Benchmark Generator for TD Mk Landscapes

Tobias van Driessel
Utrecht University
Utrecht, Netherlands
tobiasvandriessel@startmail.com

Dirk Thierens
Utrecht University
Utrecht, Netherlands
d.thierens@uu.nl

ABSTRACT

We introduce a publicly available benchmark generator for Tree Decomposition (TD) Mk Landscapes. TD Mk Landscapes were introduced by Whitley et al. to get rid of unnecessary restrictions of Adjacent NK Landscapes while still allowing for the calculation of the global optimum in polynomial time. This makes TD Mk Landscapes more lenient while still being as convenient as Adjacent NK Landscapes. Together, these properties make it very suitable for benchmarking blackbox algorithms. Whitley et al., however, introduced a construction algorithm that only constructs Adjacent NK Landscapes. Recently, Thierens et al. introduced an algorithm, CliqueTreeMk, to construct any TD Mk Landscape and find its optimum. In this work, we introduce CliqueTreeMk in more detail, implement it for public use, and show some results for LT-GOMEA on an example TD Mk Landscape problem. The results show that deceptive trap problems with higher overlap do not necessarily decrease performance and effectiveness for LT-GOMEA.

CCS CONCEPTS

• **Computing methodologies → Heuristic function construction**.

KEYWORDS

Benchmarking, Decomposable Landscapes, Dynamic Programming,

ACM Reference Format:
Tobias van Driessel and Dirk Thierens. 2021. Benchmark Generator for TD Mk Landscapes. In *2021 Genetic and Evolutionary Computation Conference-Companion (GECCO '21 Companion), July 10–14, 2021, Lille, France.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3449726.3463177

1 INTRODUCTION

Suitable benchmark functions are vital to test the effectiveness and performance of evolutionary algorithms. Ideally, these benchmark functions should be completely understood in the sense that we know their structure and, importantly, their global optimum (or optima) so that we can check if a given EA has actually found the best possible solution. A problem with designing benchmark functions is that for many interesting problem classes it is not possible to compute the global optimum efficiently. Not knowing whether an EA has found the best solution limits the practical use of the benchmark and only allows relative comparisons between different algorithms - or different parameter settings of a given algorithm - but it does not allow to evaluate the overall performance and effectiveness. For example, [6] propose an interesting class of benchmark functions, but unfortunately there is no way to efficiently compute the global optimum. Similarly, the well known NK Landscapes does not allow to compute the global optimum. For this reason, EA researchers often use the Adjacent NK Landscapes where the interaction between the variables is limited to adjacent problem variables, allowing the use of dynamic programming to compute the global optimum.

NK Landscapes form a subset of k-bounded pseudo-Boolean optimization problems due to its additional constraints: the number of subfunctions is equal to the number of variables N (= problem size), and every subfunction f_i contains variable x_i and K neighbours, thus setting the subfunction size k to $k = K + 1$. For NK Landscapes, these neighbours are K random variables, and for Adjacent NK Landscapes, these neighbours are the subsequent K variables.

Although (Adjacent) NK Landscapes are popular as a benchmark for optimization algorithms, its constraints are unnecessary for most benchmark purposes, as they turn out not to be important for most fundamental theoretical properties of NK Landscapes [9]. Whitley et al.[10] therefore recently introduced the term *Mk Landscapes* to refer to any k-bounded pseudo-Boolean optimization problem, thus a generalization of NK Landscapes without these constraints. Additionally, they introduced the term *Tree Decomposition Mk Landscapes* to refer to any Mk Landscape with a known and bounded tree-width of k. This is a generalization of Adjacent NK Landscapes, as Adjacent NK Landscapes control tree-width by only considering adjacent variables for the subfunctions, but this constraint can be loosened to allow for any Mk Landscape that still has a bounded tree-width. Ultimately, this bounded tree-width is the key to calculate the global optimum (or optima) in polynomial time.

Conveniently, the overall performance and effectiveness of algorithms can be evaluated due to this polynomial time global optimum calculation. And although the global optimum is known, black box algorithms do not know the problem structure and global optimum, and therefore linkage learning will be necessary for particular codomains to find the global optimum reliably and efficiently. The possibility of evaluating the performance and effectiveness of algorithms, together with the difficulty of Tree Decomposition (TD) Mk Landscapes for particular codomains (for blackbox algorithms), make TD Mk Landscapes well suited as a benchmark function for blackbox Genetic Algorithms. As the global optimum can be calculated efficiently by a dynamic programming algorithm, TD Mk Landscapes are not suitable in the context of graybox algorithms, however, as these do know the problem structure.

