The (1 + (λ, λ)) Genetic Algorithm for Permutations

Anton Bassin
ITMO University
Saint Petersburg, Russia
anton.bassin@gmail.com

Maxim Buzdalov
ITMO University
Saint Petersburg, Russia
mbuzdalov@gmail.com

ABSTRACT
The (1 + (λ, λ)) genetic algorithm is a bright example of an evolutionary algorithm which was developed based on the insights from theoretical findings. This algorithm uses crossover, and it was shown to asymptotically outperform all mutation-based evolutionary algorithms even on simple problems like OneMax. Subsequently it was studied on a number of other problems, but all of these were pseudo-Boolean.

We aim at improving this situation by proposing an adaptation of the (1 + (λ, λ)) genetic algorithm to permutation-based problems. Such an adaptation is required, because permutations are noticeably different from bit strings in some key aspects, such as the number of possible mutations and their mutual dependence. We also present the first runtime analysis of this algorithm on a permutation-based problem called HAM whose properties resemble those of OneMax. On this problem, where the simple mutation-based algorithms have the running time of \( \Theta(n^2 \log n) \) for problem size \( n \), the (1 + (λ, λ)) genetic algorithm finds the optimum in \( O(n^2) \) fitness queries. We augment this analysis with experiments, which show that this algorithm is also fast in practice.

CCS CONCEPTS
• Theory of computation → Theory of randomized search heuristics.

KEYWORDS
Runtime analysis, (1 + (λ, λ)) GA, permutations.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:
Anton Bassin and Maxim Buzdalov. 2020. The (1 + (λ, λ)) Genetic Algorithm for Permutations. In Genetic and Evolutionary Computation Conference Companion (GECCO ’20 Companion), July 8–12, 2020, Cancún, Mexico. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3377929.3398148

1 INTRODUCTION
The (1 + (λ, λ)) genetic algorithm (GA), proposed in [12], is a fairly recent algorithm with very interesting properties. It was the first general-purpose optimizer to outperform simple evolutionary algorithms, such as the (1 + 1) evolutionary algorithm (EA), on the simple benchmark problem OneMax asymptotically: compared to the required \( \Theta(n \log n) \) fitness evaluations in expectation, typical to hill-climbers, it needs only \( o(n\sqrt{\log n}) \) of them for an optimal fixed parameter setting, and \( O(n) \) when the simple 1/5-th rule is used to control the parameter \( \lambda \) [11].

On a larger scale, the performance of the (1 + (λ, λ)) GA on OneMax is asymptotically better than of any unbiased black-box algorithm which uses only mutations. This can be seen as a positive answer to the question of whether the crossover, as a design pattern in evolutionary algorithms, is useful even for simple problems: in artificial settings, it was known for quite a while [19], which was then followed by some evidence on combinatorial problems [27], while from the complexity perspectives the usefulness of higher-arity operators was shown already in [13]. The steady-state \((\mu + 1)\) GA has also been shown to outperform mutation-only algorithms [9, 10], although not asymptotically, and [22] clarified that non-trivial population sizes \( \mu \) are crucial for that. Similar statements for a diversity-preserving \((2 + 1)\) GA were proven in [23, 28].

The (1 + (λ, λ)) GA was also experimentally found to be competitive in application to satisfiability problems [18], which was subsequently confirmed theoretically [4] based on some of the insights from the earlier works [15, 29]. This algorithm has also been analysed on another benchmark problem, LeadingOnes [1]. However, its theoretical and practical applications were so far limited to the domain of pseudo-Boolean functions. The main reason for this seems to be the structure of the (1 + (λ, λ)) GA, which makes it more efficient in certain circumstances, but, in contrast to many other evolutionary algorithms, also complicates introducing changes, such as switching to a different domain.

We aim at changing this situation. We propose an adaptation of the (1 + (λ, λ)) GA to a different kind of the search space, the permutations. First, this is one of the most popular representations that has very different characteristics compared to bit strings and to real-valued vectors, and in particular, it allows much more structurally different choices to what a mutation and a crossover can be (cf. [2, Chapter 17]). Second, permutations are the search space for one of the most studied NP-hard combinatorial optimization problem with direct practical applications, the traveling salesperson problem. For this problem, crossover was shown to be very important by the field of gray-box optimization [25], playing roughly the same role here as for pseudo-Boolean functions [7, 31]. Finally, there are plenty of practical applications that require to optimize very complicated functions on permutations, see e.g. [16], for which the gray-box optimization cannot be an answer.

Permutations have been also considered by the theory of evolutionary algorithms, but to lesser extent compared to bit strings. The seminal paper [26] proves some important facts for various flavors of the (1 + 1) EA on the problem of sorting a permutation. In this context, a number of different fitness function to evaluate sortedness were considered, as well as a number of distinct
mutation operators. The fitness level method [30] has also been applied to sorting by inversions in [8]. In [17], the performance of hill-climbers was investigated in the presence of comparators with noise. The fitness landscape of a permutation-based linear ordering problem was investigated in [6] with regards to the transformation of a single-objective problem into a multiobjective one.

The main points of our contribution are as follows:

- we propose a modification of the \((1 + (\lambda, \lambda))\) GA suitable to solve problems defined on permutations (Section 3);
- we show that our design is sound by proving for the sorting problem, when using the exchange elementary mutation and the Hamming-distance fitness, that the \((1 + (\lambda, \lambda))\) GA, with a suitable fitness-dependent choice of \(\lambda\), solves it in expected \(O(n^2)\) fitness queries, faster than \(\Theta(n^2 \log n)\) achieved by hill-climbers (Section 4);
- we augment the proof by the experimental investigation of the running time, which shows that our modification of the \((1 + (\lambda, \lambda))\) GA is quite efficient in practice as well, and the impact of the parameter values on the running time to showcase that the \((1 + (\lambda, \lambda))\) GA on permutations is a rather interesting subject for theoretical research (Section 5).

We note that, although using fitness-dependent values of \(\lambda\) is not what a black-box optimizer should do, our theoretical bound estimates the possible performance of parameter control methods.

Finally we note that, apart from just performing an adaptation of the \((1 + (\lambda, \lambda))\) GA to some different shape of the search space, we try to summarize the first guidelines on how to do it for this algorithm in general. More precisely, we think that anyone who would like to adapt the \((1 + (\lambda, \lambda))\) GA to their own search space would necessarily have to perform the same steps as we have done, so our modifications can be used in such cases as a guideline.

