

Improving Time and Memory Efficiency of Genetic Algorithms by Storing Populations as Minimum Spanning Trees of Patches

Maxim Buzdalov

Aberystwyth University, Aberystwyth, United Kingdom

June 30, 2023

Abstract

In many applications of evolutionary algorithms the computational cost of applying operators and storing populations is comparable to the cost of fitness evaluation. Furthermore, by knowing what exactly has changed in an individual by an operator, it is possible to recompute fitness value much more efficiently than from scratch. The associated time and memory improvements have been available for simple evolutionary algorithms, few specific genetic algorithms and in the context of gray-box optimization, but not for all algorithms, and the main reason is that it is difficult to achieve in algorithms using large arbitrarily structured populations.

This paper makes a first step towards improving this situation. We show that storing the population as a minimum spanning tree, where vertices correspond to individuals but only contain meta-information about them, and edges store structural differences, or *patches*, between the individuals, is a viable alternative to the straightforward implementation. Our experiments suggest that significant, even asymptotic, improvements — including execution of crossover operators! — can be achieved in terms of both memory usage and computational costs.

This is a slightly revised and extended author's version of the GECCO'23 paper accepted to the EvoSoft workshop. That paper is available at DOI 10.1145/3583133.3596388.

1 Introduction

In evolutionary computation, the standard ways to measure the performance of an evolutionary algorithm are either to count the number of fitness function evaluations until a solution of a certain quality is found (the fixed-target perspective), or to measure the quality of the best found solution once the given number of evaluations is used (the fixed-budget perspective). Both implicitly assume that fitness evaluation dominates the computational costs of running an evolutionary algorithm.

While this assumption often comes true, the opposite also happens quite often. For example, many multiobjective algorithms, such as NSGA-II [18], and advanced continuous evolutionary algorithms, such as CMA-ES [26], have internal state update routines that dominate the computational costs asymptotically, so for large enough population size fitness evaluation will become dominated by other parts of these algorithms. However, there are also important cases where computational costs need to be considered accurately even for simple algorithms.

Auxiliary computational costs become especially noticeable if fitness evaluation takes time proportional to the individual size and hence is cheap. This often happens in benchmarking of evolutionary algorithms on problems with easy definitions: in discrete optimization, examples are ONEMAX (the number of bits set to 1), LEADINGONES (the length of the maximum prefix that has all bits set to 1), and many continuous functions, starting from SPHERE (the sum of squares for each variable), are also computationally cheap. This holds true for many practical fitness functions, such as those used for satisfiability problems [31, 48], knapsack problems [14, 49] and many other integer linear programming problems [45]. In this case, allocation of space for a new individual, as well as naïve implementation of mutation operators, require asymptotically the same number of operations as fitness evaluation, and, depending on the hidden constants, they may even take more time.

Significant decrease in computational costs is possible whenever the fitness function can be incrementally recomputed when the exact positions of changes are known. This is known as *partial evaluation* or *incremental evaluation*. One prominent example is gray-box optimization [48], which benefits from incremental fitness evaluation especially with large-scale problem sizes [17]. Improvements can often be applied to mutation operators [11, 12], but fast specialized crossovers are also possible [43]. Even outside of gray-box optimization, many fitness functions, including the ones listed in the previous paragraph, still enjoy large asymptotic runtime improvements. To benefit from this, one should be really careful with operations happening inside evolutionary algorithms, as even allocation of space for a new individual

becomes prohibitively time- and memory-consuming.

As a result, efficient implementations are known for mutation-only algorithms, few genetic algorithms with very specific crossover operators, and in the gray-box optimization contexts. In the same time, most genetic algorithms having non-trivial populations and employing crossover operators so far had no other choice than implementing most operators straightforwardly, lacking the ability to save computational resources and in particular enjoy incremental fitness evaluation. Most of this comes from the fact that there is no way to perform any kind of crossover on individuals, that could have diverged long time ago, faster than by full evaluation in an absence of an appropriate data structure. In this work, we show that it is, in fact, possible.

We propose to store the population as a graph where each individual corresponds to a vertex, and edges store the differences, or *patches*, between the individuals they connect. At any time, there is only one individual that is stored as a whole, whereas for all other individuals vertices store only the meta-information, such as the fitness value. To simplify the data structure, and also to save as much space as possible, we use as the graph the minimum spanning tree, such that the total size of all the patches is minimum possible to preserve connectivity. Should an individual be used to produce an offspring, we *promote* the only complete individual to the corresponding vertex by applying all the patches on the way that leads to that vertex. Similarly, we may compute the difference between individuals by *combining* the patches on the way that connect their vertices, which typically takes way less time than by scanning the entire individuals. Then we can use this difference to apply the crossover operator to them, which is, again, typically faster than doing it in a naïve way.

As we only begin our explorations of this topic, we consider only bit strings as the search space, and limit ourselves with a few simple algorithms, including, however, an algorithm with a non-trivial population that is able to maintain some diversity in it. We consider two problems: the benchmark problem ONEMAX and the knapsack problem, which both benefit from incremental fitness evaluation. Our experiments show a remarkable improvement in both time and memory over the naïve implementation: in most favorable conditions, the cost of one fitness evaluation becomes $O(1)$ versus $O(n)$ for problem size n , but even in the presence of population diversity, the average cost still appears to grow sublinearly.

Structure of the paper. Section 2 introduces the employed definitions, algorithms and problems, and also covers the related work on adjacent topics. Section 3 explains the proposed approach, the minimum spanning tree of patches. Section 4 presents the results of experiments and their discussion. Finally, Section 5 concludes and indicates numerous directions for future

work.

Availability of code and data. To adhere to reproducibility standards discussed in [33], our code is available on GitHub¹.

2 Preliminaries and Related Work

In this work, we consider only algorithms working on bit strings, and problems defined on bit strings. Though most of our ideas are applicable to more general classes of search spaces, and so are most of considered evolutionary algorithms, we define them on bit strings for brevity.

We denote as n the length of the bit string. The Greek letter μ typically defines the parent population size, whereas λ typically defines the number of offspring in one iteration. The term “u.a.r.” stands for “uniformly at random”. All optimizers are defined to run infinitely, whereas termination conditions are considered external to optimizer definitions.

Simple problems. We consider a benchmark problem called ONEMAX, whose (simplified) definition is as follows:

$$\text{ONEMAX} : \{0, 1\}^n \rightarrow \mathbb{R}, x \mapsto |\{i \in [1..n] \mid x_i = 1\}|.$$

This problem obviously benefits from incremental fitness evaluation: if one knows the old fitness value, the new one can be determined by examining the changed positions only.