GECCO '21 Companion, July 10–14,2021, Lille, France

ACM ISBN 978-1-4503-8351-6/21/07…$15.00
https://doi.org/10.1145/3449726.3463177

In their work, Whitley et al.[10] introduced a construction algorithm to construct TD Mk Landscapes, however, it only constructs TD Mk Landscapes for which the subfunctions form a chain, much like an Adjacent NK Landscape. Recently, Thierens et al.[8] introduced an algorithm, CliqueTreeMk, to construct any TD Mk Landscape and calculate its global optimum (or optima) using dynamic programming when its codomain values are known. In this work, we introduce CliqueTreeMk in more detail, introduce a benchmark generator that implements the algorithm and is available on GitHub, and show indicative results for the Linkage Tree Gene-pool Optimal Mixing Evolutionary Algorithm (LT-GOMEA)[3][1], a linkage learning blackbox evolutionary algorithm. These contributions aim to provide a better understanding of the CliqueTreeMk algorithm, let researchers use our implementation to generate TD Mk Landscapes and benchmark their algorithms, and show that TD Mk Landscapes could be of interest to benchmark blackbox algorithms.

2 TREE DECOMPOSITION MK LANDSCAPES

Whitley et al.[9] recently introduced the term *Mk Landscapes* to refer to any k-bounded pseudo-Boolean optimization problem, a generalization of NK Landscapes without the unnecessary constraints ($M = N, k = K+1$, and variable x_i must appear in subfunction f_i). M is the number of subfunctions and k is a constant that provides an upper bound on the interaction order size of the subfunctions, with M polynomial in N. In a later work, Whitley et al.[10] introduced Tree Decomposition Mk Landscapes, a generalization of Adjacent NK Landscapes: *Tree Decomposition Mk Landscapes* refer to any Mk Landscape with a known and bounded tree-width of k. Tree Decomposition (TD) Mk Landscapes focus on the key property to allow for the global optimum be calculated in polynomial time ($O(N \cdot 2^{2K})$ with Hammer's algorithm[5][4]); a tree decomposition with bounded and known tree-width k must be constructable from (the Variable Interaction Graph of) the Mk Landscape, with $k \in O(\log N)$. TD Mk Landscapes can be expressed by

$$f(x) = \sum_{i=1}^{M} f_i(x, C_i)$$

where $x \in X$, X represents the set of solutions over a bit string with length N, M is the number of subfunctions, f_i is the ith subfunction, and C_i is the ith subset of problem variables that form the input of f_i.

Whitley et al.[10] introduced a construction algorithm to construct TD Mk Landscapes, however, it limits the output to TD Mk Landscapes with a chain-like tree decomposition, similar to the structure of Adjacent NK Landscapes. It is therefore still limited and can not construct all TD Mk Landscapes.

It constructs a $M \times k$ matrix, where the rows correspond with the subfunctions and their variables. The variables must appear in contiguous rows and all N variables must appear in at least one row. If constructed in this way, a tree decomposition can be made with tree-width $k - 1$, where every row of the matrix is represented by a node in the tree.

3 CLIQUE TREE MK

To construct any TD Mk Landscape and calculate its global optimum, Thierens et al.[8] introduced the CliqueTreeMk algorithm. First it constructs a TD Mk Landscape and then uses the structure of the generated landscape to calculate its global optimum (or optima) efficiently.

In the context of this algorithm, we use the term clique tree rather than tree decomposition, as it makes heavily use of the concepts of cliques and separators. The output of Whitley's construction algorithm could then be regarded as a clique *chain* rather than a clique *tree*. We use the term *clique* to represent the set of problem variables in a subfunction, and the term *separator* to represent the set of overlapping problem variables between two cliques/subfunctions, as these terms reflect their properties in a clique tree/tree decomposition in a succinct manner.

The idea behind CliqueTreeMk's construction algorithm is to construct the TD Mk Landscape by directly generating a clique tree with the exact properties as required by the input topology parameters, in order to ensure that a clique tree with the required properties can be constructed, which is required by the definition of TD Mk Landscapes. Its input topology parameters are the number of subfunctions/cliques M, number of variables per subfunction/clique k, number of overlapping bits between subfunctions/cliques o, and branching factor b. The branching factor represents the number of branches in the clique tree. The problem length N can be represented by $N = (M - 1) \cdot (k - o) + k$, as the first clique/subfunction takes k variables, and every other clique/subfunction overlaps o variables with another clique/subfunction and adds $k - o$ unused variables to get to length k.