## 2 PRELIMINARIES

### 2.1 Notation

We use \(\log(x)\) for the natural logarithm of \(x\). We denote as \([a, b]\) a set of integers \([a, a + 1, \ldots, b]\) and \([n]\) is a shortcut for \([1..n]\). For a real-valued \(x\), \([x]\) means \(x\) rounded up to the nearest integer. \(\mathcal{B}(n, p)\) stands for the binomial distribution with the number of trials \(n\) and success probability \(p\).

### 2.2 The \((1 + 1)\) EA

The \((1 + 1)\) EA (Algorithm 1) applies, on every iteration, the standard bit mutation to the parent individual \(x\), which is to choose a number \(\ell\) from a binomial distribution and to flip \(\ell\) randomly chosen bits in \(x\). If the resulting offspring \(y\) is at least as good as its parent \(x\), it replaces the parent. The default mutation probability is \(1/\ell\), so that the expected value of \(\ell\) is exactly one.

In the case when the number of possible mutations is large, and it is computationally infeasible to simulate the binomial distribution by definition, many researchers approximate it with the Poisson distribution: first, this distribution is sampled for the number of mutations to apply, and then this number of mutations is sampled without replacement. The expected number of mutations is still constant, which is quite important both from theoretical and practical viewpoint because local moves are performed sufficiently often.

### Algorithm 1 The \((1 + 1)\) EA with standard bit mutation

1. \(x \leftarrow \text{uniformly from } \{0, 1\}^n\)
2. for \(i \leftarrow 1, 2, 3, \ldots\) do
3. \(\ell \sim \mathcal{B}(n, 1/n)\)
4. \(y \leftarrow \text{flip } \ell\text{ uniformly chosen bits in } x\) \(\triangleright \text{ Mutation}\)
5. if \(f(y) \geq f(x)\) then \(\triangleright \text{ Selection}\)
6. \(x \leftarrow y\)
7. end if
8. end for

### Algorithm 2 The \((1 + (\lambda, \lambda))\) GA for bit strings

1. \(n \leftarrow \text{the problem size}\)
2. \(\lambda \leftarrow \text{the population size parameter}\)
3. \(x \leftarrow \text{uniformly from } \{0, 1\}^n\)
4. for \(i \leftarrow 1, 2, 3, \ldots\) do
5. \(p \leftarrow \lambda/n, c \leftarrow 1/\ell, \lambda' \leftarrow [\lambda], \ell \sim \mathcal{B}(n, p)\)
6. for \(i \in [\lambda']\) do \(\triangleright \text{ Phase 1: Mutation}\)
7. \(x^{(i)} \leftarrow \text{flip } \ell\text{ uniformly chosen bits in } x\)
8. end for
9. \(x' \leftarrow \text{uniformly from } \{x^{(i)} \mid f(x^{(i)}) = \max\{f(x^{(i)})\}\}\)
10. for \(i \in [\lambda']\) do \(\triangleright \text{ Phase 2: Crossover}\)
11. for \(j \in [n]\) do
12. \(y^{(i)} \leftarrow x'_j\text{ with probability } c, \text{ otherwise } x_j\)
13. end for
14. end for
15. \(y \leftarrow \text{uniformly from } \{y^{(j)} \mid f(y^{(j)}) = \max\{f(y^{(i)})\}\}\)
16. Optionally adjust \(\lambda\) based on \(f(x)\) and \(f(y)\) \(\triangleright \text{ Adaptation}\)
17. if \(f(y) \geq f(x)\) then \(\triangleright \text{ Selection}\)
18. \(x \leftarrow y\)
19. end if
20. end for

### 2.3 The \((1 + (\lambda, \lambda))\) GA

Now we describe the \((1 + (\lambda, \lambda))\) GA for bit strings as proposed in [12]. The algorithm is outlined as Algorithm 2. In short, this algorithm does the following on each iteration:

- during the first phase of each iteration, it creates a intermediate population of size \(\lambda' = [\lambda]\) using a higher-than-usual mutation rate of \(\lambda/n\);
- the offspring \(x^{(i)}\) typically have much worse fitness than their parent \(x\), however, the best first-phase offspring \(x'\) has better chances to contain the new improvements, so it is selected for the second phase to undergo crossover with \(x\);
- in the second phase, the crossover is performed \(\lambda'\) times that takes the bits from \(x'\) only with probability of \(1/\lambda\), so that the outcome of the crossover has one bit different from the parent in expectation;
- the best second-phase offspring \(y\) competes directly with the parent \(x\) as in the \((1 + 1)\) EA, while also optionally adjusting the value of \(\lambda\).

With an appropriate choice of \(\lambda\), the \((1 + (\lambda, \lambda))\) GA is able to test \(\Theta(\lambda^2)\) bits with only \(\Theta(\lambda)\) fitness queries, and the moderate deviations from the optimal choice still retain good performance. For \textsc{OneMax}, the optimal value of \(\lambda\), depending on the fitness \(f\),...
is $\lambda = \sqrt{\frac{n}{n - f}}$. If $\lambda$ can depend only on $n$, [11] shows that the optimal setting for $\lambda$ and the corresponding running time are:

$$
\lambda = \sqrt{\frac{\log n}{\log \log n}}, \quad T = O\left(\pi \frac{\log n \log \log n}{\log \log n}\right).
$$

On the other hand, the simple 1/5-th rule can be applied to adjust $\lambda$ on-the-fly: if the parent is replaced by an individual with better fitness, $\lambda$ is divided by a constant $F \in (1; 2)$, otherwise it is multiplied by $F^{1/4}$. The $(1 + (\lambda, \lambda))$ GA with this rule achieves the provable $O(n)$ time on ONEMAX, which is strictly better than what is possible with the fixed $\lambda$.

### 2.4 Mutations for permutations

For the sake of self-containedness, we present a list of the most common operations used to introduce small changes to permutations, taken from [26]. Just like that paper, we limit ourselves to minimal local changes, which we call elementary mutations from now on. When an algorithm needs a global mutation, it can either use the binomial distribution or Poisson distribution to sample the number of mutations to apply in order. Below, we list these elementary mutations together with the number of possible mutations for the problem size $n$.

- **The exchange mutation**: exchange the elements at two different indices $i$ and $j$. There are $\frac{n(n-1)}{2}$ different mutations.
- **The reverse mutation**: reverse a segment of the permutation between indices $i$ and $j$. Depending on which reversals make sense, there may be up to $n(n - 1)$ different mutations, but in any case this number will be $\Theta(n^2)$.
- **The jump mutation**: move an element at index $i$ to index $j$ while the elements at the intermediate indices shift to the corresponding direction. There are $n(n-1)$ different mutations.

