Do Sophisticated Evolutionary Algorithms Perform Better than Simple Ones?

Michael Foster, Matthew Hughes, George O'Brien, Pietro S. Oliveto, James Pyle, Dirk Sudholt, James Williams
Department of Computer Science, University of Sheffield, Sheffield, UK

ABSTRACT

Evolutionary algorithms (EAs) come in all shapes and sizes. Theoretical investigations focus on simple, bare-bones EAs while applications often use more sophisticated EAs that perform well on the problem at hand. What is often unclear is whether a large degree of algorithm sophistication is necessary, and if so, how much performance is gained by adding complexity to an EA. We address this question by comparing the performance of a wide range of theory-driven EAs, from bare-bones algorithms like the (1+1) EA, a (2+1) GA and simple population-based algorithms to more sophisticated ones like the (1+(λ, λ)) GA and algorithms using fast (heavy-tailed) mutation operators, against sophisticated and highly effective EAs from specific applications. This includes a famous and highly cited Genetic Algorithm for the Multidimensional Knapsack Problem and the Parameterless Population Pyramid for Ising Spin Glasses and MaxSat. While for the Multidimensional Knapsack Problem the sophisticated algorithm performs best, surprisingly, for large Ising and MaxSat instances the simplest algorithm performs best. We also derive conclusions about the usefulness of populations, crossover and fast mutation operators. Empirical results are supported by statistical tests and contrasted against theoretical work in an attempt to link theoretical and empirical results on EAs.

1 INTRODUCTION

Numerous successful applications of Evolutionary Algorithms (EAs) to real world optimisation problems have been reported (see, e. g. [1, 4, 13]). Nevertheless, the reasons behind these successes are not well understood. In particular, given an optimisation problem, it is difficult to predict which class of EAs will be successful for that application and which parameter settings to use.

Considerable advances have been made in the theoretical analysis of EAs in recent years. Nowadays it is possible to rigorously analyse the time complexity of sophisticated EAs using populations, mutation, crossover and several elitist and stochastic selection mechanisms [6]. Such results are available for standard algorithms including the simple genetic algorithm (SGA) [22, 23] introduced by Goldberg [14] and standard steady state GAs [7, 9, 19, 27]. However, most of these analyses are performed for simple benchmark functions, such as OneMax and LeadingOnes, designed to capture specific characteristics of optimisation problems. Such results do not yet allow us to draw conclusions on the performance of standard EAs for more realistic optimisation problems.

Classical NP-hard combinatorial optimisation problems make more realistic testbeds for estimating the performance of EAs. Concerning such problems, though, most of the available theoretical results relate to extremely simplified EAs that do not use populations or crossover operators [21]. As a result, the current understanding of the performance of realistic EAs for combinatorial optimisation relies on the vast array of available experimental results [1, 4, 13].

To counteract the lack of understanding of EA behaviour, trial and testing of various algorithms and parameter settings is generally required to identify an algorithm with satisfying performance. Overall, by searching for an algorithm that produces good solutions in short time, such efforts effectively *tailor* the algorithm to the problem. At the same time some of the generality of EAs is lost, although their general applicability (i.e., their problem independence) is one of the major advantages and strengths of EAs.

Most experimental papers only report the final sophisticated algorithm tailored to the problem at hand and compare its performance to either standard EAs or problem specific algorithms for which good performance has been previously reported. A gap is left between the asymptotic runtime bounds available from theory concerning simplified EAs and the experimental results for the tailored EAs. What is particularly unclear is the amount of gain achieved by the algorithm sophistication, in terms of how much better these algorithms perform compared to simple evolutionary algorithms. In this paper we attempt a first effort to bridge this gap by comparing the successful algorithms reported in the literature for some standard combinatorial optimisation problems against a range of EAs of increasing complexity. Starting from the bare-bones (1+1) EA and gradually introducing features of more sophisticated EAs (i.e., offspring and parent populations and crossover) up to standard steady state GAs [13]. We also compare against the performance of recent theory-guided EAs such as the (1+(λ, λ)) GA [10] and *fast GAs* [11] that use a heavy-tailed mutation operator. Our aim is to contribute towards filling in the gap between the available theoretical and experimental results in evolutionary computation.

Our first aim was to identify a combinatorial optimisation problem where a standard EA with parameters tuned appropriately and the addition of a few problem specific modifications has excellent performance. For this purpose the Multidimensional Knapsack Problem (MKP) was selected, for which excellent performance has been reported for a standard steady state (μ+1) GA with genotype diversity, a problem specific repair mechanism and specialised initialisation [2]. Our experimental analysis confirms the very good performance of the algorithm and reveals how algorithmic features such as populations, crossover, diversity and higher-than-standard mutation rates are indeed necessary since simple algorithms with these features outperform their counterparts without them.

In the second part of the paper we switch our attention to the recently popular Parameterless Population Pyramid (P3) algorithm [15, 16] that combines hill climbing, a novel population management strategy and advanced crossover operators inspired by the Linkage Tree Genetic Algorithm (LTGA) [28]. Without needing to adjust any parameter values, P3 has been shown to be particularly successful for standard combinatorial optimisation problems including MaxSat and Ising Spin Glasses (ISG) [16]. Surprisingly, our experiments show that unless global optima are sought (which may be prohibitive for NP-hard problems) simple bare-bones EAs outperform more sophisticated ones including the highly-sophisticated parameterless algorithms for instances of ISG and MaxSat previously used in the literature to assess the performance of P3.

The rest of the paper is structured as follows. In the following section we define precisely the untailored algorithms and our experimental setup. In Section 3 we report on the comparisons against the problem-tailored GA for MKP. In Section 4 we present the comparisons against the P3 algorithm for the Ising Spin Glass problem and for MaxSat, respectively. We finish with a summary of the drawn conclusions.

2 PRELIMINARIES

2.1 Bare-Bones Evolutionary Algorithms

The theory of evolutionary algorithms, particularly the field of rigorous runtime analysis, has focussed on simple, bare-bones versions of evolutionary algorithms. These algorithms facilitate a theoretical analysis, while still reflecting the basic working principles of evolutionary algorithms. They also provide an excellent baseline for including features of evolutionary algorithms (e. g. parent populations, offspring populations, crossover, diversity mechanisms, etc.). Algorithm 1 shows pseudocode for these algorithms; we use the term "EA" to indicate that no crossover operator is used and the term "GA" to indicate algorithms that employ it.