The general idea of CliqueTreeMk's *construction* algorithm is to first construct clique C_0 as the root of the clique tree by assigning the first k variables from the shuffled variable list, and then generate b children cliques ($C_{j \in \text{children}_i}$) for every clique C_i until we have constructed M cliques. Each child C_j overlaps with its parent C_i for o variables, described by the separator S_j between C_i and C_j, and the remaining $k - o$ variables are taken from the shuffled variable list to complete C_j.

The *global optimum dynamic programming* algorithm then uses this clique tree structure with its cliques and separators to calculate the global optimum. It is comparable to Pelikan's[7] dynamic programming approach in the way it stores the $k - o$ remaining variables's maximizing values for the values of the o overlapping variables (separator variables). Starting at the leaves of the tree, for each separator S_j we store for each of the instances of the separator variables the maximizing variable values for its child clique C_j and the resulting score. Then, we can iterate in the reverse direction and assign values to the clique variables in C_j based on the maximizing values for its variables stored in its parent separator S_j.

We illustrate the CliqueTreeMk algorithm during these phases using an example instance with number of subfunctions/cliques $M = 7$, subfunction/clique size $k = 3$, and overlap $o = 2$. Together, these define length $N = 9$. Furthermore, we choose a branching factor $b = 2$. The construction algorithm uses fixed values for k, o, and b, but the algorithm can be extended to allow for non-fixed values during construction. Likewise for the dynamic programming algorithm. The variables are randomly ordered: $(x_4, x_2, x_7, x_5, x_1, x_9, x_3, x_8, x_6)$.

3.1 Construction

The algorithm is described in a textual version below and a pseudocode version in Algorithm 1.

(1) At the start of the algorithm, take next k variables as clique C_0. Otherwise take next already constructed clique C_i.
(2) Choose o random variables from parent clique C_i, assign to separator S_j
(3) Take next $(k-o)$ unchosen variables and add the variables from S_j to construct child clique C_j
(4) Go to step 2 until b branches have been built
(5) Go to 1 to build the whole tree

Algorithm 1: CliqueTreeMk Construction

Input: M, k, N, b, o, shuffled list of variables
Result: Clique tree
$C_0 \leftarrow$ first k variables;
$count \leftarrow 1$;
for $i \leftarrow 0$ **to** $M-2$ **do**
  for $j \leftarrow 0$ **to** $b-1$ **do**
    $S_{count} \leftarrow o$ random variables from clique C_i;
    $x \leftarrow$ next $(k-o)$ unused variables;
    $C_{count} \leftarrow S_{count} \cup x$;
    $count \leftarrow count + 1$;
    if $count == M$ **then**
      return clique tree;
  end
end

Following the algorithm with the given example instance could result in the following list of cliques: (x_4, x_2, x_7), (x_4, x_7, x_5), (x_4, x_2, x_1), (x_7, x_5, x_9), (x_4, x_7, x_3), (x_4, x_1, x_8), (x_2, x_1, x_6)
In Figure 1 we illustrate the constructed clique tree with its separators.

Essentially, the algorithm creates a clique tree / tree decomposition that adheres to the given constraints, defined by the input topology parameters. Importantly, it adheres to the running intersection property, as problem variables are either part of a single clique C_i or part of multiple cliques that are directly connected by separators. This follows from steps 2 and 3 of the textual version: During construction of a clique C_j, each variable is either taken from the unused problem variables list or copied from the parent clique C_i (and added to the separator S_j), with C_j being a child of C_i. The dynamic programming algorithm that calculates the global optimum requires this running intersection property to select the best value for variables in isolation: for $k-o$ variables at every clique and for o variables at every separator. It is able to calculate the global optimum in polynomial time due to the bounded (and known) tree-width.

3.2 Global Optimum Dynamic Programming Algorithm

To explain the dynamic programming algorithm, we first introduce it in a textual form and then we introduce it in more detail using some formulas.

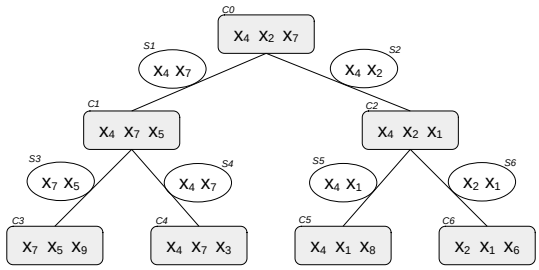

Figure 1: Example clique tree with cliques C0 to C6 and separators S1 to S6.

