

# Directed Trees: A Powerful Representation for Sorting and Ordering Problems

Benjamin Doerr

Edda Happ \*

## Abstract

We present a simple framework for dealing with search spaces consisting of permutations. To demonstrate its usefulness, we build upon it a simple  $(1 + 1)$ -evolutionary algorithm for one of the most fundamental problems in computer science, namely the problem of sorting  $n$  pairwise comparable items. We give a rigorous proof that the optimization time is at most  $O(n^2)$  with high probability. Our experimental evaluation shows that it is much better, namely around  $O(n \log n)$ . This compares favorably with the currently best  $(1 + 1)$ -EAs for sorting, for which an optimization time of  $O(n^2 \log n)$  was proven (Scharnow, Tinnefeld and Wegener (2004)) and one of similar order is observed experimentally in this work.

Our approach has the particular advantage that it does distinguish between wrong and unexplored information. This allows to retrieve partial, correct information even before the optimal solution has been found.

## 1 Introduction

Generic randomized search heuristics proved to be highly successful in solving diverse algorithmic problems. Their strength from the practitioner's point of view is that such algorithms are composed of generic parts (e.g., representations, mutation operators, fitness functions) that can easily be reused. Also, the hope is that an expert in such methods can easily solve algorithmic problems by plugging together suitable generic components without fully analyzing the problem itself. Of course, to this aim suitable representations and mutation operators must be known. A series of papers on the Euler tour problem [3–5] demonstrates how more adequate representations yield better algorithms.

In this work, we develop a new representation for permutations. Noting that there is a natural one-to-one correspondence between permutations and linear orders, we extend the search space to a larger set of orders that can be represented by trees. This allows to distinguish between information that has already been found by the algorithm and yet unknown information. This representation admits a natural mutation operator, namely choosing two elements having the same father in the partial order and making one the father of the other.

Building on this framework, we obtain a natural  $(1 + 1)$ -evolutionary algorithm for the classical problem of sorting  $n$  elements. As we shall see, this algorithm is significantly faster than previous evolutionary approaches to the sorting problem. Also, it is relatively robust, that is, several classical fitness functions can be used without worsening the optimization time. Finally, by distinguishing between wrong and unknown information, we may extract some reliable information already before the algorithm has found the terminal solution.

### 1.1 Previous Work

To the best of our knowledge, there is only one theoretical investigation on how to solve the sorting problem via evolutionary means. Scharnow, Tinnefeld and Wegener [7] work with the search space of all permutations  $\pi = (\pi(1), \dots, \pi(n))$ . They propose two different mutation operators, namely  $\text{EXCHANGE}(i, j)$  and  $\text{JUMP}(i, j)$ . The former operator swaps the elements in positions  $i$  and  $j$ , whereas the later one places the element in position  $i$  in position  $j$  and moves the elements in between one position towards  $i$ .

---

\*Max-Planck-Institut für Informatik, Campus E 1 4, 66123 Saarbrücken, Germany

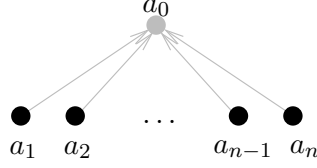


Figure 1: The initial solution. The elements of  $X = \{a_1, \dots, a_n\}$  are incomparable in this order.

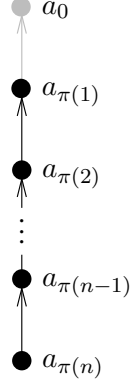


Figure 2: A permutation. If  $a_{\pi(i)} \leq a_{\pi(i+1)}$  for all  $1 \leq \pi(i) < n$  this is the optimal sorted solution.

To determine how close the current solution is to being sorted, they use several well-known measures of presortedness known in adaptive sorting [6] as fitness functions. These fitness functions are  $\text{HAM}(\pi)$  which is the number of elements in the correct position (the Hamming distance),  $\text{EXC}(\pi)$  which is the number of exchanges necessary to sort the sequence,  $\text{INV}(\pi)$  which measures how many pairs of elements  $(\pi(i), \pi(j))$  are in the wrong order,  $\text{LAS}(\pi)$  which is the length of the longest ascending subsequence, and  $\text{RUN}(\pi)$  which is the number of maximal sorted blocks (called runs).

For all of the above mentioned fitness functions they give a lower bound on the expected number of fitness function evaluations (i.e., the optimization time) of  $\Omega(n^2)$  independent of whether  $\text{EXCHANGE}$ ,  $\text{JUMP}$ , or both mutation operators are used. They prove an expected upper bound of  $O(n^2 \log n)$  for all fitness functions except  $\text{RUN}$ , which holds when both mutation operators are used, but for most fitness functions only one operator is used in the proof. For the combinations of the fitness functions  $\text{HAM}$  and  $\text{EXC}$  with the mutation operator  $\text{EXCHANGE}$  and of the fitness function  $\text{INV}$  with either  $\text{EXCHANGE}$  or  $\text{JUMP}$  they give a tight bound of  $\Theta(n^2 \log n)$ , and for the fitness function  $\text{RUN}$  they propose an expected exponential optimization time if only  $\text{JUMPs}$  are used.

## 1.2 Our Results

While the final sorting can conveniently be represented by a permutation, intermediate results of many sorting algorithms cannot. Therefore, a more natural view of sorting is that we start with an unsorted set of elements and successively by comparing elements add order to the set. Hence the approach we propose in this work is to use a sufficiently rich set of orders on the ground set of elements to be sorted as search space. This should include the empty order as natural initial solution and all linear orders (permutations) as possible final solutions. The advantage of a search space built on this paradigm is that by punishing incorrectly ordered element pairs in the fitness function, we can easily and in a natural manner ensure that any solution ever found contains only correct information. This means that also intermediate solutions contain some reliable information. Note that this cannot be realized with permutations only as search space.

