CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Cybernetics

Bachelor's Thesis

Fast Learning in Bayesian Optimization Algorithm

Matěj Vasilevski
Study programme: Cybernetics and Robotics

August 2020
Supervisor: Ing. Petr Pošík, Ph.D.

BACHELOR'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Vasilevski Matěj** Personal ID number: **474605**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Cybernetics and Robotics**

II. Bachelor's thesis details

Bachelor's thesis title in English:

Fast Learning in Bayesian Optimization Algorithm

Bachelor's thesis title in Czech:

Rychlé učení v Bayesovském optimalizačním algoritmu

Guidelines:

Algorithms ECGA (extended compact genetic algorithm) and BOA (Bayesian optimization algorithm) are population-based optimization algorithms. They are among the most powerful methods for optimization of complex black-box optimization problems with binary representation. Each generation they build a model of the structure of dependencies among individual solution components. Model learning is a time-consuming operation. For ECGA, an efficiency enhancement was proposed recently that allows to simplify and accelerate the learning without any negative effect on the algorithm performance. The goal of this project is to implement a similar method of model learning for algorithm BOA and evaluate the potential positive and negative effects on the algorithm performance.
1) Learn the principles of algorithms ECGA and BOA.
2) Explore the method used in ECGA to accelerate the model learning.
3) Apply the method to algorithm BOA adequately.
4) On a set of benchmark problems, compare the original and the modified algorithms with respect to the number of objective function evaluations required to find the solution, and with respect to the time required to run the algorithm.

Bibliography / sources:

[1] Duque, Thyago S.P.C., Goldberg, David E., Sastry, Kumara: Enhancing the Efficiency of the ECGA. PPSN 2008, Dortmund.
[2] Pelikan, M. Hierarchical Bayesian Optimization Algorithm. Springer, 2005.

Name and workplace of bachelor's thesis supervisor:

Ing. Petr Pošík, Ph.D., Analysis and Interpretation of Biomedical Data, FEE

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: **09.01.2020** Deadline for bachelor thesis submission: **14.08.2020**

Assignment valid until: **30.09.2021**

Ing. Petr Pošík, Ph.D.	doc. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor's signature	Head of department's signature	Dean's signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

. | Date of assignment receipt | Student's signature

Acknowledgement / Declaration

I would like to thank my advisor, Ing. Petr Pošík Ph.D., for his advices, corrections and valuable insights. Also, huge thanks belongs to my family and friends for the moral support.

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date 14.08.2020

.....

Abstrakt / Abstract

Evoluční algoritmy, které modelují řešený problém pomocí diskrétního pravděpodobnostního rozdělení, jsou mocné optimalizační algoritmy navržené pro řešení těžkých problémů obsahujících závislosti mezi proměnnými. Takové problémy nejde spolehlivě řešit běžnými genetickými algoritmy. Bohužel tyto pravděpodobnostní modely jsou výpočetně náročné a tyto algoritmy tak bývají pomalejší. T. Duque navrhl úpravu pro Extended compact genetic algorithm (ECGA), která je schopná 1000x zrychlit běh algoritmu na 4096bitové Trap4 funkci. V první části této práce jsme tuto úpravu úspěšně ověřili. V druhé části jsme tuto úpravu aplikovali na Bayesian optimization algorithn (BOA). Nicméně tato úprava BOA algoritmu nepřinesla očekávané výsledky. Je potřeba provést další testy, abychom mohli říct, zda a jak lze BOA zrychlit.

Klíčová slova: ECGA, BOA, stavba modelu, hladový algoritmus, extended compact genetic algorithm, bayesian optimization algorithm, replikace

Estimation of distribution algorithms are powerful optimization algorithms designed to solve hard problems with linkage that regular genetic algorithms cannot reliably solve. However, this design, which employs probability models to describe the dependencies between variables, is what makes them rather slow. T. Duque et al. proposed a speedup method for the Extended compact genetic algorithm (ECGA), and stated that the method achieved a 1000x speedup on a 4096-bit Trap4 problem. In the first part of this thesis, we have successfully replicated the results of the proposed speedup method. In the second part, we have implemented the proposed method for the Bayesian optimization algorithm (BOA). This modification, however, did not result in the expected speedup. Further tests are required to determine whether and how the BOA can be sped up.

Keywords: ECGA, BOA, model building, greedy search , extended compact genetic algorithm, bayesian optimization algorithm, replication

/ Contents

Tables / Figures

Chapter 1
Introduction

Bayesian optimization algorithm (BOA) is a popular probabilistic algorithm belonging to the class estimation of distribution algorithms, which themselves are part of a broad family of genetic algorithms. Those are potent optimization algorithms used in optimization tasks, state space search, and machine learning. Estimation of distribution algorithms are especially powerful because they employ statistics to create an explicit model of the solved problem. However, the statistics are computationally quite expensive. This work aims to explore a speedup method proposed for ECGA (Extended compact genetic algorithm) and see if it is applicable to the BOA. Speeding up BOA would enable us to use it on larger problems, or even solve problems which were previously computationally intractable.

Genetic algorithms (GAs) were started in the 1960s by John Holland. He saw them as a model of adaptation. At the same time, Ingo Rechenberg and Hans-Paul Schwefel started solving real-valued problems using methods inspired by evolution. This approach evolved into a subfield called Evolutionary strategies. Concurrently to all this, Lawrence Fogel saw the means for solving artificial intelligence in evolving a population of finite state machines; this is called Evolutionary programming. Although these approaches share the inspiration in nature and evolution, they continued to develop independently of one another. That changed in the 1990s when the three approaches have been merged under the name Evolutionary Computation. Book [1] describes the history in greater detail.

So, genetic algorithms are here and solve problems in diverse fields, such as chemistry, operations scheduling, and electronics design. Nevertheless, they lack a significant feature - linkage learning. Linkage (also epistasis) is the dependency of one variable on some other variables. In other words, the effect of one variable depends on other variables. Because GAs are generally not aware of the linkage between variables, they will often unknowingly break possibly good linkage groups. This makes it hard, if not outright impossible, for GAs to solve particular problems. One possible solution is to implement the GA in such a way that it will not break linkage. But this solution would be too tedious and impractical.

This is where Estimation of distribution algorithms (EDAs) come into play. They create an explicit model of the problem, which can have the form of a Bayesian network, or simply grouped variables with a joint probability distribution for each group. These groups, in theory, should mimic linkage groups in the solved problem. In short, the explicit model should make the algorithm aware of the linkage and enable it to solve problems that regular GAs could not. However, calculating the probability distribution is a demanding task. The algorithm has to go through the whole population and count frequencies for all possible combinations of variables from a group, making EDAs rather slow.

The speed issues bring us back to the goals of this work. First, we want to replicate the results of the proposed speedup method for ECGA. Second, we want to apply the proposed method to BOA and compare the modified BOA with the original BOA.

Chapter 2
From Genetic Algorithms to Estimation of Distribution Algorithms

Evolutionary algorithms (EAs) are optimization algorithms inspired by Charles Darwin's theory of evolution. Usually, they are used for black-box optimization tasks in both discrete and continuous space. In black-box optimization, the algorithm knows nothing about the structure of the optimized function; it can only evaluate it. EAs are very suitable for problems about which we know nothing. They can also handle dynamic problems, where the optimum changes over time.

In the first part of this chapter, we will examine a typical genetic algorithm and check its performance on a counting-ones (OneMax) problem. The second part contains a test of the typical GA on a more difficult problem, which will show why were Estimation of Distribution Algorithms (EDAs) invented and what they solve.

2.1 A simple genetic algorithm on a simple problem

Here we will describe how genetic algorithms generally work, implement an example algorithm, and see how it performs on the OneMax problem. The implemented example algorithm will be called simpleGA. Book [2] was used as a reference on GAs.

2.1.1 Generic template for genetic algorithms

Genetic Algorithms (GAs) are perhaps the most well-known subset of EAs. GAs typically work with binary strings of fixed length, but other problem encodings are possible too. The binary string here means a vector of 1s and 0s.

GAs generally maintain a set of candidate solutions called a population of individuals. The population is then evolved in generations. During a single generation, the algorithm selects the fittest individuals for reproduction, using the optimized function as the definition of fitness. Then it generates a set of offspring individuals using operations like crossover and mutation. Finally, the newly generated offsprings are incorporated into the starting population. Pseudocode for this procedure is below. Now let us have a closer look at the single steps.

```
population = create_population();
generation = 0;
while (not termination_criteria(population)){
    parents = selection(population);
    offsprings = mutate(crossover(parents));
    population = merge(population, offsprings);
    generation += 1;
}
return population.best_individual;
```

The selection promotes higher fitness individuals for reproduction; it is the exploitation part of the search process. There are many methods for selection, such as tournaments, truncation, linear ranking. Here the tournament selection with replacement was

implemented because it will be used later in ECGA too. Tournament selection of size k randomly picks k individuals from the population and only the fittest one advances. With replacement means that each individual can be chosen multiple times.

Crossover is a reproduction operator; it is the exploration part of the search process. By mixing parts of two high-fitness individuals, we hope to generate new, even better individuals. Again, more variants of crossover exist, such as 1-point, 2-point, and n-point crossover. Here we will describe and use the 2-point crossover. It works by selecting two random points that define a vector slice. Then the slices are exchanged between the two parents, which creates two new offsprings. A mutation operator can be optionally applied to these new offsprings. Mutation implemented in our example GA walks through the offsprings and flip each bit with a probability of $1/l$, where l is the length of a single individual.