The CliqueTreeMk global optimum solver follows very similar steps to the dynamic programming algorithm by Pelikan et al.[7]. The CliqueTreeMk global optimum solver traverses the clique tree from the leaves to the root, storing for each instance of separator S_i (o overlapping bits) the maximizing values for the $k-o$ variables in $C_i \setminus S_i$ with its score. The maximizing values for $C_i \setminus S_i$ are stored in K_i and the accompanying score is stored in h_i. Then, for each possible instance of the clique root C_0, the best achievable score g_0 is calculated using its children separators S_j and the stored best achievable score in h_j for that instance of the separator variables. The highest score of these possible instances is the global optimum (or global optima). To assemble the global optimum solution, C_0's maximizing instance is written to the solution and the clique tree is traversed from the root to the leaves, storing the maximizing values for the $k-o$ variables from each K_i into the solution.

If there are multiple global optima, then there are multiple maximizing instances for one or more separators S_i. Each of these maximizing instances for S_i is stored in K_i. When one of these cases of multiple maximizing instances is encountered during the assembly of the global optima, the current global optimum is copied a number of times, according to the number of maximizing instances in K_i (minus one). Finally, each of these copies is assigned one of the maximizing instances and the traversal of the clique tree is continued. Each of these global optima solutions is now considered at every remaining separator in the clique tree.

More specifically, we can define $\forall$ separators S_i:
$h_i(a_1, ..., a_o) = g_i(a_1, ..., a_o, a^*_{o+1}, ..., a^*_k)$ with
$a_1, ..., a_o \in S_i$, $a_{o+1}, ..., a_k \in C_i \setminus S_i$ and $a^*_{o+1}, ..., a^*_k$ maximizing g_i for values $a_1, ..., a_o$.
$K_i(a_1, ..., a_o) = \{a^*_{o+1}, ..., a^*_k\}$
And $\forall$ cliques C_i:
$g_i(a_1, ..., a_k) = f_i(a_1, ..., a_k) + \sum_{j \in \text{children}_i} h_j(b_1, ..., b_o)$

To illustrate these, we can define the previous specifically for our example instance. We define $\forall$ separators S_i:
$h_i(x_a, x_b) = g_i(x_a, x_b, x^*_c)$ with $x_a, x_b \in S_i$ and $x_c \in C_i \setminus S_i$ and x^*_c maximizing g_i for x_a and x_b values.
$K_i(x_a, x_b) = \{x^*_c\}$
And $\forall$ cliques C_i: $g_i(x_p, x_q, x_r) = f_i(x_p, x_q, x_r) + h_{child1}(x_p, x_q) + h_{child2}(x_p, x_r)$

Using the above formulas, we can write a shorter version of the algorithm: For every possible instance of the problem variables in C_0, calculate g_0. Calculating g_0 will recursively calculate all the g_i, h_i, and K_i values for $i > 0$. The maximum of these g_0 values

is the global optimum of the TD Mk Landscape and can be used to retrieve the bit string that achieves this fitness. This is done by acquiring the stored maximizing values for each separator S_i from K_i and assigning their values to the global optimum solution. Or in a more pseudo code way:

(1) For each possible instance of problem variables in C_0, calculate g_0
(2) Maximum g_0 is global optimum
(3) Take next separator, starting with S_1
(4) Take maximizing values from K_i, for problem variable values already in global optimum solution, and put them in global optimum solution
(5) Go to step 3 to assign all problem variable values

We illustrate the algorithm using the example used in the previous subsection. We use the following deceptive trap function for each subfunction:

$$\begin{aligned} f_i(x_a, x_b, x_c) : 111 &=> 4 \\ 000 &=> 2 \\ \text{otherwise} &=> 2 - c(x_a, x_b, x_c) \end{aligned}$$

where c returns the number of ones in the passed variable values.

We show the calculated h_i and K_i values for S_6 and S_1, as $i \in \{3, 4, 5, 6\}$ have the same h_i and K_i values and likewise for $i \in \{1, 2\}$. Then we show the construction of the global optimum using the calculation of g_0 for C_0.

$$C_6 = \{x_2, x_1, x_6\},\ S_6 = \{x_2, x_1\}$$
$$g_6(x_2, x_1, x_6) = f_6(x_2, x_1, x_6)$$

$S_6 = x_2x_1$	00	01	10	11
$h_6(x_2, x_1)$	2	1	1	4
$K_6 = x_6^*$	0	0	0	1

In the above table, we list the possible instances of the separator variables x_2 and x_1, the maximizing values of the remaining variable x_6 in C_6 for these instances (K_6), and the resulting scores for these maximizing values (h_6). Because C_6 is one of the leaves, g_6 is equal to f_6. We can see the deceptive attractor at work here, attracting any instance of the separator variables that does not contain a part of the local optimum.

$$C_1 = \{x_4, x_7, x_5\},\ S_1 = \{x_4, x_7\}$$
$$g_1(x_4, x_7, x_5) = f_1(x_4, x_7, x_5) + h_4(x_4, x_7) + h_3(x_7, x_5)$$