We also consider the *knapsack problem* [37], one of the famous NP-hard problems. Given n items, the i -th of them has the weight w_i and the value v_i , and the knapsack capacity W , one has to find a selection of these items with the maximum total value, such that their total weight does not exceed the capacity. Denoting $\bar{w} = (w_i)_{i=1}^n$, $\bar{v} = (v_i)_{i=1}^n$, for the purpose of this paper we define the genotype-to-phenotype mapping as follows:

$$K_{\bar{w}, \bar{v}, W}^P : \{0, 1\}^n \rightarrow (\mathbb{R}, \mathbb{R}), \quad x \mapsto \left(\sum_{i=1}^n x_i w_i, \sum_{i=1}^n x_i v_i \right),$$

and the phenotype-to-fitness mapping as follows:

$$K_{\bar{w}, \bar{v}, W}^F : (\mathbb{R}, \mathbb{R}) \rightarrow \mathbb{R}, \quad (W_x, V_x) \mapsto \begin{cases} V_x, & \text{if } W_x \leq W; \\ -W_x, & \text{otherwise.} \end{cases}$$

In other words, the bit string of length n defines which items are selected: bit value 1 in the i -th position selects the i -th item. The fitness function

¹<https://github.com/mbuzdalov/patch-based-ga>. This version of the paper is accompanied by the release v0.2.

Algorithm 1: Randomized local search

Input: problem size n , maximized function $f : \{0, 1\}^n \rightarrow \mathbb{R}$
1 **Initialization:** Sample $x \in \{0, 1\}^n$ u.a.r. and evaluate $f(x)$;
2 **Optimization:** **for** $t = 1, 2, 3, \dots$ **do**
3 Let y be a copy of x with one random bit flipped;
4 Evaluate $f(y)$;
5 **if** $f(y) \geq f(x)$ **then** $y \leftarrow x$;

favors increasing the total value as long as all items fit the knapsack, and decreasing the total weight if they do not.

This two-level mapping is essential to incremental fitness evaluation, as the phenotypic pair consisting of sums of weights and values can be easily recomputed on small changes, and the fitness value can follow suit, whereas with the more direct approach one would have to recompute the fitness from scratch. For simplicity, we consider the pair of sums as the fitness value, redefine the comparison operation, and avoid the explicit concept of a phenotype.

Randomized local search. Perhaps the simplest algorithm to benefit both from efficient implementation of operators and from incremental fitness evaluation is Randomized local search, or RLS, whose pseudocode is presented in Algorithm 1.

Obviously, instead of creating the entire individual y from scratch, one can just flip the i -th bit in the parent, where i is chosen u.a.r. randomly, then evaluate the changed individual, and if it is worse, then “unflip” that bit back. This way, the internal work for a single fitness evaluation, other than the first one, is limited to $O(1)$ operations.

If incremental fitness evaluation is possible, we can definitely apply it. For instance, for the knapsack problem the sum of weights is either increased or decreased by the weight of the i -th item, depending on whether it was selected or not. In this case, the total work for the single fitness evaluation needs only $O(1)$ operations.

Standard bit mutation and the $(1 + 1)$ evolutionary algorithm. For a minimalist evolutionary algorithm capable of global search, the $(1 + 1)$ evolutionary algorithm, or the $(1 + 1)$ EA, is often considered (Algorithm 2). It is very similar to RLS, except that for the mutation operator it uses *standard bit mutation* which, with probability p , flips each bit independently.

The general recommendation for a new problem, unless more knowledge is obtained, is to use $p = 1/n$ such that $np = 1$ bit is flipped on average. The number of flipped bits follows the well-known binomial distribution with

Algorithm 2: The (1 + 1) evolutionary algorithm

Input: problem size n , mutation rate p ,
maximized function $f : \{0, 1\}^n \rightarrow \mathbb{R}$

- 1 **Initialization:** Sample $x \in \{0, 1\}^n$ u.a.r. and evaluate $f(x)$;
- 2 **Optimization:** for $t = 1, 2, 3, \dots$ do
- 3 Let y be a copy of x with each bit flipped independently of others
 with probability p ;
- 4 Evaluate $f(y)$;
- 5 **if** $f(y) \geq f(x)$ **then** $y \leftarrow x$;

parameters n and p , which is reasonably well concentrated around the value np . Under these conditions it is obvious that the (1 + 1) EA also benefits a lot from incremental fitness evaluation.

But is the mutation operator capable of delivering the matching performance? Most people, including even many developers of well-known and widely-used software systems for evolutionary computation, implement sampling from binomial distributions using the straightforward naïve scheme, such as the `BitFlipMutation`² class in jMetal [35] or the `mutFlipBit`³ function in DEAP [24]. However, the properties of a binomial distribution allow sampling schemes of complexity $O(np)$, which matches the number of changes made to the individual. This fact has been known in the evolutionary computation domain for a long time, see [3, p. 238, Eq. 32.2] and, apparently independently, [29]. In fact, with some preprocessing, such sampling can be performed faster [25, 28, 30, 34, 46, 47], but since the number of changes to an individual is still $\Theta(np)$, in this domain we do not benefit much from faster sampling anyway.

As a result, it is possible to run the (1 + 1) EA with as few as $O(1)$ operations per fitness evaluation, assuming the fitness function cooperates, but it requires some care from software developers.

The (1 + (λ , λ)) genetic algorithm. Proposed a decade ago by the theoretic community, this genetic algorithm has some unique properties with regards to certain function classes [19].

Many of these properties come not just from using crossover, but from the very specific kind of crossover. It takes a parent individual x and an individual y that has been generated by flipping ℓ bits chosen u.a.r. in x . Then it generates an offspring by taking each bit from y with probability p_c and from x otherwise, treating each bit independently. In the default

²Last revision at the submission time: `BitFlipMutation.java`

³Last revision at the submission time: `mutation.py`

configuration, ℓ and p_c are tied in such a way that $\ell \cdot p_c = \Theta(1)$ with large probability.

This particular crossover, as well as the overall structure of the algorithm, allows implementing it efficiently much in the same manner as the algorithms above. As the number of bits ℓ is usually much smaller than the problem size n , one can not just generate y , but also store the list of bits that makes it different from the parent x . Then, crossover can be executed by just sampling $\ell \cdot p_c$ bits from that list and flipping them in x . Although not always explicitly mentioned in the corresponding papers, this was the mechanism to enable experiments with problem sizes n of order up to 10^6 to 10^7 in realistic times (few minutes per run) not just on problems like ONEMAX, but on MAX-SAT problems as well [8], sometimes even allowing sampling rather large values of ℓ from heavy-tailed distributions [1].