The simple theory-driven algorithms examined in this study are the (1+1) EA and $(\mu + \lambda)$ EAs and GAs with various values of μ and λ, chosen by performing preliminary experiments. All these algorithms choose parents uniformly at random from the population and create offspring using either crossover and mutation or mutation only. The crossover operator is a uniform crossover (i.e., each bit is picked independently from either parent uniformly at random), for which parents are selected uniformly at random with replacement. For mutation, standard bit mutation is used: flipping each of n bits independently with probability $1/n$. The final selection then picks the best μ individuals from the union of the λ offspring and the μ parents.

In addition, modern theory-inspired EAs are investigated in the form of fast GAs [11] that only differ in the choice of mutation operator (see Section 2.2) and the (1+(λ, λ)) GA [10] (see Section 2.3).

2.2 Fast Genetic Algorithms

Traditionally, evolutionary algorithms that use a bit-string representation set the mutation rate to be $1/n$ where n is the length of the bit-string. For the simple EAs and GAs in this paper we apply this mutation operator. However, for fast genetic algorithms [11], a heavy-tailed, random mutation rate α/n is used. This value is computed once per generation, by selecting α from a discrete power distribution $D_{n/2}^{\beta}$ where $\beta > 1$. Following [11], we choose $\beta = 1.5$ and α is determined according to $\Pr(\alpha = i) = i^{-\beta}/\sum_{i=1}^{n/2} i^{-\beta}$. Mutation then flips each bit independently with probability α/n. The result of this approach is a mutation rate that is normally set to low values, but can occasionally reach very high values.

Algorithm 1 Scheme of (μ+λ) EA and (μ+λ) GA

```
Randomly initialise the initial population with $\mu$ individuals
Evaluate initial population
while solution not found and max evaluations not reached do
    For GA: create $\lambda$ offspring by choosing $\lambda$ pairs of parents
    uniformly at random with replacement and applying uniform
    crossover to them
    For EA: create $\lambda$ offspring by copying $\lambda$ individuals chosen
    uniformly at random from the population
    Mutate offspring
    Evaluate offspring
    Select the best $\mu$ individuals from the union of parents and
    offspring to form the next generation
```

Genetic Algorithms using the heavy-tailed mutation operator were called *Fast Genetic Algorithms* in [11]. Following this, we call it the *fast mutation* operator and refer to the (μ+λ) EA with fast mutation as (μ+λ) Fast-EA and likewise for the (μ+λ) GA and (μ+λ) Fast-GA.

2.3 Self-adjusting (1+(λ, λ)) GA

The (1+(λ, λ)) GA [10] features a unique approach to offspring production. In each generation a set of λ offspring is produced from the parent x through mutation and are evaluated. The best individual x' is selected from the offspring and a crossover step occurs with x and x' as parents. λ children from the crossover are produced, evaluated and the best child y is selected as the final offspring.

Mutation Operator. The (1+(λ, λ)) GA flips ℓ bits chosen uniformly at random (u. a. r.) from the parent. In each generation, the step size ℓ is sampled from a binomial distribution $\text{Bin}(n, \lambda/n)$ where λ/n can be regarded as the mutation probability.

Crossover Operator. As previously mentioned, the best offspring x' is taken from the mutation stage and a crossover operator is performed between x and x' to produce λ children. The probability of a child inheriting a bit from each parent is set to $p(x) = 1 - 1/\lambda$ and $p(x') = 1/\lambda$ respectively. This is designed so that, in most cases, any offspring produced will be similar to the original parent. We use the self-adjusting variant of the (1+(λ, λ)) GA, shown in Algorithm 2. For this algorithm, the value of λ is reset accordingly for each generation:

$$\lambda' = \begin{cases} \max(\frac{\lambda}{F}, 1) & \text{if } f(y) > f(x) \\ \min(\lambda F^{\frac{1}{4}}, N) & \text{otherwise} \end{cases} \tag{1}$$

where F is the update strength and N is the problem size. In our study we set $F = 1.5$.

It was observed in [15] and [3] that the self-adjusting (1+(λ, λ)) GA algorithm can get stuck if λ grows too large. We addressed this by implementing one of the modifications to the original algorithm used in [15]: if $\lambda \geq n$, where the mutation rate has become 1.0, search is restarted with $\lambda = 1$ from a uniform random search point. Goldman and Punch justify this restart strategy by saying that "this point is only reached when the algorithm has stalled for a significant number of generations" [15].

Algorithm 2 Self-adjusting (1+(λ, λ)) GA [10]

```
Initialise and evaluate initial parent $x$; set $\lambda = 1$.
while solution not found and max evaluations not reached do
    Select step size $\ell$ from $\mathrm{Bin}(n, \lambda/n)$
    for $i = 1$ to $\lambda$ do
        Create offspring[$i$] by flipping $\ell$ bits from $x$ u. a. r.
    Select best individual $x'$ from offspring
    for $j = 1$ to $\lambda$ do
        offspring[$j$] $\leftarrow$ crossover($x'$, $x$)
    Select best individual $y$ from offspring
    if $f(y) > f(x)$ then $x \leftarrow y; \lambda \leftarrow \max\{\lambda/F, 1\}$
    if $f(y) = f(x)$ then $x \leftarrow y; \lambda \leftarrow \min\{\lambda F^{1/4}, n\}$
    if $f(y) < f(x)$ then $\lambda \leftarrow \min\{\lambda F^{1/4}, n\}$
```

2.4 Experimental Setup

In all experiments we stop each algorithm after a time budget of 10,000 fitness evaluations and record averages of the best fitness in the final population. This scheme is repeated for increasing problem sizes. The reason for recording the number of function evaluations is that in practical applications this is often the computationally most expensive operation. In contrast to wall-clock times, this measure is also independent from the actual hardware used. Theoretical runtime analyses conventionally study the number of function evaluations, giving us a solid baseline for discussing the results.

The time budget of 10,000 fitness evaluations is fixed, while the problem size is varied. This means that for small problems, we expect all algorithms to find high-quality solutions with ease, whereas for large problems we may not expect to find global optima in the allotted time. Since we use a range of different problem sizes, our experiments include problems that are easy and hard under the allocated time budget. This effect is particularly evident for Ising Spin Glasses and MaxSat, where we will see that the solution quality deteriorates with increasing problem size (see Section 4). For the MKP the time budget seems generous enough to allow all algorithms to find solutions of reasonable quality (see Section 3); in fact, the solution quality slightly increases when more objects are being considered. This might be due to the fact that many objects offer more combinations to achieve a good packing of the knapsack.

