The Linkage Tree Genetic Algorithm

Dirk Thierens

Institute of Information and Computing Sciences
Universiteit Utrecht, The Netherlands
dirk.thierens@cs.uu.nl

Abstract. We introduce the *Linkage Tree Genetic Algorithm (LTGA)*, a competent genetic algorithm that learns the linkage between the problem variables. The LTGA builds each generation a *linkage tree* using a hierarchical clustering algorithm. To generate new offspring solutions, the LTGA selects two parent solutions and traverses the linkage tree starting from the root. At each branching point, the parent pair is recombined using a crossover mask defined by the clustering at that particular tree node. The parent pair competes with the offspring pair, and the LTGA continues traversing the linkage tree with the pair that has the most fit solution. Once the entire tree is traversed, the best solution of the current pair is copied to the next generation. In this paper we use the normalized variation of information metric as distance measure for the clustering process. Experimental results for fully deceptive functions and nearest neighbor NK-landscape problems with tunable overlap show that the LTGA can solve these hard functions efficiently without knowing the actual position of the linked variables on the problem representation.

1 Introduction

In general, the search bias of recombination operators can be beneficial to the search efficiency in the following two cases:

1. Different partial structures of two good solutions can be juxtaposed by crossover to form a new good solution.
2. Common partial structures shared by two good solutions are shielded from crossover disruption, and the new solution inherits the common partial structures, while it randomly samples the subspace where the parents disagree.

In this paper we focus on the first case. In a standard genetic algorithm this case can only work if the partial structures are not disrupted too often by recombination. This can be achieved by designing the solution representation and/or the crossover operator in an appropriate way. However, if there is not enough domain knowledge to design a suitable representation and/or crossover operator, we have to induce this knowledge from a population of solutions. Learning what variables form important partial solutions - and therefore should be protected from disruption by crossover - is called *linkage learning*. During the past decade a number of linkage learning evolutionary algorithms have been proposed [1][4][9].

R. Schaefer et al. (Eds.): PPSN XI, Part I, LNCS 6238, pp. 264–273, 2010.

In this paper we introduce an alternative linkage learning GA called the *Linkage Tree Genetic Algorithm (LTGA)*. The LTGA builds each generation a linkage tree using a hierarchical clustering algorithm. To generate new offspring it traverses the linkage tree and uses the clustering specified at each tree node as a crossover mask. The next section explains how the LTGA works. Section 3 discusses the distance measure we use to learn the linkage between variables and groups of variables. We also compare the LTGA to related linkage learning GAs. Section 4 shows experimental results of the LTGA on deceptive trap functions and nearest neighbor NK-landscape problems with tunable overlap. Finally, Section 5 concludes the paper.

2 Linkage Tree Genetic Algorithm

2.1 Linkage Tree

Linkage learning evolutionary algorithms aim to identify which variables should be treated as a dependent set of variables during the exploration phase. In this paper we learn linkage between variables - and groups of variables - by building a hierarchical cluster using a proximity distance that measures how correlated the variables or groups of variables are in the current population.

Definition 1. *The* Linkage Tree *of a population of solutions is the hierarchical cluster tree of the problem variables using an agglomerative hierarchical clustering algorithm with a distance measure D. The distance measure $D(X_1, X_2)$ measures the degree of dependency between two sets of variables X_1 and X_2.*

Algorithm. HIERARCHICAL CLUSTERING

1. Compute the proximity matrix using metric D.
2. Assign each variable to a single cluster.
3. Repeat until one cluster left:
4. Join two nearest clusters c_i and c_j into c_{ij}.
5. Remove c_i and c_j from the proximity matrix.
6. Compute distance between c_{ij} and all clusters.
7. Add cluster c_{ij} to the proximity matrix.

An agglomerative hierarchical clustering algorithm proceeds bottom-up. First, each problem variable is assigned to a single cluster. Then, the clustering algorithm recursively joins the closest clusters until only one cluster is left. For a problem of length ℓ the linkage tree has ℓ leaf nodes (the clusters having a single problem variable) and $\ell - 1$ internal nodes. Each (internal or leaf) node of the linkage tree divides the set of problem variables into two mutually exclusive subsets. One subset is the cluster of variables at that node, while the other subset is the complementary set of problem variables. The LTGA uses this division of the problem variables as a crossover mask. The variables specified in the cluster of a specific node are swapped between two parent solutions to generate an offspring