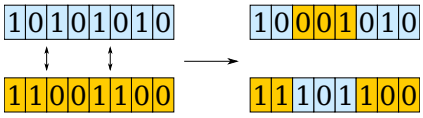

Figure 2.1. Crossover operator.

The two standard approaches for merging offsprings are generational replacement and steady-state replacement. The generational approach replaces the old population with the new offsprings. The steady-state approach replaces only a fraction of the population in every generation. Here the generational approach was chosen, mainly for the ease of implementation.

2.1.2 Performance check on OneMax problem

SimpleGA with population size 160 has been tested on a 40-bit instance of the OneMax problem (see eq. (5.1) in chapter 5 for the definition). We can see in fig. 2.2 that SimpleGA managed to solve OneMax easily. This is the expected result because OneMax is a somewhat primitive problem; it is used as a sanity check that the algorithm under test optimizes. Running the algorithm 100 times and averaging the achieved fitness yields 40.0, which means the optimum was found every time.

2.2 The motivation for the Estimation of Distribution Algorithms

We have seen that simpleGA handled the OneMax problem smoothly. Now we will try a more difficult n-bit Concatenated Trap4 problem (n-bit Trap4), see eq. (5.4) for the definition.

From figure 2.3, we can see that simpleGA failed to reach the optimal solution; averaging results from 100 runs gives only 48.04. This is because Trap4 contains dependencies between variables (all bits in one block must be zero to achieve the optimum) and simpleGA does not know about them. Let us modify the crossover operation to exchange only whole blocks of size 4 between the parents, so simpleGA will not unintentionally break optimal blocks of all 0s.

Figure 2.4 shows that simpleGA did a bit better with the modified crossover. Average over 100 runs is 49.07, which is an improvement. However, tailoring the crossover to specific problems is not always feasible because the problem structure may not be known. And uncovering the problem structure is why the Estimation of Distribution

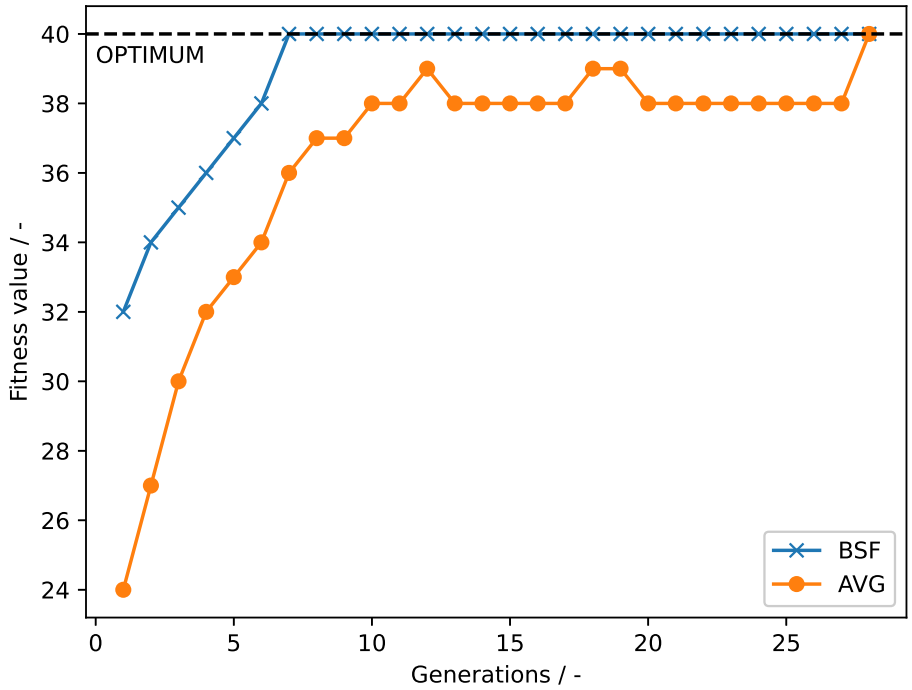

Figure 2.2. Simple GA with population size 160 on 40-bit instance of OneMax problem.

Algorithms (EDAs) were developed. EDAs create an explicit probability model of the population, enabling them to solve previously difficult problems efficiently. One can also gain insight into the problem by inspecting the model during evolution. A generic pseudocode for EDAs is below.

```
population = create_population();
while (not termination_criteria(population)){
    parents = selection(population);
    model = create_model(parents);
    offsprings = sample(model);
    population = merge(population, offsprings);
}
return population.best_individual;
```

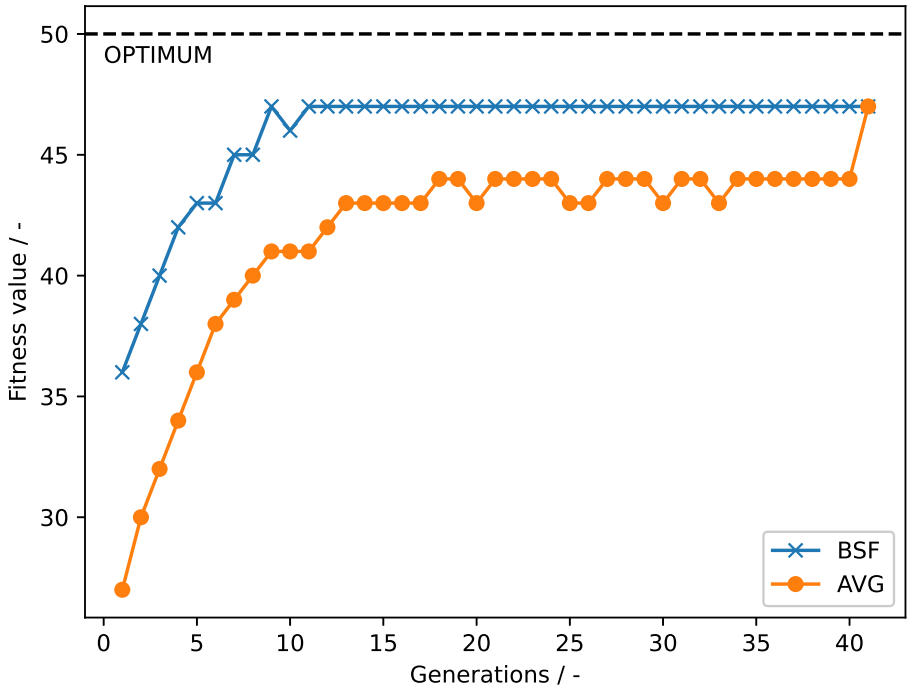

Figure 2.3. SimpleGA with population size 160 on 40-bit Trap4.

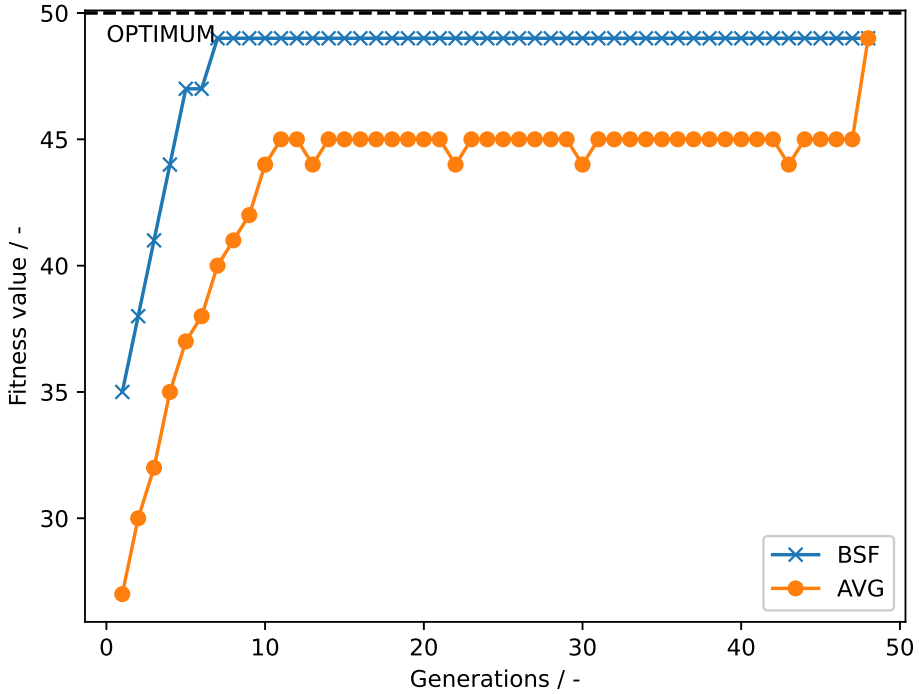

Figure 2.4. SimpleGA with population size 160 and modified crossover on 40-bit Trap4.

Chapter 3
Extended Compact Genetic Algorithm

Extended Compact Genetic Algorithm (ECGA) is a discrete EDA that uses Marginal Product Model (MPM) to represent dependencies between variables. In this chapter, we will see how ECGA works first. Then we will look at a relaxed version of the greedy search by T. Duque, which lowers the run time of ECGA.

3.1 How ECGA learns the structure

In order to learn linkage, ECGA creates an MPM model of the population. A greedy search is used to rebuild the model from scratch in every generation. In this section, we will describe two criteria that form a model evaluation metric and the greedy search for the best model.

