Missouri Estimation of Distribution Algorithms Laboratory

Implementation of the Dependency-Tree Estimation of Distribution Algorithm in C++

Martin Pelikan

MEDAL Report No. 2006010

September 2006

Abstract

This technical report describes how to download, compile and use the C++ source code of the dependency-tree estimation of distribution algorithm, which uses dependency trees to model promising solutions and sample the new ones. Candidate solutions are represented by fixed-length strings with a finite number of values in each string position (the code is *not* restricted to binary strings). The report also describes how to modify the code to solve new optimization problems. Additional documentation can be found in the provided software package, which includes a detailed documentation in PDF, HTML, and LaTeX formats.

Keywords

Estimation of distribution algorithm, dependency tree, probabilistic models, implementation, optimization, evolutionary computation,C++.

Missouri Estimation of Distribution Algorithms Laboratory (MEDAL)
Department of Mathematics and Computer Science
University of Missouri–St. Louis
One University Blvd., St. Louis, MO 63121
E-mail: medal@cs.umsl.edu
WWW: http://medal.cs.umsl.edu/

Implementation of the Dependency-Tree Estimation of Distribution Algorithm in C++

Martin Pelikan
Missouri Estimation of Distribution Algorithms Laboratory (MEDAL)
Dept. of Math and Computer Science, 320 CCB
University of Missouri at St. Louis
One University Blvd., St. Louis, MO 63121
pelikan@cs.umsl.edu

Abstract

This technical report describes how to download, compile and use the C++ source code of the dependency-tree estimation of distribution algorithm, which uses dependency trees to model promising solutions and sample the new ones. Candidate solutions are represented by fixed-length strings with a finite number of values in each string position (the code is *not* restricted to binary strings). The report also describes how to modify the code to solve new optimization problems. Additional documentation can be found in the provided software package, which includes a detailed documentation in PDF, HTML, and LaTeX formats.

Keywords: Estimation of distribution algorithm, dependency tree, probabilistic models, implementation, optimization, evolutionary computation,C++.

1 Introduction

Estimation of distribution algorithms (EDAs) (Baluja, 1994; Mühlenbein & Paaß, 1996; Larrañaga & Lozano, 2002; Pelikan, Goldberg, & Lobo, 2002) replace standard crossover and mutation operators of genetic and evolutionary algorithms by building a probabilistic model of selected solutions and sampling the built model to generate new candidate solutions. This paper discusses how to download, compile and use the C++ implementation of the dependency-tree EDA, which was one of the first EDAs to consider dependencies between variables in the construction of the probabilistic model (Baluja & Davies, 1997).

Before discussing the algorithm and its implementation, let's briefly look at the main reasons for implementing the dependency-tree estimation of distribution algorithm. First of all, the reason for considering EDAs as the staring point is that EDAs outperform other types of evolutionary algorithms on broad classes of challenging problems because they are able to adapt to the problem and deal effectively with both linkage learning and mixing (Goldberg, 2002; Larrañaga & Lozano, 2002; Pelikan, 2005), which are the two driving forces of exploration in selectorecombinative genetic algorithms (Goldberg, 2002). But why should we use dependency trees when we can use more advanced models instead, such as multiply connected Bayesian networks? The most important reason to prefer simple models is *efficiency* because simple models can usually be created much faster than the complex ones and they still often provide sufficient accuracy (Baluja & Davies, 1997; Pelikan & Mühlenbein, 1999).

```
Estimation of distribution algorithm (EDA)
  t := 0;
  generate initial population P(0);
  while (not done) {
    select population of promising solutions S(t);
    build probabilistic model M(t) for S(t);
    sample M(t) to generate O(t);
    incorporate O(t) into P(t);
    t := t+1;
  };
```

Figure 1: Pseudocode of the estimation of distribution algorithm (EDA).

2 Algorithm Description

Estimation of distribution algorithms (EDAs) (Mühlenbein & Paaß, 1996; Larrañaga & Lozano, 2002; Pelikan, Goldberg, & Lobo, 2002) evolve a population (multiset) of potential solutions to the given optimization problem using a combination of techniques from evolutionary computation and machine learning.

Like standard evolutionary algorithms (Goldberg, 1989; Holland, 1975), EDAs start with a randomly generated population of potential solutions. Each iteration updates the population using selection and variation operators. The selection operator selects a population of promising solutions from the current population. However, instead of using variation operators inspired by natural evolution and genetics, EDAs create new solutions by *building a probabilistic model* of selected solutions and *sampling the built model* to generate new candidate solutions. The new candidate solutions are incorporated into the original population and the next iteration is executed unless some termination criteria have been met. Ideally, the quality of candidate solutions generated by the probabilistic model improves over time, and, after a reasonable number of iterations, the model should generate only the best optima. The EDA pseudocode can be found Figure 1.

