Estimation-of-Distribution Algorithms

CSC_42021_EP Project

Contact: Martin (martin.krejca@polytechnique.edu)

General remarks. The source code for this project, the report, as well as the final defense need to be in **English.** You can use **any programming language** as long as the source code is formatted *and annotated* such that it is evident what it does even if one is not familiar with the language.

In general, you can **use any libraries** that you want. However, since some languages might provide shortcuts for certain tasks deemed important in this project, **you need to implement certain algorithms and data structures yourself.** This is stated explicitly in this document. When in doubt, reach out to me.

Please implement *efficient* and *legible* algorithms. It should be easy to understand what you are doing, but do not simplify to a degree where the performance starts to suffer.

If you do not know what programming language to use, consider D or Nim, not because they are best suited for this task, but because they are fun.

1 Context

A plethora of real-world optimization problems does not allow to obtain specific information about the objective function, for example, because it relies on simulations or is simply too complex to reason about. In these settings, where the problem acts as a black box, general-purpose optimizers thrive. One important class of these optimizers are *estimation-of-distribution algorithms* [PHL15] (EDAs). EDAs optimize a function f by refining a *probabilistic model* of the solution space of f. Starting with a uniform distribution, the model is updated iteratively by first creating samples from the model and then updating the model, based on the most promising samples. Ideally, this approach converges quickly to only producing optimal samples.

In this project, we implement and study the *significance-based compact genetic algorithm* [DK20] (sig-cGA, Algorithm 1), which is an EDA that has very good theoretical run time guarantees. We compare it to other (simple) general-purpose optimizers, and we observe its performance on more complex benchmarks.

2 General Terminology

We only consider *pseudo-Boolean* objective functions in this project, that is, functions that map bit strings to floating-point values. Formally, for a *problem size* $n \in \mathbb{N}_{\geq 1}$, we consider *fitness*

functions $f\colon \{0,1\}^n \to \mathbb{R}$. We call each $\boldsymbol{x} \in \{0,1\}^n$ an *individual*, and $f(\boldsymbol{x})$ the *fitness* of $\boldsymbol{x}$. For all $a, b \in \mathbb{N}$, let $[a..b] = [a,b] \cap \mathbb{N}$ and $[b] = [1..b]$. Moreover, we denote for all $i \in [n]$ the bit at position i of $\boldsymbol{x}$ by $\boldsymbol{x}_i$. Last, $|\boldsymbol{x}|_0$ denotes the number of 0s of $\boldsymbol{x}$, and $|\boldsymbol{x}|_1$ its number of 1s.

Matching the domain $\{0,1\}^n$, we consider probabilistic models that create random bit strings. To this end, we only consider *univariate* models of size n, which means that we determine the bit for each of the n positions based on a single model variable. In more detail, we use a *frequency vector* $\boldsymbol{p} \in [0,1]^n$, where for each $i \in [n]$, the *frequency* $\boldsymbol{p}_i$ denotes the independent probability that the sample has a 1 at position i. That is, for a sample $\boldsymbol{y}$ according to $\boldsymbol{p}$, assuming that $0^0 = 1$ here, we have for all $\boldsymbol{x} \in \{0,1\}^n$ that

$$\Pr[\boldsymbol{y} = \boldsymbol{x} \mid \boldsymbol{p}] = \prod_{i \in [n]} \boldsymbol{p}_i^{\boldsymbol{x}_i} \cdot (1 - \boldsymbol{p}_i)^{1-\boldsymbol{x}_i}. \tag{1}$$

When updating a frequency vector $\boldsymbol{p}$, we make use of *borders* that prevent each frequency in $\boldsymbol{p}$ to reach the extreme values 0 or 1, as the model may be stuck at incorrect values otherwise. We use the common borders of $\frac{1}{n}$ and $1 - \frac{1}{n}$. That is, the *restriction of* $\boldsymbol{p}$ to its borders is another frequency vector $\widetilde{\boldsymbol{p}}$ such that for all $i \in n$, we have

$$\widetilde{\boldsymbol{p}}_i = \max\left\{\min\left\{\boldsymbol{p}_i, 1 - \tfrac{1}{n}\right\}, \tfrac{1}{n}\right\}. \tag{2}$$

Last, since one of the competing algorithms is going to be an evolutionary algorithm (EA), which maintains a single best-so-far solution, we also define an operator that randomly changes a given individual directly. Such operators are known as *mutation*, their input is called the *parent*, and the output is called the *offspring*. We only consider *standard bit mutation* here. That is, given a parent $\boldsymbol{x} \in \{0,1\}^n$, the offspring $\boldsymbol{y} \in \{0,1\}^n$ is created by first copying $\boldsymbol{x}$ and then deciding for each position independently to flip it with probability $\frac{1}{n}$. That is, for each $i \in [n]$, we have $\Pr[\boldsymbol{y}_i = \boldsymbol{x}_i] = 1 - \frac{1}{n}$ and $\Pr[\boldsymbol{y}_i = 1 - \boldsymbol{x}_i] = \frac{1}{n}$.