3 COMPARISON AGAINST A SUCCESSFUL GA

In this section we compare the simple bare bones EAs and GAs against a more sophisticated GA that has been reported to be successful in the literature for the Multidimensional Knapsack Problem.

3.1 The Multidimensional Knapsack Problem

The Multidimensional Knapsack Problem (MKP) aims to maximise the total value of selected objects without surpassing any of the constraints of the knapsack. This can be defined more formally as:

$$\text{Maximise} \sum_{j=1}^{n} p_j x_j \quad \text{subject to} \quad \sum_{j=1}^{n} r_{ij} x_j < b_i \text{ for all } 1 \le i \le m$$

where n is the number of objects, m is the number of constraints, x_j is a one if object j has been put in the bag and a zero otherwise, p_j is the value of object j, r_{ij} is the value of constraint i for object j and b_i is the maximum value of constraint i (e.g. maximum weight).

A well-stated MKP has the additional constraints $\forall i, j \colon p_j > 0$ and $\forall i \colon r_{ij} \le b_i < \sum_{j=1}^{m} r_{ij}$ to prevent trivial problems. Any x_j where $p_j \le 0$, will never be selected as it does not increase the size of $p_j x_j$. For any x_j where $r_{ij} > b_i$, there is never any capacity to select it. If $b_i \ge \sum_{j=1}^{m} r_{ij}$ then the entire set of x would fit, so there is no reason to not select all the objects. The fitness function is chosen as $\sum_{j=1}^{n} p_j x_j$ if all constraints are met, and as 0 otherwise.

3.2 Chu and Beasley's GA

Chu and Beasley [5] introduced a GA incorporating problem specific knowledge for the MKP. They presented a thorough experimental analysis comparing their GA against the state-of-the-art exact solver CPLEX and several heuristics from the literature (i.e., those proposed by Magazine and Oguz [20], Volgenant and Zoon [29] and Pirkul [24]). Chu and Beasley considered a large set of randomly generated test instances of large size that are difficult to solve exactly. They showed that their proposed GA achieved higher quality solutions on average using a much lower computational effort than those produced by CPLEX (the instances were too large thus it was not computationally tractable for CPLEX to find the optimal solutions. CPLEX was stopped when the tree memory exceeded 42 MB or after 1,800 CPU seconds). While the other heuristics terminated earlier than the GA, the latter algorithm produced solutions of considerably higher quality, still within reasonable time.

Chu and Beasley's GA is described in Algorithm 3. The algorithm is essentially a standard steady-state (μ+1) GA with $\mu = 100$, a slightly higher parent selection pressure than uniform (i.e. size 2 tournament selection), a higher mutation probability than the standard one, a genotype diversity mechanism and a problem specific repair mechanism to fix infeasible solutions before they are evaluated. While the algorithm is not very different from a standard steady-state GA, we believe the parameters have undergone considerable tuning for the achievement of the reported results. Our aim is to compare it with bare-bones algorithms with standard parameter settings and quantify the difference in performance.

In Chu and Beasley's GA the population is initialised with a set of feasible solutions. These are obtained by starting with an empty knapsack and randomly adding items until adding another one exceeds a constraint maximum. A binary tournament selection method is used to select parents for crossover. Tournament selection works by creating two pools of N individuals and then selecting the individual with the highest fitness from each pool. A binary tournament uses $N = 2$. Uniform crossover is used to create the new offspring and then each bit is flipped independently with probability $2/n$. If the offspring is already contained in the population, then it is discarded (genotype diversity). Otherwise it is added to the population as long as it is a feasible solution and an individual chosen at random amongst those with the lowest fitness is removed.

Since the binary string representation allows the generation of infeasible solutions (i.e. $\sum_{j=1}^{m} r_{ij} x_j > b_i$ for some b_i) after crossover and mutation are applied, a greedy heuristic was implemented by Chu and Beasley [5] that guarantees to transform an infeasible solution into a feasible one. This ensures that only valid solutions reach the fitness function, meaning that it can be defined as simply $f(s) = \sum p_j x_j$. The repair operator is a heuristic applied to any individual that is not a feasible solution to the problem. It has two

Algorithm 3 Knapsack GA

```
Set iterations to 0
Initialise 100 random individuals
Evaluate each individual
Find highest evaluated individual and store as BestSolution
while Iterations < MaxIterations do
    Select 2 parents ($P_1$ and $P_2$) using tournaments
    Create $C$ from uniform crossover of $P_1$ and $P_2$
    Mutate $C$
    Make $C$ feasible with repair operator
    if $C$ is duplicate of current population then
        Discard $C$
    else
        Evaluate $C$
        Remove worst individual from population
        if C evaluates better than BestSolution then
            Set BestSolution to $C$
    Increment iterations by one
return BestSolution and score of BestSolution
```

phases: the drop-phase and the add-phase. The drop-phase finds the biggest object marked with a one and drops it if the bag is overfull. It repeats this until the knapsack is feasibly filled. The add-phase then iterates over all objects in the reverse order and adds objects that are not in the bag if they fit. This leaves bags that are always filled as much as possible in a greedy way, but not overfilled.

3.3 Experimental Setup

Since previous MKP test instances from the literature were of small size and generally easy to solve, Chu and Beasley developed a large number of randomly generated problem instances to test their genetic algorithm. They generated a total of 270 problems. The number of constraints, m, was set to 5, 10, and 30 and the number of variables, n was set to 100, 250, and 500. The capacity constraints b_i were chosen as $b_i = \alpha \sum_{j=1}^{n} r_{ij}$ where α is called *tightness ratio*; lower values of α lead to more constrained problems. 30 problems were generated for each (m, n) combination, 10 for each of three different tightness ratios (i.e., $\alpha := \{0.25, 0.5, 0.75\}$). The problems were made available in the OR-library[1]. For all of the 27 problem structures (except for 2) Chu and Beasley's GA performed at least as well as CPLEX and the average performance across all problems was much better with considerably lower computation time.

All algorithms use the procedure from Chu and Beasley's GA [5] to generate an initial population of feasible solutions. All algorithms are terminated after 10,000 fitness function evaluations. The best fitness value in the final population is recorded and then averages across 10 runs for each of the 27 settings are reported.