$S_1 = x_4x_7$	00	01	10	11
$h_1(x_4, x_7)$	2 + 2 + 2 = 6	0 + 4 + 1 = 5	0 + 1 + 4 = 5	4 + 4 + 4 = 12
$K_1 = x_5^*$	0	1	1	1

Because C_1 does have children cliques, the calculation of h_1 and thus of g_1 does involve the h_i values of its children, h_4 and h_3.

$$C_0 = \{x_4, x_2, x_7\},\ S_0 = \emptyset$$
$$g_0(x_4, x_2, x_7) = f_0(x_4, x_2, x_7) + h_2(x_4, x_2) + h_1(x_4, x_7)$$

$x_4x_2x_7$	000	...	111
$g_0(x_4, x_2, x_7)$	2 + 6 + 6 = 14		4 + 12 + 12 = 28

Finally, we calculate the g_0 values for all possible instances of the problem variables in C_0. Here we have illustrated just two cases, instances 000 and 111 for $x_4x_2x_7$. Note that this table differs from the two before in the things we calculate; here we don't calculate h_i values, as there is no separator. Instead, we calculate all g_0 values and record the maximum value as the global optimum (or global optima).

For this example, the global optimum value is 28. The maximizing instance for C_0, while considering the rest of the clique tree using dynamic programming, is $x_4^*x_2^*x_7^* = 111$. We can now traverse the clique tree to assign the other bits of the global optimum solution. First, $S_1 = \{x_4, x_7\}$, as is shown in the table for C_6 / S_6, so we insert the values of x_4 and x_7 from our global optimum solution, which are 1 and 1. For instance $x_4x_7 = 11$, $K_1 = x_5^* = 1$, so we assign value 1 to x_5 in our global optimum solution. After doing this for all separators, our global optimum solution is 111111111.

4 EXAMPLE

Our implementation of the CliqueTreeMk algorithm can be found on GitHub[1], here we show some results with our benchmark generator implementation to illustrate its ease of use. Its main functionality is the generation of problems and the calculation of these problems's global optimum, however, it can also generate some input codomain files for the problem generation. The codomain files generation should make it easy to generate a TD Mk Landscape problem from scratch and benchmark an algorithm with it.

4.1 Problem Generation

The problem generator can take as input a configuration folder, a codomain folder, a configuration file, or a codomain file. Here, we highlight how to use the generator with a configuration file and codomain file, and refer the reader to the documentation for the instructions on how to run the generator with multiple configuration files in a folder or multiple codomain files in a folder.

4.1.1 Configuration Input. We create a configuration file to generate deceptive trap problems with topology parameters in a range, in this case we use $M \in \{1, ..., 49\}$, $k = 5$, $o = 1$, $b = 1$:

```
M 1 50
k 5 6
o 1 2
b 1 2
deceptive-trap
```

As options for the codomain we currently offer: *Random*, *Deceptive Trap*, *NKq*, *NKp*, and *Random Deceptive Trap* (a combination of the two). Here we have chosen the deceptive trap function.

Then we use the executable *problem_generator* to generate the codomain files and the problems (25 for each configuration), and find the global optimum for each problem:

```
problem_generator configuration_file -n 25 FILE
    CODOMAIN_OUT PROBLEM_OUT
```

where `CODOMAIN_OUT` and `PROBLEM_OUT` are the (existing) output codomain folder and output problem folder.

[1] https://github.com/tobiasvandriessel/problem-generator

4.1.2 Codomain Input. Instead of generating the codomain and then generate a problem with this generated codomain, one can use an existing codomain file to create a TD Mk Landscape problem. The executable offers the following subcommand for this purpose:

```
problem_generator codomain_file CODOMAIN_FILE
    PROBLEM_FILE_OUT
```

4.1.3 Codomain File Structure. The input codomain files should have the following structure:

```
M K O B
CODOMAIN_VALUE_1
...
CODOMAIN_VALUE_LAST
```

where `M`, `K`, `O`, and `B` represent the to be inserted values of M, k, o and b, and `CODOMAIN_VALUE_1 ... CODOMAIN_VALUE_LAST` represent the $M \cdot 2^k$ decimal codomain values, each on a new line.

4.1.4 Problem File Structure. The output problem files have the following structure:

```
M K O B
GLOB_OPT_VAL
NUM_GLOB_OPT
GLOB_OPT_1
...
GLOB_OPT_LAST
CLIQUE_INDICES_1
...
CLIQUE_INDICES_LAST
```

where `GLOB_OPT_VAL` represents the global optimum (optima) value, `NUM_GLOB_OPT` represents the number of global optima, `GLOB_OPT_1 ... GLOB_OPT_LAST` represent the global optima solutions, and `CLIQUE_INDICES_1 ... CLIQUE_INDICES_LAST` represent the problem variables in each clique.