We defer the detail to the following section, but sketch the main concepts and results now. We shall not need all orders on the ground set  $X$  (which consists of the elements to be sorted) in our search space. Since we aim at a linear order on  $X$ , we can restrict ourselves to orders that can be defined via assigning a predecessor to some elements of

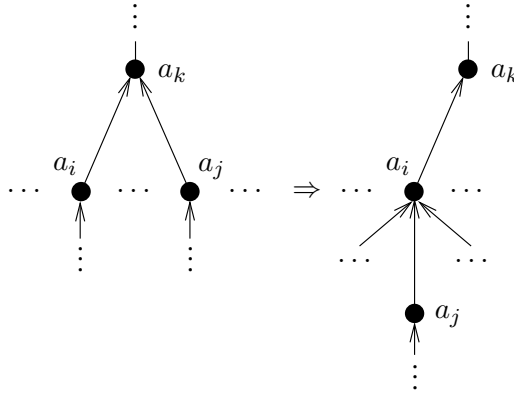


Figure 3: An elementary mutation.

the set (meaning that the predecessor is ‘smaller’ than the element itself). In our case, this leads to directed forests in which each tree is directed towards a unique root. Since dealing with separate trees may be less convenient, we add an artificial element  $a_0$  that is known to be smaller than all elements and arcs from the tree roots to this new element. This ensures that our search space can be represented by all directed trees on  $X \cup \{a_0\}$  such that the tree is directed towards its root  $a_0$ . As desired, this search space includes the empty order, represented by the tree having all elements of  $X$  as children of  $a_0$  (cf. Figure 1), and all permutations, represented by trees that are simple directed paths ending in  $a_0$  (cf. Figure 2).

This representation admits a natural local mutation operator: We choose two elements having the same predecessor (thus being sibling vertices in the tree) and make the first one the new predecessor (i.e., the father) of the second one (cf. Figure 3). We will present two ways to choose the sibling vertices.

As fitness functions, we shall use several measures of presortedness. Since we aim at having no incorrect information (an element having a predecessor that actually is larger than itself), we shall punish occurrences of such situations heavily.

As already discussed, this framework ensures that we shall only have correct information in our population. We feel that this is a very desirable feature. To analyze the algorithmic efficiency, we build a simple  $(1 + 1)$ -evolutionary algorithm from the components just discussed. As common when analyzing evolutionary algorithms, we will analyze the optimization time, i.e. the number of evaluations of the fitness function, instead of the runtime. We will see that the optimization time and the number of element comparisons (the common measure when analyzing sorting algorithms) have the same order of magnitude. We prove that our algorithm has an expected optimization time of  $O(n^2)$ , where  $n := |X|$  is the number of elements to be sorted. Note that this is already faster than the current best evolutionary solutions [7] having a proven expected optimization time of  $O(n^2 \log n)$ . Our  $O(n^2)$  bound is relatively robust with respect to the fitness function. It holds for any fitness function that punishes wrong information heavily, but does not punish finding any new information. Examples include the number of comparable pairs (in the individual) or the sum of distances between elements and their predecessors.

Next, we show that the optimization time is in fact a useful measurement of the efficiency of the algorithm, since each step of the algorithm can be computed efficiently (i.e., in constant or in logarithmic time, depending on how we choose the sibling vertices for the mutation).

On the experimental basis, our approach is even better than the  $O(n^2)$  bound. We conducted several experiments that suggest that the expected optimization time is around  $\Theta(n \log n)$ . They clearly show that it is of smaller order than  $n \log^2 n$ . To have a comparison with the previous work, we also implemented their algorithms. The results indicate that the expected optimization time coincides with the proven upper bounds of  $O(n^2 \log n)$ .

## 2 A Representation for Permutations and a $(1 + 1)$ -EA for Sorting

Given a ground set  $X = \{a_1, \dots, a_n\}$  and a total order  $\leq$  on  $X$ . The sorting problem is the problem of finding an ordered sequence  $(a_{\pi(1)}, \dots, a_{\pi(n)})$  such that  $a_{\pi(i)} \leq a_{\pi(i+1)}$  for all  $1 \leq \pi(i) < n$  where  $\pi$  is a permutation of  $\{1, \dots, n\}$ . It is well known that any sorting algorithm based on comparisons only needs in the worst case a runtime of  $\Omega(n \log n)$ . The final ordered sequence can be identified with a permutation of the elements of  $X$ . However, the intermediate results of many sorting algorithms cannot be represented this way. Thus, we consider a different search space that can represent a wider range of orders, namely directed forests where each component is directed towards its root and an arc  $(a_i, a_j)$  means  $a_j \leq a_i$ . This can nicely be achieved by assigning predecessors to some of the elements. Since it is more convenient to work with a single connected component instead of with different trees, we add an artificial element  $a_0$  not belonging to  $X$  that is known to be smaller than all other elements. All tree roots will be connected by an arc to this new element  $a_0$ . Note that not all partial orders can be represented by such a tree, however the representation can help evolutionary algorithms to find linear orders more efficiently.

We aim at algorithms that successively find new information and add it to the current solution. Using our representation and an adequate fitness function (see below) it is easy to achieve that any intermediate solution contains for any two elements of  $X$  either the correct order (if  $a_i \leq a_j$  than  $a_i$  is an ancestor of  $a_j$  in the tree) or no information. In the beginning we have no information, and thus the initial solution  $\mathcal{I}_{init}$  should be the empty order which is represented by a tree where each  $a_i$  for  $1 \leq i \leq n$  has an arc towards  $a_0$  (cf. Figure 1) instead of a random initial solution. This way, all elements of  $X$  are incomparable. The final solution will be a permutation of the elements of  $X$  which in the tree representation will be a simple directed path ending in  $a_0$  (cf. Figure 2). If for every arc  $(a_{\pi(i+1)}, a_{\pi(i)})$  it holds that  $a_{\pi(i)} \leq a_{\pi(i+1)}$ , then this permutation is sorted.

A natural local mutation operator will try to add information to the current solution by assigning a new predecessor to some vertex. Since the ordering contained in any intermediate solution is correct and we do not wish to destroy correct information, the local mutation operator picks two vertices having the same father (two sibling vertices) and makes the first one the new father of the second (cf. Figure 3). We propose and use the following two probability distributions to choose the sibling vertices. Either (i) pick one of the fathers having at least two children uniformly at random and then pick two of its children uniformly at random or (ii) choose a pair of sibling vertices uniformly at random from all pairs of sibling vertices.