Such crossover can be interpreted as mutation subsampling. While not making large difference on bit strings, this consideration actually enabled porting this algorithm to other search spaces, such as permutations [4], with similar performance improvements compared to naïve implementations. However, the techniques employed to achieve these improvements cannot be readily used with other algorithms.

General form of “good” mutation and crossover operators. Evolutionary algorithms capable of optimizing arbitrary problems over certain search spaces shall possess certain invariance properties in order to avoid preferring some (otherwise equivalent) problem instances over other. For example, in continuous optimization, the obvious requirements are translation invariance and scaling invariance, whereas many algorithms, such as CMA-ES, also have rotational invariance (and hence invariance over general affine transformations). Having such properties generally improves reliability of black-box optimization algorithms [27], whereas algorithms pretending to be black-box but lacking many of these properties are often used in fraudulent publications [2].

For the search space consisting of bit strings of a fixed length, there are two such properties: invariance over flipping a bit (which is a rough equivalent of translation invariance) and invariance over shuffling positions of bits. Algorithms that satisfy these properties are called *unbiased black-box optimization algorithms* [32], a similar concept exists for arbitrary discrete search spaces [39]. This concept can be extended to variation operators, such as mutation and crossover operators. For mutation operators over bit strings, any unbiased mutation operator can be represented as follows [32]:

- sample a number $\ell \in [0..n]$;
- when applied to an individual x , copy it and flip in the copy ℓ bits

without replacement at positions sampled u.a.r.

For instance, the standard bit mutation with mutation rate p samples ℓ from a binomial distribution with parameters n and p . Similarly, any crossover operator can be represented as follows [7, 10, 21, 22]:

- sample a function f from $d \in [0..n]$ to a pair of numbers $\ell_d \in [0..d], \ell_s \in [0..n - d]$;
- when applied to two individuals x_1 and x_2 , compute the number of differing bits d and then compute ℓ_d and ℓ_s using the function f ;
- copy x_1 and flip in the copy ℓ_d bits at positions where x_1 and x_2 differ, and also flip ℓ_s bits at positions where x_1 and x_2 are same, both without replacement and sampled u.a.r.

Similar definitions exist for operators of higher arities, as well as for operators returning multiple offspring [7]. This view enables a unified framework and a single-entry implementation for support of arbitrary operators of a given arity.

The $(\mu + 1)$ genetic algorithm. As an example of a genetic algorithm with a non-trivial population that cannot be easily adapted to the use of high-performance variation operators we consider the $(\mu + 1)$ genetic algorithm, presented in Algorithm 3. This is essentially a steady-state algorithm, where on each iteration a new individual is generated either by only mutation, or by crossover followed by mutation, and then the worst individual is removed from the population.

Note that uniform crossover, followed by standard bit mutation, can be expressed in the above-mentioned framework as follows: if the distance between the parents is d , then sample the number of differing bits to flip ℓ_d from the binomial distribution $\text{Bin}(d, 1/2)$, and the number of same bits to flip ℓ_s from the binomial distribution $\text{Bin}(n - d, p_m)$.

Despite its simplicity, the $(\mu + 1)$ GA can be shown to be provably better than mutation-only algorithms even on simple problems [9], furthermore proofs exist that increasing the population size improves the performance [15]. With only a simple additional diversity-preserving mechanism it becomes capable of efficiently jumping over large valleys [16], and even in its original form it is capable of preserving enough diversity to do essentially the same [20].

Potential diversity of the population makes it a problem to perform crossovers efficiently, as in the absence of any extra information the distance between the individuals is essentially unknown and can change a lot

Algorithm 3: The $(\mu + 1)$ genetic algorithm

Input: problem size n , mutation rate p_m , crossover probability p_c , maximized function $f : \{0, 1\}^n \rightarrow \mathbb{R}$

- 1 **Initialization:** Let the population $X = \emptyset$;
- 2 **for** $i = 1, 2, \dots, \mu$ **do**
- 3 Sample $x_i \in \{0, 1\}^n$ u.a.r. and evaluate $f(x_i)$;
- 4 Add x_i to the population: $X \leftarrow X \cup \{x_i\}$;
- 5 **Optimization:** **for** $t = 1, 2, 3, \dots$ **do**
- 6 Sample random number c ;
- 7 **if** $c < p_c$ **then**
- 8 Sample x from X u.a.r., let y be a copy of x with each bit flipped independently of others with probability p_m ;
- 9 **else**
- 10 Sample x_1, x_2 from X u.a.r. with replacement;
- 11 Apply uniform crossover to x_1 and x_2 , obtain x' : on each position, x' takes a value from either x_1 or x_2 with probability $1/2$;
- 12 Let y be a copy of x' with each bit flipped independently of others with probability p_m ;
- 13 Evaluate $f(y)$;
- 14 Add y to the population: $X \leftarrow X \cup \{y\}$;
- 15 Remove the individual with the smallest fitness from X , breaking ties u.a.r.;

over the run. To the best of our knowledge, this paper is the first to present a mechanism that is capable of performing crossovers in sublinear time in practice.

Gray-box optimization. This is the prominent area of research where, based on a deeper understanding of the problem, efficient operators are possible to design with significant improvement in computational time, search efficiency, or both. A large number of works essentially belonging to gray-box optimization implement mutation operators by essentially synthesizing the difference vectors and storing only a few complete individuals [5,6,17]. These works either use only mutation operators or invoke crossover operators only infrequently to amortize their cost.

With more understanding of the structure of the problem, fast deterministic mutation operators and linear-time deterministic recombination that synthesizes the best out of 2^q individuals from two individuals whose

difference is partitioned into q independent regions, is possible [11, 43, 44, 48], which leads to state-of-the-art solutions in the particular problem domains. However, these advanced techniques cannot be used when the problem admits incremental fitness evaluation, but the degree of linkage between variables is too high, of which the knapsack problem and its variations are good examples.

Individuals as immutable balanced trees. An approach that targets essentially the same problem, but in a significantly different way, was recently proposed in [38]. For a class of problems where it is not only possible to perform incremental fitness evaluation, but also to store the partial evaluation results for parts of individuals, optimization can benefit by representing individuals as immutable balanced trees that store evaluation results for subtrees. Note that both ONEMAX and the knapsack problem, which we consider in this paper, belong to this class, but, for instance, most satisfiability problems do not.

In the case of bit strings, a mutation of $O(1)$ bits corresponds to $O(\log n)$ time and memory to construct a new individual and compute its fitness. What is more, the single-point crossover (and any similar crossover with a constant number of exchange points) can also be performed in time and memory of $O(\log n)$. Note that such crossover operators are not unbiased, and direct implementation of, say, uniform crossover using this approach does not result in any performance improvements. However, further development of this approach, such as aggressive subtree caching, can probably make it more similar to our approach in many aspects.