In this implementation, the probabilistic model built for the selected population of solutions forms a dependency tree where each variable is conditioned on its predecessor (see Figure 2 for an example dependency tree). The overall distribution is then given as the product of the marginal probability of the variable at the root of the tree and the conditional probabilities of the remaining variables given their predecessors. Using probabilistic trees is a common approach in machine learning because learning more general models, such as multiply connected Bayesian networks, is sometimes intractable. Tree probabilistic models have also been used in EDAs; the model used in this paper is formed analogically to the one in the COMIT algorithm (Baluja & Davies, 1997).

For more details on the model learning and sampling algorithms for dependency trees, please see Baluja and Davies (1997). The remainder of this technical report focuses on the details of the implementation and usage of the dependency-tree EDA.

3 Downloading the Code

The code of the dependency-tree EDA can be downloaded from the following web address:

```
http://medal.cs.umsl.edu/files/dt-eda.tar.gz
```

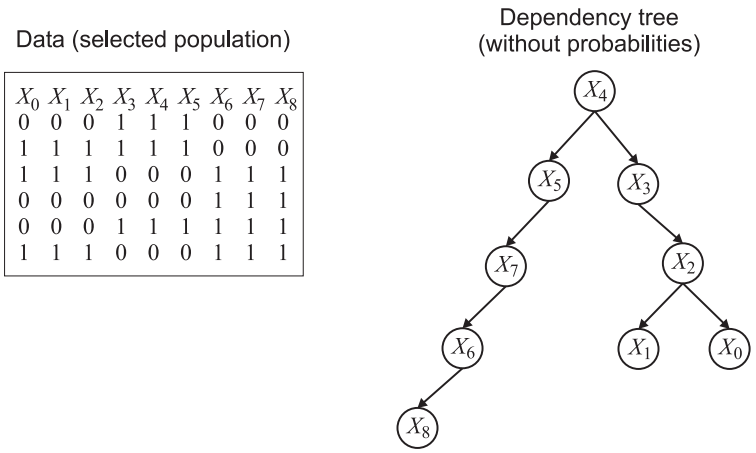

Figure 2: An example dependency tree for a population of 6 strings of 9 bits each. After examining the population, strong correlations can be found among the group of the first three string positions (X_0, X_1, and X_2), and analogical correlations can be found for the groups (X_3, X_4, X_5) and (X_6, X_7, X_8). The dependency tree captures many of these interactions by relating the most strongly correlated variables using conditional dependence. In addition to the structure, the model would still have to define the conditional probabilities of each variable given its parent, which can also be learned from the given sample.

In UNIX systems, you can probably use `wget`, `links`, or `lynx` to download the file in the text mode. After downloading the file `dt-eda.tar.gz`, move it into the directory where you want the source code folder to reside. Then, use your favorite compression package to unpack the downloaded archive. In UNIX systems, you can use `gunzip` and `tar` from the directory with the downloaded archive to uncompress the package as follows:

```
gunzip dt-eda.tar.gz
tar xvf dt-eda.tar.gz
```

In most UNIX systems, you can also use a single command to uncompress the package:

```
tar xvzf dt-eda.tar.gz
```

If you use `tar` or `gunzip` and `tar`, a directory `dt-eda` would be created upon successful completion of `tar`. In other compression software, there should be an option to preserve the relative path names of extracted files, if this option is not enabled by default.

Let's now assume that you successfully unpacked all the files into the directory `dt-eda`. After examining the content of this directory, you should find the following files and directories (probably in a different order and/or format):

```
Makefile                  bisection.cpp     stats.cpp        obj-function.hpp
README                    eda.cpp           status.cpp       parse-input.hpp
doc/                      heap.cpp          tree-model.cpp   random.hpp
example_input             main.cpp          MT.hpp           stats.hpp
example_input_big         obj-function.cpp  bisection.hpp    status.hpp
example_input_bisection   parse-input.cpp   eda.hpp          tree-model.hpp
MT.cpp                    random.cpp        heap.hpp         documentation.pdf
howto.pdf
```

README contains basic information about the code. howto.pdf contains the text of this document. documentation.pdf contains the documentation generated from the source code using Doxygen. The subdirectory doc contains the Doxygen-generated documentation in HTML and LaTeX formats. The file extensions cpp and hpp denote the source code of the dependency-tree EDA. Makefile is used for compiling the package using standard GNU make. Finally, the files example_input, example_input_bisection, and example_input_big denote example input parameter files, which can be provided to the algorithm as a command line argument.