3.4 Experimental Results

We compare the simple algorithms against Chu and Beasley's GA in two different settings. In the first setting no algorithm uses the repair operator (top part of Figure 1) and in the second they all use it (bottom part of Figure 1). Each plotted point represents the best found fitness averaged over the 10 instances with parameters (α, n, m) and normalised to the interval $[0, 1]$ by dividing by the optimal value of the LP relaxation.

As expected all the algorithms perform much better with the repair operator than without.

In the following we report on results of statistical tests executed as follows. We performed Wilcoxon signed-rank tests for all pairs of algorithms where the final fitness values were paired according to problem instances. We performed two-sided tests to check whether the two input distributions differ or not, followed by one-sided tests both ways to confirm which algorithm has the higher mean rank. We report a comparison as statistically significant if the p-values of the two-sided test and that of the respective one-sided test both satisfied $p \leq 0.01$, that is, a confidence level[2] of 0.01.

Results without repair. Without repair, Chu and Beasley's algorithm considerably outperforms all the bare-bones EAs and GAs in solution quality; all comparisons are statistically significant. Concerning the latter, the standard crossover algorithms perform better than those using only mutation. Amongst the crossover based algorithms, those using both parent and offspring populations (i.e., (20+20) GA and (20+20) Fast-GA) produce better solutions than those creating only one offspring per generation (i.e., $\lambda = 1$). All these comparisons are statistically significant. Figure 1 suggests that fast mutations are beneficial as the (20+20) Fast-GA seems to outperform the (20+20) GA on most instances. However, there is no statistically significant difference between these two algorithms. Fast mutation alone, without populations and crossover, does not perform better than non-fast mutation-only algorithms. Hence we cannot conclude that fast mutations are better than non-fast ones.

There is a striking performance gap between switching crossover on and off. The solution qualities of both the (20+20) GA and (20+20) Fast-GA are statistically significantly higher than those of the (20+20) EA. Finally, the (1+(λ, λ)) GA performs statistically significantly worse than all other algorithms, even when restarting the algorithm to avoid problems with diverging parameter λ.

From our experiments it is clear that populations and crossover are helpful for the problem. However, it is unclear whether the larger population (i.e. $\mu = 100$) suffices for Chu and Beasley's GA to achieve its performance. Nevertheless, given that crossover is clearly useful, it would be surprising if the genotype mechanism was not helping too.

Results with repair. With the repair mechanism, Chu and Beasley's algorithm is better than all other algorithms with statistical significance, except for the (20+20) Fast-GA. We compared Chu and Beasley's algorithm against the (20+20) Fast-GA using Mann-Whitney U tests for each instance and found that on all instances the results were either not statistically significant or showed a significant advantage for Chu and Beasley. The (20+20) Fast-GA, in turn, was better than all simple algorithms, except for the (20+20) GA. The (1+(λ, λ)) GA ranks joint last (alongside the (20+20) EA), with its

[1] see http://people.brunel.ac.uk/~mastjjb/jeb/info.html.

[2] In the remainder, we use pairwise tests to compare the considered algorithms. We did not apply Bonferroni correction, hence the confidence level might be increased. The confidence level of 0.01 was chosen low enough to allow for meaningful conclusions. We often use pairwise tests to compare groups of algorithms that differ in one design feature (e.g. comparing algorithms with crossover against their direct counterparts without crossover, while keeping all other parameters the same). In many such comparisons all pairwise tests gave significant results in favour of the same group, which is very strong evidence that one group performs better than the other.