pair. For instance, assume LTGA crosses the parent pair $(00001111, 00110011)$ at the internal node of the linkage tree with cluster (x_0, x_1, x_4, x_5). The values at position 0, 1, 4 and 5 are swapped which results into the offspring pair $(00000011, 00111111)$. The LTGA evaluates the fitness of the offspring and holds a competition between the parent pair and the offspring pair. If one of the children is better than both parents the offspring pair replaces the parent pair, and LTGA continues to traverse the linkage tree with the new pair. If none of the two children is better than both parents, LTGA continues its tree traversal with the parent pair. When the tree is completely traversed, the best solution of the current pair is copied to the next generation. To increase the efficiency, LTGA always checks whether the offspring solutions are actually different from their parents, if not no call to the fitness function is done, and the algorithm simply proceeds with its tree traversal.

The order in which the tree is traversed, is the opposite order of the merging of the clusters by the hierarchical clustering algorithm. Therefore, LTGA first crosses the clusters which are the least dependent on each other - this is, they are least linked. The clustering algorithm initially pushes all single variable clusters on a stack. Then, it iteratively joins the two closest clusters and pushes the joint cluster on the stack. When all clusters are merged the stack will consist of $2\ell - 1$ clusters, ℓ of them being single variable clusters. During tree traversal, LTGA simply pops the clusters of the stack and uses them as crossover mask. Note that half of the crossovers are actually single bit flips, so LTGA is also performing a bitwise search. Of course this is redundant in applications where the LTGA is combined with another local search algorithm, in this case the single variable clusters are not pushed on the stack, effectively reducing the number of crossover masks by half.

Algorithm. Linkage Tree Genetic Algorithm (LTGA)

```
 1. Create initial population of size N.
 2. Repeat until stop criteria met:
 3.  Build the linkage tree.
 4.  Do for N solution pairs:
 5.   Traverse one step in the linkage tree.
 6.   Set crossover mask to the current clustering.
 7.   Cross solution pair using the crossover mask.
 8.   When one offspring is better than both parents:
 9.    Replace parent pair with the offspring pair.
10.   If tree fully traversed:
11.    Copy best solution to next population.
12.    Else, go to step 5.
```

3 Hierarchical Clustering Using Mutual Information

The hierarchical clustering algorithm requires a distance measure that measures the amount of linkage between two clusters of variables. In [7] a hierarchical clustering algorithm is outlined that uses a mutual information based distance

measure. In this paper we will use the same distance measure to build the linkage tree of a population of solutions. Assume X_k is a discrete, random variable with probability mass function $p(X_k)$. The entropy H is then defined as $H(X_k) = -\sum_i p_i(X_k) \log p_i(X_k)$. The mutual information I between a set of random variables is defined as $I(X_1, \ldots, X_\ell) = \sum_{k=1}^{\ell} H(X_k) - H(X_1, \ldots, X_\ell)$. Mutual information is particularly interesting for use in a hierarchical clustering algorithm due to its *grouping property*. The grouping property states that the mutual information between three clusters of random variables C_1, C_2 and C_3 is equal to the sum of the mutual information between two clusters C_1 and C_2, plus the mutual information between the union of the two clusters $C_1 \cup C_2$ and $C3$: $I(C_1, C_2, C_3) = I(C_1, C_2) + I((C_1 \cup C_2), C_3)$. It is important to realize that the mutual-information I is a similarity measure between objects but is not a distance measure. Hierarchical clustering requires a distance measure between clusters to build the cluster decomposition. A distance measure based on mutual information is the *variation of information* $d(X_1, X_2)$ which is the difference between the joint entropy $H(X_1, X_2)$ and the mutual information $I(X_1, X_2)$:

$$d(X_1, X_2) = H(X_1, X_2) - I(X_1, X_2) = H(X_1|X_2) + H(X_1|X_2).$$

In hierarchical clustering we are comparing clusters of different sizes so it is preferable to normalize the distance measure $d(X_1, X_2)$ by dividing it by the total information as represented by the entropy H:

$$D(X_1, X_2) = \frac{d(X_1, X_2)}{H(X_1, X_2)} = 2 - \frac{H(X_1) + H(X_2)}{H(X_1, X_2)}.$$

Interestingly, D is a metric with $0 \leq D(X_1, X_2) \leq 1$. In the experiments in Section 4 we will use this metric as distance measure for the hierarchical clustering.