A fitness function is some measure to determine how close a candidate solution  $\mathcal{I}$  is to being optimal. We will propose one fitness function here and will show later that it can be replaced by several other fitness functions. Since we search for a correct ordering of the  $n$  elements of  $X$ , it is intuitive to give a positive reward (e.g. 1) for every correctly ordered pair of vertices in the tree. Since we want to avoid incorrect orderings completely, we give a sufficiently high punishment (e.g.  $-n^2$ ) for any wrong ordering of two elements. That way, we get the following fitness function.

$$f(\mathcal{I}) := \sum_{1 \leq i, j \leq n} f(a_i, a_j)$$

where

$$f(a_i, a_j) := \begin{cases} 1 & \text{if } a_i \leq a_j \text{ and } a_i = p^k(a_j) \\ & \text{for some } 1 \leq k < n, \\ -n^2 & \text{if } a_i \leq a_j \text{ and } a_j = p^k(a_i) \\ & \text{for some } 1 \leq k < n, \\ 0 & \text{otherwise} \end{cases}$$

and

$$p^k(a_j) = \overbrace{p(p(\dots p(a_j)))}^k = a_i \text{ for some } 1 \leq k < n \\ \text{if } a_i \text{ is an ancestor of } a_j.$$

```

(1 + 1)-EA FOR SORTING
Initialization:
1  $\mathcal{I} \leftarrow \mathcal{I}_{init}$  where  $\mathcal{I}_{init}$  is the empty order.
2 repeat
    Mutation:
3     Pick  $S$  according to  $\Pr[S = k] = \frac{1}{e \cdot k!}$ .
4      $\mathcal{I}^0 \leftarrow \mathcal{I}$ 
5     for  $\ell = 1$  to  $S + 1$ 
6         do
7             Choose two sibling vertices  $a_i, a_j$ 
            by Dist1 or Dist2.
8             Generate  $\mathcal{I}^\ell$  from  $\mathcal{I}^{\ell-1}$  by making  $a_i$ 
9             the father of  $a_j$ .
    Selection:
10    if  $f(\mathcal{I}^{S+1}) \geq f(\mathcal{I})$ 
11    then  $\mathcal{I} \leftarrow \mathcal{I}^{S+1}$ 
12    until  $\mathcal{I}$  is optimal

Dist1:
13 Choose an element having at least 2 children uniformly at
    random.
14 Choose 2 of the children uniformly at random.

Dist2:
15 Choose 2 sibling vertices uniformly at random from all
    pairs of sibling vertices.

```

Figure 4: The  $(1 + 1)$ -EA for sorting.

The value of this fitness function is 0 for the initial individual and  $\frac{1}{2}n(n - 1)$  for the optimal one. We will see in Section 3 that we can use a number of other (and possibly easier) fitness functions instead and in Section 4 how the fitness function can be computed efficiently.

Given the described representation, local mutation operator and fitness function, the following  $(1 + 1)$ -evolutionary algorithm for sorting (from now on called  $(1 + 1)$ -EA) naturally arises from these components. It starts with the initial solution  $\mathcal{I} = \mathcal{I}_{init}$  described above. This solution  $\mathcal{I}$  is modified by a mutation step to get a new solution  $\mathcal{I}'$ . The mutation step of the “classical”  $(1 + 1)$ -evolutionary algorithm on bit-strings of length  $n$  flips each bit with probability  $\frac{1}{n}$ . Since this is not possible for more elaborate problems, we try to simulate this behavior. Hence, as has been proposed in [7], we pick a number  $S$  at random according to a Poisson distribution<sup>1</sup>  $\Pr_\lambda[S = k] = \frac{\lambda^k}{k!} e^{-\lambda}$  with parameter  $\lambda = 1$  and repeat the local mutation described above  $S + 1$  times on  $\mathcal{I}$ . A selection step replaces  $\mathcal{I}$  by  $\mathcal{I}'$  if the fitness  $f(\mathcal{I}')$  of the new solution  $\mathcal{I}'$  is not worse than the fitness of  $\mathcal{I}$ . Together with the above described fitness function this selection step assures that no solution containing incorrectly ordered vertex pairs is ever accepted. The mutation and selection steps are repeated until the optimal solution is found. Pseudo-code for the  $(1 + 1)$ -EA for sorting  $n$  elements is given in Figure 4.

The  $(1 + 1)$ -EA has several benefits over the evolutionary algorithm using permutations proposed by Scharnow, Tinnefeld, and Wegener [7]. For one, using the tree representation our algorithm can also be used to find a wider range of partial orders. Another advantage is that even if the algorithm has not finished sorting the elements completely, the preliminary result is still useful, as for every pair of elements it either gives the correct order or no order. Last but not least, we will see that using this data structure, we outperform the algorithms based on permutations.

<sup>1</sup>We use the Poisson distribution with  $\lambda = 1$  since it is the limit of the Binomial distribution for  $n$  trials with probability  $\frac{1}{n}$  each

### 3 Analysis of the Optimization Time

In this section, we prove bounds for the optimization time of the  $(1+1)$ -EA, that is, the number of fitness evaluations until the optimal solution is found. We show that an upper bound of  $O(n^2)$  holds with overwhelming probability. A natural  $\Omega(n \log n)$  lower bound is derived from classical theory. Finally, we show that both bounds in fact hold for any fitness function that rewards finding additional information but forbids accepting wrong information.

For the upper bound analysis we need the following Chernoff bound [1].

**Theorem 1** (Chernoff Bound). *Let  $X_1, \dots, X_t$  be mutually independent random variables with  $\Pr[X_i = 1] = p$  and  $\Pr[X_i = 0] = 1 - p$  for all  $i$ . Let  $X := \sum_{i=1}^t X_i$ . Then for all  $\delta \in (0, 1]$ ,*

$$\Pr[X < (1 - \delta)\mathbb{E}[X]] < \exp\left(-\frac{1}{2}\delta^2\mathbb{E}[X]\right).$$

Now we can prove the following theorem, which holds regardless of which of the two probability distributions proposed in the previous section is used to choose the two elements for the mutation step.

**Theorem 2.** *As long as the probability to pick the sibling vertices  $a_i$  and  $a_j$  for the mutation step is the same as to pick  $a_j$  and  $a_i$ , the  $(1+1)$ -EA needs with overwhelming probability<sup>2</sup> at most  $O(n^2)$  steps to find the optimal solution.*

*Proof.* For any two elements  $a_i$  and  $a_j$  the probability to choose  $a_i$  and  $a_j$  in a local mutation, and thus to make  $a_i$  the predecessor of  $a_j$ , is the same as the probability to choose  $a_j$  and  $a_i$ . Since one of the mutations increases the fitness by at least one and the other one is rejected, with a probability of  $\frac{1}{2}$  a local mutation increases the fitness by at least one.