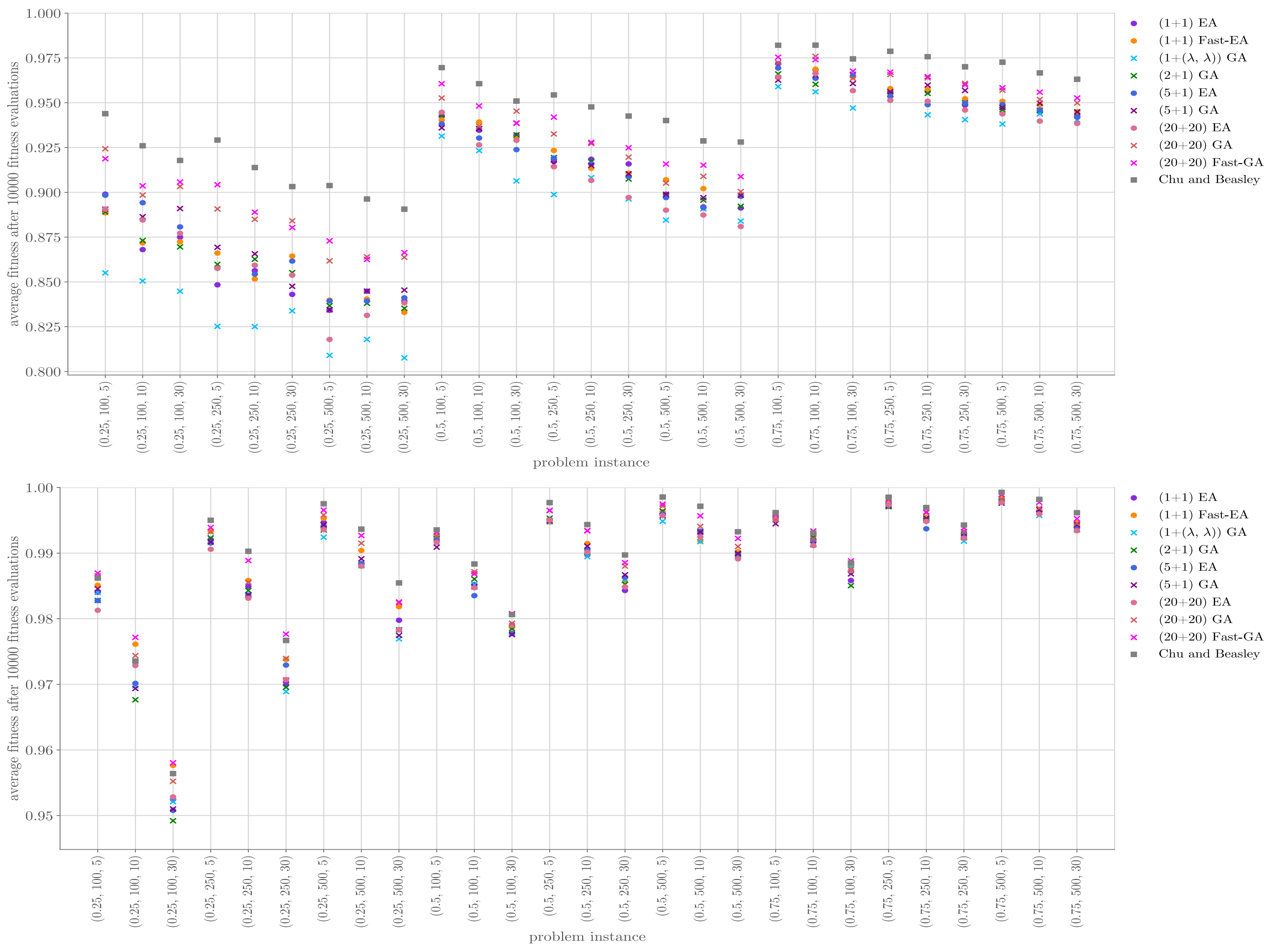

Figure 1: Averages for the best fitness in the final population for MKP and algorithms without repair (top) and with repair (bottom). For simple algorithms, circles show mutation-only algorithms and crosses show algorithms with crossover. Instances from the OR-library with tightness ratio α, n variables and m constraints are labelled (α, n, m).

performance being statistically significantly worse than that of all other algorithms, except for the (20+20) EA.

3.5 Discussion and Related Theoretical Work

From runtime analysis we know the expected runtime of the bare bones algorithms for exploitation (OneMax) and at escaping local optima (Jump) for functions of unitation (i.e. functions that only depend on the number of 1-bits). The steady-state GAs outperform the EAs for both OneMax (i.e., with expected runtimes at most $((3/4)e + o(1))n \ln n$ versus at least $(e - o(1))n \ln n$ [7]) and Jump (i.e., with expected runtimes $O(n^{k-1} \log n)$ versus $\Theta(n^k)$ [9]).

Furthermore, we essentially also know the expected runtime of Chu and Beasley's GA for both OneMax and Jump functions. Since it is a steady-state (μ+1) GA (i.e., $\mu = 100$) with genotype diversity and tournament selection for reproduction we know from [26] that its expected runtime for OneMax is at most $(e/2 + o(1))n \ln n$. We also know from the analysis for Jump [18] that if uniform parent selection was used instead of tournament selection then the expected runtime of Chu and Beasley's GA to escape a local optimum with basin of attraction of k bits would be roughly $O(4^k + \mu n k^2)$. Since the tournaments are of size 2, selection is approximately uniform.

Hence, for unitation functions we expect the EAs to be outperformed by the GAs at hillclimbing and at escaping local optima and similarly the GAs to be outperformed by Chu and Beasley's GA. Interestingly we get similar (at least rank-wise) experimental results for the NP-hard MKP. While the theoretical results hold for unitation functions, where diversity may be created via mutation by swapping the positions of a 0-bit with a 1-bit, MKP clearly does not display the unitation function characteristics. Also the theoretical results are related to runtimes until optimal solutions are found and not to approximate solutions as in the presented experiments. Nevertheless, we see a similar trend. In Section 4 we will see that for other NP-hard problems very different conclusions may be drawn.

Algorithm 4 Main loop of P3 (adapted from [15–17])

```
Generate uniform random solution $x$
Perform local search on $x$
**if** $x$ not in pyramid **then**
    Add $x$ to level 0 of the pyramid and update clusters
**for all** levels $i = 0, 1, 2, \dots$ of the pyramid **do**
    Cluster-based crossover of $x$ with level $i$ to create $y$
    **if** $f(y) > f(x)$ **and** $y$ is not in pyramid **then**
        Add $y$ to pyramid level $i + 1$ and update clusters
        $x \leftarrow y$
```

4 COMPARISON AGAINST A SUCCESSFUL PARAMETERLESS ALGORITHM

We now compare the bare-bones EAs and GAs against the Parameterless Population Pyramid (P3), a parameterless algorithm for which good results have been reported for the Ising Spin Glasses problem and MaxSat amongst others.

P3 is a genetic algorithm conceived by Goldman and Punch [15], notable by the fact that it does not use conventional generations of a solution population. The model instead uses a pyramid-like set of sorted populations. It utilises a combination of local search and an advanced crossover operator taken from the Linkage Tree Genetic Algorithm (LTGA) [28] that tries to cluster variables to identify sets of bits that should be kept together during crossover operations. P3 was designed as a method for performing evolutionary optimisation without requiring any user-specified parameters [16], and it showed excellent performance results in empirical tests on a range of combinatorial problems like Ising Spin Glasses, NK-landscapes, MaxSat and synthetic problems [15, 16]. The P3 algorithm was proven to show competitive performance to the best known unbiased Genetic Algorithms across a range of problems, from unimodal to deceptive ones and highly multimodal ones such as H-IFF [17].

The main loop of P3 is shown in Algorithm 4; pseudocode for all subroutines can be found in [15–17]. P3 builds up a pyramid of populations: each "level" in the pyramid refers to a separate subpopulation. This pyramid is built bottom-up, with higher levels being more likely to contain better fitness as new, improved solutions are propagated to higher levels of the pyramid. P3 maintains a hierarchical set of gene clusters in each level of the pyramid. These are determined using a greedy construction procedure that iteratively merges the most similar clusters (where similarity is based on the pairwise entropy of genes) to create larger clusters. These clusters are used during crossover: a newly created solution is first turned into a local optimum by local search (hill climbing), and then it is crossed with the bottom level of the pyramid. For each cluster of genes crossover searches all individuals on the considered level to find gene values for said cluster that improve the fitness. If crossover leads to an improvement of fitness, the outcome is added to the next higher level of the pyramid. This process is iterated until no more improvements can be found. Code for Goldman and Punch's implementation is freely available on GitHub[3].

[3]See: https://github.com/brianwgoldman/FastEfficientP3

4.1 Ising Spin Glasses and Experimental Setup

The Ising Spin Glasses (ISG) problem is a popular combinatorial benchmark problem derived from physics. It concerns subatomic particles and desires a ground state s to be found which minimises energy. The problem is represented as a two-dimensional torus with edge set E where each site i has an atomic spin $s_i \in \{-1, +1\}$. The energy of a state is affected by the interactions between neighbours i and j called J_{ij}. Then the energy is given by a Hamiltonian function, which in physics is to be minimised: $H(s) = -\sum_{\{i,j\} \in E} J_{ij} s_i s_j$. We obtain a fitness function for maximisation by changing the sign, using a straightforward binary encoding for the spins s_i and normalising to the interval $[0, 1]$, with 1 being the optimum fitness.