Task 1. Implement reasonable data structures for individuals and for frequency vectors, and implement standard bit mutation that operates on your individuals.

The frequency vector should allow to create individuals according to equation (1), and it should be able to update single frequencies by setting them to a given floating-point number and then restricting the frequency according to equation (2).

The run time for standard bit mutation, *after copying the input*, should be linear in the number of flipped bits (plus some constant). For example, if we do not flip any bit, a run time of $\Theta(n)$ is unacceptable and should be rather $O(1)$. Explain why your implementation satisfies the run time requirement. Moreover, compute the expected run time of your implementation in big-O, disregarding the copy step.

3 The Significance-Based Compact Genetic Algorithm

The sig-cGA [DK20] (Algorithm 1) is an EDA that aims to *maximize* a pseudo-Boolean function of problem size $n \in \mathbb{N}_{\geq 1}$. It is based on the classic *compact genetic algorithm* [HLG99] (cGA, Algorithm 3), which is—despite its name—also an EDA and not a genetic algorithm. The sig-cGA

only features three values for its frequencies: $\frac{1}{n}$, $\frac{1}{2}$, and $1 - \frac{1}{n}$. We can think of these values as having a high confidence for a specific bit value (in case of the frequency being close to 0 or 1) or as being indecisive about which value is better. The decision whether to update a frequency is based on the statistical significance of a value in a history of promising bit values. These bit values are determined by selecting the better out of two samples each iteration.

Significance Estimation

We explain how to determine for a frequency vector $\boldsymbol{p} \in [0,1]^n$ as well as a position $i \in [n]$ whether to update $\boldsymbol{p}_i$, based on a history of samples that we specify further below. We note that this history is the result of selecting *promising* samples from those that we create over time.

Since our stochastic models are univariate, when creating $m \in \mathbb{N}_{\geq 1}$ samples according to $\boldsymbol{p}_i$, the bits in these m samples follow a binomial distribution with m trials and a success probability $\boldsymbol{p}_i$. Hence, by Chernoff bounds, the actual number of 0s and 1s within these samples is strongly concentrated around its expectation, which we call μ. Given a user-defined *confidence* $\varepsilon \in \mathbb{R}_{>0}$, the Chernoff bounds tell us which deviation from μ occurs with probability at most $n^{-\varepsilon}$. We use this deviation as an indicator to update the frequency.

Formally, we define the *significance function* sig as a function of a frequency, a history of bit values, and the confidence to the set of decisions, namely, $\{\text{down}, \text{stay}, \text{up}\}$, such that for all $p \in [0,1]$, all $H \in \{0,1\}^m$, and all $\varepsilon \in \mathbb{R}_{>0}$, we have

$$\operatorname{sig}(p, H, \varepsilon) = \begin{cases} \text{up} & \text{if } p \in \{\frac{1}{n}, \frac{1}{2}\} \text{ and } |H|_1 \geq mp + \varepsilon \max\{\sqrt{mp\ln(n)}, \ln(n)\}, \\ \text{down} & \text{if } p \in \{\frac{1}{2}, 1-\frac{1}{n}\} \text{ and } |H|_0 \geq mp + \varepsilon \max\{\sqrt{mp\ln(n)}, \ln(n)\}, \text{ and} \\ \text{stay} & \text{else.} \end{cases} \tag{3}$$

Note that mp is the expectation of the binomial distribution of the samples and that the first term in the maximum is essentially the standard deviation multiplied by a logarithmic factor in order to reduce the error probability to the order of at most $n^{-\varepsilon}$. The maximum ensures that we see at least a logarithmic number of samples before we make any decision (as we cannot be very confident otherwise).

Maintaining a History of Bits

Let $m \in \mathbb{N}$. A history $H \in \{0,1\}^m$ is simply a sequence of bits. The sig-cGA maintains a history *per position* and adds exactly one bit per iteration to each history. We describe below how to add a bit $b \in \{0,1\}$ to H (and consequently increase it to length $m+1$).

Since we make in each iteration a decision of whether to update a frequency, scanning an increasing history of bits for a significance is inefficient. Luckily, equation (3) only depends on the number of 0s and 1s of the history, not the history itself. Hence, it makes more sense to only save these values instead of an entire history. However, since the significance may only be recent and thus only observable in a smaller part of the history, the original sig-cGA scans H in exponentially increasing subsequences, always started at the most recent bit that was added. Due to this, we consider two versions of maintaining a history: a simplified one and the original one. We use the notation from above.