3.1 Related Work

The LTGA is related to the ClusterMI [3] and to the Dependency Structure Matrix Genetic Algorithm (DSMGA) [13]. The DSMGA uses a (non-hierarchical) clustering algorithm to learn the linkage between the variables. DSMGA applies a bitwise hillclimbing search algorithm and a Minimum Description Length (MDL) measure to find a clustering that accurately models the building-block structure of the problem. This model is used to perform building-block wise crossover. The ClusterMI computes a hierarchical clustering using the mutual information between all the problem variables. To compute the distance between 2 clusters of variables ClusterMI computes the mean of the mutual information between the variables of each cluster. However, as mentioned above, mutual information is a similarity measure, not a distance measure, and it is not clear what it actually means to take the average mutual information as the distance measure for the hierarchical clustering algorithm. Obviously, taking the average mutual information between variables as a measure between clusters is much more efficient than computing an actual distance between clusters. Perhaps it

might be a good compromise to use the average linkage distance D between variables as an approximation of the distance measure between clusters.

The main difference between ClusterMI and DSMGA on the one hand, and the LTGA on the other, is that the former two are Estimation of Distribution Algorithms (EDAs) whose aim is to learn a probabilistic model of the current population. Therefore, they try to learn the exact partitioning model that directly represents the building-block structure of the fitness function. Both the ClusterMI and the DSMGA use a MDL measure as originally proposed in the Extended Compact Genetic Algorithm (ECGA) [5] to induce this particular clustering. Learning a single partitioning however makes these algorithms vulnerable to modeling errors. This vulnerability expresses itself by the rather large minimal population sizes required for entropy-based model building in discrete estimation of distribution algorithms [14]. The LTGA is more robust in that sense. It builds a complete linkage tree and recombines parent solutions at different levels of dependencies - or linkage - between variable clusters. LTGA does not generate new solutions by sampling from a probability distribution. Instead, it repeatedly recombines a parent pair in search of a better offspring. This extensive exploration of the parent pair is guided by the structure of the fitness landscape, as expressed by the induced linkage tree.

The LTGA can actually be looked upon as a hybrid between standard genetic algorithms and estimation of distribution algorithms. By building a probabilistic model EDAs capture the global structure of the fitness landscape. Single good solutions however might contain certain detailed information that is not (well) captured in the global probability model. A GA using crossover or mutation might be better suited to preserve the intricate details of a good solution. Through the extensive recombination and mutation - combined with the family elitism - the LTGA inherits the exploration and exploitation capabilities of GAs, while the guidance by the linkage tree makes it respect the global structure of the fitness landscape as done by EDAs.

Finally, recent work on network crossover appears to share some interesting similarities with the linkage tree crossover [6] [11]. In future work we plan to compare these methods.

4 Experimental Results

4.1 Deceptive Trap functions

We have tested the LTGA on the deceptive mk-trap function DTF [2]. DTF is a binary, additively decomposable function composed of m trap functions DT_i, each defined on a separate group of k bits (the total problem length is $\ell = mk$): $DTF(x_1 \dots x_\ell) = \sum_{i=0}^{m-1} DT_i(x_{ik} \dots x_{ik+k-1})$ with $x_i \in \{0, 1\}$. Call u the number of bits in such a group that are equal to 1:

$$DT_i(x_{ik} \dots x_{ik+k-1}) = \begin{cases} k, & if\ u = k \\ k - 1 - u, & otherwise. \end{cases}$$

Clearly, the global optimal solution is the string of all 1-bits. The number of local optima is $2^m - 1$, and a hillclimbing algorithm quickly becomes trapped in one of these local optima. Furthermore, all schemata of order less than k are deceiving. This means that the schema fitness of the schemata containing the local optima - consisting of bits 0 - are better than the competing schemata that contains the optimal bits 1. Any standard GA that uses a disruptive crossover operator - like uniform crossover - in combination with a moderate selection pressure, will quickly converge to the deceptive local optima. When the individual trap functions are tightly linked - this is, they are represented in the string as consecutive blocks - a GA using a position biased crossover like 1- or 2-point crossover will quickly find the global optimum. However, when the trap functions are randomly scattered over the string, no position biased crossover can mix the different optimal substrings to form the global optimum. In order to find the global optimum, a standard GA will have to apply a crossover operator that is not position biased, like uniform crossover, however, to balance the high disruption rate of uniform crossover the GA will have to increase the selection pressure dramatically. Unfortunately, with such a high selection pressure the population size also needs to increase dramatically in order to prevent premature convergence. As a result, the number of function evaluations needed to find the optimal solution by the standard GA scales exponentially in the problem length - this is, the number of trap functions DT_i for any fixed length k [4][12].