3 Minimum Spanning Tree of Patches

Our data structure to work with the population consists of two parts: the persistent lightweight graph and the mutable memory-heavy part of size $\Theta(n)$.

The graph. We maintain the entire population as a graph that describes the structure of the population.

- Vertices of this graph represent individuals. Only a limited amount of information about an individual is permanently stored in the vertex. Currently, this information consists of the fitness value of the individual, and a Boolean value which indicates whether an individual is still in the population (that is, it has not been discarded).
- Edges represent differences between individuals, or *patches*. An edge from vertex A to vertex B contains an immutable piece of information that tells how to convert individual A to individual B . In the general

case, the graph is directed, and an edge from B to A shall produce an opposite effect to an edge from A to B . However, for bit strings, these effects coincide: the edges contain a list of bit indices to flip.

This graph is maintained in such a way that it is a minimum spanning tree, where the weight of an edge is the size of the patch (that is, the number of indices to flip). As even in the general case it is reasonable to expect that the size of the patch is equal to the size of its negation, for the purpose of building the tree we can safely use the characteristic size of a patch for a weight.

The memory-heavy part. Apart from this graph, there are two more values which use $\Theta(n)$ memory. The first of these values is what we call the *complete individual*, which is a bit string of size n containing one of the individuals currently in the population. We also maintain a pointer to the vertex that corresponds to this individual. The second value is what we call the *mutable patch*: as opposed to immutable patches stored at the edges, the mutable patch has the size of $\Theta(n)$ words. Its purpose is to store a set of (zero-based) indices, which is a subset of $[0..n - 1]$, with the following supported operations:

1. Clear the set.
2. Add or remove a given element.
3. Test whether a given element is contained in the set.
4. Report the number of elements currently in the set.
5. Add a random element which is not yet in the set.
6. Add a given number of random elements not yet in the set while *simultaneously* removing another given number of random elements which are already in the set.
7. Create an immutable set containing the same elements.

The mutable patch can be implemented with some minimal care atop of one permutation π of the set $[0..n - 1]$, its inverse, and the size of the patch s , such that elements $\{\pi(0), \dots, \pi(s - 1)\}$ are the elements constituting the patch. For further details, please refer to the class `MutableIntSet` in the repository.

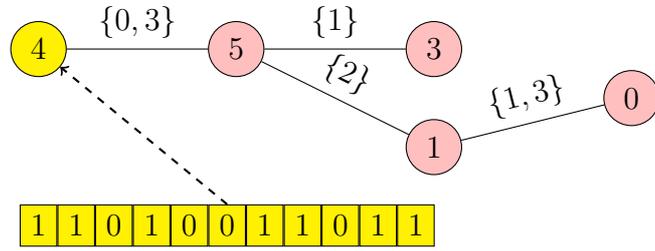
Using operations (2)–(4), we can combine in-place the mutable patch and one of the immutable patches. This allows, in particular, to traverse the entire graph using depth-first search (in time proportional to the sum of sizes

of all patches) and to find out which vertex results in the smallest cardinality of the set. When adding a new individual to the population, this is used to find the existing individual with which this new individual will be connected by an edge to ensure that the graph is the minimum spanning tree. Similarly, by starting a depth-first search from the first parent and terminating it when the second parent is encountered allows to compute the difference between the parents, which will be used later in the crossover operator.

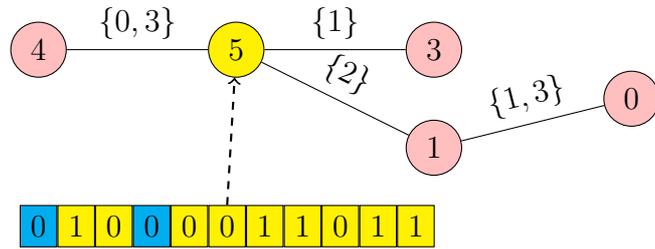
Operation (5) is used to implement mutation: given the number of bits to flip, we first clear the set, then add the necessary number of not yet existing elements. Similarly, operation (6) is used to implement crossover: assuming the set contains the bit indices in which the parents differ, we “flip” the given number of randomly chosen bits at positions where the parents differ and where they are the same, and after that the set will contain exactly those bit indices which have to be flipped in the first parent. For both operators, operation (7) is used to create a new immutable patch from the result of applying the operator, that will then be used either to generate a new individual or to create a permanent link from that individual to the existing population.

The operations. The following operations are supported.

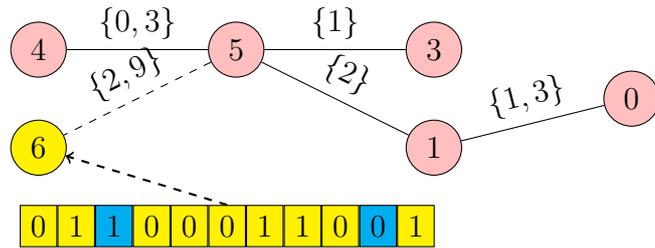
- Create a random individual. The first such call creates the memory-heavy part, initializes the *complete individual* randomly, and creates the first vertex in the graph. All subsequent calls effectively perform mutation with $\ell \sim \text{Bin}(n, 1/2)$, such that the offspring is statistically identical to a randomly generated individual.
- Mutate ℓ bits in the given individual x . First, the *complete individual* is promoted to match the individual x , which is performed by the depth-first search from x until the vertex matching the complete individual is found, then by returning back and applying patches from the reverse edges. Then, the mutable patch is initialized by adding ℓ random elements that are not yet added. Next, starting from this state of the mutable patch, the entire graph is traversed completely in order to update it to reflect the new minimum spanning tree. When an edge is traversed, the patch written on that edge is prepended to the mutable patch, and the edge length between the current vertex and the new individual is the size of the mutable patch. The graph is rebuilt to become the new minimum spanning tree by an adaptation of an algorithm from [13], which works in linear time. When this algorithm ends, the shortest edge from an existing individual to the newly created individual is used to compute the fitness of the latter incrementally. An example application of the mutation operator is given in Fig. 1.