MPM is a probability model that divides variables into partitions (groups) and determines joint probability distribution for each partition. Figure 3.1 shows an example population and the accompanying model. The joint probability distribution of the first partition, which contains variables 1 and 2, is calculated by counting the occurrences of every possible combination (of variables 1 and 2) and dividing the counts by population size. For example, a combination '01' of the first two variables is not present in the population, therefore the probability of combination '01' is 0. Both of the two partitions left contain only a single variable, so the probability is a count of 0s (1s) divided by the population size.

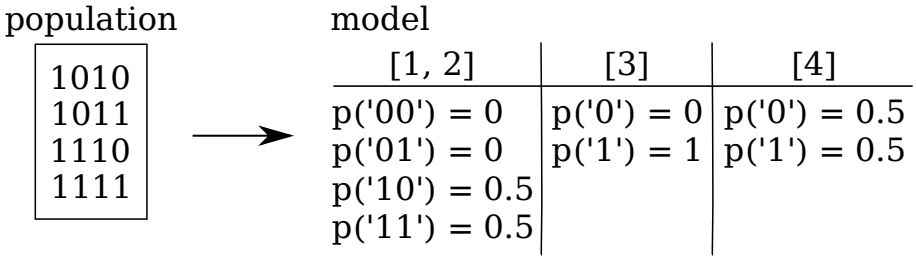

Figure 3.1. Example population and corresponding model.

The model is evaluated using the Combined Complexity Criterion (CCC), which we are trying to minimize. CCC consists of two parts: Model Complexity (MC) and Compressed Population Complexity (CPC). Equation (3.1) defines the MC; N is the population size, I is a partition, and S_I is its size (number of variables in the partition). Equation (3.2) defines the CPC; M_I is the joint probability distribution from partition I. The CPC represents population compression using entropy. Here the entropy of joint probability distribution is $\sum -p \log_2(p)$, where ps are nonzero probabilities of respective combinations.

$$MC = \log_2(N+1) \sum_{I \in \text{partitions}} (2^{S_I} - 1) \tag{3.1}$$

$$CPC = N \sum_{I \in \text{partitions}} \text{entropy}(M_I) = N \sum_{I \in \text{partitions}} \sum_{p \in M_I} -p \log_2(p) \tag{3.2}$$

ECGA uses a greedy search to find the best model of the population in one generation. The search starts by putting each variable in its own partition. ECGA then goes through every possible pair of partitions and tries to merge it to see the CCC difference. After that, the pair that reduces CCC the most is then merged in the model. This procedure is repeated until it is no longer possible to reduce CCC by merging two partitions. After this, we can end the search and sample the final model to create a new population. In the next generation, the model is again built from the beginning. Figure 3.2 shows the greedy search procedure.

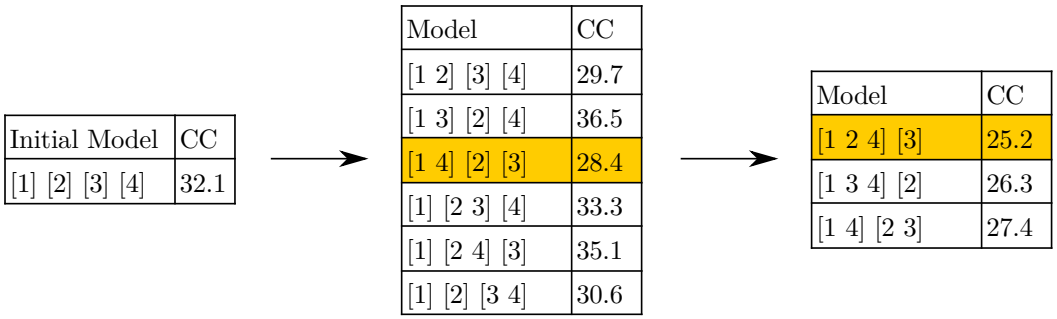

Figure 3.2. Greedy search evaluating all possible pairs.

Selection is the last notable part of ECGA. After sampling a new population from the model, there are no correlations between variables from different partitions. It is the selection's role to recorrelate specific combinations of genes that correlate with high fitness [3].

3.2 Improving the speed of ECGA

Model building is the main reason why ECGA is so computationally expensive. Let us take a look at the search's pseudocode below and analyze its time complexity. In the worst case, the algorithm will merge partitions until it ends with one large partition with all variables. Because we start with n partitions (one partition per variable) and one iteration of the while loop reduces the number of partitions by one, the while loop will be executed $n-1$ times at most (to get to one large partition). In the body of the while loop, all pairs of partitions are merged. That is $m*(m-1)/2$ merges, where m is the number of partitions in a particular iteration. Summing the merges over all loop iterations gives us

$$\begin{aligned}
\sum_{k=0}^{n-1} \frac{1}{2}(n-k)(n-k-1) &= \sum_{l=1}^{n} \frac{1}{2}(n-l+1)(n-l) \\
&= \sum_{l=1}^{n} n^2 - 2nl + l^2 + n - l \\
&= n^3 - 2n\frac{n(n+1)}{2} + \frac{n(n+1)(2n+1)}{6} + n^2 - \frac{n(n+1)}{2} \\
&= \frac{1}{6}n^3 + \frac{1}{3}n
\end{aligned} \tag{3.3}$$

Thus the greedy search is $\mathcal{O}(n^3)$.

```
// greedy search procedure
initialize partitions - one partition per variable
repeat:
    for (each pair of partitions):
        merge the pair and evaluate CCC
    if an improvement is possible: update model with the best pair
    else: end the search
```

Duque, Goldberg, and Sastry suggested an improved model building procedure to reduce the time complexity from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^2)$ [4]. The proposed pseudocode is below. They achieved about 10 times speedup for a 32-bit instance of Trap4 and more than 1000 times speedup for a 4096-bit instance. Instead of evaluating all pairs of partitions, one partition is chosen to be fixed and merged with other partitions. Pseudocode for this is below. The fixed partition is selected using a round-robin policy. The breaking probability P starts at 0.01 and is cooled down by 10 % every loop iteration.

```
// improved greedy search
initialize partitions - one partition per variable
repeat:
    choose a partition A
    for (each other partition B):
        merge A and B
        evaluate CCC
        undo the merge
    merge the best pair if there was one
    with some probability P, break a random partition
    cool down the probability P
    end the search if no improvement is possible
```

Chapter 4
Bayesian Optimization Algorithm

Bayesian Optimization Algorithm (BOA) is an EDA that uses Bayesian networks to model the dependencies among variables. In this chapter, we will start by reviewing Bayesian networks and their scoring metrics. Then we will look at the modification of BOA's model search procedure.

4.1 Bayesian networks

Bayesian Network (BN) is a probability model that captures conditional probabilities between variables. BN is described by a Directed Acyclic Graph (DAG), an acyclic graph without directed cycles. Each node in the DAG represents a single variable, and the edges represent dependencies between variables. However, DAG is not everything; it only describes the structure of BN. The other half of BN are conditional probabilities of every node and its parents. Formally, the BN encodes a joint probability distribution of random vector $X = (X_1, X_2, \ldots, X_n)$, whose variables X_i satisfy the conditional dependencies and independencies described in the DAG. π_i in eq. (4.1) is a set of X_i's parent variables.

$$p(X) = \prod_{i=1}^{n} p(X_i|\pi_i) \tag{4.1}$$

4.1.1 Metrics for Bayesian networks

There are two types of scoring metrics for BNs: Bayesian metrics and Minimum description length metrics. Bayesian-Dirichlet (BD) metric is a Bayesian metric that estimates the likelihood of a Bayesian network B for a given data D. In our case, the data are the individuals in BOA's population. $BD(B)$ is given by eq. (4.2), where:

- Γ is the Gamma function ($\Gamma(x) = (x-1)!$).
- $p(B)$ is the prior probability of a network being the network B. It can be used to bias the search.
- The last product over x_i runs over all possible values for x_i. Here we operate on binary strings, so the possible values are only 0 and 1.
- The product over π_i runs over all possible combinations of values of parents from Π_i.
- $m(\pi_i)$ is the number of instances where the parents are the specific combination π_i.
- $m(x_i, \pi_i)$ is the number of instances where $X_i = x_i$ and $\Pi_i = \pi_i$.
- $m'(\pi_i)$ and $m'(x_i, \pi_i)$ are prior information about $m(\pi_i)$ and $m(x_i, \pi_i)$, respectively.

$$BD(B) = p(B) \prod_{i=1}^{n} \prod_{\pi_i} \frac{\Gamma(m'(\pi_i))}{\Gamma(m'(\pi_i) + m(\pi_i))} \prod_{x_i} \frac{\Gamma(m'(x_i, \pi_i) + m(x_i, \pi_i))}{\Gamma(m'(x_i, \pi_i))} \tag{4.2}$$

Here we will use the K2 metric, which is a special case of the BD metric, where we substitute $m'(x_i, \pi_i) = 1$ and $m'(\pi_i) = \sum_{x_i} m'(x_i, \pi_i)$. Because x_i has only two possible

values, we can simplify it to $m'(\pi_i) = \sum_{x_i=1,2} 1 = 2$. Eq. (4.3) is the K2 metric after substituting the equations above. We can also take a log of the K2 metric (eq. (4.4)) to make the metric computation less CPU intensive.