An example problem generated:

```
2 5 1 1
1.9
2
101000111
010111000
5 3 2 1 7
1 0 6 4 8
```

4.2 Experiment

To show the potential of the TD Mk Landscape benchmark, we conducted a simple experiment: We generated deceptive trap problems with increasing problem size N and overlap o, and ran the Linkage Tree Gene-pool Optimal Mixing Evolutionary Algorithm (LT-GOMEA) on these generated problems to quantify the effect of this increase in o for the difficulty of the problem. LT-GOMEA is a blackbox algorithm, and thus does not have any problem structure information, that tries to learn the linkages between the problem variables to learn the problem structure. LT-GOMEA has shown state-of-the-art performance for discrete, Carthesian-space optimization problems[2], and should therefore show just how difficult and non-trivial TD Mk Landscapes can be. Because we know the global optimum (or optima) of the generated problems, we can evaluate the overall performance and effectiveness of LT-GOMEA.

Configuration input: $M \in \{m \mid N \leq 150\}$, $k = 5$, $o \in \{0, 1, ..., 4\}$, $b = 2$. Where problem size $N = (m - 1) \cdot (k - o) + k$. Note that preliminary experiments indicate that the branching factor b seems to have a big impact on the number of global optima.

The codomain used for the experiment is the deceptive trap function, where we generate for each subfunction a random bit string of length k to be the local optimum and its inverse to be the deceptive attractor. The local optimum has a score of 1.0, the deceptive attractor has a score of 0.9 and any other bit string has score $0.9 - d \cdot \frac{0.9}{k}$, where d is the hamming distance to the local deceptive attractor.

Per configuration instance we generated 25 problems, and for each of these problems, we ran LT-GOMEA 3 times. For the runs where LT-GOMEA manages to find the global optimum, we record the first hitting time. The first hitting time is the number of function evaluations until the global optimum or one of the global optima was found by the algorithm. To record the first hitting times, we need to ignore any unsuccessful LT-GOMEA runs, as these did not find the global optimum. So, for the 3 runs of LT-GOMEA, we filter out any runs that did not find the global optimum and take the median first hitting time for the remaining successful runs. Then we take the median value from the median first hitting times for the 25 generated problems, where again any runs that did not find the global optimum were filtered out. This median value is recorded together with the problem size of the configuration. Besides this first hitting time, we record the effectiveness of LT-GOMEA for every configuration. We measure the effectiveness by counting the number of problems out of 25 (for the current configuration) for which at least 1 LT-GOMEA run found the global optimum, or one of the global optima in case the fitness function has multiple global optima (note that LT-GOMEA is not designed to be a multi-modal EA, so one should not expect it to return all global optima simultaneously). Also note that when we filter out unsuccessful runs in the first hitting time calculation, we still record these unsuccessful runs in the effectiveness for that configuration instance.

We use LT-GOMEA with the population sizing-free scheme as introduced in [1], but we use its discrete cartesian version. We set the No Improvement Stretch (NIS) to $1 + log_{10}(P_i)$, where P_i is the population size of LT-GOMEA instance i. Forced Improvement (FI)[3] is run if the best fitness in a population did not improve for more generations than this NIS. We use premature stopping to stop any LT-GOMEA instance when a LT-GOMEA instance with a bigger population size has a higher average fitness. A LT-GOMEA instance is also stopped when the fitness variance in the population of a LT-GOMEA instance is equal to or smaller than 0.00001. When the global optimum score is found, we stop execution (of all LT-GOMEA instances) and record the current number of evaluations as the first hitting time. Finally, we also stop execution as soon as we hit 300,000 evaluations, with every partial evaluation counting as an evaluation as well.

The results are shown in Figure 2; the upper graph shows the first hitting time for increasing values of the problem size N, with a

line per overlap o setting, and the lower graph shows the effectiveness. When the effectiveness of an overlap configuration decreases below 50%, it is not considered reliable anymore and therefore its performance is not plotted anymore in the upper graph. To emphasize this decision, we have highlighted the 50% effectiveness mark in the lower graph with a horizontal line.