Note that, unlike the common mutations for bit strings, most of the presented mutations can be not commutative, that is, when having two elementary mutations $m_1$ and $m_2$, it makes a difference whether one applies $m_1$ and then $m_2$, or $m_2$ and then $m_1$. Among other things, this poses a difficulty for the $(1 + (\lambda, \lambda))$ GA, because it is now much less trivial to identify a good mutation if we suspect there is one.

### 2.5 Problems ONE MAX and HAM

We often refer to the ONE MAX problem, which is defined on bit strings of length $n$ as a maximization problem as follows:

$$
\text{ONE MAX}(x) = \sum_{i=1}^{n} [x_i = 1].
$$

For benchmarking our modification of the $(1 + (\lambda, \lambda))$ GA we use a permutation-based problem which was called HAM in [26]. This problem can be formally defined as follows:

$$
\text{HAM}(\pi) = \sum_{i=1}^{n} [\pi_i = i].
$$

where $n$ is the problem size. Informally, it counts the number of positions at which the queried permutation coincides with the identity (sorted) permutation. As our algorithms are unbiased as in [24], our analysis covers the whole class of similar problems, namely, $\text{HAM}_p(\pi) = \sum_{i=1}^{n} [\pi_i = p_i]$, where $p$ is a “guessed” permutation.

In this paper, we use the exchange mutations as the only considered elementary mutations. These make a good starting point, because from the point of the Hamilton problem these elementary mutations are the most local operations possible. A number of considerations in this paper would also make sense for different elementary mutations, and of course for other problems, however, the theoretical analysis and experimental investigations are performed solely for HAM and exchange mutations.

We shall now shortly reconsider the basic properties of elementary exchange mutations, which are detailed in [26]. The number of elementary exchange mutations is $\binom{n}{2} = \frac{n(n-1)}{2}$. Assuming that the current Hamming distance to the optimum is $d$, these elementary mutations can be classified as follows, assuming we evaluate the effect of each mutation independently of others.

- **Mutations which exchange two positions that were guessed right**: There are $\binom{n-d}{2} = \frac{(n-d)(n-d-1)}{2}$ mutations of this sort, and each of them increases the Hamming distance by 2, because none of the new positions become guessed right.
- **Mutations which exchange a position that was guessed right and a position that was guessed wrong**: There are $(n-d)d$ mutations of this sort, and each of them increases the Hamming distance by 1, because none of the new positions become guessed right.
- **Mutations which exchange two positions that were guessed wrong**: There are $\binom{d}{2} = \frac{d(d-1)}{2}$ mutations of this sort. Depending on what happens, the following effects apply:
  - both new positions are still guessed wrong: this does not change the Hamming distance;
  - one of the new positions becomes guessed right: this decreases the Hamming distance by 1;
  - both new positions become guessed right: this decreases the Hamming distance by 2.

It is clear that for each position that is guessed wrong there is exactly one mutation that makes it right. However, if a mutation decreases the Hamming distance by 2, it is counted twice. If there are $x$ such mutations, there are $d - 2x$ mutations that decrease the Hamming distance by 1. Hence there are at least \( \left\lceil \frac{d}{2} \right\rceil \) and at most $d$ elementary mutations, $\Theta(d)$ in total, that decrease the Hamming distance. The expected distance decrease, assuming every elementary mutation is chosen with equal probability, is exactly $\frac{2d}{n(n-1)}$.

The latter consideration allows to easily re-prove the results from [26, Theorem 3] using suitable modern tools, such as multiplicative drift theorems [14, 21] for upper and lower bounds.

The effect of applying several elementary exchange mutations may not cumulate if these mutations modify a certain position more than once. This effect can result in deviations of both signs, as will be shown in the following examples. For convenience we denote a mutation that exchanges positions $i$ and $j$ by $\langle i, j \rangle$. In the few examples below, we work with permutations of size 3 and assume that the optimum is $[1, 2, 3]$. 


• Consider a permutation [2, 1, 3]. A mutation (1, 2) decreases the Hamming distance by 2, and a mutation (1, 3) increases it by 1. However, applying (1, 2) and then (1, 3) together retains the Hamming distance unchanged (and does not decrease it by 1). Applying them in a different order results in a different permutation, however, the Hamming distance is not decreased by 1 as well.

• Consider a permutation [2, 3, 1]. A mutation (1, 3) decreases the Hamming distance by 1, and a mutation (2, 3) does not change it. However, applying (1, 3) and then (2, 3) decreases the Hamming distance by 3 (not by 1), and applying them in a different order increases it by 1.

As a result, we see that whenever we have a number of offspring and elementary mutations overlap in some of them, it is difficult for an algorithm with a structure similar to the \((1 + (\lambda, \lambda))\) GA to reliably determine which of these offspring contains a promising mutation.

3 THE \((1 + (\lambda, \lambda))\) GA FOR PERMUTATIONS

In this section, we present our modification of the \((1 + (\lambda, \lambda))\) GA suitable for solving optimization problems on permutations and discuss the corresponding design choices and their consequences. The pseudocode of the algorithm is outlined in Algorithm 3 with the differences highlighted in blue.

The key differences between this modification and the original \((1 + (\lambda, \lambda))\) GA for bit strings are summarized below.

1. Most parameters of the algorithm now depend on the number of possible mutations \(m\) (for permutations, \(m = \Theta(n^2)\)) rather than the problem size \(n\). In fact, it is just a pure coincidence that the number of mutations and the problem size coincide for the most investigated problems on bit strings. Technically, there may exist problems which would benefit from being able to apply more than \(n\) elementary mutations at once; with the default choice, this would not be possible.

2. The order of elementary mutations matters. This may introduce an additional implementation detail in the algorithm. For example, there exist certain ways applicable to bit strings that allow generation of the bit flip indices without sampling the (pseudo)random number generator for every bit, which are based on a quite nice distribution of the distance between the successive indices [20]. If the elementary mutations are encoded as integer numbers, and such a method is applied, it would generate the indices in an increasing order, which would severely alter the distribution of elementary mutation lists. For this reason, an explicit shuffle of elementary mutations may be a good recommendation.