$$K2(B) = p(B) \prod_{i=1}^{n} \prod_{\pi_i} \frac{1}{(m(\pi_i)+1)!} \prod_{x_i} m(x_i, \pi_i)! \tag{4.3}$$

$$\log(K2(B)) = \log(p(B)) + \sum_{i=1}^{n} \sum_{\pi_i} \Big(- \log((m(\pi_i)+1)!) + \sum_{x_i} \log(m(x_i, \pi_i)!) \Big) \tag{4.4}$$

An example of the Minimal description length metric is Bayesian Information Criterion (BIC). It is very similar to the CCC used in ECGA. Eq. (4.6) defines BIC, where H is a conditional entropy (eq. (4.5)), and N is the population size.

$$H(X_i|\Pi_i) = - \sum_{x_i, \pi_i} p(x_i, \pi_i) \log_2(p(x_i|\pi_i)) \tag{4.5}$$

$$BIC(B) = \sum_{i=1}^{n} \left(- H(X_i|\Pi_i)N - 2^{|\Pi_i|} \frac{log_2(N)}{2} \right) \tag{4.6}$$

4.2 Speeding up the search

A greedy search is used to build the Bayesian network in BOA. The search procedure is akin to the one in ECGA. The search starts with an empty graph (no edges) and performs elementary graph operations on all pairs of nodes until it is no longer possible to improve the metric. Elementary operations are edge addition, edge removal, and edge reversal. The search procedure should also limit the number of incoming edges into a single node when applying edge addition and reversal. This prevents the creation of too complex networks because Bayesian metrics often have higher sensitivity and capture random dependencies from noise. Minimum description length metrics, on the other hand, tend to favor overly simple models [5].

```
---------------------- Greedy search in BOA --------------------------
B = initialize bayesian network B to an empty net
repeat:
    for each node:
        for each other node:
            try graph operations on the node pair and compute metric(B)
    if (it was possible to improve metric(B)):
        apply the best operation to B
    else:
        return the network B
```

BOA, unfortunately, suffers the same fate as ECGA. It is rather slow because estimating probabilities from the population is quite demanding on the computer. However, we can try the same improvement as we did with ECGA in section 2.2. Here the improved search procedure randomly selects a node and applies the graph operations to it. This should reduce the search complexity as it did with ECGA and make BOA run faster.

```
------------------- Improved greedy search in BOA ----------------------
B = initialize bayesian network B to an empty net
repeat:
    randomly select a node
    for each other node:
        try graph operations on the node pair and compute metric(B)
    if (it was possible to improve metric(B)):
        apply the best operation to B
    else:
        return the network B
```

4.2.1 Note on used implementation

In this work, I implemented the search modification for BOA on top of a simple implementation of BOA from Martin Pelikan [6]. This simple BOA allows only edge additions. Most of the algorithm parameters were left on default, except for parameters like population size and problem size, which we need to change.

Chapter 5
Experiments

In this chapter, we will review 4 test problems and a bisection method. Those will be useful in the experiments on both regular and improved version of ECGA and BOA, where we will assess the algorithms' performance from the bisection results.

In order to avoid typing concatenated over and over again, we will establish a convention that names like n-bit EqPairs4/SlidingXor4/Trap4 mean n-bit concatenated EqPairs4/SlidingXor4/Trap4. For example, n-bit concatenated trap4 is a function composed of n/4 blocks of Trap4 in series. See figure 5.1 for a graphic explanation.

5.1 Test Functions

5.1.1 OneMax

OneMax is an elementary optimization problem. It serves more as a test that given algorithm works correctly and optimizes. The optimal solution is a vector of all 1s.

$$f_{\text{OneMax}}(x) = \sum_{i=1}^{n} x_i, \quad \text{where } x = (x_1, \ldots, x_n) \text{ is an input vector of length } n \tag{5.1}$$

5.1.2 Equal Pairs

N-bit Equal Pairs (EqPairsN) function, as its name suggests, calculates the number of equal pairs of bits in the input vector. The interactions between variables are rather weak; the optimal value for each bit is influenced by the previous bit. EqPairsN has two optima: vector of all 0s and vector of all 1s.

$$f_{EqPair}(x, y) = \begin{cases} 1, & \text{if } x = y \\ 0, & \text{if } x \neq y \end{cases}$$
$$f_{EqPairsN}(x) = 1 + \sum_{i=2}^{N} f_{EqPair}(x_{i-1}, x_i) \tag{5.2}$$

5.1.3 Sliding Xor

N-bit SlidingXor (SlidingXorN) is composed of two functions: AllEqual and Xor of 3 variables. AllEqual evaluates to 1 only for input vector of either all 0s or all 1s. Xor of 3 variables returns 1 if logical xor of the first two variables is equal to the third variable. Optimum of SlidingXorN is a vector of all 0s.

$$f_{AllEqual}(x) = \begin{cases} 1, & \text{if } x = (0, 0, \ldots, 0) \text{ or } x = (1, 1, \ldots, 1) \\ 0, & \text{if } x \neq y \end{cases}$$

$$f_{Xor}(x,y,z) = \begin{cases} 1, & \text{if } x \oplus y = z \\ 0, & \text{otherwise} \end{cases}$$

$$f_{SlidingXorN}(x) = 1 + f_{AllEqual}(x) + \sum_{i=3}^{N} f_{Xor}(x_{i-2}, x_{i-1}, x_i) \tag{5.3}$$

5.1.4 Trap function

The TrapN is a deceptive function. It takes a vector of length N and outputs the sum of 1s (like OneMax), except when the vector is all 0s, then the output is N+1.

$$f_{\text{TrapN}}(x) = \begin{cases} N+1, & \text{if } x \text{ is all 0s} \\ \sum_{i=1}^{N} x_i, & \text{otherwise} \end{cases} \tag{5.4}$$

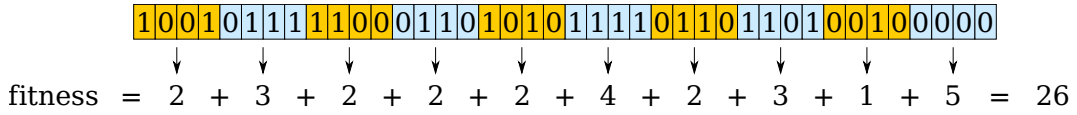

Figure 5.1. Evaluation of 40-bit Concatenated Trap4 function.

5.2 Bisection method

The bisection method searches for the minimum population size required to find the optimum of the problem 24 times out of 25 runs. The bisection works by maintaining an upper and lower population limits. These limits together form an interval that is repeatedly halved until the limits are close enough. The search starts by finding the upper limit, with which the tested algorithm can find the optimum in all 25 runs. The lower limit is set to some minimal value (for example, 1). Then the bisection averages the two limits and runs the algorithm 25 times with population size set to the average. If the optimum is found at least 24 times, the upper limit is set to the average, or else the lower limit is set to the average. Then this process repeats until the limits are not close enough, e.g., the difference between upper and lower limit is less than 1 % of their average.

Output of the bisection method is a set of parameters that describe an algorithm's runtime on a given problem. The parameters are averaged over the 25 runs when the algorithm runs with the minimal popsize. Minimal popsize is the smallest population size required to find the optimal solution in 24 out of 25 runs. Fitness evaluations (fitevals) is the number of evaluations of the problem function. Fitevals is an essential criterion because the fitness function can be resource-intensive. Also, the number of fitness evaluations can be used to compare two or more algorithms directly. CCC evaluations is the number of model evaluations using CCC. It enables us to track the computational complexity of the greedy search. Lastly, the duration is the time an algorithm needs to find a solution.

```
-------------------------- Bisection Method --------------------------
lower_limit = 1
upper_limit = 1
repeat:
    run the algorithm 25 times with population_size = upper_limit
    if (the alg. finds the optimum every time) then break
    else double the upper_limit
```

```
repeat:
    average the upper and lower limits
    run the algorithm 25 times with population_size = the average
    if (the alg. finds the optimum at least 24 times)
        then upper_limit = average
    else lower_limit = average
    if (upper and lower limit are too close) then break
return averaged data from the last 25 runs
```

5.3 Experiment procedure

In the experiments, we will run the tested algorithm 25 times through the bisection method on all 4 problems (OneMax, EqPairs4, SlidingXor4, Trap4) and average the results. This averaging is being done to obtain more accurate estimates of minimal popsize and the other parameters. We will also run a single bisection on 400-bit instances of the problems above, to gain insight into how the improved algorithm scales with problem size.

5.4 Experiment on ECGA

In this experiment, we aim to replicate the speedup results from [4]. We should achieve speedup of about 10 on a 32-bit Trap4 and more than 1000x speedup on 4096-bit Trap4. However, running bisection on 4096-bit Trap4 is very slow. That is why we use only 400-bit problem instances to see how the speedup scales, because even those 400-bits can take up to a full week to finish one bisection. Running the 4096-bit problem instances seemed intractable on a relatively new desktop computer (CPU is AMD Ryzen 5 3600).

Figures 5.2-5.5 show averaged bisection results for all 4 fitness functions we have yet introduced: OneMax, Trap4, EqPairs4, SlidingXor4. Table 5.1 presents results of single bisection run on 400-bit instances of problems mentioned above to gain insight into how the improved greedy search scales with problem size.

