponding output files with respect to the `<base>` (the base of the file names given by the `outputFile` parameter):

`<base>.log`
: Log-file, including all standard outputs (except for the message about waiting for the Enter key to be pressed).

`<base>.model`
: The models (the description of the used network, defined by the decision graphs for each variable) used for generating offspring each generation. Frequencies are also included.

`<base>.fitness`
: The best, average, and the worst fitness, with the number of current generation and the number of fitness evaluations performed in a format that is easily visualized by visualization tools as `gnuplot`.

The log-file includes all standard outputs that are explained in the previous section and therefore no additional explanation is necessary.

In the file with the models used to generate the offspring each generation, the number of current generation and the decision graphs for each variable including the conditional frequencies contained in the leaves are stored. Each node is denoted by the number of variable it corresponds to starting with 0. The number of a variable is followed by the decision graph that lists vertices in a pre-order ordering shifting each level to the right proportionally to the depth. An example decision graph follows:

```
0:
   2
       p(x0=1|...)=0.54
       1
           p(x0=1|...)=0.63
           p(x0=1|...)=0.27
```

The above graph corresponds to the 0th variable. The root has a label 2 (it's split on the 2nd variable) and it has two children. The left child (corresponding to the value 0 of the 2nd variable) includes the probability $p(X_0 = 1|X_2 = 0) = 0.54$. The right child of the root is split on the 1st variable, and has two children, containing $p(X_0 = 1|X_2 = 1, X_1 = 0) = 0.63$, and $p(X_0 = 1|X_2 = 1, X_1 = 1)$. Only decision trees are displayed completely correctly, because the algorithm can display only the trees. Decision graphs are displayed by duplication of nodes with multiple parents.

In the file with the extension `fitness` including the best, worst and average fitness values with respect to the number of current generation and the number of fitness evaluations performed so far, each line includes 5 numbers. The following list describes the values (in the order from the leftmost)

1) The number of current generation

2) The number of fitness calls performed so far

3) The best fitness in a current population

4) The average fitness in a current population

5) The worst fitness in a current population

An example of a line produced by the `boa` with a specified output file name with additional extension `fitness` is shown in Figure 7.2. The line says that in generation 7, after performing 4500 fitness calls, the best fitness in the population is 9.8, the average fitness is 9.1862, and the worst fitness is 8.400001 (this is an error caused by floating-point operations, the correct value is 8.4).

```
   7    4500   9.800000   9.186200   8.400001
```

Figure 4: Example line in the output file including the information about the fitness.

8 The Code

In this section, we shortly describe the function of each of the source files. Thereafter, we provide the instructions for plugging in a new test function into the existing code.

8.1 Brief Description of the Source Files

The following list briefly describes what functions are located in each source file. A similar description is located at the beginning of the corresponding source files among with the information about the author and the date of a last modification. For each function, a detailed description of its purpose and its input parameters are presented before its definition. The source files are heavily commented.

`args.cc`
: Functions for manipulation with arguments passed to a program.

`bayesian.cc`
: Functions for construction and use of Bayesian networks (not including some functions for manipulation with the operators).

`boa.cc`
: Functions for the initialization of the BOA, the BOA itself and a done method for the BOA.

`decisionGraph.cc`
: Definition of the `decisionGraph` class.

`fitness.cc`
: The definition of fitness functions; in order to add a fitness one has to add it here (plus the definition in the header file fitness.h); see documentation or the instructions below.

`frequencyDecisionGraph.cc`
: Definition of the `frequencyDecisionGraph` class that extends `decisionGraph`.

`getFileArgs.cc`
: Functions for reading the input file, printing the description of the parameters that can be processed, and the related.

`graph.cc`
: The definition of classes `OrientedGraph` and `AcyclicOrientedGraph` for manipulation with oriented graphs.

`header.cc`
: Prints out the header saying the name of the product, its author, the date of its release, and the file with input parameters (if any).

`help.cc`
: help (arguments description, input file parameters description).

`labeledTreeNode.cc`
: Definition of the `labeledTreeNode` class used by decision and frequency decision graphs.

`main.cc`
: Main routine and the definition of input parameters.

`mymath.cc`
: Commonly used mathematical functions.

`operator.cc`
: Types and functions to manipulate various operators on decision graphs. The operators themselves are defined within `frequencyDecisionGraph` and `decisionGraph` classes; here there are only functions that allow easier storage of alternatives in the model construction.

`population.cc`
: Functions for manipulation with the populations of strings and the strings themselves.

`random.cc`
: Random number generator related functions (random generator is based on the code by Fernando Lobo, Prime Modulus Multiplicative Linear Congruential Generator (PMMLCG).

`recomputeGains.cc`
: A function calling the metric repeatedly in order to recompute the gains for all edge additions ending in a particular node.

`replace.cc`
: The definition of replacement replacing the worst portion of the original population and the divide and conquer function it uses to separate the worst.

`select.cc`
: The definition of tournament selection.

`stack.cc`
: The definition of a class `IntStack` (a stack for `int`).