3. The lists of elementary mutations are stored along the mutants and are, in fact, their complete synonyms. If a problem in hand allows incremental fitness re-evaluation, one is no longer required to store the entire mutant in a separate memory: a difference is enough, and its size would typically be much smaller than the size of the entire individual.

4. Crossover is now mutation subsampling: it is performed by taking the ordered list of elementary mutations that describes a mutant, picking each elementary mutation with probability \(c\) and applying them again to the parent.

Algorithm 3 The \((1 + (\lambda, \lambda))\) GA for permutations

1. \(n \leftarrow \) the problem size
2. \(\lambda \leftarrow \) the population size parameter
3. \(M \leftarrow \) the set of possible mutations for problem size \(n\)
4. \(m \leftarrow |M|\)
5. \(x \leftarrow \) uniformly from \(\Pi_n\)
6. for \(i \leftarrow 1, 2, 3, \ldots\) do
7. \(p \leftarrow \lambda/m, c \leftarrow 1/\lambda, \lambda' \leftarrow [\lambda], \ell \sim B(m, p)\)
8. for \(i \in [\lambda']\) do
9. \(M^{(i)} \leftarrow \) sample \(\ell\) uniformly chosen mutations from \(M\)
10. Shuffle \(M^{(i)}\)
11. \(x^{(i)} \leftarrow x\) with mutations from \(M^{(i)}\) applied in order
12. end for
13. \(x' \leftarrow \) uniformly from \(\{x^{(i)} \mid f(x^{(i)}) = \max\{f(x^{(i)})\}\}\)
14. \(M' \leftarrow \) the corresponding list of mutations
15. for \(i \in [\lambda']\) do
16. \(s \sim B(\ell, c)\)
17. \(C' \leftarrow s\) random elements of \(M'\) with order preserved
18. \(y^{(i)} \leftarrow x\) with mutations from \(C'\) applied in order
19. end for
20. \(y \leftarrow \) uniformly from \(\{y^{(i)} \mid f(y^{(i)}) = \max\{f(y^{(i)})\}\}\)
21. Optionally adjust \(\lambda\) based on \(f(x)\) and \(f(y)\) \(\rightarrow\) Adaptation
22. if \(f(y) \geq f(x)\) then
23. \(x \leftarrow y\) \(\rightarrow\) Selection
24. end if
25. end for

In fact, the original \((1 + (\lambda, \lambda))\) GA can also be seen this way, and efficient implementations which enabled experimenting with problem sizes up to \(2^{25}\) in [4] already use this concept internally. For permutations, this also allows efficient implementation if incremental fitness re-evaluation is possible. However, it can also be that certain permutation-specific techniques, such as operating on cycles rather than positions, can be beneficial, which we leave for the future work.

5 The order of elementary mutations used in the mutant is preserved in crossover when more than one elementary mutation is chosen. This is done in order to reduce the chances of misguiding the fitness-based reproduction selection in the \((1 + (\lambda, \lambda))\) GA and hence to improve the performance: if two non-commuting mutations make a big improvement together, the order-preserving crossover has the bigger probability to take them together.

Note that the proposed changes are, in fact, independent from the particular set of mutations and even from the permutation representation itself. Similar to other evolutionary algorithms, and contrary to the impression that the structure of the \((1 + (\lambda, \lambda))\) GA is overfitted to bit strings, the proposed modification may be used for almost arbitrary problem representation.

We are optimistic that our modification can be used with a large variety of representations and a large variety of possible mutation sets for every such representation. This includes problems over several permutations, such as various scheduling problems, or problems defined on binary strings together with mutations that preserve the number of chosen bits, which appear in practice.
4 RUNNING TIME ANALYSIS

In this section, we are going to analyze the $(1 + (\lambda, \lambda))$ GA using exchange mutations on the HAM problem. We call an elementary mutation good if it improves the fitness when applied to the parent, and all other elementary mutations we consider to be bad. As long as our generalized modification of the $(1 + (\lambda, \lambda))$ GA explicitly operates with the lists of elementary mutations, we do not distinguish a mutant and a list of elementary mutations that generated it.

It is crucial for the $(1 + (\lambda, \lambda))$ GA to reliably distinguish, by using fitness values only, the mutants that were constructed using at least one good elementary mutation (the good mutants) from those mutants which were constructed using only bad elementary mutations (the bad mutants). The original $(1 + (\lambda, \lambda))$ GA has no problems with that on ONEMAX: if the parent’s fitness is $f$ and $\ell$ bits are flipped in each mutant, all bad mutants have fitness of $f - \ell$ and all good mutants have fitness of at least $f - \ell + 2$. For less trivial problems telling good and bad mutants apart becomes more difficult. This happens, for instance, on MAX-SAT problems [4, 18] and on linear functions [3], and requires much more involved proofs.

The proof from [4] uses the following technique. For each iteration of the $(1 + (\lambda, \lambda))$ GA it introduces an artificial fitness threshold $\tau$ and pessimistically considers an iteration to be successful only if the fitness of every good mutant is strictly greater than $\tau$ and the fitness of every bad mutant is strictly less than $\tau$. This appeared to be easier than using more fine-grained approaches, however, the precision of the resulting bounds is not known and may be imperfect. Note, however, that the choice of $\tau$ may influence the degree of pessimism, and whenever the probability of success $p = p(\tau)$ depends on $\tau$, the true success probability is at least the supremum of $p(\tau)$ across all possible $\tau$.

We use the same idea in our analysis in an even more restricted and simplified form. Recall that every elementary mutation results in a fitness change belonging to $\{-2, -1, 0, +1, +2\}$. We introduce a threshold $\tau \in \{-2, -1, 0\}$ and consider the following definition.

**Definition 4.1.** An iteration is good with respect to threshold $\tau$ if:

- there exists exactly one mutant with at least one good elementary mutation, this mutation uses positions $i_1$ and $i_2$, all other elementary mutations use neither of these positions, and they increase the fitness by at least $\tau$ as they are applied;
- in all other mutants, all elementary mutations increase the fitness by at most $\tau$ as they are applied.

If an iteration satisfies this definition, the good mutant gets a fitness advantage over all other mutants, thus it is selected for reproduction. Subsequently, with a constant probability, one of its good elementary mutations is directly applied on the parent. Note that the “as they are applied” clarifications used in Definition 4.1 make us consider the effect of other mutations in the current context. In particular, they allow bad elementary mutations to present in the mutants of the first type, assuming they do not reduce the fitness too much, and even good elementary mutations to present in the mutants of the second type if their goodness is masked by some other elementary mutations.