Figure 5.2 shows the results on the OneMax problem. Minimal popsizes for ECGA and improved ECGA are within the same order of magnitude, and so are the fitness evaluations. CCC evaluations show clearly that the greedy search complexity dropped from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^2)$. As the problem size increases by an order (from 10^1 to 10^2), the CCC evals increase by 3 and 2 orders of magnitude, respectively. Finally, the duration shows a speedup of about 13 on 40-bit OneMax. On 400-bit OneMax, the speedup is about 200. These numbers indicate speedup even greater than expected, but let us not forget that the original article measured speedup on Trap4, while this is OneMax. We can conclude that on the OneMax problem, the improvement works and scales well.

Figure 5.3 shows the results on the EqPairs4. The minimal population size grows faster for the improved ECGA, but it is not a concern as long as the duration is shorter than that of regular ECGA. The fitness evaluations are more or less the same, which is neither good nor bad. There is about 10x speedup on 32-bit EqPairs4 while keeping fitness evaluations more or less the same. On a 400-bit EqPairs4, the fitness evaluations for improved ECGA are within the same order as the regular ECGA, but there is a 45x speedup. The decision to benchmark on 400-bits instead of 4000-bits makes it difficult to gauge the speedup effect, but qualitatively we can conclude that the improvement works and scales well on EqPairs4.

Results from the SlidingXor4 are in figure 5.4. The minimal population size for improved ECGA grows faster than that of regular ECGA. But the speedup is here:

1.5x on 32-bits, about 30x on 400-bits. The fitness evaluations grow faster for the improved ECGA; however, it seems the difference might plateau on larger problem sizes - improved ECGA keeps 10x the fitness evaluations on both 100-bit and 400-bit. Here the conclusion is the same as for EqPairs4, the speedup qualitatively works and scales with problem size.

It is interesting that even though the EqPairs4 should be an easier problem than SlidingXor4, ECGA seems to handle the SlidingXor4 better - the number of fitness evaluations is lower and the duration is shorter.

Figure 5.5 shows results on the Trap4. Here it looks that not much has changed. The minimal population size and fitness evaluations are slightly larger for the improved ECGA, but it is in the same order of magnitude. The durations are about the same. However, there is a 32x speedup on 400-bit Trap4, while the fitness evaluations are only doubled for the improved ECGA.

parameter	OneMax		EqPairs4	
Popsize / -	133	226	1.98e3	7.94e3
Fitevals / -	1.87e3	3.90e3	3.99e5	5.46e5
CCC evals / -	1.07e8	3.54e5	4.04e8	1.25e6
Duration / s	94.9	0.5	3.22e3	71.6
parameter	SlidingXor4		Trap4	
Popsize / -	242	726	6.85e3	2.04e4
Fitevals / -	4.30e3	4.34e4	1.93e5	3.94e5
CCC evals / -	1.46e8	6.56e5	1.78e8	7.48e5
Duration / s	184.7	6.2	3.16e3	97.4

Table 5.1. Results of one bisection run on 400-bit problem instances. The left column is for regular ECGA, the right is for improved ECGA.

5.4.1 Conclusion from experiment on ECGA

In the end, we can conclude that the proposed improved greedy search qualitatively works. The number of fitness evaluations is within the same order of magnitude as the number for regular ECGA, except for 400-bit SlidingXor4 (improved ECGA required 10x more fitness evaluations). Also, the significant drop in CCC evals matches the results from the original article.

The speedup was not always noticeable at smaller problem sizes (10 to 100 bits), but it certainly is here. However, it is hard to say if the speedup matches the declared 1000x on 4096-bit Trap4 because of the rather unfortunate decision to benchmark on 400-bits instead of 4096-bits. Perhaps a better solution would be to reduce the number of runs in one bisection (e.g., from 25 to 10) to get at least some results from 4096-bits. Maybe ask if there is a stronger computer available/what to do, or simply start the tests early enough so they would finish before thesis deadline. Furthermore, different scaling of the speedup can be explained by various factors. The article is 12 years old and both HW (CPUs, memory speeds) and SW (compilers) have progressed. Plus, the speedup depends on the implementation.

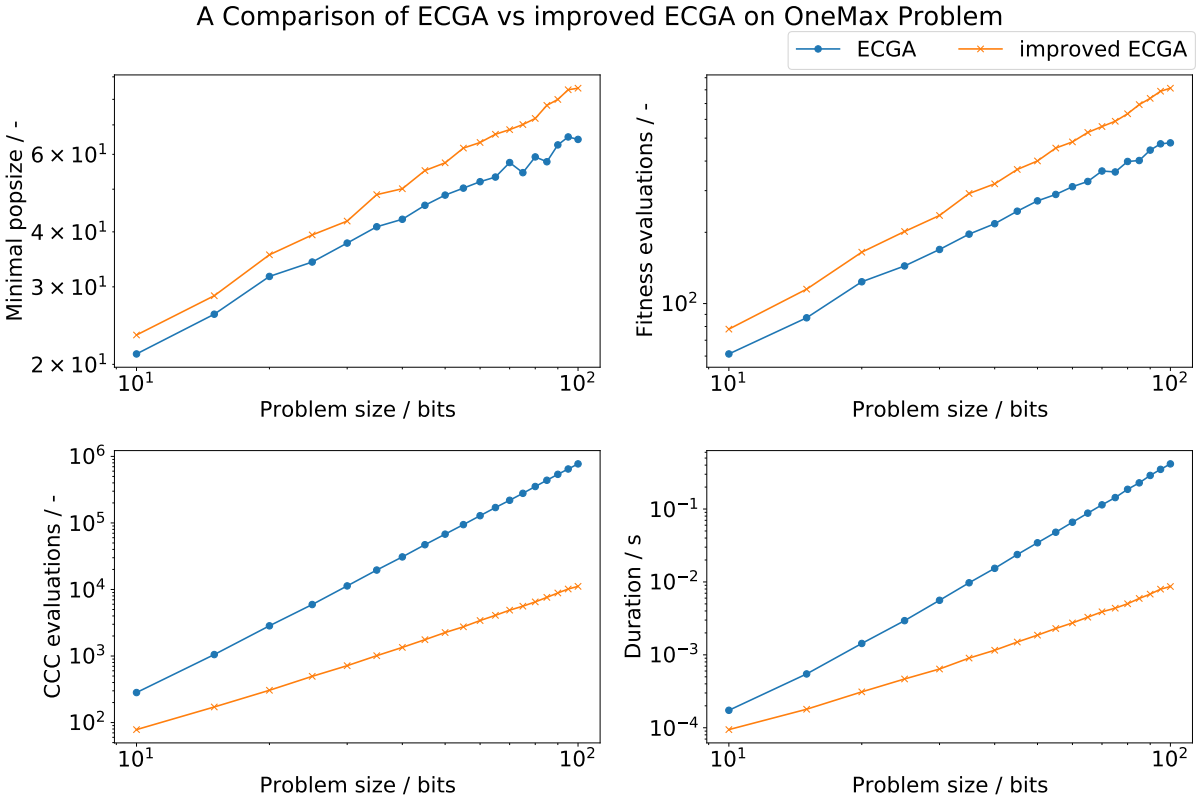

Figure 5.2. Averaged bisection results of ECGA and improved ECGA on OneMax problem.

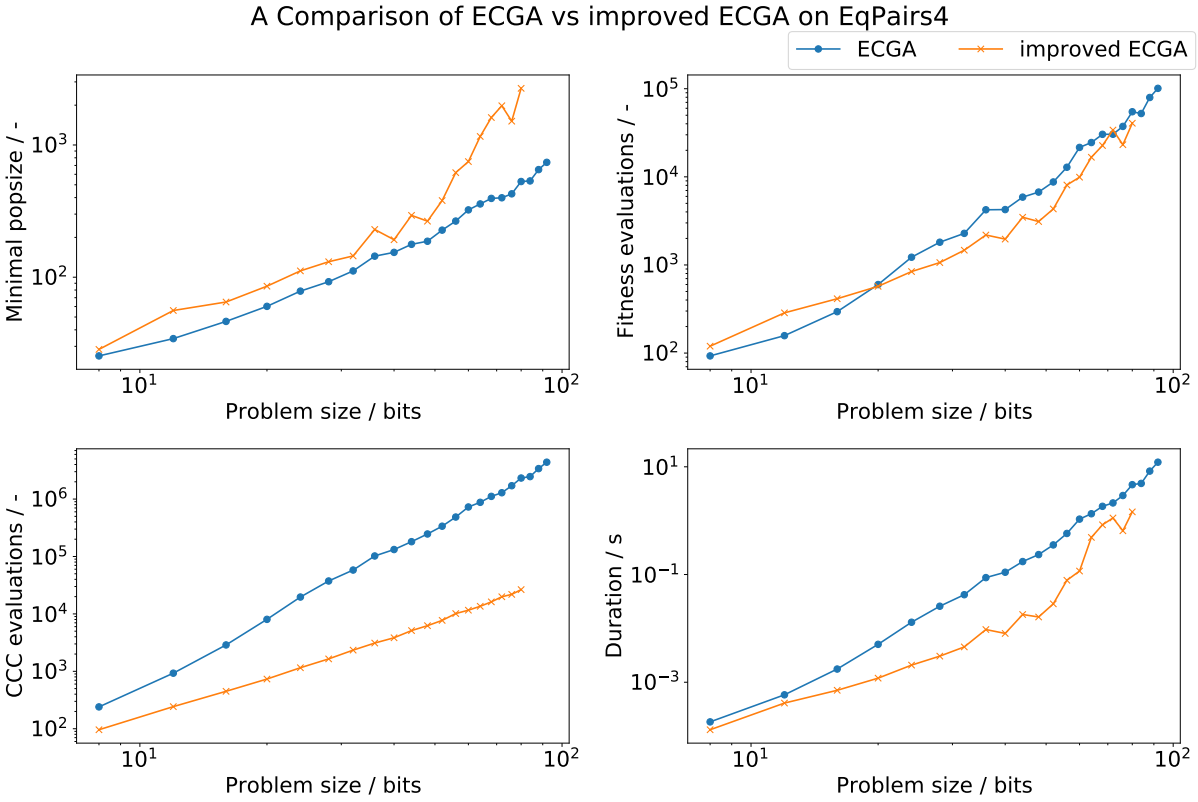

Figure 5.3. Averaged bisection results of ECGA and improved ECGA on EqPairs4 problem.