We used 5000 ISG test problems that were included in the P3 repository. This consists of 200 problems of 25 different sizes, along with optimal solutions. In the generation of these instances, the interactions J_{ij} had been chosen uniformly at random from $\{-1, +1\}$.

P3 as well as the simple algorithms were run until either the optimum solution was found, or at least 10,000 fitness function evaluations were made at the end of the iteration[4]. For each of the 5,000 instances each algorithm was run once, and the best fitness values in the final population were averaged over all 200 instances of the same problem size.

4.2 The MaxSat Problem & Experimental Setup

MaxSat is a classical combinatorial problem: given n variables $x_1, \dots, x_n \in \{0, 1\}^n$ and a set of clauses, conjunctions of literals that include variables and their negations, the task is to find an assignment of variables that satisfies a maximum number of clauses.

We use the procedure and code by Goldman and Punch [15, 16] to randomly generate problem instances with known optima. They generated a target solution uniformly at random and then generated clauses that are satisfied by it. This was done by choosing the variable indices uniformly at random and setting the signs of literals such that at least one literal is satisfied by the target solution. To avoid biasing search towards finding the target easily, Goldman and Punch [15] used a $1/6$ probability that all clause signs match the target, a $1/6$ probability that two clause signs match the target, and a $4/6$ probability that only one sign matches. This resulted in a problem instance where the target solution is guaranteed to be an optimum, although other optima may exist. The clause to variable ratio was set to a standard value of 4.27. All algorithms were compared on the same randomly generated instances.

The fitness function is then simply taken as the fraction of satisfied clauses, with a value of 1.0 being optimal. As before for ISG, we report averages of the best fitness values in the final population, averaged over 200 instances for the same problem size.

4.3 Experimental Results

The results for ISG shown in Figure 2 and those for MaxSat shown in Figure 3 are remarkably similar, hence we describe them together.

[4]P3 can spend many function evaluations in one iteration due to the computationally expensive crossover process spanning various levels of the pyramid. In some cases the threshold of 10,000 evaluations was exceeded considerably, giving P3 an advantage over other algorithms. We also considered a variant of P3 where after exceeding 10,000 function evaluations the last iteration was discarded, leading to a potential disadvantage. Both variants performed similarly in comparison to other algorithms. Hence for simplicity we only show results for the more advantageous setting.

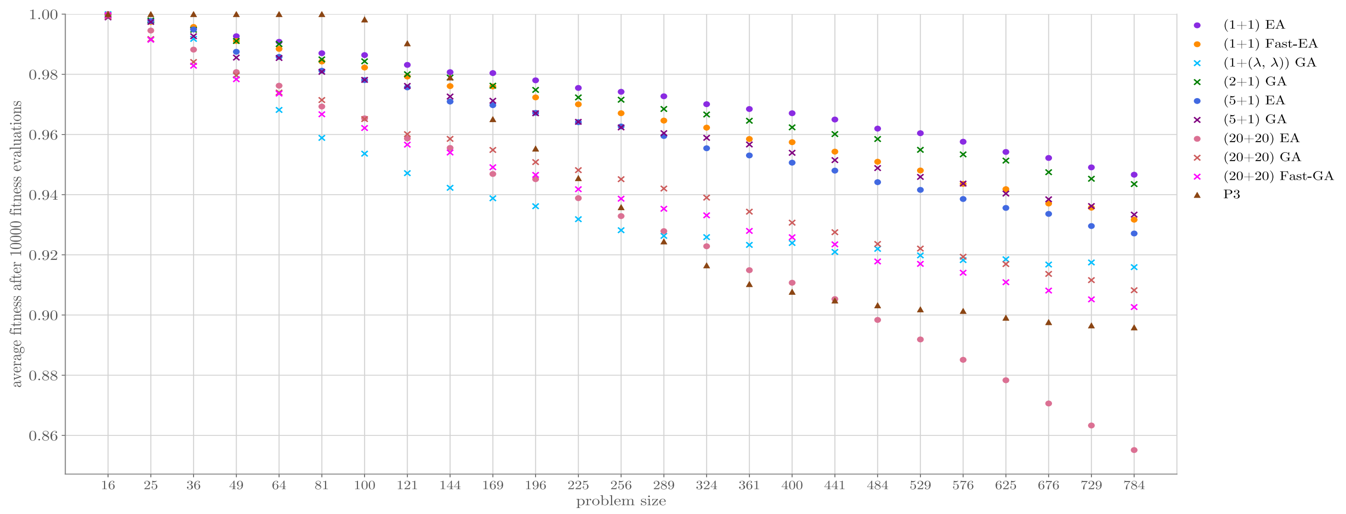

Figure 2: Averages for the best fitness in the final population for Ising Spin Glasses. Amongst the simple algorithms, circles show mutation-only algorithms and crosses show algorithms with crossover.

The first surprising result is that while P3 outperforms all algorithms for small problem sizes, its performance deteriorates drastically as the problem sizes increase. In order to get more insights through statistical tests, we divided the problems into small and large problem sizes. For ISG we define small problems as $n \leq 144$ and large problems as $n \geq 324$. For MaxSat we define small problems as $n \leq 196$, and large problems as $n \geq 324$.

All statistical tests involving P3 on small MaxSat instances failed to indicate statistical significance. These instances are so easy that all algorithms find the optimum in most runs. Hence we focus on (1) comparisons across large instances and (2) comparisons across all instances. The following comparisons are all statistically significant.

Results for large instances. For large ISG problems, P3 is outperformed by all algorithms, except for the (20+20) EA. P3 outperforms the (20+20) EA, hence the latter algorithm shows the worst performance. All these comparisons also hold for MaxSat, however there was no statistically significant difference between P3 and the (20+20) Fast-GA. While this does not rule out that P3 would find better solutions on average given sufficient time, its performance is worse than that of the simple EAs within the available time budget.