4 Compiling the Code

The source code has been tested with g++ compiler from GNU Compiler Collection on Linux, Windows, Mac OS X and Sun Solaris but it should be straightforward to compile it using other C++ compilers.

If you are using g++, to compile the code, go to the dt-eda directory and run make. Otherwise, make appropriate modifications in Makefile and run the make command afterwards. After successful compilation, there will be an executable called dt-eda or dt-eda.exe, depending on the system you are using.

Of course, you do not have to use the provided Makefile and can compile the source code in another way instead.

5 Source Code Files

The source code consists of the following files:

- obj-function.cpp
  Implements the objective function to maximize. Additionally, functions are defined that (1) specify the number of symbols in each string position and (2) verify optimality of candidate solutions. This is most likely the only part of the source code that the user really needs to access. A header for obj-function.cpp is obj-function.hpp.

- main.cpp
  Defines the main function, which processes command-line arguments, loads the input parameter file (if the filename is specified), and runs the algorithm.

- parse-input.cpp
  Parses input parameter file if the user has specified the filename as a command line argument. If the user wants to modify current parameters or add new ones, this is the file to start with. A header for parse-input.cpp is parse-input.hpp.

- eda.cpp
  Encodes most functions related to the estimation of distribution algorithm except for the model building and model sampling procedures, which are located in tree-model.cpp. A header for eda.cpp is eda.hpp.

- tree-model.cpp
  Implements TreeModel class, which contains all functions for learning and sampling models in the form of probabilistic trees. A header for tree-model.cpp is tree-model.hpp.

- `stats.cpp`
  Implements basic statistics manipulation functions to collect data from populations and combine the results of multiple runs. The user may want to extend or modify this part as well. A header for `stats.cpp` is `stats.hpp`.

- `MT.cpp`
  Mersenne Twister random generator. A header for `MT.cpp` is `MT.hpp`.

- `random.cpp`
  Simple interface functions to Mersenne Twister random number generator. A header for `random.cpp` is `random.hpp`.

- `heap.cpp`
  Procedures for manipulating a binary max-heap, which is used in the Prim's algorithm in the construction of the dependency-tree model. A header for `heap.cpp` is `heap.hpp`.

- `status.cpp`
  A text-mode status bar used in the verbose mode. A header for `status.cpp` is `status.hpp`.

The code is heavily commented and documented, especially the parts that are expected to be modified by the user (especially `obj-function.cpp`). The subdirectory `doc` contains additional documentation in various formats generated with Doxygen, including more examples of the user-defined components.

6 Running the Code

There are two basic modes that the algorithm operates in:

- *Execute a single run.*
  In the single-run mode, the user specifies all parameters (unspecified parameters are filled in from default values), and the algorithm proceeds by making one run with the specified parameters. Example input files for this mode are `example_input` and `example_input_big`.

- *Run bisection to determine optimal population size.*
  In the bisection mode, the algorithm also starts with user-defined parameters, but then it uses bisection to find the minimum population size that would ensure reliable convergence (Sastry, 2001). Since bisection uses a number of independent runs, using bisection takes longer than running the algorithm just once with adequate parameter settings. Nonetheless, bisection provides us with good estimates of algorithm performance with adequate parameter settings without having to play around with parameters much. Furthermore, the performance we obtain with bisection should be close to the optimal performance. The final result corresponds to the averages over the successful runs with the optimal population size. There is an example input file for the bisection mode called `example_input_bisection` included in the main directory. Running bisection runs with `quiet_mode=0` should help to understand the mechanics of bisection.

Without any command line arguments, `dt-eda` proceeds with the default values of all parameters and executes one run on a simple default problem.

For more complex use, we need to use command line arguments. `dt-eda` accepts one command line argument, which can be either of the following:

- `[input parameter file name]`
  The program loads parameters from the file specified by this string parameter. All parameters not specified in the input file will be set to their default values.
- `--help`
  The program displays a short help on the basic use of the code.
- `--version`
  The program displays its version.

To get help, just execute the following:

```
./dt-eda --help
```

To run the algorithm with the default parameter values, run the following:

```
./dt-eda
```

To run the algorithm with parameter settings specified in the file `example_input`, which is provided in this distribution, run the following:

```
./dt-eda example_input
```

In the last two cases, you should see a quick successful run. You can look at the `example_input` file to get an idea of how it defines the basic parameters.

Other example parameter files are `example_input_bisection` and `example_input_big`. For example, try to run the following command:

```
./dt-eda example_input_bisection
```

The standard output will show all steps of the bisection algorithm, which attempts to find a good estimate of an adequate population size for reliable convergence to the global optimum (Sastry, 2001). To turn off all but the summary output, set `quiet_mode=1` in the input file.