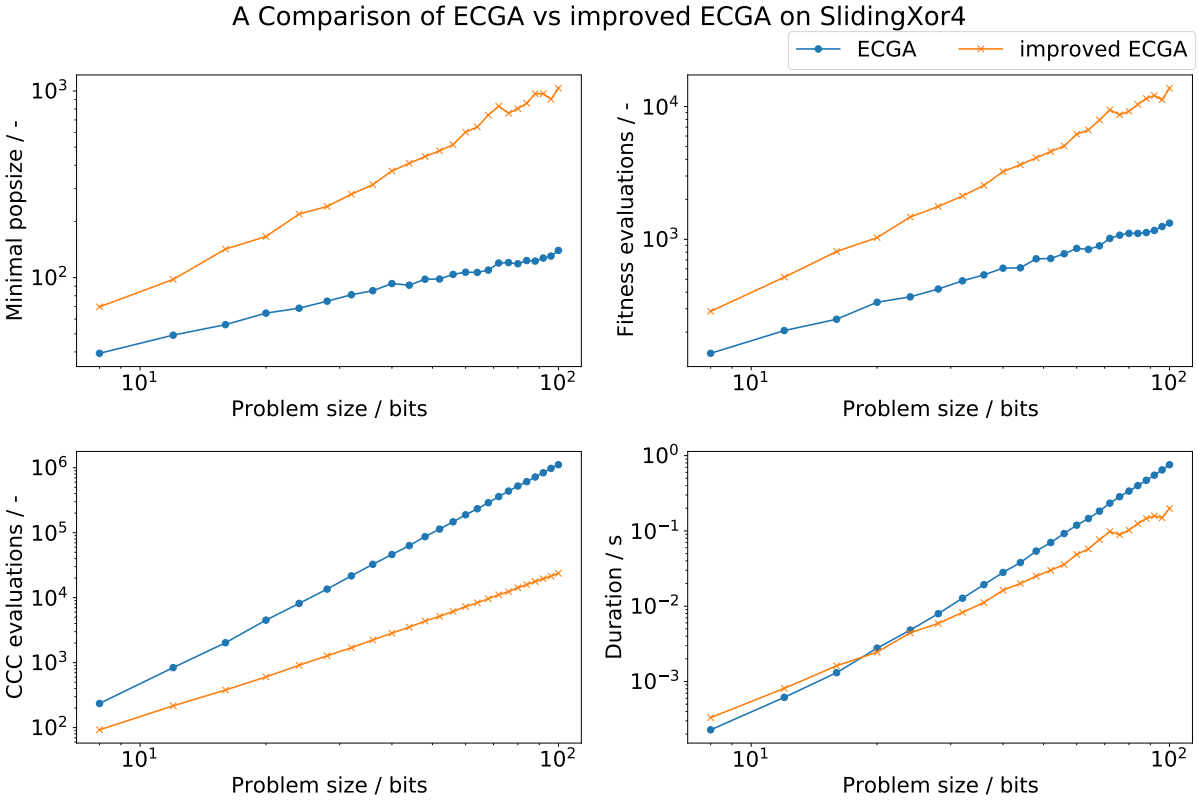

Figure 5.4. Averaged bisection results of ECGA, improved ECGA on SlidingXor4 problem.

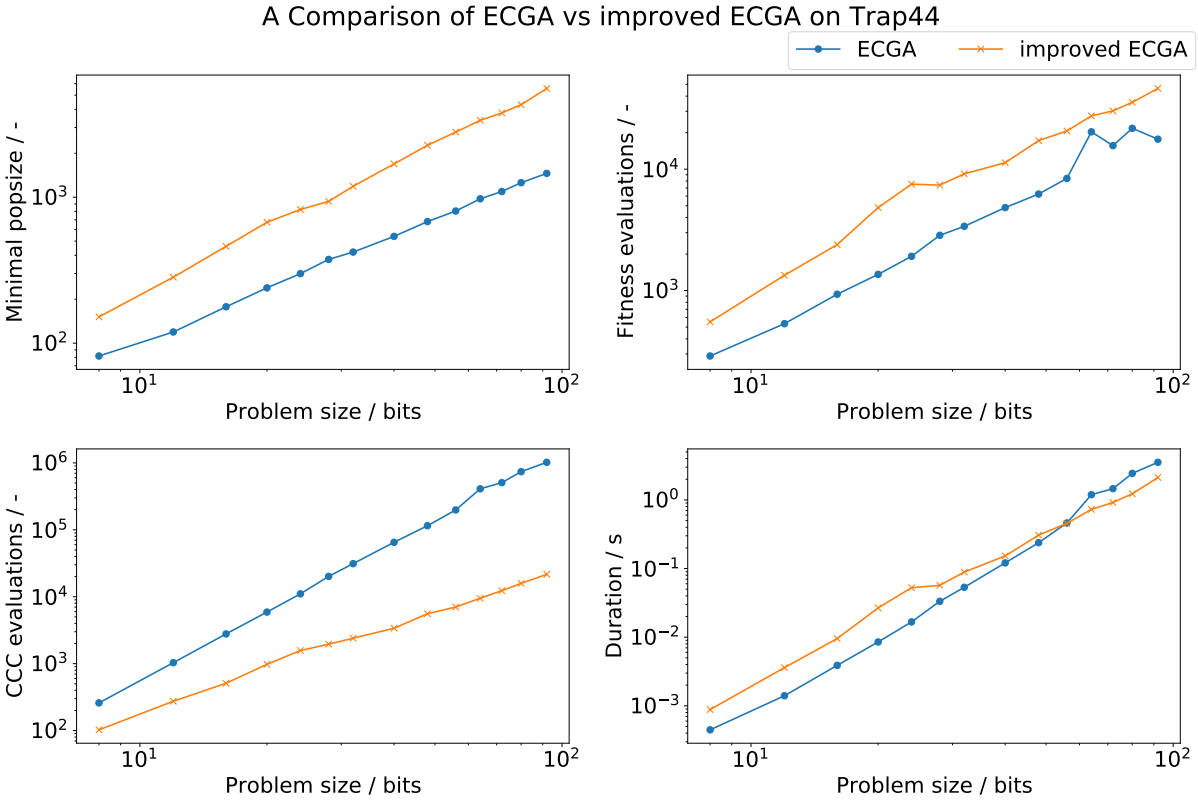

Figure 5.5. Averaged bisection results of ECGA and improved ECGA on Trap4 problem.

parameter	OneMax		EqPairs4	
Popsize / -	726	234	889	8.9e3
Fitevals / -	2.3e4	6.2e3	9.0e4	9.0e5
K2 evals / -	1.5e7	3.5e6	4.8e7	2.1e7
Duration / s	20.9	4.3	67.9	220.1
parameter	SlidingXor4		Trap4	
Popsize / -	4.5e3	6.1e3	1.2e4	3.0e4
Fitevals / -	2.2e5	4.4e5	3.5e5	3.0e6
K2 evals / -	2.8e7	2.0e7	1.8e7	3.2e7
Duration / s	185.9	124.2	282.7	1.2e3

Table 5.2. Results of one bisection run on 400-bit problem instances. The left column is for regular BOA, the right is for improved BOA.

5.5 Experiments on BOA

In this experiment, we aim to reproduce the speedup results from [4], but this time on BOA. For tests on BOA, we will use the same procedure we used for ECGA (described in section 5.3).

Figure 5.6 shows a comparison of BOA vs. improved BOA (BOA with improved greedy search) on the OneMax Problem. Improved BOA seems to run faster with fewer fitness evaluations; the speedup is about 10x on 100-bit OneMax. Nevertheless, looking at table 5.2, we can see that the speedup rather diminishes; it dropped from 10x to 5x. If we run the bisection on 1000-bit OneMax, we can see that the speedup drops to 4x. This is concerning because the speedup is somewhat small (4-10x), and it does not scale with problem size.

Figure 5.7 shows results on the EqPairs4 problem. There is a 10-15x speedup on smaller problem sizes, but all parameters of improved BOA suddenly worsen at 80-bits. It has been measured multiple times to verify it is not a random perturbation. Table 5.2 shows that is does not get better with larger problem sizes. In fact, regular BOA is 3 times faster than the improved BOA on 400-bit EqPairs4; the fitness evaluations are also 1 order of magnitude smaller.