The (1+1) EA turns out to be the best algorithm for large ISG instances as it outperforms the (2+1) GA and the (2+1) GA outperforms (1+1) Fast-EA, (1+(λ, λ)) GA, (5+1) EA, (20+20) GA and (20+20) Fast-GA. The (5+1) GA seems to perform slightly better as it is *not* outperformed by the (2+1) GA (no statistical significance). For MaxSat the same can be said, however there is no statistically significant difference between the (1+1) EA and the (2+1) GA, leaving these as the best two algorithms.

Results across all instance sizes. The comparisons between simple algorithms give similar results to large instance sizes: the (1+1) EA outperforms all other simple algorithms for ISG, followed by the (2+1) GA, whereas for MaxSat these two algorithms both outperform all other simple algorithms, with no statistically significant difference between (1+1) EA and the (2+1) GA. Interestingly, for both ISG and MaxSat the (1+(λ, λ)) GA performs significantly worse than (1+1) EA, (2+1) GA, (1+1) Fast-EA, and (5+1) GA, while beating the (20+20) EA with statistical significance.

Amongst the simple algorithms the ranking is very different compared to the experiments for the MKP problem. While there the use of populations and crossover is beneficial, for these instances of ISG and MaxSat the use of offspring populations is detrimental: the (20+20) EA, (20+20) GA and the (20+20) Fast-GA are all outperformed by all algorithms with population sizes 1, 2 and 5. For population size 20, crossover is beneficial since for both problems the (20+20) Fast-GA outperforms the (20+20) EA.

4.4 Discussion and Related Theoretical Work

The experimental results suggest that algorithms that are good at exploitation show the best performance for both ISG and MaxSat. Indeed algorithms evolving a single lineage like the (1+1) EA and the (2+1) GA perform better than those with larger populations, hence populations do not seem to be necessary. Our findings also suggest that P3 is not very effective under limited time budgets as it seems to spend many function evaluations in early stages of a run building up a pyramidal population model. While this strategy does pay off when larger budgets are being used [15, 16], it is unhelpful when the available computation time is small.

There is related theoretical work on the performance of the (1+1) EA on MaxSat [12], showing that the (1+1) EA is provably efficient (runs in time $O(n \log n)$ with probability $1 - o(1)$) when clauses are generated uniformly at random and the clause density is at least logarithmic. However, there are no theoretical results for the (1+1) EA for constant clause densities as used here, which are more challenging as they are not biased towards the optimum.

We compare the findings against theoretical results for OneMax as another example where exploitation is crucial. In the absence of crossover, populations are provably not necessary for OneMax; in fact, the (1+1) EA is an optimal algorithm amongst all mutation-only algorithms [25]. It is also known that crossover in a small population

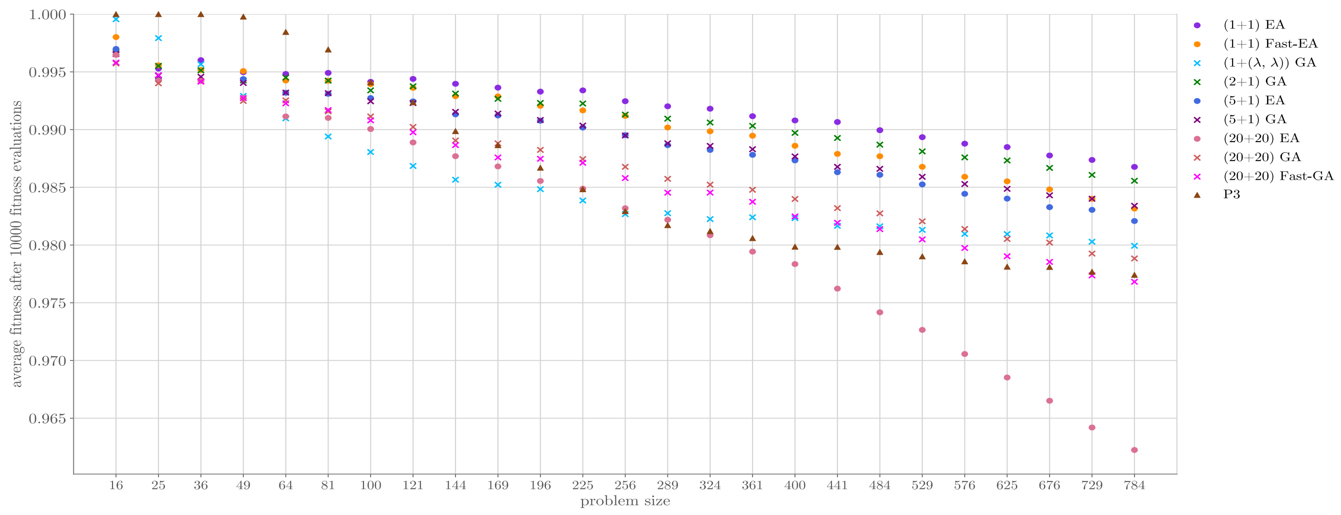

Figure 3: Averages for the best fitness in the final population for MaxSat instances. Amongst the simple algorithms, circles show mutation-only algorithms and crosses show algorithms with crossover.

leads to faster algorithms [7, 26]: both the (2+1) GA and the (5+1) GA have better expected runtimes than the (1+1) EA for OneMax [7, 8]. This effect does not show in the present experimental results for ISG and MaxSat. One conjecture is that for these problems creating diversity for crossover to be effective is harder than for functions of unitation where it is sufficient to exchange the positions of a 1-bit and a 0-bit. Nevertheless, crossover is still beneficial, since the (20+20) Fast-GA outperforms the (20+20) EA.

The $(1+(\lambda, \lambda))$ GA is outperformed by several algorithms. This is unexpected since in experiments from [15] the $(1+(\lambda, \lambda))$ GA was the runner-up to P3 for large problem sizes of the same planted model considered here. Also recent theoretical results [3] showed that the $(1+(\lambda, \lambda))$ GA is more efficient on a different, easier, class of planted MaxSat instances than any of the simple algorithms studied here. In [15] the algorithms were allowed to run much longer. Given the smaller budgets the $(1+(\lambda, \lambda))$ GA performs better than P3, while they both perform poorly compared to the more simple algorithms. Given that the $(1+(\lambda, \lambda))$ GA has better performance on the set of instances considered in [3], we conclude that the algorithm displays poorer performance on more difficult instances.