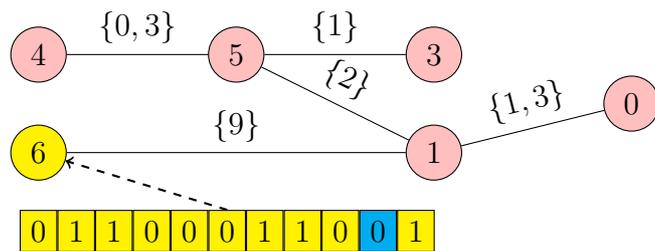
(a) Initial state



(b) The complete individual is promoted to vertex with fitness 5



(c) A new offspring is created



(d) A patch of the minimum size is found for the new offspring

Figure 1: Minimum spanning tree of patches. Vertices contain fitness values, edges are labeled with patches: the sets of bit indices to flip. An example of performing a mutation operator on an individual with fitness 5 is shown

- Apply crossover to the given parents x_1 and x_2 using a function f to obtain the numbers of bits to flip in the “same” and “differing” bit groups. First, the *complete individual* is promoted to match x_1 , and then the mutable patch is set to the difference between x_1 and x_2 using depth-first search started from x_1 until x_2 is encountered. Then, the distance between the parents is obtained from the mutable patch, the function f is called, and the simultaneous flip is called on the mutable patch. Finally, the procedure similar to the one in the mutation operator is followed to create the new vertex and connect it to the rest of the graph.
- Discard an individual. The individual is marked as discarded, and if it has degree 1 in the graph, it is removed from the graph. Then, if the former neighbor has also been marked as discarded, the same procedure is called recursively. In the operations above, when computing the shortest distance and finding an individual with that distance, only vertices not marked as discarded are used.

All these operations traverse the entire graph at least once (for discarding an individual, this is only the worst-case bound), which takes time proportional to the sum of sizes of all the patches in the graph (each individual creation also adds $\Theta(n)$ time and memory). This total size is small for all the algorithms which are known to benefit from working with patches and from incremental fitness evaluation, and can potentially be large for large diverse populations. However, when an optimizer has worked for some time, even if it maintains diversity, it may happen that a large portion of bit indices has converged to the same values for the entire population, just because all other values are strictly worse. The proposed data structure will automatically benefit from such conditions, as the largest patch sizes, and hence the average distances between the individuals, will get smaller over time.

To measure the potential speed-ups from using the minimal spanning tree of patches, we conducted a series of experiments, which we report in the next section.

4 Experiments

In our experiments, we use the algorithms mentioned in Section 2 with the following notation used:

- RLS: Randomized local search;

- (1 + 1): the (1 + 1) evolutionary algorithm using standard bit mutation with flip probability $1/n$;
- (2 + 1): the ($\mu + 1$) genetic algorithm with $\mu = 2$ using standard bit mutation with flip probability $1.2/n$ as suggested in [36] and crossover probability 0.9;
- (10 + 1): the ($\mu + 1$) genetic algorithm with $\mu = 10$ using standard bit mutation with flip probability $1.4/n$ as suggested in [15] and crossover probability 0.9.

For the ($\mu + 1$) genetic algorithm, we use different population sizes to see how they affect the performance of the proposed data structure. All these algorithms are evaluated using both the naïve approach of handling populations and the proposed data structure. In all cases, mutation operators using binomial distributions are implemented using efficient approaches.

As the project is implemented in Scala 3, which uses a Java virtual machine, to measure the running times we need to warm up the virtual machine, which we do by observing the reported running times over periods of one second (or more if the optimizer runs longer) and normalizing them over the number of fitness evaluations. The obtained values, with a semantic of average operation time, are remarkably stable, so we report their means and standard deviations over 10 such one-second attempts.

To perform the experiments, we use a dedicated Linux server with kernel 6.1.19, OpenJDK Java virtual machine version 11.0.18, Scala version 3.2.2, running on an Intel Core i7-8700 CPU clocked at 3.2GHz.

4.1 OneMax

The first series of our experiments uses the ONEMAX problem with different values for the problem size n to see how the proposed data structure works in favorable conditions. We consider problem sizes $n = 2^k$ for $k \in [5..13]$. We run each algorithm until the optimum is found, then report the operation times as specified above. The results are presented in Fig. 2.

This figure shows that the time of a single operation scales linearly with the problem size in all algorithms using the naïve population handling approach, but in all algorithms using the proposed data structure it stays constant as the problem size grows. It can be seen, however, that as the population size μ grows in the ($\mu + 1$) genetic algorithm, the cost of an operation seems to scale linearly with the population size. This is expected, as each operation traverses the entire population to determine the nearest neighbor. The slow increase of the running times as the problem size reaches

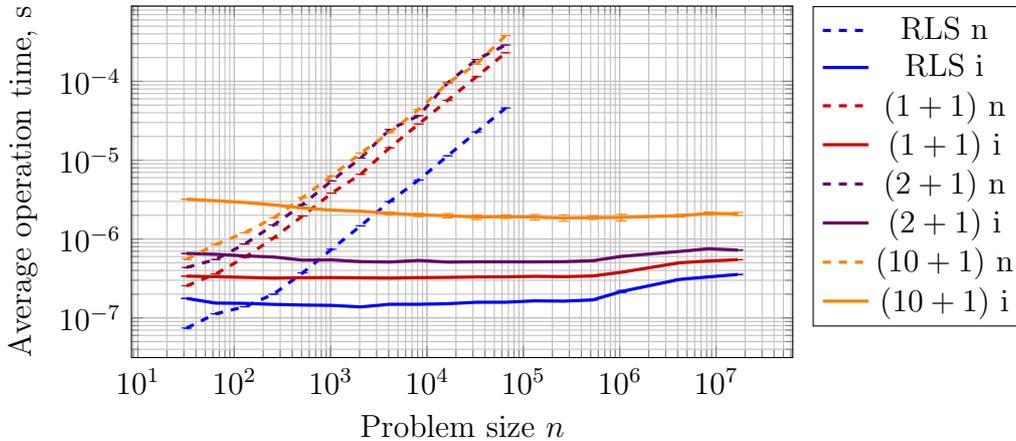


Figure 2: Operation times for ONEMAX. The last letter of the algorithm’s name means “n” for “naïve” and “i” for “incremental”. Means and standard deviations are shown, the latter almost always invisible.

10^6 can be explained by the data structure no longer fitting the L2 cache of the processor, which results in performance degradation by a factor of up to three.

4.2 Knapsack Problem, Fixed Budget

A similar experiment was conducted for the knapsack problem, where for each problem size $n = 2^k$ for $k \in [5..13]$ a random uncorrelated instance was generated with weights and values in the range $[10000..20000]$ and the capacity equal to half the sum of weights. It is known that such instances are generally easy [37]. However, we did not aim at obtaining the optimum, instead, we set a computational budget of 25000 fitness evaluations. Fig. 3 shows the plots of the operation times for the considered algorithms.