Results on SlidingXor4 (fig. 5.8) are quite even. Runtime duration is almost the same for both algorithms, and so are the minimum popsizes and fitness evaluations. Table 5.2 shows that there is a slight speedup of 1.5, for the cost of double the fitness evaluations. The growth of K2 evaluations is about the same for both regular and improved BOA, which is not what we expected. Thus the improvement does not work on SlidingXor4.

Finally, the results on Trap4 are not great either (fig. 5.9). Duration and minimal popsize of the improved BOA seems to be on a par with the regular BOA. Fitness evaluations are about 4 times bigger on a 100-bit problem instance for the improved BOA. Furthermore, K2 evals are the worst here; they seem to grow slightly faster for the improved BOA. On 400-bit Trap4, the improved BOA fails to catch up with the regular BOA (tab. 5.2. It requires 10x more fitness evaluations, 2x more K2 evaluations, and is 4x slower than the regular BOA.

5.5.1 Experiment conclusion

It is clear from the experiment results that the improved BOA does not function as expected. It performs worse than the regular BOA on EqPairs4 and Trap4. The 1.5x

speedup on SlidingXor is rather negligible, and it comes with the price of double the fitness evaluations. Only OneMax showed some meaningful speedups (10x on 100-bit OneMax), but it diminishes with larges problem size. So, the improved BOA works neither from the outside (no speedup) nor the inside (K2 evals did not decrease).

One possible explanation is that the search procedure has been impaired too much, so the model search has to run a lot longer to reach the final model, which cannot be further improved. At that point it matches the performance of regular greedy search, if not worse. Another explanation is that the current BOA implementation is too simple. First, it can only add edges; there are neither edge removals nor edge reversals. Second, the implemented improved search is somewhat shallow. It selects a node and then searches among edges ending in the selected node. But it could very well search among the edges starting from the selected node. This possibility has, unfortunately, slipped my mind.

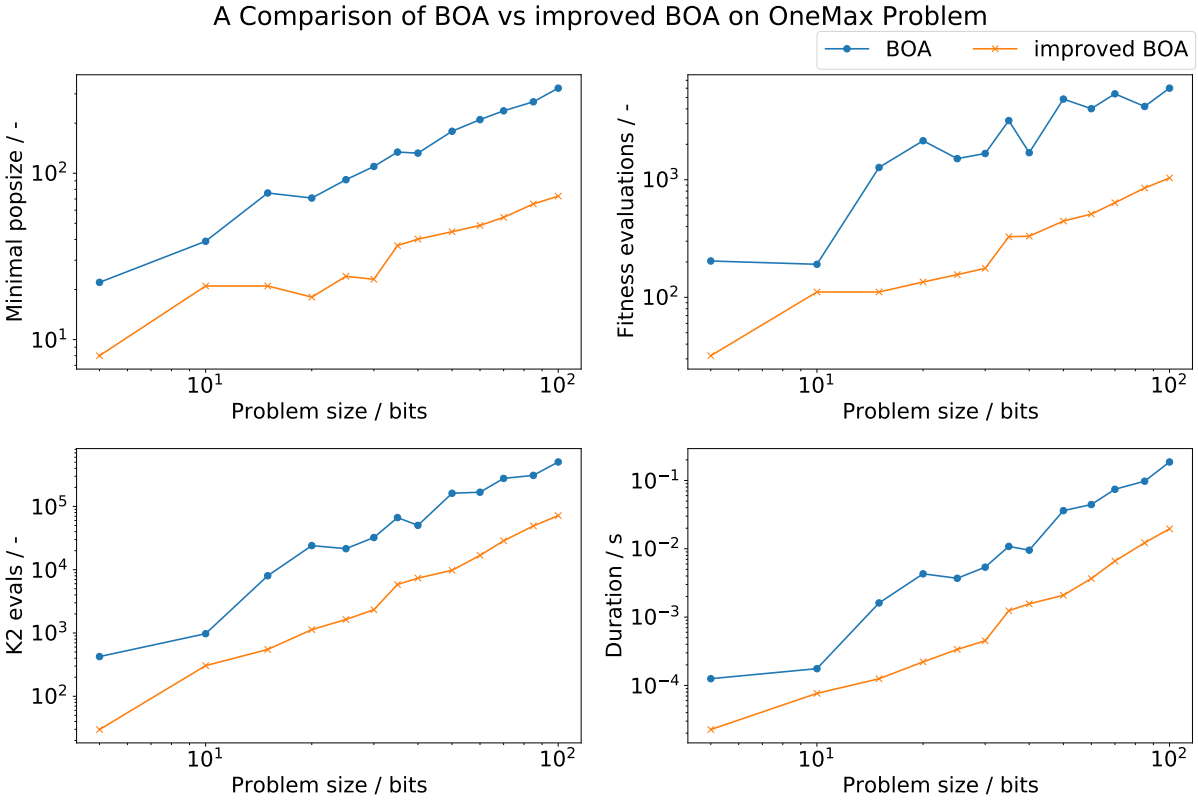

Figure 5.6. Averaged bisection results of BOA and improved BOA on OneMax problem.

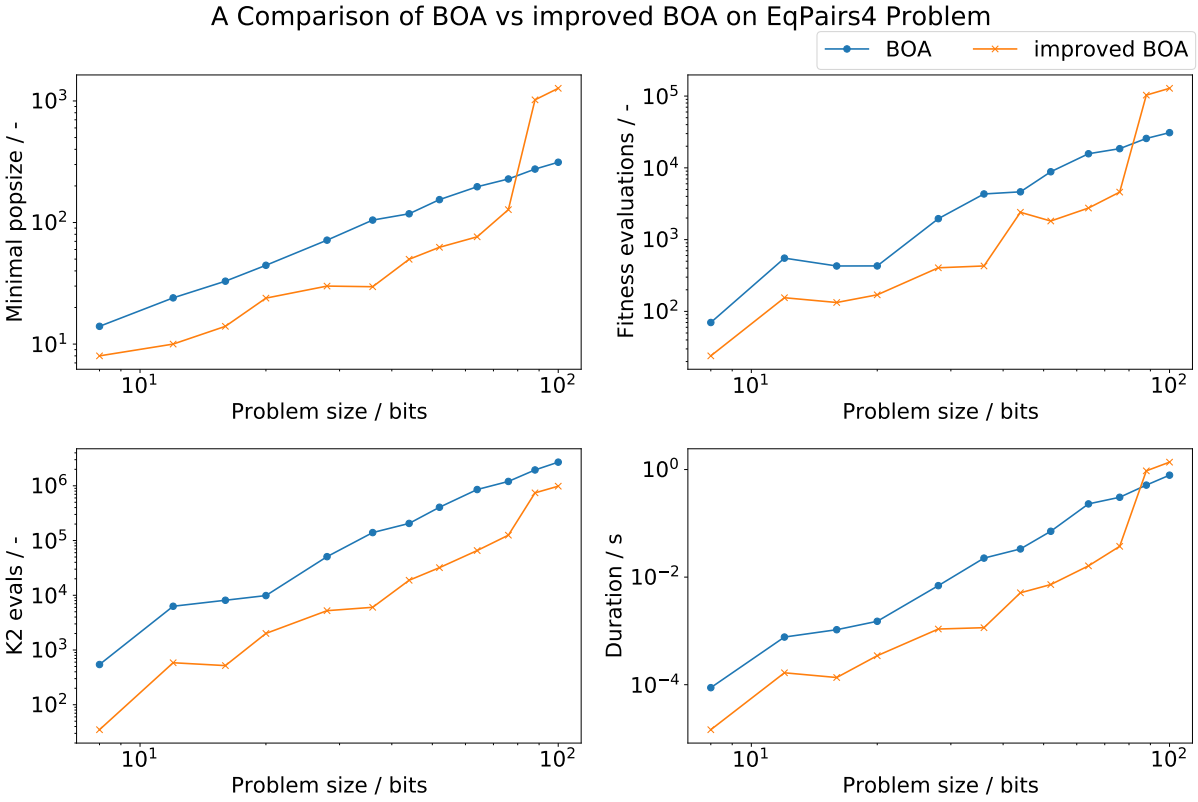

Figure 5.7. Averaged bisection results of BOA and improved BOA on EqPairs4 problem.