Now we consider all $\tau \in \{-2, -1, 0\}$ and bound the probabilities that an iteration is good with respect to threshold $\tau$. We will often use the well-known fact that $(1 - 1/x)^{x-1} \geq e^{-1}$ for all $x \geq 1$.

**Lemma 4.2.** For HAM with a problem size $n$, an iteration of the $(1 + (\lambda, \lambda))$ GA with the parent fitness $f$ and $\lambda$ mutants created by applying $\ell$ elementary mutations is good with respect to threshold $\tau = 0$ with probability at least

$$\frac{\lambda \ell}{n} \cdot e^{-\frac{2\ell(\lambda - 1)}{n} - \frac{2\ell \min(n, f + 2\ell + 1)}{n \min(n, f + 2\ell + 1)}}.$$

**Proof.** To ease the notation, we always assume that the fitness is always within $[0; n]$ and omit the min and max clauses in the corresponding locations.

We first estimate the probability of an individual to belong to the second clause of Definition 4.1. The number of good elementary mutations is at most $n - f$, so the probability of not increasing fitness by applying a single elementary mutation to an individual with fitness $f$ is

$$P_1^- \geq 1 - \frac{2(n - f)}{n(n - 1)},$$

so, taking into account that each such mutation decreases the fitness by 0, 1 or 2, the probability of not increasing fitness on each of $\ell$ applications of single elementary mutations is, assuming $n \geq 4$:

$$P_\ell^- \geq \prod_{i=0}^{\ell-1} \left(1 - \frac{2(n - (f - 2i))}{n(n - 1)}\right) \geq \left(1 - \frac{2(n - (f - 2\ell))}{n(n - 1)}\right)^\ell \geq \left(1 - \frac{2}{n - 1}\right) \ell \geq e^{-\frac{2\ell}{n-1}}. \quad (1)$$

We proceed with estimating the probability of an individual to belong to the first clause of Definition 4.1. The number of good elementary mutations is at least $\frac{n - f}{2}$, and the number of elementary mutations which do not decrease fitness and do not touch the positions affected by a given good elementary mutation is exactly $(n - (f + 2))(n - (f + 3))/2$. Taking into account that each such mutation can only increase the fitness by 0, 1 or 2, we get the following bound for the probability of generating such an individual:

$$P_\ell^+ \geq \frac{n - (f + 2\ell)}{n(n - 1)} \prod_{i=0}^{\ell-1} \left(\frac{n - (f + 2i + 2)}{n(n - 1)}\right) \geq \frac{(n - (f + 2\ell - 1))^{2\ell-1}}{n^\ell(n - 1)^\ell} = \frac{\ell \cdot (n - (f + 2\ell + 1))^{2\ell-1}}{n^\ell(n - 1)^\ell} \geq \frac{\ell}{n} \left(1 - \frac{f + 2\ell + 1}{n}\right)^{2\ell} \geq \frac{\ell}{n} \cdot e^{-\frac{2\ell(f + 2\ell + 1)}{n - f - 2\ell - 1}}. \quad (2)$$

Now we get together (1) and (2), remembering that there is exactly one first-clause individual (which can appear as any of the $\lambda$ individuals) and $\lambda - 1$ second-clause individuals:

$$P_{\tau=0} = \lambda \cdot (P_\ell^-)^{\lambda-1} \cdot P_\ell^+ \geq \frac{\lambda \ell}{n} \cdot e^{-\frac{2\ell(\lambda - 1)}{n} - \frac{2\ell \min(n, f + 2\ell + 1)}{n \min(n, f + 2\ell + 1)}}. \quad \square$$

**Lemma 4.2** essentially tells that, whenever $f = \Theta(\sqrt{n})$, it is sufficient to maintain $\lambda, \ell = \Theta(\sqrt{n})$ in order to have constant progress. This is indeed quite natural in the very beginning of the optimization process: when the fitness is small, most elementary mutations are applied to the positions which are not guessed right. However, if $f = \Theta(n)$, one needs at most polylogarithmic values of $\lambda$ and $\ell$ to achieve inverse polynomial probabilities of being good with respect to $\tau = 0$, hence $\tau < 0$ is necessary to consider.
LEMMA 4.3. For HAM with a problem size n, an iteration of the 
$(1 + (λ, λ))$ GA with the parent fitness $f ≥ 3$ and $λ$ mutants created 
by applying $ℓ$ elementary mutations is good with respect to threshold 
$τ = −1$ with probability at least:

$$\frac{λℓ}{n + f − 3} \cdot \left( \frac{f + 1}{n − 1} \right)^{(λ−1)\ell} \cdot 2(2(2^ℓ + 3) + (f + 2)(f − 3)) \cdot \left( \frac{1}{n(n − 1)} \right)^\ell .$$

PROOF. Similarly to the previous lemma, we estimate the proba-

bility of decreasing the fitness by at least 1 by applying $ℓ$ randomly

chosen elementary mutations as follows:

$$P_ℓ^≤ \geq \prod_{i=0}^{ℓ−1} \left( \frac{2(f − 2i)}{n − 1} - \frac{(f − 2i)(f − 2i + 1)}{n(n − 1)} \right) \geq \frac{2(f − 2f + 1)}{n − 1} \cdot \left( \frac{f + 1}{n(n − 1)} \right)^\ell = \frac{f + 1}{n − 1} \cdot \left( \frac{2f}{n(n − 1)} \right)^\ell .$$

Now we estimate the probability of an individual to belong to the 
first clause of Definition 4.1 as follows:

$$P_ℓ^+ \geq \sum_{i=0}^{ℓ−1} \frac{n − (f + 2i)}{n(n − 1)} \cdot \left( \frac{n − (f + 2i)(n + (f + 2i − 3))}{n(n − 1)} \right) \geq \sum_{i=0}^{ℓ−1} \frac{n − (f + 2i − 2)}{n(n − 1)} \cdot \left( \frac{n − (f + 2i − 2)(n + f − 3)}{n(n − 1)} \right)^\ell = \frac{ℓ}{n + f − 3} \cdot \left( \frac{n − (f + 2i − 2)(n + f − 3)}{n(n − 1)} \right)^\ell.$$

We prove the lemma by combining (3) and (4), remembering that 
there is exactly one first-clause individual (which can appear as any 
of the $λ$ individuals) and $λ − 1$ second-clause individuals.