In the previous section we have seen that the fitness of the initial solution is 0 and the fitness of the optimal solution is  $\frac{1}{2}n(n-1)$ . If the  $(1+1)$ -EA has not found the optimal solution yet, the fitness of the individual is smaller than  $\frac{1}{2}n(n-1)$ . Since every step that is accepted increases the fitness by at least 1, the fitness has then been increased by less than  $\frac{1}{2}n(n-1)$  mutation steps. We show for any constant  $\lambda > e$  that if the  $(1+1)$ -EA did  $t = \lambda n(n-1)$  steps with overwhelming probability at least  $\frac{1}{2}n(n-1)$  steps would be accepted mutation steps performing a single local mutation.

Let  $t'$  be the number of mutation steps the  $(1+1)$ -EA needs to find the optimal solution. We count the number of accepted mutation steps that consist of a single local mutation. For that we define  $\{0, 1\}$ -valued random variables  $X_i$  for  $1 \leq i \leq t'$  by  $X_i = 1$  if the  $i$ -th mutation step of the  $(1+1)$ -EA consists of a single local mutation and increases the fitness and  $X_i = 0$  otherwise. The probability that the  $i$ -th mutation step consists of a single local mutation is  $\frac{1}{e}$  by definition of the Poisson distribution. Thus  $\Pr[X_i = 1] = p := \frac{1}{2e}$  and  $\Pr[X_i = 0] = 1 - p$ . Let  $\lambda > e$  be a constant. If  $\lambda n(n-1) > t'$  define the mutually independent random variables  $X_i$  for  $t' < i \leq \lambda n(n-1)$  as  $\Pr[X_i = 1] = p$  and  $\Pr[X_i = 0] = 1 - p$ . Then, all  $X_i$  are mutually independent. Let  $X := \sum_{i=1}^{\lambda n(n-1)} X_i$ , thus the expected value of  $X$  is  $\mathbb{E}[X] = \frac{\lambda}{2e}n(n-1)$ .

Thus, if the  $(1+1)$ -EA has not found the optimal solution after  $\lambda n(n-1)$  steps, the fitness has been increased by less than  $\frac{1}{2}n(n-1)$  mutation steps, and  $X < \frac{1}{2}n(n-1)$ . If we use  $\delta = \frac{\lambda - e}{\lambda}$ , we have that  $\delta \in (0, 1]$  and  $(1 - \delta)\mathbb{E}[X] = \frac{1}{2}n(n-1)$ . Thus, we can use Theorem 1 to get

$$\begin{aligned} & \Pr \left[ \begin{array}{l} \text{The } (1+1)\text{-EA has not found the} \\ \text{optimal solution after } \lambda n(n-1) \text{ steps} \end{array} \right] \\ & \leq \Pr[X < \frac{1}{2}n(n-1)] \\ & = \Pr[X < (1 - \delta)\mathbb{E}[X]] \\ & < \exp\left(-\frac{1}{2}\delta^2\mathbb{E}[X]\right) \\ & = \exp\left(-\frac{(\lambda - e)^2}{4e\lambda}n(n-1)\right). \end{aligned}$$

Thus, we have shown that with overwhelming probability the  $(1+1)$ -EA needs at most  $O(n^2)$  mutation steps to find the optimal solution.  $\square$

<sup>2</sup>“With overwhelming probability” means that an event happens with probability at least  $1 - 2^{-\Omega(n^\epsilon)}$  for a constant  $\epsilon > 0$ .

**Theorem 3.** *In the worst case, the  $(1 + 1)$ -EA needs at least an expected number of  $\Omega(n \log n)$  steps to find the optimal solution.*

*Proof.* A mutation step chooses a random number  $S$  drawn from a Poisson distribution  $\text{Pois}(\lambda = 1)$  with parameter  $\lambda = 1$  and performs  $S + 1$  local mutations doing one comparison each. Since the expected value for a Poisson distribution is  $\lambda$ , a mutation step performs in expectation two comparisons. Thus, if the  $(1 + 1)$ -EA performs  $t$  mutation steps to find the optimal solution, then it performs in expectation  $2t$  comparisons. Since any randomized algorithm needs at least an expected number of  $\Omega(n \log n)$  comparisons [2], the  $(1 + 1)$ -EA needs in expectation at least  $\Omega(n \log n)$  mutation steps to find the optimal solution.  $\square$

This proof also reveals that the optimization time and the number of element comparisons, the common measure for analyzing sorting algorithms, have the same order of magnitude.

Although the fitness function given in Section 2 is the natural choice for a fitness function, it does not appear to be very easy to analyze. However, we can easily choose a different fitness function, as long as we follow certain restrictions.

**Lemma 4.** *If the  $(1 + 1)$ -EA uses a fitness function  $f'$  that awards additional correct order information with a non-negative reward and prevents the acceptance of incorrect order information, the behavior of the  $(1 + 1)$ -EA is the same as if it uses the original fitness function  $f$ .*

*Proof.* Consider a mutation step consisting of  $S + 1 \geq 1$  local mutations. A local mutation chooses two sibling vertices and makes one the father of the other. Differently put, it chooses two elements that are incomparable in the current solution and makes them comparable. Since there is no way to make two comparable elements incomparable, no local mutation can destroy the effects of another local mutation. Hence, if one of the local mutations introduces incorrect order information into the candidate solution, no other local mutation can undo the wrong information, and both fitness function,  $f$  and  $f'$ , assure the rejection of the new individual.

Otherwise, only correct order information is introduced, and thus the new individual has a fitness that is greater than the fitness of the old individual by a non-negative amount, and thus is accepted using either of the two fitness functions.

Thus we have proven that if the original fitness function  $f$  accepts or rejects the new individual so does the alternative fitness function  $f'$ .  $\square$

Note that we only assure that the acceptance behavior of  $f$  and  $f'$  is the same, meaning  $f(\mathcal{I}') \geq f(\mathcal{I})$  if and only if  $f'(\mathcal{I}') \geq f'(\mathcal{I})$ . However, the fitness values of  $f$  and  $f'$  can differ for the same individual. Hence Lemma 4 is not transferable to evolutionary algorithms working with more than one candidate solution in the population and one as offspring.

## 4 Implementation Details and Analysis of the Actual Runtime

In this section we show that each component of the  $(1 + 1)$ -EA can be computed highly efficiently. That implies that the bounds for the optimization time given in the previous section also hold up to a constant factor for the runtime if we choose the father of the two sibling vertices uniformly at random. If we choose the two sibling vertices uniformly at random from all pairs of sibling vertices, the runtime increases by a factor of  $\log n$ .