5 CONCLUSIONS

We have provided an extensive study comparing simple theory-guided EAs against sophisticated EAs known to perform well on difficult combinatorial problems. We investigated when and why sophistication in the EA design is beneficial, and quantified the performance gain through adding complexity. We considered whether the theoretical insights obtained on simpler problems extend to more complex ones and whether the simple and basic algorithms are competitive with the problem-tailored ones. The conclusions strongly depend on the problem at hand as we obtained very different results for the MKP compared to those for Ising Spin Glasses (ISG) and MaxSat.

For the MKP, no simple algorithm was able to outperform the tailored GA by Chu and Beasley independent from the problem size or the available computational budget [5]. Amongst the simple algorithms, population-based GAs with crossover performed the best and the $(1+(\lambda, \lambda))$ GA showed the worst performance.

In sharp contrast, for ISG and MaxSat, where P3 was reported to outperform many other algorithms [15, 16], we found that for large instances, where the time budget was small compared to the problem size, P3 performed poorly. For these problems, populations were found to be detrimental and the simplest algorithm, the (1+1) EA, performs best, for large instances and across all instances.

We conclude that the usefulness of EA features such as populations and crossover strongly depends on the problem at hand. For instance, the (20+20) Fast-GA using populations, crossover and fast mutation operators is amongst the best algorithms for MKP and amongst the worst algorithms for ISG and MaxSat.

ACKNOWLEDGMENTS

The research leading to these results has received funding from the EPSRC under grant agreement no EP/M004252/1. This paper is based upon work from COST Action CA15140 "Improving Applicability of Nature-Inspired Optimisation by Joining Theory and Practice (ImAppNIO)" supported by COST (European Cooperation in Science & Technology). The authors thank Mark Wineberg for advice on statistical tests.

REFERENCES

[1] E. Alba, G. Luque, and S. Nesmachnow. Parallel metaheuristics: Recent advances and new trends. *International Transactions in Operational Research*, 20(1):1–48, 2013.
[2] J. E. Beasley. Or-library: Distributing test problems by electronic mail. *The Journal of the Operational Research Society*, 41(11):1069–1072, 1990.
[3] M. Buzdalov and B. Doerr. Runtime analysis of the $(1+(\lambda,\lambda))$ genetic algorithm on random satisfiable 3-CNF formulas. In *Proc. of GECCO 2017*, pages 1343–1350. ACM, 2017.
[4] R. Chiong, T. Weise, and Z. Michalewicz. *Variants of Evolutionary Algorithms for Real-World Applications*. Springer, 2011.
[5] P. Chu and J. Beasley. A genetic algorithm for the multidimensional knapsack problem. *Journal of Heuristics*, 4(1):63–86, 1998.
[6] D. Corus, D. C. Dang, A. V. Eremeev, and P. K. Lehre. Level-based analysis of genetic algorithms and other search processes. *IEEE Transactions on Evolutionary Computation*, 22(5):707–719, 2018.

[7] D. Corus and P. S. Oliveto. Standard steady state genetic algorithms can hill-climb faster than mutation-only evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 22(5):720–732, 2018.

[8] D. Corus and P. S. Oliveto. On the benefits of populations for the exploitation speed of standard steady-state genetic algorithms. In *Proc. of GECCO 2019*, pages 1452–1460. ACM, 2019.

[9] 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.

[10] B. Doerr, C. Doerr, and F. Ebel. From black-box complexity to designing new genetic algorithms. *Theoretical Computer Science*, 567:87–104, 2015.

[11] B. Doerr, H. P. Le, R. Makhmara, and T. D. Nguyen. Fast genetic algorithms. In *Proc. of GECCO 2017*, pages 777–784, 2017.

[12] B. Doerr, F. Neumann, and A. M. Sutton. Time complexity analysis of evolutionary algorithms on random satisfiable k-cnf formulas. *Algorithmica*, 78(2):561–586, Jun 2017.

[13] A. Eiben and J. Smith. *Introduction to Evolutionary Computing*. Natural Computing Series. Springer Berlin Heidelberg, 2015.

[14] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[15] B. W. Goldman and W. F. Punch. Parameter-less population pyramid. In *Proc. of GECCO 2014*, pages 785–792. ACM, 2014.

[16] B. W. Goldman and W. F. Punch. Fast and efficient black box optimization using the parameter-less population pyramid. *Evolutionary Computation*, 23(3):451–479, 2015.

[17] B. W. Goldman and D. Sudholt. Runtime analysis for the parameter-less population pyramid. In *Proc. of GECCO 2016*, pages 669–676. ACM, 2016.

[18] T. Jansen and I. Wegener. The analysis of evolutionary algorithms-a proof that crossover really can help. *Algorithmica*, 34(1):47–66, 2002.

[19] J. Lengler. A general dichotomy of evolutionary algorithms on monotone functions. In *Proc. of PPSN 2018*, pages 3–15. Springer, 2018.

[20] M. J. Magazine and O. Oguz. A heuristic algorithm for the multidimensional zero-one knapsack problem. *European Journal of Operations Research*, 16:319–326, 1984.

[21] F. Neumann and C. Witt. *Bioinspired Computation in Combinatorial Optimization*. Springer-Verlag, 2010.

[22] P. S. Oliveto and C. Witt. On the runtime analysis of the simple genetic algorithm. *Theoretical Computer Science*, 545:2–19, 2014.

[23] P. S. Oliveto and C. Witt. Improved time complexity analysis of the simple genetic algorithm. *Theoretical Computer Science*, 605:21–41, 2015.

[24] H. Pirkul. A heuristic solution procedure for the multiconstrained zero-one knapsack problem. *Naval Research Logistics*, 34:161–197, 1987.

[25] D. Sudholt. A new method for lower bounds on the running time of evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 17(3):418–435, 2013.

[26] D. Sudholt. How crossover speeds up building-block assembly in genetic algorithms. *Evolutionary Computation*, 25(2):237–274, 2017.

[27] A. Sutton. Crossover can simulate bounded tree search on a fixed-parameter tractable optimization problem. In *Proc. of GECCO 2018*, pages 1531–1538. ACM, 2018.

[28] D. Thierens and P. A. N. Bosman. Hierarchical problem solving with the linkage tree genetic algorithm. In *Proc. of GECCO 2013*, pages 877–884. ACM, 2013.

[29] A. Volgenant and J. A. Zoon. An improved heuristic for multidimensional 0-1 knapsack problem. *Journal of the Operations Research Society*, 41:963–970, 1990.