We hypothesised that the problems would become more difficult to solve for LT-GOMEA with increasing overlap o, however, the results paint a different picture. Problems with overlap 1 and 2 do require more evaluations and have a lower effectiveness than problems without any overlap, and are very close in both the required number of evaluations as well as the effectiveness. Interestingly, problems with overlap setting 3 have a lower number of required evaluations and a higher effectiveness than overlap settings 1 and 2, so one could regard problems with overlap 3 as easier. Likewise, problems with an overlap of 4 variables require less evaluations and have a higher effectiveness than problems with overlap setting 3. Finally, and perhaps most surprising of all, the results show that problems with overlap 4 are solved using fewer evaluations than problems with overlap 0, for problem sizes $N \leq 60$. Importantly, however, problems with overlap 0 are always solved, whereas problems with overlap 4 are not.

5 CONCLUSIONS

This paper aimed 1) to provide a better understanding of the CliqueTreeMk algorithm, 2) let researchers use our implementation to generate TD Mk Landscapes and benchmark their algorithms, and 3) show that TD Mk Landscapes could be of interest to benchmark blackbox algorithms.

First, we introduced the CliqueTreeMk benchmark generator for TD Mk Landscapes by Thierens et al.[8] in more detail. We have shown the main difference with the construction algorithm by Whitley et al.[10]: it is able to construct TD Mk Landscapes with a clique *tree* rather than a clique *chain*, due to the branching factor configuration parameter b. With this branching factor, it is able to construct any TD Mk Landscape with fixed k, o, and b. In the example section, we have illustrated the usage of our implementation of CliqueTreeMk, which is publicly available on GitHub and designed to be easy to use for researchers. Finally, we have reported on a simple experiment to show the variation in difficulty one can already achieve with the change of one parameter of the TD Mk Landscape; the number of overlapping bits between subfunctions o. The results show that the performance and effectiveness of LT-GOMEA do not necessarily decrease with increasing overlap o for deceptive trap problems.

By varying the codomain of the landscape, multiple types of problems can be created. TD Mk Landscapes are well suited to serve as benchmark functions for blackbox Genetic Algorithms that are not given the structural problem information as specified by the clique tree. Specifically, for particular codomains - including deceptive functions - linkage learning techniques will be necessary to be able to find the global optima reliably and efficiently. Experimental studies of genetic algorithms greatly benefit from the availability of suitable and well understood benchmark functions.

In the future, one might consider extending the CliqueTreeMk algorithm for variable (but bounded) k, o, and b. If implemented, this would then allow for truly any TD Mk Landscape to be generated and its optimum calculated.

REFERENCES

[1] Peter A.N. Bosman, Ngoc Hoang Luong, and Dirk Thierens. 2016. Expanding from discrete cartesian to permutation Gene-Pool Optimal Mixing Evolutionary Algorithms. *GECCO 2016 - Proceedings of the 2016 Genetic and Evolutionary Computation Conference* (2016), 637–644. https://doi.org/10.1145/2908812.2908917

[2] Peter A.N. Bosman and Dirk Thierens. 2013. More concise and robust linkage learning by filtering and combining linkage hierarchies. *GECCO 2013 - Proceedings of the 2013 Genetic and Evolutionary Computation Conference* (2013), 359–366. https://doi.org/10.1145/2463372.2463420

[3] Peter A N Bosman and Dirk Thierens. 2012. Linkage Neighbors , Optimal Mixing and Forced Improvements in Genetic Algorithms Categories and Subject Descriptors. *Gecco 2012* x (2012), 585–592.

[4] Yves Crama, Pierre Hansen, and Brigitte Jaumard. 1990. The basic algorithm for pseudo-Boolean programming revisited. *Discrete Applied Mathematics* 29, 2-3 (1990), 171–185. https://doi.org/10.1016/0166-218X(90)90142-Y

[5] Peter L Hammer, Ivo Rosenberg, and Sergiu Rudeanu. 1963. Application of discrete linear programming to the minimization of Boolean functions. *Rev. Mat. Pures Appl* 8 (1963), 459–475.

[6] Kei Ohnishi, Shota Ikeda, and Tian-Li Yu. 2020. A Test Problem with Difficulty in Decomposing into Sub-Problems for Model-Based Genetic Algorithms. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion* (Cancún, Mexico) *(GECCO '20)*. Association for Computing Machinery, New York, NY, USA, 221–222. https://doi.org/10.1145/3377929.3389993

[7] Martin Pelikan, Kumara Sastry, David E. Goldberg, Martin V. Butz, and Mark Hauschild. 2009. Performance of evolutionary algorithms on NK landscapes with nearest neighbor interactions and tunable overlap. *Proceedings of the 11th Annual Genetic and Evolutionary Computation Conference, GECCO-2009* (2009), 851–858. https://doi.org/10.1145/1569901.1570018

[8] Dirk Thierens and Tobias van Driessel. 2021. A Benchmark Generator of Tree Decomposition Mk Landscapes. In *Proceedings of the Genetic and Evolutionary Computation Conference 2021 (GECCO '21)*. Association for Computing Machinery, p–p+1.