In the **first experiment**, we test the LTGA on deceptive functions with deception length $k = 5$. The number of mk-trap functions varies from 5 to 20 with increments of 5, the problem length thus varies from 25 to 100 with increments of 25. We fix the population size at $N = 128$ which is enough to find the optimal solution in at least 24 out of 25 independent runs. In this experiment the initial population is a population of random local optimal strings, obtained by running a bitwise hillclimbing algorithm. During the LTGA search no local search is applied anymore. Table 1 shows the first hitting time, this is the number of function evaluations needed to find the global optimum for the first time. We also show the growth ratio of the median number of function evaluations, and the growth ratio of the CPU runtime. The ratio are relative to the value of the smallest problem length ($\ell = 25$). A least squares fit indicates that the growth ratio of the number of functions evaluations is of order $\Theta(\ell \log \ell)$, while the growth ratio of the CPU runtime is of order $\Theta(\ell^{2.9} \log \ell)$.

Table 1. The 1^{st}, 2^{nd}, and 3^{rd} quartile of the number of functions evaluations needed to hit the global optimum for the first time. The population size $P = 128$. The last two columns show the growth ratio of the number of evaluations and the CPU runtime.

Length	Blocks	Evals Q1	Evals Q2	Evals Q3	Evals Ratio	Runtime Ratio
25	5	13095	15069	16253	1	1
50	10	38073	43933	45074	2.9	4
75	15	65889	70397	73615	4.7	10
100	20	96367	97818	102146	6.5	25

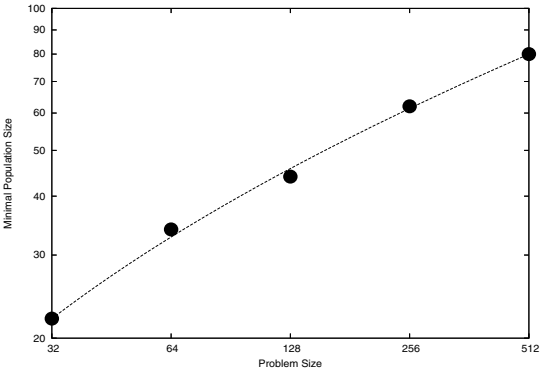

Fig. 1. The minimal population size needed for $k = 4$ problems with increasing problem length. The fitted curve is of order $\Theta(m^{0.14} \ln m)$.

In the **second experiment**, we investigate how the minimal population size scales with the problem length. The deception length is fixed at $k = 4$, the problem length varies from $\ell = 32$ to $\ell = 512$ each time doubling the string length, which corresponds to the number of blocks $m = 8, 16, 32, 64, 128$. In order to be able to compare the LTGA's scaling behavior with that of the ECGA and the ClusterMI algorithms in Figure 1 of [3], we count a run successful when at least 29 out of 30 runs have at least $m - 1$ trap functions correct. There is also no local search involved, so the LTGA is run on a random initial population. Figure 1 shows the results and a least-squares fit of $P = 8\ m^{0.14} \ln m$. LTGA's scaling behavior is thus $\Theta(m^{0.14} \ln m)$, while the ECGA and the ClusterMI have a scaling behavior of $\Theta(m^{1.1} \ln m)$. For instance for $\ell = 256$ or $m = 64$ the minimal population size is $P = 60$ for LTGA, while it is well above $P > 10000$ for ECGA and ClusterMI. Of course, that does not mean that LTGA requires less functions evaluations. To generate a single solution for the next population, LTGA traverses the entire linkage tree and evaluates a new offspring couple at each internal node where the offspring is different from the parents. Actually, in this experiment the overall number of function evaluations are similar for LTGA, ECGA, DSMGA, and ClusterMI.