LEMMA 4.3 is noticeably harder to use than Lemma 4.2. However, 
one can notice that, whenever $c_1 \cdot n ≤ f ≤ c_2 \cdot n$ for constants 
$0 < c_1 < c_2 < 1$, all the exponentiation bases in both (3) and (4) 
are bounded by constants that are less than one. In particular, for 
$f = n/2$ the probability is roughly $\frac{\lambda\ell}{\ln n}$, which is maximized 
at $\lambda\ell = \log(4/3)$ to 0.176/n. As a result, we can only bound the 
progress $(1 + (λ, λ))$ GA in the middle of optimization by $Θ(n)$ fitness 
evaluations in expectation per each single fitness improvement, 
thus yielding $O(n^2)$ running time in this region.

LEMMA 4.4. For HAM with a problem size n, an iteration of the 
$(1 + (λ, λ))$ GA with the parent fitness $f ≥ 3$ and $λ$ mutants created 
by applying $ℓ$ elementary mutations is good with respect to threshold 
$τ = −2$ with probability at least:

$$\frac{λℓ}{n + f − 3} \cdot \left( \frac{f + 1}{n − 1} \right)^{(λ−1)\ell} \cdot 2(2(2^ℓ + 3) + (f + 2)(f − 3)) \cdot \left( \frac{1}{n(n − 1)} \right)^\ell .$$

PROOF. Similarly to Lemma 4.2, to ease the notation, we always 
assume that the fitness is always within $[0; n]$ and omit the min 
and max clauses in the corresponding locations.

There are exactly $f(f − 1)/2$ elementary mutations which de-
crease the fitness by 2, hence the probability component that cor-
sponds to have a bad mutant in the right shape reads as follows:

$$P_ℓ^≤ \geq \prod_{i=0}^{ℓ−1} \frac{(f − 2i)(f − 2i + 1)}{n(n − 1)} \geq \left( \frac{1}{n(n − 1)} \right)^\ell \geq \frac{1}{n(n − 1)}.$$

On the other hand, good mutants are now allowed to mutate the 
remaining positions in any possible way, which results in:

$$P_ℓ^+ \geq \sum_{i=0}^{ℓ−1} \frac{n − (f + 2i)}{n(n − 1)} \cdot \left( \frac{n − (f + 2i)(n + (f + 2i − 3))}{n(n − 1)} \right)^\ell \geq \sum_{i=0}^{ℓ−1} \frac{n − (f + 2i − 2)}{n(n − 1)} \cdot \left( \frac{n − (f + 2i − 2)(n + f − 3)}{n(n − 1)} \right)^\ell = \frac{ℓ}{n + f − 3} \cdot \left( \frac{n − (f + 2i − 2)(n + f − 3)}{n(n − 1)} \right)^\ell.$$

We prove the lemma by combining (5) and (6), remembering that 
there is exactly one first-clause individual (which can appear as any 
of the $λ$ individuals) and $λ − 1$ second-clause individuals. □

LEMMA 4.4 is much more similar to Lemma 4.2 in the shape of 
the final result, but not entirely symmetrical. For instance, when 
the distance to the optimum $n − f$ is $O(1)$, the lower bound on 
success probability is roughly proportional to $\frac{λ\ell}{n} e^{−λ^2/n}$ assuming 
$λ ≈ f$. This is maximized at $λ = (2n/3)^{1/3}$, which yields the success 
probability $Ω(n^{-4/3})$, so finding two last positions requires $Ω(n^{4/3})$ fitness 
evaluations in expectation with the optimal choice of $λ$.

Now we are going to formulate the main result of this section.

THEOREM 4.5. With the optimal fitness-dependent parameters, the 
running time of the $(1 + (λ, λ))$ GA on HAM is $O(n^2)$.

PROOF. We use the method of fitness levels, hence we pessimisti-

cally assume that each fitness improvement is minimum possible.

We consider the following ranges of the fitness value $f$:

1. $f = O(\sqrt{n})$. In this case, we set $λ = Θ(\sqrt{n})$. As $ℓ = Θ(λ)$ 
with the constant probability, we choose $τ = 0$ and apply 
Lemma 4.2, which yields a constant probability for an it-
eration to be good. In this mode, the algorithm increases 
the fitness by at least 1 in $O(1)$ iterations and $O(\sqrt{n})$ fitness 
evaluations.

2. $f = O(\sqrt{n})$, $f ≤ c_1 n$ for some constant $0 < c_1 < 1/2$. In 
this case, we still use $τ = 0$ and hence Lemma 4.2, but we 
set $λ = Θ(n/f)$. The probability that an iteration is good is 
$Ω(n/f^2)$, and the expected number of fitness evaluations until 
the fitness update is $O(f^2/n + n/f) = O(f)$. The total 
number of fitness evaluations spent in this phase is

$$\sum_{f=\sqrt{n}}^{c_1 n} O(f) = O(n^2).$$

3. $c_1 n ≤ f ≤ c_2 n$ for some constants $0 < c_1 < 1/2 < c_2 < 1$.

In this case, we choose $τ = −1$ and use Lemma 4.3. We set 
$λ = 1$, so that the algorithm essentially performs local search 
with the improvement probability of $Ω(1/n)$, the expected
time until improvement \(O(n)\) and the total number of fitness evaluations in this phase being \(O(n^2)\).

(4) \[c2n < f < n - \Theta(n^{1/3})\]. We choose \(\tau = -2\) and use Lemma 4.4. From this lemma, the optimal setting \(\lambda = \Theta(\sqrt{n/(n - f)})\) can be derived for the current fitness. The probability that an iteration is good is \(\Theta(\lambda(n - f)/n^2) = \Theta(1/n)\), so the expected number of fitness evaluations until an improvement is \(\Theta(\sqrt{n^3/(n - f)})\). The total number of fitness evaluations spent in this phase is

\[
\sum_{f=c2n}^{n/2} O\left(\frac{n^{3/2}}{\sqrt{n - f}}\right) = O(n^2).
\]

(5) \[n - \Theta(n^{1/3}) < f\]. We still use \(\tau = -2\) and Lemma 4.4, which this time recommends setting \(\lambda = \Theta(n^{1/3})\). This leads to the improvement probability of \(\Theta(n^{-4/3})\), the expected number of evaluations until improvement \(O(n^{5/3})\) and the total number of evaluations spend in this phase to be \(O(n^2)\).