[9] Darrell Whitley. 2015. Mk landscapes, NK landscapes, MAX-kSAT: A proof that the only challenging problems are deceptive. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. 927–934.

[10] Darrell L Whitley, Francisco Chicano, and Brian W Goldman. 2016. Gray box optimization for Mk landscapes (NK landscapes and MAX-kSAT). *Evolutionary computation* 24, 3 (2016), 491–519.

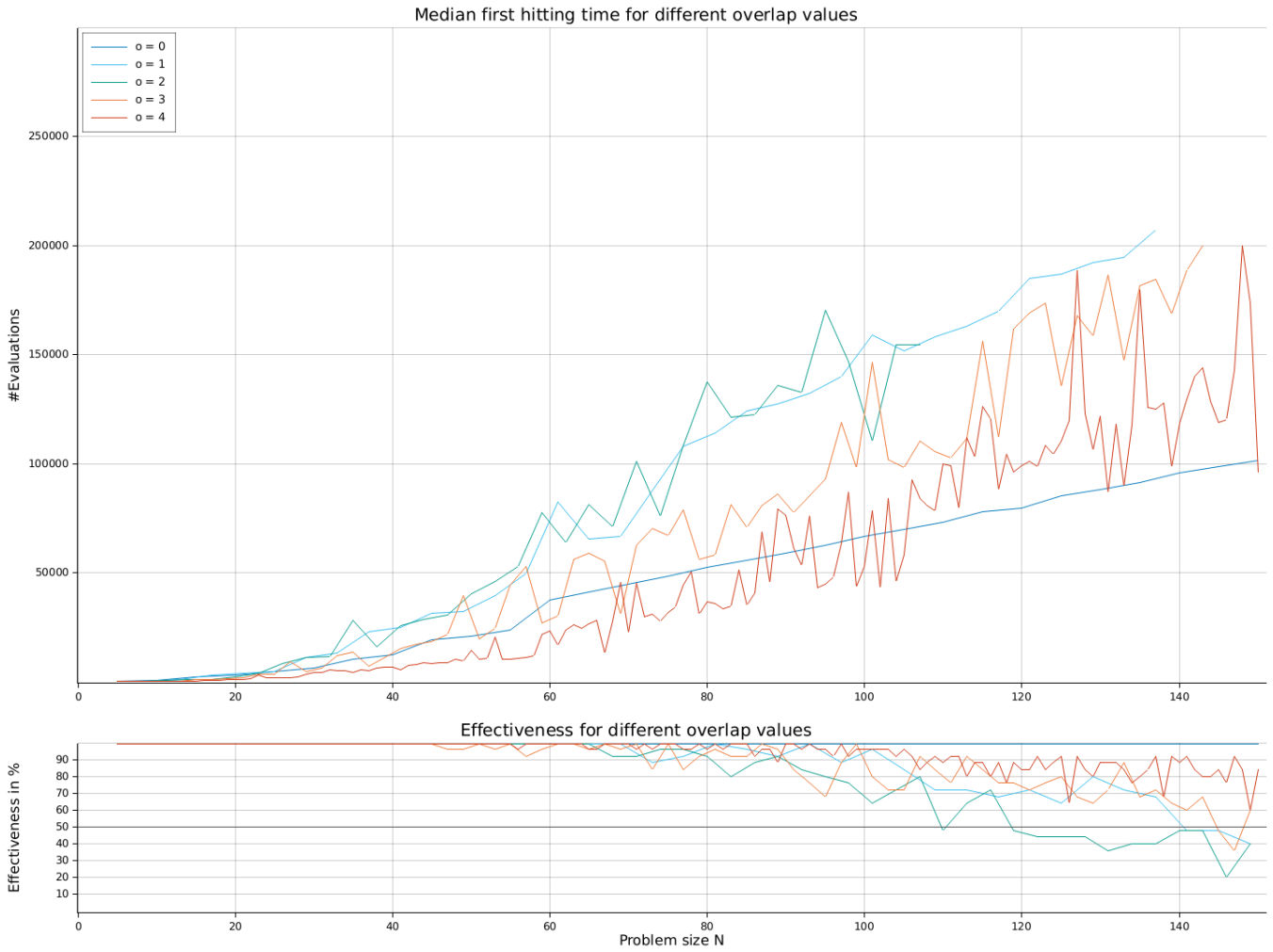

Figure 2: Performance and effectiveness of LT-GOMEA for different overlap values