Algorithm 1: The significance-based compact genetic algorithm (sig-cGA). Given the confidence $\varepsilon \in \mathbb{R}_{>0}$, the problem size $n \in \mathbb{N}_{\geq 1}$, and the pseudo-Boolean function f, the sig-cGA *maximizes* f. It stops after a user-defined termination criterion.

$t \leftarrow 0$;
$\boldsymbol{p}^{(t)} \leftarrow (\frac{1}{2})_{i \in [n]}$;
for $i \in [n]$ **do** $H_i \leftarrow$ empty history;
while termination criterion not met **do**
  $\boldsymbol{x}^{(t,1)}, \boldsymbol{x}^{(t,2)} \leftarrow$ two independent samples from $\boldsymbol{p}^{(t)}$ (as in equation (1));
  if $f(\boldsymbol{x}^{(t,1)}) < f(\boldsymbol{x}^{(t,2)})$ **then** swap $\boldsymbol{x}^{(t,1)}$ and $\boldsymbol{x}^{(t,2)}$;
  for $i \in [n]$ **do**
    add $\boldsymbol{x}_i^{(t,1)}$ to H_i;
    for each subsequence h of H_i **do**
      $s \leftarrow \mathrm{sig}(\boldsymbol{p}_i^{(t)}, h, \varepsilon)$ (as in equation (3));
      $\boldsymbol{p}_i^{(t+1)} \leftarrow$ **switch** s **do**
        case up **do** $1 - \frac{1}{n}$;
        case down **do** $\frac{1}{n}$;
        otherwise do $\boldsymbol{p}_i^{(t)}$;
      if $s \neq$ stay **then**
        $H_i \leftarrow$ empty history;
        break;
  $t \leftarrow t + 1$;

Simplified history. Instead of storing H fully, we just store its length m as well as the two values $|H|_0$ and $|H|_1$. When adding b, we increase the length by one as well as either $|H|_0$ or $|H|_1$, depending on whether $b = 0$ or $b = 1$, respectively.

In order for this concept to be consistent with the following one, we treat the condensed version of H that consists of the three values above as a single subsequence.

Original history. In the original sig-cGA, the history is scanned from the latest bit to the oldest, in subsequences of exponentially increasing size. This aims at recognizing significances that only showed up recently, after no significance was detected for a long time.

More formally, we assume that the most recent bit in H is H_1 and that the oldest bit is H_m. The subsequences that the sig-cGA considers are all contiguous and all start at index 1.

In general, this approach is similar to the previous one, condensing each of the $\Theta(\log(m))$ subsequences to three values. However, since each newly added bit shifts in theory which bit belongs to which subsequence, it is a bit tricky to condense the subsequences directly. Thus, the sig-cGA follows a slightly different approach: It keeps a singly linked list L, where each element represents a condensed summary about the bits that are not present in the previous subsequences. That is, each element stores how many bits the respective part of the history contains, as well as how many 0s and 1s.

Algorithm 2: Sketch of the consolidation of a singly linked list L as required by the original history maintenance of the sig-cGA. This procedure does not return anything, but it changes the state of L.

```
curr ← L.head;
if curr is null then return;
next ← curr.next;
alreadySeenDouble ← false;
while next is not null do
    if curr.size = next.size then
        if alreadySeenDouble then
            m ←merge curr and next into a single element that summarizes their data;
            curr ← m and remove next from L (updating curr.next);
            next ← curr.next;
            alreadySeenDouble ← false;
        else
            alreadySeenDouble ← true;
            curr ← next;
    else curr ← next;
```

When adding a new bit b to H, we adjust the summary of the head of L if L summarizes fewer than $\ln(n)$ bits (due to equation (3))[1]. Otherwise, we add a new head to the list that summarizes only b, and then we consolidate L (Algorithm 2). The consolidation of L merges iteratively two consecutive elements of L that have the same size if and only if three consecutive elements have the same size. This leads to doubling the size of an element (and removing the one whose summary is included in the other one) and thus to elements that summarize subsequences of exponentially increasing sizes, noting that each size appears at most twice in L.

Task 2. Implement procedures for computing the significance function as well as both types of maintaining a history of bits.

For the original history of bits, how many merges occur at most when consolidating? Provide an example of an instance that leads to the maximum number of merges.

The Full sig-cGA

The complete sig-cGA is given in Algorithm 1. It builds a history for each position based on the better of two independent samples. Each history is scanned in increasing subsequences, and the update is performed and the history reset at the first significance found.

1 The actual sig-cGA always adds a new element of size 1, which leads to some wasted space and operations

Algorithm 3: The compact genetic algorithm (cGA). Given the hypotehtical population size $K \in \mathbb{R}_{\geq 1}$, the problem size $n \in \mathbb{N}_{\geq 1}$, and the pseudo-Boolean function f, the cGA *maximizes* f. It stops after a user-defined termination criterion.