In this experiment, the operation times of all the algorithms using the naïve population handling approach still scale linearly with the problem size, which is not surprising. With the proposed data structure, the runtimes seem to be constant for the algorithms with small population sizes, but only until the problem size of few thousands. After that size, the runtimes seem to get worse until the transition around the size of 10^5 , after which they also become linear. This, however, is easily explained by the domination of the initial fitness evaluation: as this experiment operates with a fixed computational budget for evaluations, for very large problem sizes computation of the first few fitness values takes more time than the rest of the optimization.

Even despite the deficiency of the experimental setup, we can see that

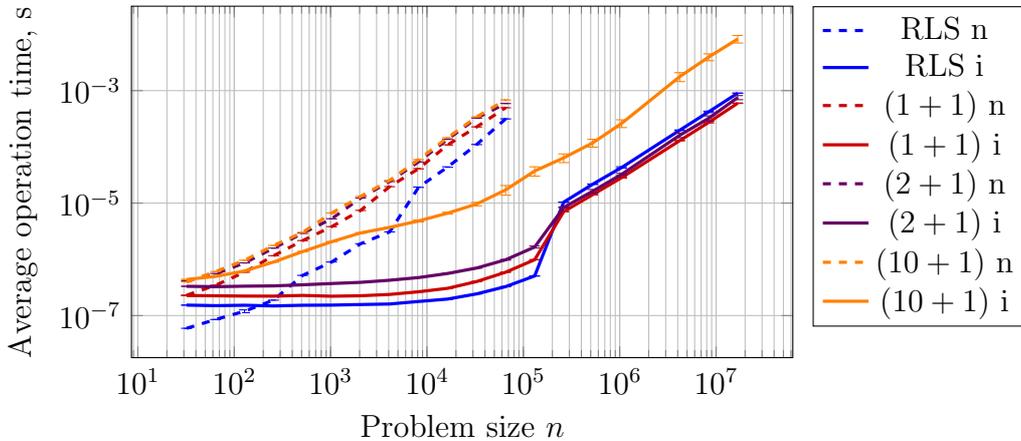


Figure 3: Operation times for the knapsack problem with budget of 25000 fitness evaluations. The last letter of the algorithm’s name means “n” for “naïve” and “i” for “incremental”. Means and standard deviations are shown, the latter almost always invisible.

the $(10 + 1)$ GA shows an increasing trend even for the small problem size, which, however, is significantly slower than the one of the naïve approach. This can be an indication that the $(10 + 1)$ GA manages to maintain some diversity in the population, which is immediately reflected in the operation times, however, the proposed data structure still is quite efficient even in these conditions.

4.3 Knapsack Problem, Varying Budget

One of the possible measures of diversity in the population, which is available almost for free in the proposed data structure, is the sum of sizes of all the patches: the larger the sum, the bigger the diversity. Additionally, this sum may serve as an approximation of the work each variation operator has to do, but it is unclear how good this approximation is.

We performed an additional experiment with the knapsack problem. In this experiment, we use the $(10 + 1)$ GA exclusively. We use one particular knapsack problem instance with $n = 10^4$, which was randomly generated using the procedure from the previous experiment. However, this time we record the average operation time, as well as the average patch size, every 10 evaluations until the computational budget of 10^5 is reached.

The plot of average patch sizes against the number of evaluations is presented in Fig. 4, and the plot of average operation times is presented in Fig. 5. Both plots are somewhat similar: in the beginning, both quantities

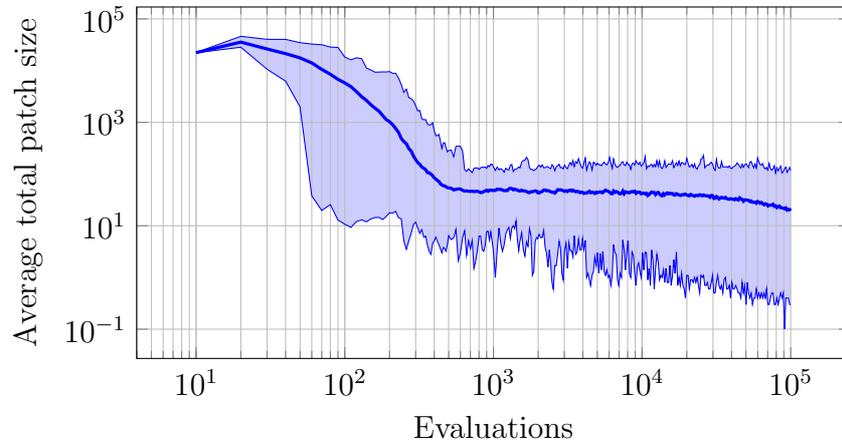


Figure 4: Average patch sizes for the $(10 + 1)$ GA on the knapsack problem with $n = 10^4$ as a function of the number of fitness evaluations. Means, minima and maxima are shown.

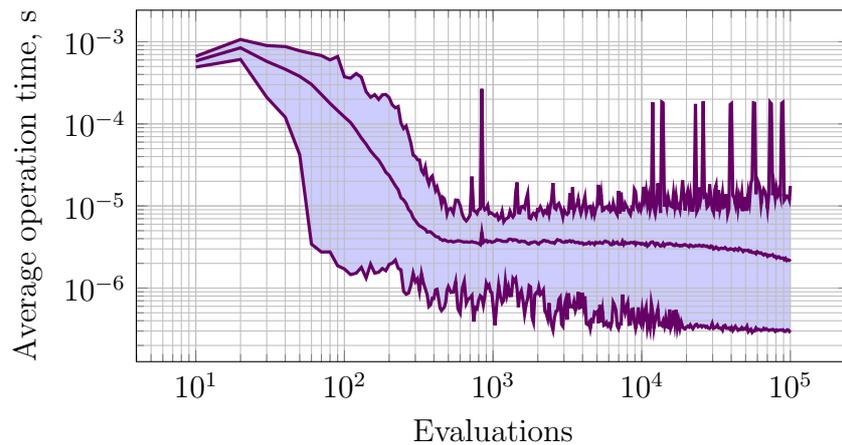


Figure 5: Average operation times for the $(10 + 1)$ GA on the knapsack problem with $n = 10^4$ as a function of the number of fitness evaluations. Means, minima and maxima are shown.

quickly reach the maximum value, then they drop to the small values, and starting from roughly 600 fitness evaluations they decrease very slowly. The initial growth phase seems somewhat unexpected, but it can be explained by a number of individuals which did not survive, but have not yet been deleted from the tree. The quick decrease corresponds to the gradual loss of diversity, which is not unexpected.