4.2 Nearest Neighbor NK-Landscape with Tunable Overlap

In the previous section we investigated the ability of the LTGA to learn what problem variables should be linked together. Deceptive trap functions are ideal benchmark functions to test this. In this section we want to investigate how the LTGA deals with problems having overlap between the different subproblems. Therefore, we look at the performance on NK-landscape problems with nearest neighbor interactions and tunable overlap (nn-NK) [8][10]. A big advantage

of nn-NK problems above the more general NK problems is that we can compute the global optimal solution with dynamic programming when using the knowledge of the position of the subproblems [8]. The LTGA of course does not have access to this positional information, its task is to learn the linkage from the current population of solutions. The nn-NK are interesting as benchmark functions because we can tune the amount of overlap between the subproblems and see whether the LTGA is still successful in finding the global optimum. Formally, a nn-NK function is defined by its length (ℓ), the size of the subproblems (k), the amount of overlap between the subproblems (o), and the number of subproblems (m). The first subproblem is defined at the first k string positions. The second subproblem is defined at the last o positions of the first subproblem and the next $(k - o)$ positions. All remaining subproblems are defined in a similar way. As an example, a nn-NK problem with $\ell = 65$, $k = 5$, $0 = 3$, and $m = 31$ has the following positions of the subproblems: $(0\,1\,2\,3\,4)(2\,3\,4\,5\,6)(4\,5\,6\,7\,8)\ldots(56\,57\,58\,59\,60)(58\,59\,60\,61\,62)(60\,61\,62\,63\,64)$. The relationship between the problem variables is $\ell = k + (m - 1)(k - o)$.

Table 2. Results for fixed length ($\ell = 65$) nn-NK problem with varying overlap

Overlap	Blocks	PopSize	Evals Q1	Evals Q2	Evals Q3	Runtime Ratio
1	16	180	123394	141147	165765	1
2	21	230	191619	224935	243720	1.5
3	31	240	217810	249200	265669	2
4	61	240	179407	220736	235499	2

Table 3. Results for fixed overlap ($o = 4$) nn-NK problem with varying length

Length	Blocks	PopSize	Evals Q1	Evals Q2	Evals Q3	Evals Ratio	Runtime Ratio
20	16	70	1719	4154	5613	1	1
40	36	140	38400	54021	76181	13	15
60	56	220	158549	182514	210568	44	100
80	76	380	393156	478951	540138	115	350
100	96	600	936167	1013846	1120435	244	1000

Table 2 and 3 show the number of function evaluations of the first hitting time and the growth ratio. Table 2 fixes the problem length $\ell = 65$ and varies the overlap $o \in \{1, 2, 3, 4\}$, or equivalently the number of blocks $m \in \{16, 21, 31, 61\}$. Table 3 fixes the overlap $o = 4$ and varies the problem length $\ell \in \{20, 40, 60, 80, 100\}$, or equivalently the number of blocks $m \in \{16, 36, 56, 76, 96\}$. Both tables show the minimal population size required to find the global optimum in at least 24 out of 25 independent runs. To determine this value we start with a population that is sufficiently large to reliably find the global optimum. Next we iteratively decrease the population size time by 10, until we encounter the first size where the LTGA no longer finds the global optimum at

least 24 out of 25 runs. For each independent run a different random nn-NK problem is generated. We also run a bitwise hillclimber to each random initial solution. The LTGA thus learns its first linkage tree from a population of local optima. Thereafter, the hillclimber is not applied anymore.

In Table 2 it can be seen that for a fixed length problem the population size, the number of function evaluations, and the CPU runtime, hardly varies as a function of the overlap (especially for the overlap values $o =\in \{2, 3, 4\}$). Table 3 shows how the population size, the number of function evaluations, and the growth ratio of the number of function evaluations and the CPU runtime varies with increasing problem length (or increasing block number). A least squares fit indicates a scaling behavior of $\Theta = (\ell^{1.5} \ln \ell)$ for the minimal population size, $\Theta = (\ell^{3.1} \ln \ell)$ for the median number of function evaluations, and $\Theta = (\ell^{4.4} \ln \ell)$ for the corresponding CPU runtime.

To investigate the use of learning the linkage tree we have also run the LTGA without measuring the distance between clusters. So whenever the distance measure D is called to compute a value, we simply return a random number in [0...1]. This basically replaces our linkage guided crossover by uniform random crossover masks. Almost none of these runs ever found the global optimum which shows that learning the linkage in a hierarchical tree is beneficial even for problems with overlapping building blocks.