We will use the following datastructures.

- Array *tree* contains a solution tree by storing for each element  $a_i$  its predecessor  $p(a_i)$ .
- Array *num\_children* contains for every element the number of its children.
- Array *num\_descendants* contains for every element the number of its descendants.
- Array *fathers* contains in arbitrary order the elements having at least 2 children.

- For every element  $a_i$  array  $children\_i$  contains its children in arbitrary order.
- The complete binary tree  $sibling\_pairs$  has the  $n$  elements to be sorted as leaves. The labels of the vertices hold the number of sibling vertex pairs that are children of one of the leaves of this subtree.

As we have seen in the previous section, the expected number of local mutations in a mutation step is 2. Thus for the expected runtime analysis it suffices to assume that only mutation steps consisting of a single local mutation are performed.

Creating the initializing the above mentioned datastructures is done once and needs  $\Theta(n)$  time. As we have assumed that only a single local mutation is performed per step, the updates on these datastructures can in every step be done in constant time. To avoid the creation of a new array  $tree$  in every step of the algorithm, we keep two copies of  $tree$  and perform a constant number of changes per step on them.

The fitness function can be computed in linear time by doing a depth-first traversal of the tree during which the tree-levels of the elements are summed up. If an incorrect ordering of some vertex and its father is found a negative number is returned. However, by remembering which local mutation step was performed, instead of recomputing the fitness function in every step, we can update the fitness function in constant time using the information stored in  $num\_descendants$ .

A local mutation step first chooses a pair of sibling vertices and then makes the first vertex the father of the second. The second part can obviously be done in constant time. We proposed two ways of choosing sibling vertices. The first one picks a father having at least two children uniformly at random and then chooses uniformly at random two of its children. Clearly the information stored in the arrays  $fathers$  and  $children\_i$  suffices to choose the random sibling pair in constant time. Since the arrays are not ordered, with the help of array  $num\_children$  the update can also be done in constant time. The second possibility to choose the sibling vertices is to pick a sibling pair uniformly at random from all pairs of sibling vertices. To do so, we traverse the binary tree  $sibling\_pairs$  until we find the father having the chosen pair as children and from the array  $children\_i$  we get the two sibling vertices. Clearly, finding the children as well as updating the stored information can be done in  $O(\log n)$  time.

Thus, if we use the first variant of finding the sibling vertices for the mutation step, a step of the algorithm can be performed in expected constant time, and if the second variant is used, it can be performed in expected logarithmic time.

## 5 Experimental Results

We implemented the  $(1 + 1)$ -EA described in the previous sections as well as the evolutionary algorithm for sorting presented in [7]. This section contains the results of the experiments we did on both implementations.

The bounds we proved in the previous section are not tight. However, they already show that the expected optimization time of our  $(1 + 1)$ -EA using a tree representation is at least as good as the expected optimization time of the  $(1 + 1)$ -evolutionary algorithm given in [7] which we will from now on name  $(1 + 1)$ -EA<sub>p</sub> since it uses permutations as representation. Our upper bound is  $O(n^2)$ , whereas the general lower bound of the  $(1 + 1)$ -EA<sub>p</sub> is  $\Omega(n^2)$  and  $\Omega(n^2 \log n)$  for several of the combinations of fitness function and mutation operator (in particular those combinations that seem most adequate). We will see that the real expected optimization time of our algorithm is much closer to the lower bound of  $\Omega(n \log n)$  than to the upper bound. The expected optimization time of the  $(1 + 1)$ -EA<sub>p</sub> appears to be close to  $\Theta(n^2 \log n)$  as the proven bounds indicate.

As described in Section 2, a mutation step of our evolutionary algorithm chooses two elements and makes the first one the father of the second. The two ways we proposed to choose these elements are (i) choosing the father of the two elements uniformly at random or (ii) choosing a sibling pair uniformly at random from all pairs of sibling vertices. We implemented both variants of the algorithm and did test runs. For each variant we let the algorithm solve 100 instances for the inputs sizes 100, 300,  $\dots$ , 5900. The results (cf. Figure 5) clearly indicate that the average optimization time of our algorithm is only slightly higher than linear in the number of elements to be sorted. Figure 6 shows the same optimization times divided by  $n \log n$  in the first graph and divided by  $n \log^2 n$  in the second graph. These graphs show that the real expected optimization time seems to lie between  $\Omega(n \log n)$  and  $O(n \log^2 n)$ .

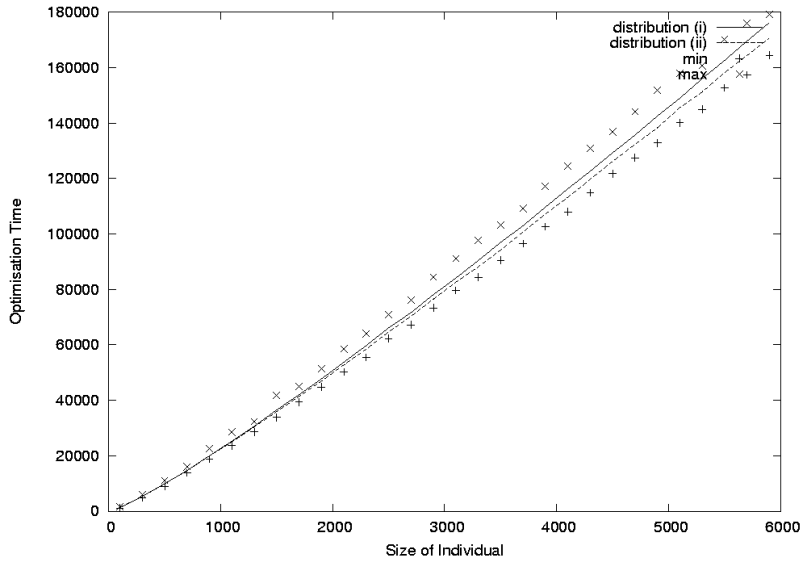


Figure 5: The graph shows the optimization times for the  $(1 + 1)$ -EA using the distributions (i) and (ii) in the mutation step, averaged over 100 runs each. Additionally the minimum and maximum optimization times over all 200 runs are given.