`startUp.cc`
: A start-up function for processing the arguments passed to the program and the function returning the name of the input file if any was used.

`statistics.cc`
: Functions that compute and print out the statistics during and after the run.

`utils.cc`
: Functions use elsewhere for swapping values of the variables of various data types.

8.2 Implemented Test Functions

We have implemented the following test functions (ordered by their number).

Number	Description
0	OneMax (bit-count) function
1	Quadratic function without overlapping
2	A deceptive function of order 3 without overlapping
3	A trap function of order 5 without overlapping
4	A bipolar function of order 6 without overlapping
5	A deceptive function of order 3 with 1-bit overlap between adjacent building blocks

For the definitions of the quadratic function and the trap function of order 5, see Pelikan and Mühlenbein (1999). For the definitions of the rest of the functions, see Pelikan et al. (1998) or Pelikan et al. (1999). In the latter two papers, there is a typo in the definition of the trap-5 function.

8.3 How to Plug-in a New Test Function

The instructions to plug in a new test function, also provided in `fitness.cc` follow:

1. Create a function with the same input parameters as other fitness functions defined in this file (e.g., `onemax`) that returns the value of the fitness given a binary chromosome of a particular length (sent as input parameters to the fitness). Place the function in the source file `fitness.cc`

2. Put the function definition in the `fitness.h` header file (look at `onemax` as an example).

3. In file `fitness.cc`, increase the counter `numFitness` and add a structure to the array of the fitness descriptions `fitnessDesc` as described below. For compatibility of recent input files, put it at the end of this array in order not to change the numbers of already defined and used functions. The structure has the following items (in this order), we also provide an example in the form of how the items are set for `onemax` function:

   a) a string description of the function (informative in output data files). For `onemax` the description is `"ONEMAX"`.

   b) a pointer to the function (simple "&" followed by the name of a function). For onemax this is `&onemax`, since this function is defined in a function named `onemax`.

   c) a pointer to the function that returns true if an input solution is globally optimal and false if this is not the case. If such function is not available, just use `NULL` instead. The optimum of `onemax` is in 111...1 and therefore the function `areAllGenesOne` (which returns true if the input string has 1's on all positions) is used. This item is therefore set to `&areAllGenesOne` with onemax function.

   d) a pointer to the function for initialization of the particular fitness function (not used in any of these and probably not necessary for most of the functions, but in case reading input file would be necessary or so, it might be used in this way). Use `NULL` if there is no such function. In `onemax`, no initialization is necessary and therefore this item is set to `NULL`.

e) a pointer to the "done" function, called when the fitness is not to be used anymore, in case some memory is allocated in its initialization; here it can be freed. Use `NULL` if there is no need for such function. In `onemax`, no such function is necessary and therefore this item is also set to `NULL`.

4. The function will be assigned a number equal to its ordering number in the array of function descriptions `fitnessDesc` minus 1 (the functions are assigned numbers consequently starting at 0); so its number will be equal to the number of fitness definitions minus 1 at the time it is added. Its description in output files will be the same as the description string (see 3a).

9 Final Comments

The code is distributed for academic purposes with absolutely no warranty of any kind, either expressed or implied, to the extent permitted by applicable state law. We are not responsible for any damage resulting from its proper or improper use.

If you have any comments or identify any bugs, please contact the author (email is a preferred way of communication).

Acknowledgments

The work was sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grants F49620-94-1-0103, F49620-95-1-0338, F49620-97-1-0050, and F49620-00-0163. Research funding for this work was also provided by a grant from the National Science Foundation under grant DMI-9908252. Support was also provided by a grant from the U. S. Army Research Laboratory under the Federated Laboratory Program, Cooperative Agreement DAAL01-96-2-0003. The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research, the National Science Foundation, the U. S. Army, or the U.S. Government.

References

Pelikan, M., Goldberg, D. E., & Cantú-Paz, E. (1998). *Linkage problem, distribution estimation, and Bayesian networks* (IlliGAL Report No. 98013). Urbana, IL: University of Illinois at Urbana-Champaign, Illinois Genetic Algorithms Laboratory.

Pelikan, M., Goldberg, D. E., & Cantú-Paz, E. (1999). BOA: The Bayesian optimization algorithm. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M., & Smith, R. E. (Eds.), *Proceedings of the Genetic and Evolutionary Computation Conference GECCO-99*, Volume I (pp. 525–532). Orlando, FL: Morgan Kaufmann Publishers, San Fransisco, CA.

Pelikan, M., Goldberg, D. E., & Sastry, K. (2000). *Bayesian optimization algorithm, decision graphs, and Occam's razor* (IlliGAL Report No. 2000020). Urbana, IL: University of Illinois at Urbana-Champaign, Illinois Genetic Algorithms Laboratory.

Pelikan, M., & Mühlenbein, H. (1999). The bivariate marginal distribution algorithm. In Roy, R., Furuhashi, T., & Chawdhry, P. K. (Eds.), *Advances in Soft Computing - Engineering Design and Manufacturing* (pp. 521–535). London: Springer-Verlag.