What is interesting, however, is that the patch size does not drop to values

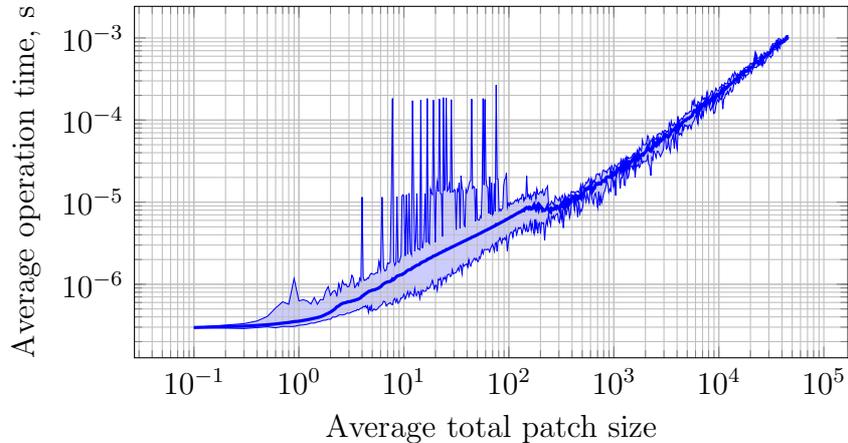


Figure 6: Average operation times for the $(10 + 1)$ GA on the knapsack problem with $n = 10^4$ as a function of the average patch size. Means, minima and maxima are shown.

of 0 or 1 for a long time, which indicates that a noticeable amount of diversity is still present in the population. For up to 10^4 evaluations, the average total patch size is greater than 30 for a population of 10, which cannot happen if the individuals are set aside by single bit flips. It can happen though that there are just two different individuals located at a large distance, with the rest of the population being their copies. However, this can still result in different population dynamics compared to the population-less algorithms.

Since Fig. 4 and 5 look very similar, one may expect that the total patch size and the average operation time are strongly correlated, which also matches the employed algorithms well. Fig. 6 plots the average operation time as a function of the average total patch size. While the correlation is indeed high (the Pearson correlation coefficient was 0.9327), Fig. 6 suggests that the dependency might be more complicated. Though the sublinear parts of the employed algorithms can affect the linearity for very small patch sizes, the explanation of the abrupt change of the trends around the patch size of 200 may require additional investigations.

5 Conclusion and Future Work

We proposed a data structure for storing populations, the minimum spanning tree of patches, that has a good potential of improving time and memory complexity of population-based genetic algorithms in a similar way that is already possible for mutation-only algorithms and gray-box optimizers.

Different search spaces. Most if not all of the proposed ideas can be implemented in discrete search spaces other than bit strings. Care should be taken, however, of how exactly the patches are combined, and especially reversed. Permutations already present a challenge there, as a related paper shows [4].

Hash functions for non-revisiting algorithms. Polynomial hash functions can be incrementally computed in a way similar to fitness functions. This can be used to maintain a hash table of all individuals in the population and prevent sampling the same individual more than once. Note that a patch of size zero will otherwise be created for such a duplicate, however, filtering these out based on hash tables is more efficient.

Not only trees to make paths shorter. The minimality of the minimum spanning tree can be, to some extent, sacrificed for making the average paths between individuals shorter. It is not clear what is the most efficient design of such a graph, but most likely the number of edges should still be linear in the number of vertices.

More than one complete individual. If the population size is large enough, or if there is significant diversity in the population, the total size of all the patches may be of the same order, or more, as the size of an individual. The data structure may then benefit from using more than one complete individual: a request to perform an operator on an individual may be fulfilled by, for instance, the closest complete individual. The proper management of the positioning of the complete individuals may be tricky and should be a subject of further investigation.

Large populations. It is undesirable to traverse the entire population in the case it becomes large. Advanced balanced trees and path handling data structures, such as splay trees [41] and link-cut trees [40], have a potential to improve the performance further. They may also help in a better, non-lazy, handling of discarding individuals by treating the minimum spanning tree as a dynamic structure. With more effort, it is probably possible to implement specialized queries on large populations, such as searching for individuals at a certain distance faster than by complete enumeration. With an efficient implementation, this may go as far as remembering all the individuals sampled throughout the search process.

Higher arities. Similarly to supporting crossovers, it is possible to support higher-arity operators, for instance ternary operators common to differential evolution [23, 42]. This requires only a moderate modification of the data structure behind the mutable patch, which would still occupy the linear amount of memory.

Naïve bootstrapping. Finally, the proposed data structure can be used not from the very start of the algorithm, but once the diversity decreases

somewhat, which would alleviate the problem of large overheads during the first iterations of the algorithms found in the experiments.

References

- [1] D. Antipov, M. Buzdalov, and B. Doerr. Fast mutation in crossover-based algorithms. *Algorithmica*, 84(6):1724–1761, 2022.
- [2] Q. Askari, I. Younas, and M. Saeed. Critical evaluation of sine cosine algorithm and a few recommendations. In *Proceedings of Genetic and Evolutionary Computation Conference Companion*, pages 319–320, 2020.
- [3] T. Bäck, D. B. Fogel, and Z. Michalewicz, editors. *Evolutionary Computation 1: Basic Algorithms and Operators*. Institute of Physics Publishing, 2000.
- [4] A. Bassin and M. Buzdalov. The $(1 + (\lambda, \lambda))$ genetic algorithm for permutations. In *Proceedings of Genetic and Evolutionary Computation Conference Companion*, pages 1669–1677. ACM, 2020.
- [5] A. Bouter, T. Alderliesten, and P. A. Bosman. Achieving highly scalable evolutionary real-valued optimization by exploiting partial evaluations. *Evolutionary Computation*, 29(1):129–155, 2021.
- [6] A. Bouter, T. Alderliesten, C. Witteveen, and P. A. N. Bosman. Exploiting linkage information in real-valued optimization with the real-valued gene-pool optimal mixing evolutionary algorithm. In *Proceedings of Genetic and Evolutionary Computation Conference*, pages 705–712, 2017.
- [7] N. Bulanova and M. Buzdalov. On binary unbiased operators returning multiple offspring. In *Proceedings of Genetic and Evolutionary Computation Conference Companion*, pages 1395–1398, 2017.
- [8] M. Buzdalov and B. Doerr. Runtime analysis of the $(1 + (\lambda, \lambda))$ genetic algorithm on random satisfiable 3-CNF formulas. In *Proceedings of Genetic and Evolutionary Computation Conference*, pages 1343–1350, 2017.
- [9] E. Carvalho Pinto and C. Doerr. A simple proof for the usefulness of crossover in black-box optimization. In *Parallel Problem Solving from Nature – PPSN XV, Vol. 2*, number 11102 in Lecture Notes in Computer Science, pages 29–41. 2018.