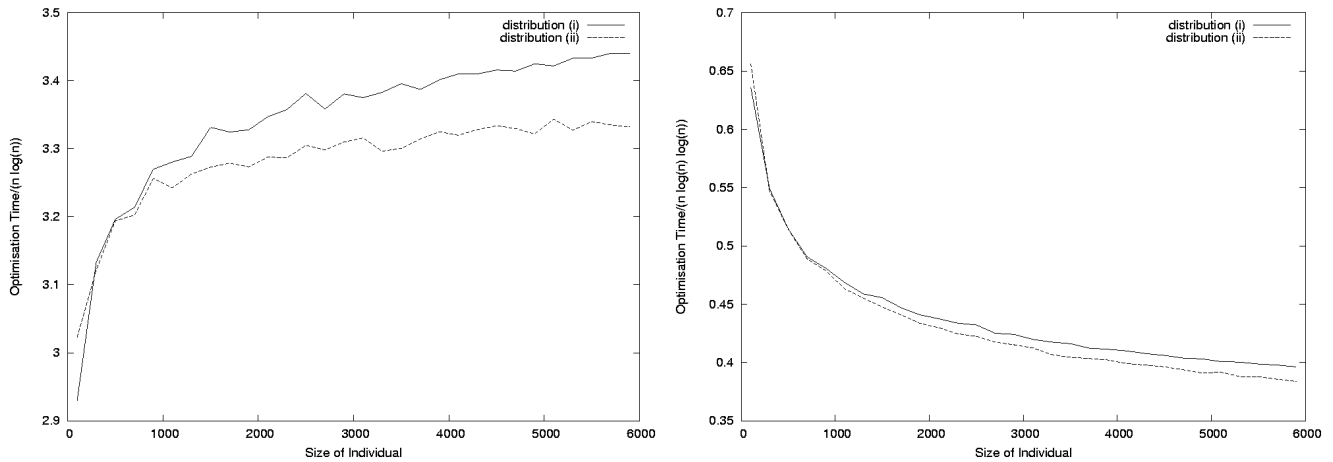


Figure 6: The graphs show the average optimization times given in Figure 5 divided by  $n \log n$  and  $n \log^2 n$ . The plots indicate that the actual optimization time is around  $\Theta(n \log n)$ .

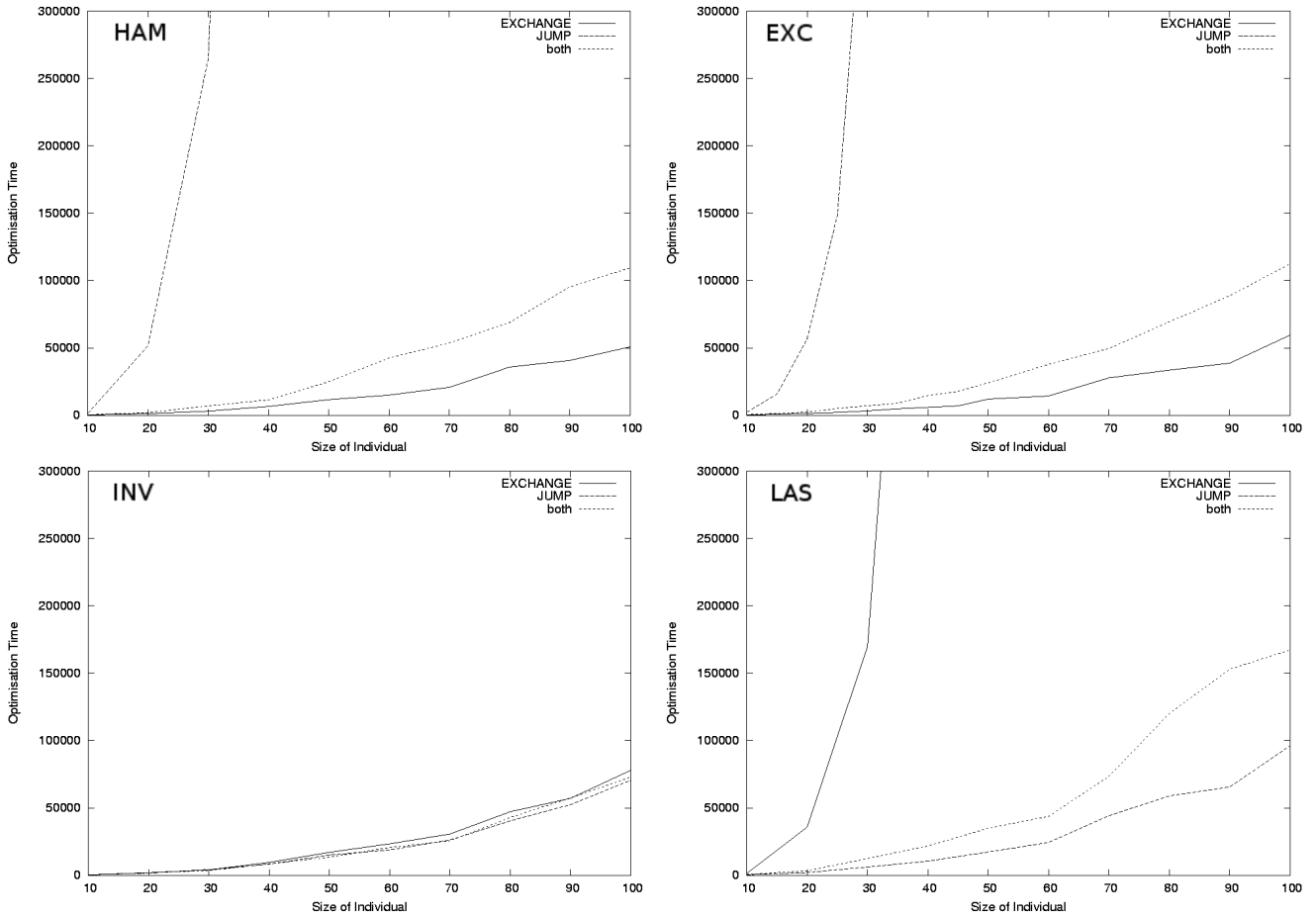


Figure 7: The first graph shows the average optimization times of the  $(1 + 1)$ -EA<sub>p</sub> using the fitness function HAM and either only the JUMP or only the EXCHANGE operator or both. The next graphs show the same for the  $(1 + 1)$ -EA<sub>p</sub> using the fitness functions EXC, INV, and LAS. Note that, allowing e.g. an optimization time of 50000, all algorithms can only handle instances with  $n \leq 100$  whereas our algorithm works up to  $n \leq 2000$  (cf. Figure 5).