As a result, each range is traversed by the algorithm in time \(O(n^2)\). Since the number of ranges is constant, this proves the theorem. \(\square\)

5 EXPERIMENTS

In this section, we present our experimental results. The implementation of all the experiments is available on GitHub\(^1\). For the sake of performance, we use a set of performance-improving modifications similar to what has been done in [18]:

- the number of elementary mutations for mutants is sampled from a conditional distribution \([\ell \sim B(m, p) \mid \ell > 0]\);
- similarly, the number of elementary mutations to take during crossover is sampled from \([s \sim B(\ell, c) \mid s > 0]\);
- if the crossover offspring takes all mutations from the mutant, it is not re-evaluated and hence not counted towards the number of fitness evaluations.

For a survey of some modifications of this sort please refer to [5].

5.1 Running Times

Figure 1 presents the running times of several algorithms on the HAM problem. To ensure the comparison with popular local search algorithms with the \(\Theta(n^2 \log n)\) performance, we considered the permutation variants of the \((1 + 1)\) evolutionary algorithm and randomized local search (RLS); the tested \((1 + 1)\) EA also uses the conditionally positive binomial distribution. Since it is unlikely that anyone would ever use the \((1 + (\lambda, \lambda))\) GA with the fitness-dependent parameter setting, we considered the following choices:

- the static setting \(\lambda = 10\);
- the default self-adjustment method as in [11];
- the logarithmically capped self-adjustment method as was proposed in [4];
- the problem size dependent static choice \(\lambda = 2 \log(n + 1)\).

The logarithmically capped version behaved exactly as the problem size dependent static choice, so we do not plot the former. The problem sizes were chosen as powers of two from \(2^8\) to \(2^{17}\); the upper limit was chosen so that the experiments could have been done in time, as the single run at \(n = 2^{17}\) reached and exceeded 100 billion fitness evaluations. For every algorithm and every problem size, 100 independent runs were performed; the only exception was the the default self-adjustment method, which started to deteriorate quite early and so was excluded from the further experimentation.

Figure 1 presents the running times divided by \(n^2\) with the logarithmic absissa axis and the linear ordinate axis. In such plots, \(O(n^2)\) algorithms produce horizontal plots and \(\Theta(n^2 \log n)\) algorithms produce plots with a steady slope.

In Figure 1 we can easily see that the \((1 + (\lambda, \lambda))\) GA with the logarithmic choice of \(\lambda\) in fact demonstrates a \(\Theta(n^2)\) performance. This is moderately surprising, since the theoretical analysis from Lemma 4.3 suggests a \(o(n^2)\) upper bound for the middle range of fitness values when \(\lambda = \Theta(\log n)\). This indicates that a more precise analysis is necessary to fully understand what is happening for these fitness values. We can also see that this version of the \((1 + (\lambda, \lambda))\) GA starts outperforming RLS already at a rather small problem size \(2^8\), which indicates a high practical efficiency of the proposed algorithm.

The static parameter version, \(\lambda = 10\), can be seen to deviate from the horizontal line towards the higher problem sizes. This is expected, because theoretical investigations suggest that the performance of this algorithm with a constant population size would be \(O((n^2 \log n)/\lambda)\), similar to the \((1 + (\lambda, \lambda))\) GA on bit strings. Finally, the default self-adjustment method performs much worse than others, which happens because it makes \(\lambda\) grow too high in the fitness ranges where no constant progress is ever possible.

5.2 Parameter Landscape Analysis

To get a better understanding for how the optimal parameters actually look like, we performed a simple landscape analysis. For a few problem sizes \(n \in \{2^8, 2^9, 2^{10}, 2^{11}\}\) we performed 10000 runs of the \((1 + (\lambda, \lambda))\) GA with the fixed value of \(\lambda\) taken from a multiplicative lattice with a step of 1.05. For every \(\lambda\) and every fitness (or, put alternatively, every possible distance to the optimum) we approximated the expected number of fitness evaluations until improvement. Figure 2 presents this as a heatmap, where the color at the point \((d, \lambda)\) stands for the relative probability of improvement compared to the maximum of \((d, \lambda')\) over all tested \(\lambda'\). The yellow color signifies near-optimal values of \(\lambda\), whereas colors towards violet indicate inferior values.

Most theoretical insights appear to be visible in the pictures. In particular, the best values of \(\lambda\) for maximal distances appear to be very close to \(\sqrt{n}\), and the distance range of roughly \([2030; 2048]\) where the optimal \(\lambda\) remains unchanged can be seen in the lower right picture. The small-distance end also features the best values of \(\lambda\) close to the optimum that increase with \(n\) roughly as \(2n^{1/3}\).

The only unexpected thing is that the optimal \(\lambda\) near \(f = n/2\) appears to be non-constant, probably logarithmic, which could not be predicted from our analysis, but which possibly explains the unexpectedly good performance of the \(\lambda = 2 \log(n + 1)\) version.

6 CONCLUSION

We presented an extension to the \((1 + (\lambda, \lambda))\) GA that is capable of efficiently solving permutation-based problems, and whose structure is robust enough to allow applications to completely different problem representations. We conducted theoretical analysis of this

\(^1\)https://github.com/mbuzdalov/generic-onell
algorithm on the permutation-based problem HAM, the Hamming distance, which is probably the first theoretical analysis of a black-box crossover-based algorithm on permutation-based problems.

This combination of an algorithm and a problem poses a number of intriguing questions to the theory community. We have seen five different modes of optimal parameter settings, which require greater parameter values at both ends of the fitness range, and which is quite far from being perfectly symmetrical. This makes it a good benchmark problem for self-adjustment methods that should be able to increase and decrease the parameter in a timely manner.

What is more, we still cannot explain why this algorithm is more efficient around the fitness $f = n/2$ than we can show theoretically. This may force the development of better proof methods inspired by the particular features of the $(1 + (\lambda, \lambda))$ GA. Finally, we hope to investigate the $(1 + (\lambda, \lambda))$ GA on other permutation-based problems.

**ACKNOWLEDGMENTS**

This research was supported by the Russian Science Foundation, grant number 17-71-20178.
REFERENCES

[1] Denis Antipov, Benjamin Doerr, and Vitalii Karavaev. 2019. A tight runtime analysis for the $(1 + (λ,λ))$ GA on LeadingOnes. In Foundations of Genetic Algorithms. 169–182.

[2] Thomas Bäck, David B. Fogel, and Zbigniew Michalewicz (Eds.). 2000. Evolutionary Computation 1: Basic Algorithms and Operators. Institute of Physics Publishing.