7 Changing the Objective Function and Other Problem Specifics

The default objective function is a modified onemax, which takes a string of integers from 0 to 2 and returns a sum of all the integers. The maximum is thus in the string of all 2s. The only reason to not use binary strings was to stress that this implementation does *not* assume the binary representation.

Of course, the user will want to specify other objective functions and other representations, using for example strings with characters that can obtain 10 values. To incorporate these modifications, one should edit the file `obj-function.cpp`.

There are three important functions in the file `obj-function.cpp`:

- `double objective_function(int *x, int n)`
  This is the objective function. On input, there is an array of integers `x` and the string length `n`. The function returns the value of the objective function. Each character of the input string can obtain values from 0 to the number of its values minus one. The task is to *maximize* the objective function. One example of an objective function is included in the source code, another example can be found in the documentation in the `doc/` folder.

- `void set_num_vals(int *num_vals, int n)`
  This function sets the values of the vector `num_vals` that specifies the number of values of each character in a string. Each character then takes values from 0 to its maximum value minus 1. This function must be defined even if one uses binary strings, in which case all entries of the `num_vals` vector are set to 2. The vector has already been allocated once this function is called, so there is no need for memory allocation of any kind. An example implementation of this function can be found in the source code; another example can be found in the documentation.

- `int is_optimal(int *x, int n, int *num_vals)`
  This function verifies whether a given candidate solution is a global optimum (but it can be used also for specifying *good enough* solutions if this is the goal). If it is not possible to design such a function, one can use a function that always returns 0. However, the function must be defined.

8 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 from its proper or improper use.

Feel free to use, modify and distribute the code with an appropriate acknowledgment of the source, but in all resulting publications please include the following citation:
Martin Pelikan (2006). *Implementation of the Dependency-Tree Estimation of Distribution Algorithm in C++.* MEDAL Report No. 20060010, Missouri Estimation of Distribution Algorithms Laboratory, University of Missouri in St. Louis, MO.

If you find any bugs in the source code, please report them to Martin Pelikan at

`pelikan@cs.umsl.edu`

Acknowledgments

This project was sponsored by the National Science Foundation under CAREER grant ECS-0547013, the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant FA9550-06-1-0096, and the University of Missouri in St. Louis through the High Performance Computing Collaboratory sponsored by Information Technology Services, and the Research Award and Research Board programs.

The U.S. Government is authorized to reproduce and distribute reprints for government purposes notwithstanding any copyright notation thereon. Most experiments were completed at the Beowulf cluster at the University of Missouri–St. Louis.

References

Baluja, S. (1994). *Population-based incremental learning: A method for integrating genetic search based function optimization and competitive learning* (Tech. Rep. No. CMU-CS-94-163). Pittsburgh, PA: Carnegie Mellon University.

Baluja, S., & Davies, S. (1997). Using optimal dependency-trees for combinatorial optimization: Learning the structure of the search space. *Proceedings of the International Conference on Machine Learning*, 30–38.

Goldberg, D. E. (1989). *Genetic algorithms in search, optimization, and machine learning*. Reading, MA: Addison-Wesley.

Goldberg, D. E. (2002). *The design of innovation: Lessons from and for competent genetic algorithms*, Volume 7 of *Genetic Algorithms and Evolutionary Computation*. Kluwer Academic Publishers.

Holland, J. H. (1975). *Adaptation in natural and artificial systems*. Ann Arbor, MI: University of Michigan Press.

Larrañaga, P., & Lozano, J. A. (Eds.) (2002). *Estimation of distribution algorithms: A new tool for evolutionary computation*. Boston, MA: Kluwer.

Mühlenbein, H., & Paaß, G. (1996). From recombination of genes to the estimation of distributions I. Binary parameters. *Parallel Problem Solving from Nature*, 178–187.

Pelikan, M. (2005). *Hierarchical Bayesian optimization algorithm: Toward a new generation of evolutionary algorithms*. Springer-Verlag.

Pelikan, M., Goldberg, D. E., & Lobo, F. (2002). A survey of optimization by building and using probabilistic models. *Computational Optimization and Applications*, *21*(1), 5–20. Also IlliGAL Report No. 99018.

Pelikan, M., & Mühlenbein, H. (1999). The bivariate marginal distribution algorithm. *Advances in Soft Computing—Engineering Design and Manufacturing*, 521–535.

Sastry, K. (2001). *Evaluation-relaxation schemes for genetic and evolutionary algorithms*. Master's thesis, University of Illinois at Urbana-Champaign, Department of General Engineering, Urbana, IL. Also IlliGAL Report No. 2002004.