The  $(1 + 1)$ -EA<sub>p</sub> to which we compare our algorithm comes in many flavors. For one, 5 different fitness functions are used (cf. also Section 1.1), namely HAM, EXC, INV, LAS, and RUN. Additionally, two different mutation operators, JUMP and EXCHANGE, were proposed.

First we wanted to find out for each fitness function whether it is best to use both mutation operators or only one of them, and if so which one. To this aim, we let the  $(1 + 1)$ -EA<sub>p</sub> solve for each fitness function except RUN 10 random instances for the input sizes 10, 20, . . . 100 either using only JUMP or only EXCHANGE or using both with probability  $\frac{1}{2}$  each (cf. Figure 7). Since RUN has a very high optimization time, we chose to let this variant of the  $(1 + 1)$ -EA<sub>p</sub> solve 10 random instances for the input sizes 2, 4, . . . , 20 (cf. Figure 8). The results indicate that for the following experiments we should choose the mutation operator EXCHANGE for the fitness functions HAM and EXC and for the fitness functions LAS and RUN only JUMP. Also for the fitness function INV is the variant using only JUMP best, however, INV is the only fitness function where all three variants are comparable. Note that the tight bounds in [7] were given for HAM and EXC using only the EXCHANGE operator and for INV using either operator, which coincides with the best mutation operators we determined for these fitness functions.

Having found the adequate mutation operators for each fitness function, we chose to first have 10 runs for each fitness function on input sizes  $\{1, 2, \dots, 19\}$ . Since RUN seems to have an expected exponential optimization time, we then let the  $(1 + 1)$ -EA<sub>p</sub> for the other fitness functions make 10 runs of the input sizes between 10 and 300 with a step size of 10. As can easily be seen (cf. Figure 9), RUN has a very high and probably exponential expected

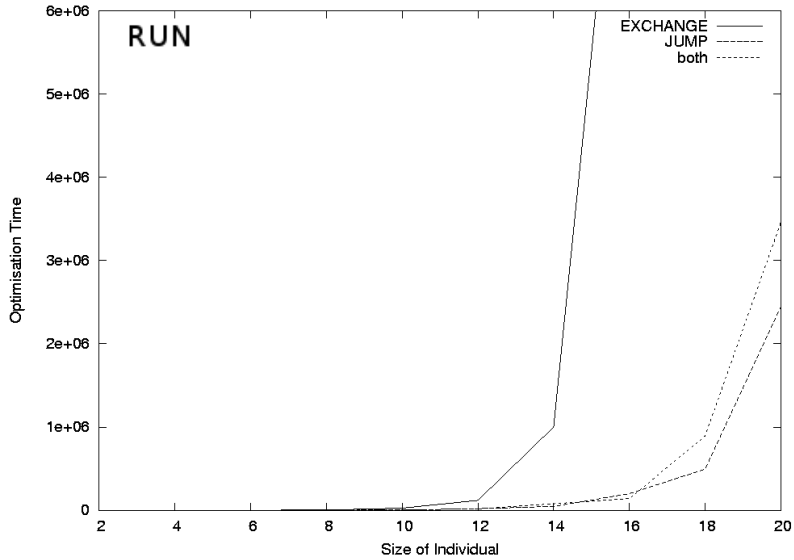


Figure 8: The graph shows the average optimization times of the  $(1 + 1)$ -EA<sub>p</sub> using the fitness function RUN and either only the JUMP or only the EXCHANGE operator or both.

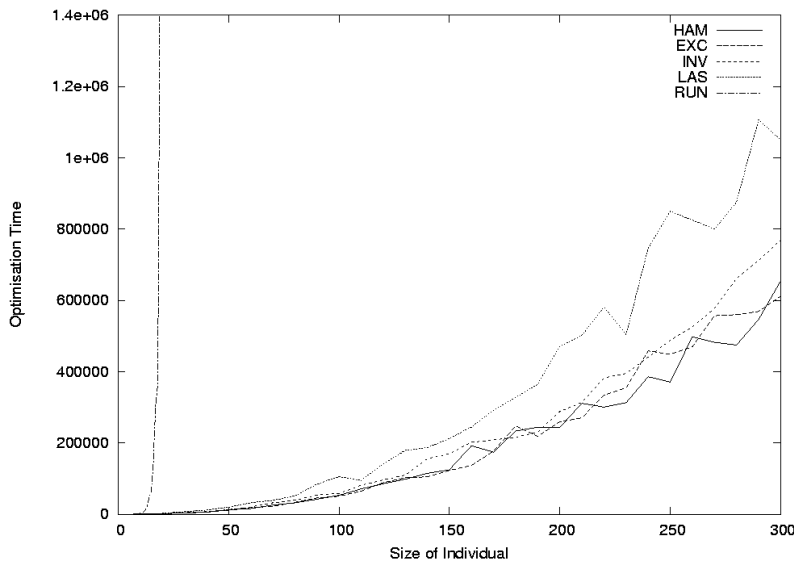


Figure 9: The graph shows the average optimization times of the  $(1 + 1)$ -EA<sub>p</sub> for all fitness functions using the best mutation operator for the respective fitness function.