5 Conclusion

We have introduced the Linkage Tree Genetic Algorithm (LTGA). A linkage tree is the tree obtained by a hierarchical clustering procedure using a distance measure that represents the linkage between the problems variables or between clusters of problem variables. Each generation, the LTGA builds a new linkage tree from the current population. To generate new offspring solutions, the LTGA chooses two parent solutions and traverses the entire linkage tree starting from the root. At each tree node the LTGA recombines the two parent solutions according to the crossover mask specified by the clustering at that particular node. Whenever one of the two offspring has a better fitness score than both parents, the offspring pair replaces the parent pair, and the LTGA continues its traversal of the tree, now crossing the offspring pair. If none of the offspring improves on both its parents, LTGA simply continues with the parent pair. When the entire linkage tree has been visited, the best solution from the current pair is copied to the population of the next generation.

In this paper we have used as distance measure the normalized variation of information metric. This metric is based on mutual information between items and cluster of items, and exploits the grouping property of mutual information to build a hierarchical or nested set of clusters. Experimental results for fully deceptive functions and nearest neighbor NK-landscape problems with tunable overlap show that the LTGA can solve these hard functions efficiently without knowing the actual position of the linked variables on the problem representation.

References

1. Chen, Y.-P., Lim, M.-H. (eds.): Linkage in Evolutionary Computation. Studies in Computational Intelligence, vol. 157. Springer, Heidelberg (2008)
2. Deb, K., Goldberg, D.E.: Analyzing deception in trap functions. In: Whitley, L.D. (ed.) Proceedings of the Second Workshop on Foundations of Genetic Algorithms, pp. 93–108. Morgan Kaufmann, San Francisco (1993)
3. Duque, T.S., Goldberg, D.E.: A new method for linkage learning in the ECGA. In: Proceedings of the 11th annual conference on Genetic and Evolutionary Computation, pp. 1819–1820. ACM, New York (2009)
4. Goldberg, D.E.: The Design of Innovation: Lessons from and for Competent Genetic Algorithms. Kluwer Academic Publishers, Dordrecht (2002)
5. Harik, G.R., Lobo, F.G., Sastry, K.: Linkage learning via probabilistic modeling in the extended compact genetic algorithm. In: Pelikan [9], pp. 39–61
6. Hauschild, M., Pelikan, M.: Network crossover performance on NK landscapes and deceptive problems. Technical Report MEDAL No. 2010003, Missouri Estimation of Distribution Algorithms Laboratory (January 2010)
7. Kraskov, A., Stögbauer, H., Andrzejak, R.G., Grassberger, P.: Hierarchical clustering using mutual information. EuroPhysics Letters 70(2), 278–284 (2005)
8. Pelikan, M., Sastry, K., Butz, M.V., Goldberg, D.E.: Performance of evolutionary algorithms on random decomposable problems. In: Runarsson, T.P., Beyer, H.-G., Burke, E.K., Merelo-Guervós, J.J., Whitley, L.D., Yao, X. (eds.) PPSN 2006. LNCS, vol. 4193, pp. 788–797. Springer, Heidelberg (2006)
9. Pelikan, M., Sastry, K., Cantú-Paz, E. (eds.): Scalable Optimization via Probabilistic Modeling. Studies in Computational Intelligence, vol. 33. Springer, Heidelberg (2006)
10. Pelikan, M., Sastry, K., Goldberg, D.E., Butz, M.V., Hauschild, M.: Performance of evolutionary algorithms on NK landscapes with nearest neighbor interactions and tunable overlap. In: Raidl, G. (ed.) Proceedings of the 11th annual conference on Genetic and Evolutionary Computation, pp. 851–858. ACM, New York (2009)
11. Stonedahl, F., Rand, W., Wilensky, U.: Crossnet: a framework for crossover with network-based chromosomal representations. In: Proceedings of the 10th annual conference on Genetic and Evolutionary computation, pp. 1057–1064. ACM, New York (2008)
12. Thierens, D., Goldberg, D.E.: Mixing in genetic algorithms. In: Proceedings of the International Conference on Genetic Algorithms (ICGA), pp. 38–47 (1993)
13. Yu, T.-L., Goldberg, D.E.: Conquering hierarchical difficulty by explicit chunking: substructural chromosome compression. In: Cattolico, M. (ed.) Proceedings of the 8th annual conference on Genetic and Evolutionary computation, pp. 1385–1392. ACM, New York (2006)
14. Yu, T.-L., Sastry, K., Goldberg, D.E., Pelikan, M.: Population sizing for entropy-based model building in discrete estimation of distribution algorithms. In: Proceedings of the 9th annual conference on Genetic and Evolutionary Computation, pp. 601–608. ACM, New York (2007)