[3] Anton Bassim and Maxim Budalov. 2019. The 1/S-th rule with rollbacks: On self-adjustment of the population size in the $(1 + (λ,λ))$ GA. In Proceedings of Genetic and Evolutionary Computation Conference Companion. 277–278. https://arxiv.org/abs/1904.04867

[4] Maxim Budalov and Benjamin Doerr. 2017. Runtime analysis of the $(1 + (λ,λ))$ genetic algorithm on random satisfiable 3-CNF formulas. In Proceedings of Genetic and Evolutionary Computation Conference. 1343–1350.

[5] Eduardo Carvalho Pinto and Carola Doerr. 2018. Towards a more practice-aware runtime analysis of evolutionary algorithms. https://arxiv.org/abs/1812.08493

[6] Josu Ceberio, Borja Calvo, Alexander Mendiburu, and Jose A. Lozano. 2019. Multi-objectivizing combinatorial optimization problems by means of elementary landscape decompositions. Evolutionary Computation 27, 2 (2019), 291–311.

[7] Francisco Chicano, Darrell Whitley, Gabriela Ochoa, and Renato Tinós. 2017. Optimizing one million variable NK landscapes by hybridizing deterministic recombination and local search. In Proceedings of Genetic and Evolutionary Computation Conference. 753–760.

[8] Dogan Corus, Duc-Cuong Dang, Anton V. Eremeev, and Per Kristian Lehre. 2018. Level-based analysis of genetic algorithms and other search processes. IEEE Transactions on Evolutionary Computation 22, 5 (2018), 707–719.

[9] Dogan Corus and Pietro Simone Oliveto. 2018. Standard steady state genetic algorithms can hillclimb faster than mutation-only evolutionary algorithms. IEEE Transactions on Evolutionary Computation 22, 5 (2018), 720–732.

[10] Dogan Corus and Pietro Simone Oliveto. 2019. On the benefits of populations on the exploitation speed of standard steady-state genetic algorithms. In Proceedings of Genetic and Evolutionary Computation Conference. 1452–1460.

[11] Benjamin Doerr and Carola Doerr. 2018. Optimal static and self-adjusting parameter choices for the $(1 + (λ,λ))$ genetic algorithm. Algorithmica 80, 5 (2018), 1658–1709.

[12] Benjamin Doerr, Carola Doerr, and Franziska Ebel. 2015. From black-box complexity to designing new genetic algorithms. Theoretical Computer Science 567 (2015), 87–104.

[13] Benjamin Doerr, Daniel Johannsen, Timo Kötzing, Per Kristian Lehre, Markus Wagner, and Carola Winzen. 2013. Faster black-box algorithms through higher arity operators. In Proceedings of Foundations of Genetic Algorithms. 163–172.

[14] Benjamin Doerr, Daniel Johannsen, and Carola Winzen. 2012. Multiplicative Drift Analysis. Algorithmica 64, 4 (2012), 673–697.

[15] Benjamin Doerr, Frank Neumann, and Andrew M. Sutton. 2015. Improved runtime bounds for the $(1+1)$ EA on random 3-CNF formulas based on fitness-distance correlation. In Proceedings of Genetic and Evolutionary Computation Conference. 1415–1422.

[16] Vitaliy Feoktistov, Stéphane Pietravalle, and Nicolas Haslert. 2017. Optimal experimental design of field trials using differential evolution. In Proceedings of Congress on Evolutionary Computation. 1690–1696.

[17] Tomáš Gavenčiak, Barbara Geissman, and Johannes Lengler. 2019. Sorting by swaps with noisy comparisons. Algorithmica 81 (2019), 796–827.

[18] B. W. Goldman and W. F. Punch. 2014. Parameter-less population pyramid. In Proceedings of Genetic and Evolutionary Computation Conference. 785–792.

[19] Thomas Jansen and Ingo Wegener. 2002. The analysis of evolutionary algorithms—A proof that crossover really can help. Algorithmica 34 (2002), 47–66.

[20] Thomas Jansen and Christine Zarges. 2011. Analysis of evolutionary algorithms: From computational complexity analysis to algorithm engineering. In Proceedings of Foundations of Genetic Algorithms. 1–14. https://doi.org/10.1145/1967654.1967656

[21] Timo Kötzing and Martin Krejča. 2019. First-hitting times under drift. Theoretical Computer Science 796 (2019), 51–69.

[22] Pietro Simone Oliveto, Dirk Sudholt, and Carsten Witt. 2020. A tight lower bound on the expected runtime of standard steady state genetic algorithms. In Proceedings of Genetic and Evolutionary Computation Conference. to appear.

[23] Eduardo Carvalho Pinto and Carola Doerr. 2018. 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. 29–41.

[24] Jonathan Rowe and Michael Voise. 2011. Unbiased Black Box Search Algorithms. In Proceedings of Genetic and Evolutionary Computation Conference. 2035–2042.

[25] Danilo Sipoli Sanches, Darrell Whitley, and Renato Tinós. 2017. Improving an exact solver for the traveling salesman problem using partition crossover. In Proceedings of Genetic and Evolutionary Computation Conference. 337–344.

[26] Jens Scharnow, Karsten Tinnefeld, and Ingo Wegener. 2004. The analysis of evolutionary algorithms on sorting and shortest path problems. Journal of Mathematical Modelling and Algorithms 3 (2004), 349–366.

[27] Dirk Sudholt. 2005. Crossover is provably essential for theising model on trees. In Proceedings of Genetic and Evolutionary Computation Conference. 1161–1167.

[28] Dirk Sudholt. 2017. How Crossover Speeds up Building Block Assembly in Genetic Algorithms. Evolutionary Computation 2 (2017), 237–274.

[29] Andrew M. Sutton and Frank Neumann. 2014. Runtime Analysis of Evolutionary Algorithms on Randomly Constructed High-Density Satisfiable 3-CNF Formulas. In Proceedings of Parallel Problem Solving from Nature. Number 8672 in Lecture Notes in Computer Science. 942–951.

[30] Ingo Wegener. 2003. Methods for the analysis of evolutionary algorithms on sorting and shortest path problems. In Foundations of Genetic Algorithms. 1–14. https://doi.org/10.1145/1967654.1967656

[31] L. Darrell Whitley, Francisco Chicano, and Brian W. Goldman. 2016. Gray box optimization for Mk landscapes (NK landscapes and MAX-kSAT). Evolutionary Computation 24, 2 (2016), 491–519.