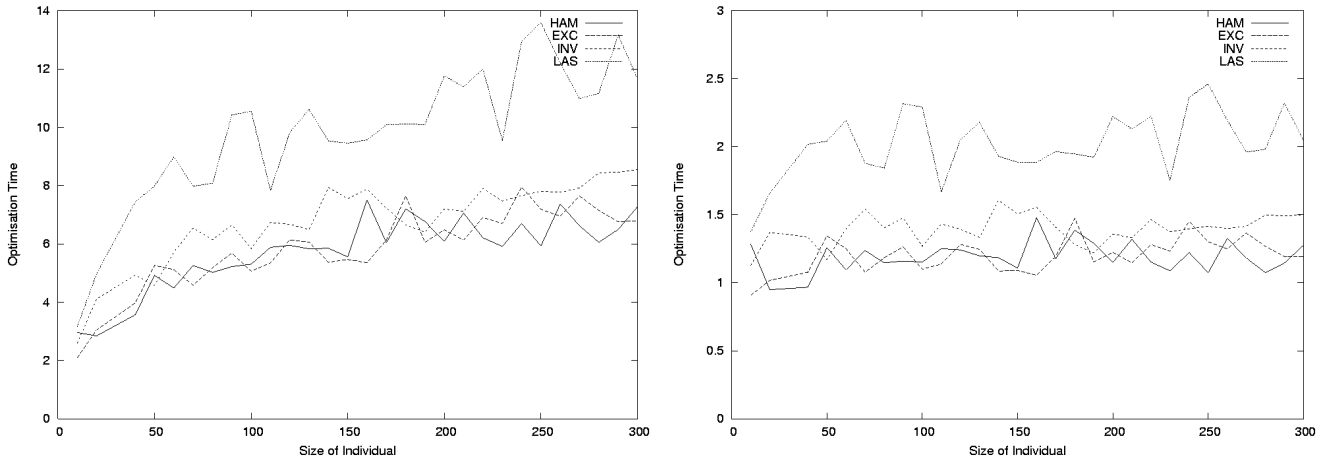


Figure 10: The graphs show the average optimization times of the  $(1 + 1)$ -EA<sub>p</sub> for all fitness functions except RUN divided by  $n^2$  (first graph) and by  $n^2 \log n$  (second graph).

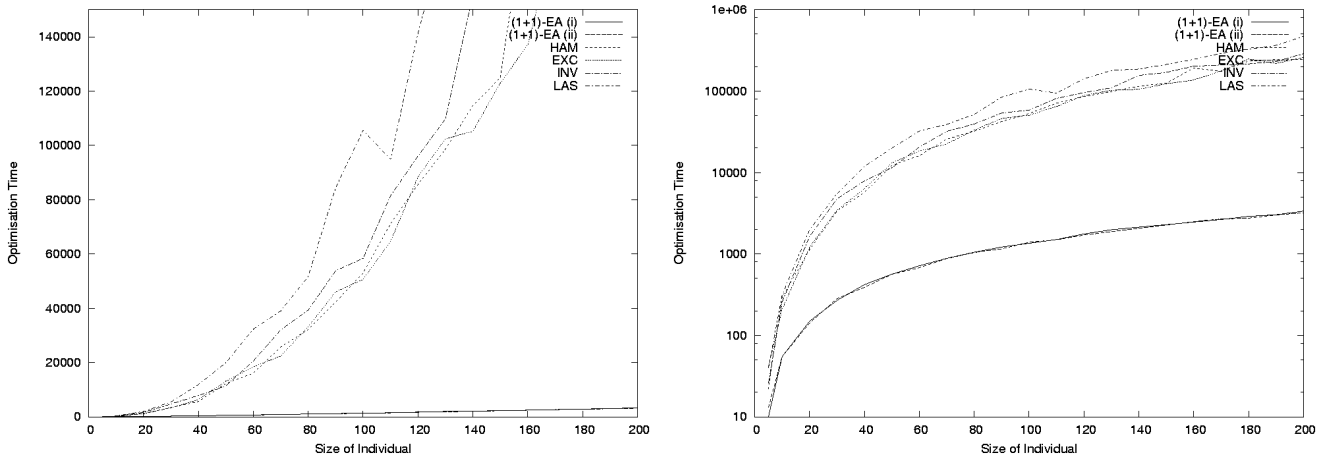


Figure 11: The graphs show the average optimization times of the  $(1 + 1)$ -EA versus the average optimization times of the  $(1 + 1)$ -EA<sub>p</sub>, the first one scaling the  $y$ -axis linearly, the second one logarithmically.

runtime, as suggested in [7]. The others seem to have an expected optimization time of  $O(n^2 \log n)$  as Figures 9 and 10 indicate.

Finally, we directly compare our  $(1 + 1)$ -EA and the  $(1 + 1)$ -EA<sub>p</sub>. For this, we let the  $(1 + 1)$ -EA (with both variants for the mutation step) and the  $(1 + 1)$ -EA<sub>p</sub> (with the fitness functions HAM, EXC, INV, and LAS and the according mutation operators) solve 10 random instances for the input sizes 10, 20,  $\dots$ , 200 (cf. Figure 11). Note that even using the logarithmic scale it is barely possible to distinguish the two variants of the  $(1 + 1)$ -EA. The comparison clearly shows that our algorithm outclasses the  $(1 + 1)$ -EA<sub>p</sub>.

## 6 Conclusion and Future Work

In this paper, we suggested a novel approach to problems where the final solution is a permutation (linear order). When applied to the problem of sorting  $n$  comparable items, it has the structural advantage of disallowing incorrect information. A simple  $(1 + 1)$ -EA built upon this framework was faster than previous such algorithms both theoretically (proven asymptotic bounds) and experimentally. A problem left open in this work is giving a mathematical proof that the evolutionary algorithm proposed indeed has an expected optimization time of  $\Theta(n \log n)$  as observed in the experiments. Another research direction is extending our framework to problems of finding non-linear orders

from pairwise comparisons. We are optimistic that our approach via orders instead of permutations turns out to be useful here, too.

## Acknowledgement

The authors thank an unknown friendly reviewer for his thorough reading of this paper and the proposed improvements.

## References

- [1] N. Alon and J. H. Spencer. *The Probabilistic Method*. Wiley, 2nd edition, 2000.
- [2] T. H. Cormen. *Introduction to algorithms*. MIT Press, Cambridge, Mass., 2. ed., 4. print. edition, 2003.
- [3] B. Doerr, N. Hebbinghaus, and F. Neumann. Speeding up evolutionary algorithms through restricted mutation operators. In T. P. Runarsson, H.-G. Beyer, E. K. Burke, J. J. M. Guervs, L. D. Whitley, and X. Yao, editors, *PPSN*, volume 4193 of *Lecture Notes in Computer Science*, pages 978–987. Springer, 2006.
- [4] B. Doerr and D. Johannsen. Adjacency list matchings — an ideal genotype for cycle covers. In *Genetic and Evolutionary Computation Conference (GECCO-2007)*, pages 1203–1210. ACM, 2007.
- [5] B. Doerr, C. Klein, and T. Storch. Faster evolutionary algorithms by superior graph representation. In *FOCI*, pages 245–250. IEEE, 2007.
- [6] O. Petersson and A. Moffat. A framework for adaptive sorting. *Discrete Appl. Math.*, 59(2):153–179, 1995.
- [7] J. Scharnow, K. Tinnefeld, and I. Wegener. The analysis of evolutionary algorithms on sorting and shortest paths problems. *Journal of Mathematical Modelling and Algorithms*, 3(4):349–366, 2004.