- [10] E. Carvalho Pinto and C. Doerr. Towards a more practice-aware runtime analysis of evolutionary algorithms. <https://arxiv.org/abs/1812.00493>, 2018.
- [11] F. Chicano, D. Whitley, G. Ochoa, and R. Tinós. Optimizing one million variable NK landscapes by hybridizing deterministic recombination and local search. In *Proceedings of Genetic and Evolutionary Computation Conference*, pages 753–760, 2017.
- [12] F. Chicano, D. Whitley, and A. M. Sutton. Efficient identification of improving moves in a ball for pseudo-Boolean problems. In *Proceedings of Genetic and Evolutionary Computation Conference*, pages 437–444, 2014.
- [13] F. Chin and D. Houck. Algorithms for updating minimal spanning trees. *Journal of Computer and System Sciences*, 16:333–344, 1978.
- [14] P. Chu and J. Beasley. A genetic algorithm for the multidimensional knapsack problem. *Journal of Heuristics*, 4(1):63–86, 1998.
- [15] D. Corus and P. S. Oliveto. On the benefits of populations for the exploitation speed of standard steady-state genetic algorithms. *Algorithmica*, 82:3676–3706, 2020.
- [16] D. Dang, T. Friedrich, T. Kötzing, M. S. Krejca, P. K. Lehre, P. S. Oliveto, D. Sudholt, and A. M. Sutton. Escaping local optima using crossover with emergent diversity. *IEEE Transactions on Evolutionary Computation*, 22(3):484–497, 2018.
- [17] K. Deb and C. Myburgh. A population-based fast algorithm for a billion-dimensional resource allocation problem with integer variables. *European Journal of Operational Research*, 261(2):460–474, 2017.
- [18] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multi-objective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.
- [19] B. Doerr, C. Doerr, and F. Ebel. From black-box complexity to designing new genetic algorithms. *Theoretical Computer Science*, 567:87–104, 2015.
- [20] B. Doerr, A. Echarghaoui, M. Jamal, and M. S. Krejca. Lasting diversity and superior runtime guarantees for the $(\mu + 1)$ genetic algorithm, 2023.

- [21] B. Doerr, D. Johannsen, T. Kötzing, P. K. Lehre, M. Wagner, and C. Winzen. Faster black-box algorithms through higher arity operators. In *Proceedings of Foundations of Genetic Algorithms*, pages 163–172, 2011.
- [22] B. Doerr and C. Winzen. Reducing the arity in unbiased black-box complexity. *Theoretical Computer Science*, 545:108–121, 2014.
- [23] B. Doerr and W. Zheng. Working principles of binary differential evolution. *Theoretical Computer Science*, 801:110–142, 2020.
- [24] F.-A. Fortin, F.-M. de Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, 2012.
- [25] J. E. Gentle. *Random Number Generation and Monte Carlo Methods*. Springer, 2nd edition, 2003.
- [26] N. Hansen and A. Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9:159–195, 2001.
- [27] N. Hansen, R. Ros, N. Mauny, M. Schoenauer, and A. Auger. Impacts of invariance in search: When CMA-ES and PSO face ill-conditioned and non-separable problems. *Applied Soft Computing*, 11(8):5755–5769, 2011.
- [28] W. Hörmann. The generation of binomial random variates. *Journal of Statistical Computation and Simulation*, 46(1-2):101–110, 1993.
- [29] T. Jansen and C. Zarges. Analysis of evolutionary algorithms: From computational complexity analysis to algorithm engineering. In *Proceedings of Foundations of Genetic Algorithms*, pages 1–14, 2011.
- [30] V. Kachitvichyanukul and B. W. Schmeiser. Binomial random variate generation. *Communications of the ACM*, 31(2):216–222, 1988.
- [31] S. Kirkpatrick and B. Selman. Critical behavior in the satisfiability of random Boolean expressions. *Science*, 264(5163):1297–1301, 1994.
- [32] P. K. Lehre and C. Witt. Black-box search by unbiased variation. *Algorithmica*, 64:623–642, 2012.
- [33] M. López-Ibáñez, J. Branke, and L. Paquete. Reproducibility in evolutionary computation. *ACM Transactions on Evolutionary Learning and Optimization*, 1(4):14:1–14:21, 2021.

- [34] G. Marsaglia and W. W. Tsang. The ziggurat method for generating random variables. *Journal of Statistical Software*, 5(8), 2000.
- [35] A. J. Nebro, J. J. Durillo, and M. Vergne. Redesigning the jMetal multi-objective optimization framework. In *Proceedings of Genetic and Evolutionary Computation Conference Companion*, pages 1093–1100, 2015.
- [36] P. S. Oliveto, D. Sudholt, and C. Witt. Tight bounds on the expected runtime of a standard steady state genetic algorithm. *Algorithmica*, 84:1603–1658, 2022.
- [37] D. Pisinger. *Algorithms for Knapsack Problems*. PhD thesis, University of Copenhagen, 2 1995.
- [38] E. Pitzer and M. Affenzeller. Cheating like the neighbors: Logarithmic complexity for fitness evaluation in genetic algorithms. In *Proceedings of IEEE Congress on Evolutionary Computation*, pages 1431–1438, 2021.
- [39] J. Rowe and M. Vose. Unbiased black box search algorithms. In *Proceedings of Genetic and Evolutionary Computation Conference*, pages 2035–2042, 2011.
- [40] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.
- [41] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of ACM*, 32(3):652–686, 1985.
- [42] R. Storn and K. Price. Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11(4):341–359, 1997.
- [43] R. Tinós, D. Whitley, and G. Ochoa. A new generalized partition crossover for the traveling salesman problem: Tunneling between local optima. *Evolutionary Computation*, 28(2):255–288, 2020.
- [44] R. Tinós, L. Zhao, F. Chicano, and D. Whitley. NK hybrid genetic algorithm for clustering. *IEEE Transactions on Evolutionary Computation*, 22(5):748–761, 2018.
- [45] N. Veerapen, G. Ochoa, M. Harman, and E. K. Burke. An integer linear programming approach to the single and bi-objective Next Release problem. *Information and Software Technology*, 65:1–13, 2015.

- [46] A. J. Walker. An efficient method for generating discrete random variables with general distributions. *ACM Transactions on Mathematical Software*, 3(3):253–256.
- [47] A. J. Walker. New fast method for generating discrete random numbers with arbitrary frequency distributions. *Electronics Letters*, 10(8):127–128, 1974.
- [48] L. D. Whitley, F. Chicano, and B. W. Goldman. Gray box optimization for Mk landscapes (NK landscapes and MAX-kSAT). *Evolutionary Computation*, 24(3):491–519, 2016.
- [49] C. Wishon and J. R. Villalobos. Robust efficiency measures for linear knapsack problem variants. *European Journal of Operational Research*, 254(2):398–409, 2016.