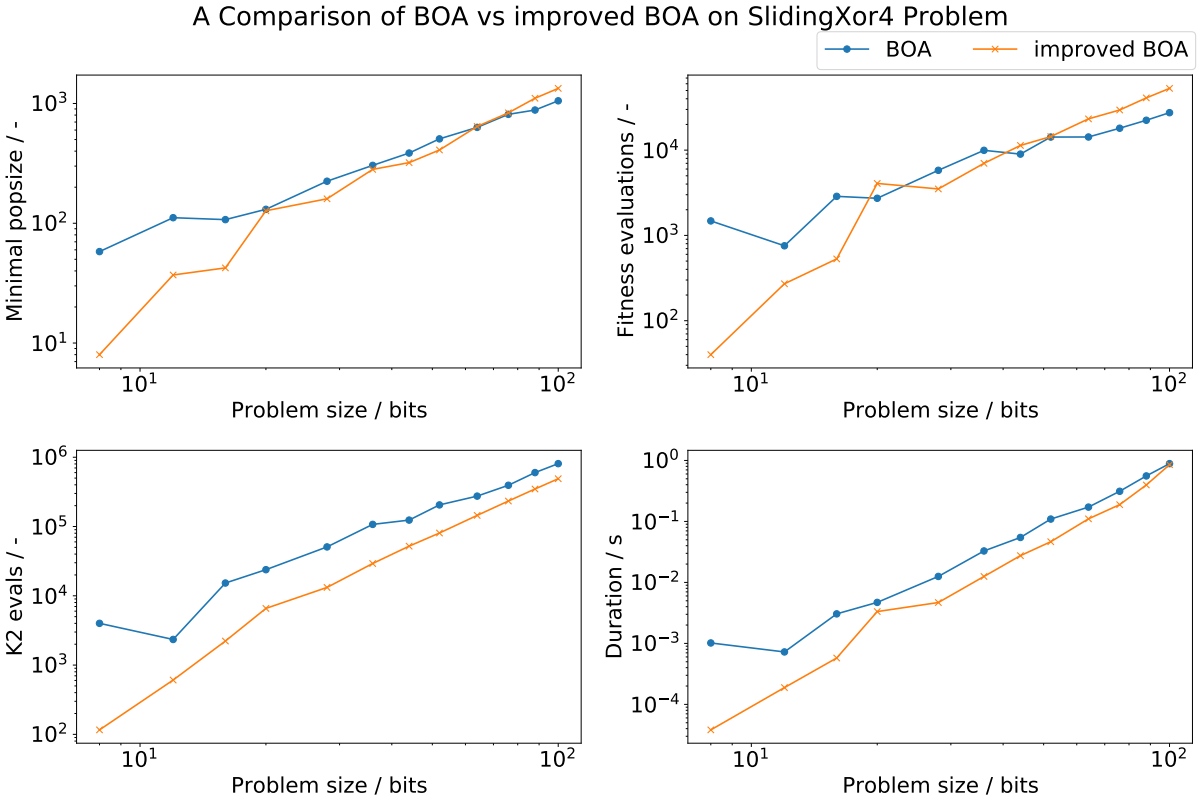

Figure 5.8. Averaged bisection results of BOA and improved BOA on SlidingXor4 problem.

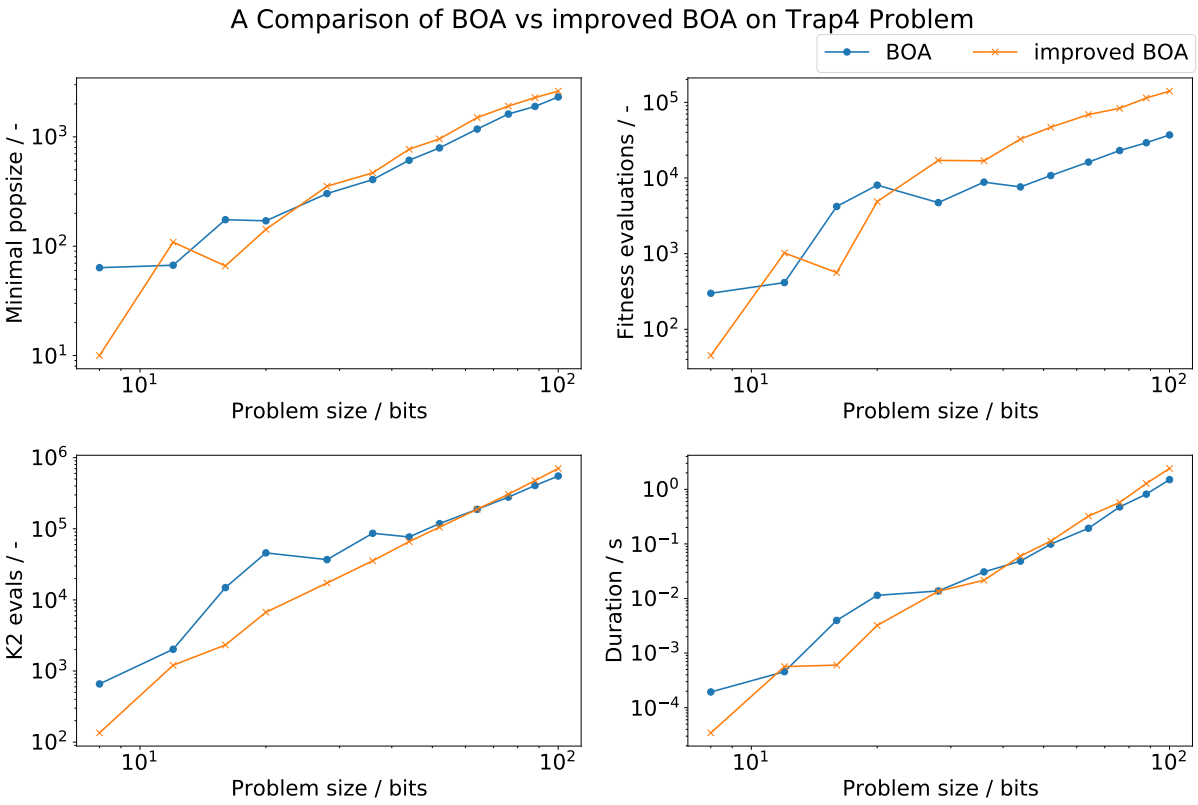

Figure 5.9. Averaged bisection results of BOA and improved BOA on Trap4 problem.

Chapter 6
Conclusion

The goal of this work was twofold. The first part was about replicating the speedup method for ECGA proposed by Duque et al. In the second part, we applied the same method on BOA and aimed to reproduce the speedup.

The first replication part was successful. The improved ECGA shows a significant drop in CCC evaluations on all 4 problems, which confirms that the model building procedure has reduced complexity. The speedups were confirmed on all the tested problems, while the original paper tested only the Trap4 function. Plus, the improved ECGA managed to keep the number of fitness evaluations at the levels of regular ECGA (except for SlidingXor4), which is great. The only shortcoming is that we did not replicate the 1000x speedup on 4096-bit Trap4 because of the computational complexity. Our strategy here has been to estimate the speedup from results on 400-bit problems, which I would say worked, but it does not strictly verify the declared results.

In the second part, the speedup method has been applied to BOA. However, this improved BOA failed to meet expectations. Neither did it run faster than regular BOA, nor did the number of K2 evaluations drop. The easy explanation for this failure would be that the improved BOA simply lost its ability to create a model of the problem, and would require the model search to run much longer to solve the problems. The fact that improved BOA runs slightly faster than regular BOA on OneMax, but slower on anything else, supports this. Another explanation is that the implementation of improved BOA lacks two edge operations (removal and reversal), and tries to connect only incoming edges to the selected node. More tests, with the missing features above, have to be done to see if BOA can be sped up.

References

[1] Jong, Kenneth A. De. *Evolutionary computation*. 2006 ed. Cambridge: MIT Press, 2006. ISBN 9780262041942.

[2] Luke, Sean. *Essentials of Metaheuristics*. second ed. Lulu, 2013. ISBN 978-1-300-54962-8.
`https://cs.gmu.edu/~sean/book/metaheuristics/`.

[3] Harik, Georges R., Fernando G. Lobo, and Kumara Sastry. Linkage Learning via Probabilistic Modeling in the Extended Compact Genetic Algorithm (ECGA). *Scalable Optimization via Probabilistic Modeling*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 39-61. Available from DOI 10.1007/978-3-540-34954-9_3.
http : //link.springer.com/10.1007/978 − 3 − 540 − 34954 − 9_3.

[4] Duque, Thyago S.P.C., David E. Goldberg, and Kumara Sastry. Improving the efficiency of the extended compact genetic algorithm. *Proceedings of the 10th annual conference on Genetic and evolutionary computation - GECCO '08*. New York, New York, USA: ACM Press, 2008, Vol. 2008, pp. 467-. Available from DOI 10.1145/1389095.1389181.
`http://portal.acm.org/citation.cfm?doid=1389095.1389181`.

[5] Pelikan, Martin. *Hierarchical Bayesian Optimization Algorithm*. Berlin: Springer, 2005. ISBN 978-3-540-23774-7.

[6] Pelikan, Martin. *A Simple Implementation of the Bayesian Optimization Algorithm (BOA) in C++ (version 1.0)*.
`https://web.archive.org/web/20200125101323/http://medal-lab.org/files/99011.pdf`. The source code is available at https://web.archive.org/web/20170327084810/http://medal-lab.org/files/sBOA.tar.gz .

Appendix A
Attachment content

The following files are in the attachment.

- benchmarking: a small python library for benchmarking
- CMakeLists.txt: cmake file used to compile programs. Instructions are in readme.md.
- eigen: a math library used in the ECGA (chapter 3) code. It is under the MPL2 license.
- grafy.py: a python helper script to automatically generate and save figures to pdf
- moje_ecga: my implementation of ECGA for the chapter 3
- pcg-cpp: a random number generator library for C++. It is under the MIT license.
- pelikan_simple_boa: M. Pelikan's simple BOA implementation in C++.
- pybind11: a C++ library used to create Python bindings
- pyeda: a small program that takes both ECGA and BOA in C++ and exposes them in Python.
- readme.md: file with compile instructions
- requirements.txt: file with Python packages requirements.
- run_benchmarks.py: a Python script that uses pyeda and benchmarking to run benchmarks on ECGA and BOA
- simplega: my implementation of the simpleGA used in chapter 2