$t \leftarrow 0$;
$\boldsymbol{p}^{(t)} \leftarrow (\frac{1}{2})_{i \in [n]}$;
while termination criterion not met **do**
  $\boldsymbol{x}^{(t,1)}, \boldsymbol{x}^{(t,2)} \leftarrow$ two independent samples from $\boldsymbol{p}^{(t)}$ (as in equation (1));
  if $f(\boldsymbol{x}^{(t,1)}) < f(\boldsymbol{x}^{(t,2)})$ **then** swap $\boldsymbol{x}^{(t,1)}$ and $\boldsymbol{x}^{(t,2)}$;
  for $i \in [n]$ **do**
    $\boldsymbol{p}_i^{(t+1)} \leftarrow \boldsymbol{p}_i^{(t)} + \frac{1}{K}\left(\boldsymbol{x}_i^{(t,1)} - \boldsymbol{x}_i^{(t,2)}\right)$;
  restrict $\boldsymbol{p}^{(t+1)}$ to the interval $[\frac{1}{n}, 1 - \frac{1}{n}]$ (as in equation (2));
  $t \leftarrow t + 1$;

Task 3. Implement the full sig-cGA. Leave it to the user to provide a fitness function as well as a termination criterion. For this project, it is sufficient if the termination criterion only depends on a single individual.

4 Competing Algorithms

We compare the sig-cGA to two easy algorithms. One of them is the *compact genetic algorithm* [HLG99] (cGA, Algorithm 3), which is an EDA and acted as the inspiration for the sig-cGA. The other one is the $(1 + 1)$ evolutionary algorithm ($(1 + 1)$ EA, Algorithm 4), which is closely related to random local search but uses a global search operator. Both algorithms maximize a given pseudo-Boolean function f of problem size $n \in \mathbb{N}_{\geq 1}$.

cGA Like the sig-cGA, the cGA (Algorithm 3) only creates two samples per iteration. However, it uses both samples in order to perform an update. To this end, it uses a parameter $K \in \mathbb{R}_{\geq 1}$ called the *hypothetical population size*. During an update, for each position, it changes the frequency by $\frac{1}{K}$ in the direction of the fitter offspring.

(1 + 1) EA Different from the cGA and the sig-cGA, the $(1 + 1)$ EA (Algorithm 4) maintains an explicit *population* of a single individual. Iteratively, it creates a single offspring via standard bit mutation and keeps the better of these two, favoring the offspring.

Task 4. Implement the cGA and the $(1 + 1)$ EA.

Algorithm 4: The (1 + 1) evolutionary algorithm ((1 + 1) EA). Given the problem size $n \in \mathbb{N}_{\geq 1}$ and the pseudo-Boolean function f, the (1 + 1) EA *maximizes* f. It stops after a user-defined termination criterion.

$t \leftarrow 0$;
$\boldsymbol{x}^{(t)} \leftarrow$ individual from $\{0,1\}^n$ chosen uniformly at random;
while termination criterion not met **do**
  $\boldsymbol{y}^{(t)} \leftarrow$ result from standard bit mutation applied to $\boldsymbol{x}^{(t)}$;
  $\boldsymbol{x}^{(t+1)} \leftarrow$ **if** $f(\boldsymbol{y}^{(t)}) \geq f(\boldsymbol{x}^{(t)})$ **then** $\boldsymbol{y}^{(t)}$;
  else $\boldsymbol{x}^{(t)}$;
  $t \leftarrow t + 1$;

5 Experimental Evaluation

We aim at assessing the performance quality of the different algorithms that we discussed above. To this end, we are interested in the number of fitness function evaluations until an optimal solution is created for the first time. Since each algorithm only terminates after a full iteration, this is the same as counting the number of iterations until the algorithm terminates, multiplying this number by the number of fitness evaluations per iteration, and then adding the number of fitness evaluations from the initialization.

We assume that the fitness of an individual is evaluated exactly once upon its creation and then never again. Hence, the sig-cGA and the cGA have *two* fitness evaluations per iteration, with no evaluation during the initialization, and the (1 + 1) EA has *one* fitness evaluation per iteration, with one evaluation during the initialization.

We note that for the sake of brevity, when we refer to the sig-cGA, we always mean two algorithms, that is, one for each version of how to maintain the history.

Benchmark Functions

We consider three common pseudo-Boolean benchmark functions, all subject to maximization and all with the all-1s bit string as their unique optimum.

OneMax The OneMax benchmark is often considered to be the simplest benchmark. It is a unimodal function consisting of a slope that leads directly to the global maximum. Formally,

$$\text{OneMax}\colon \boldsymbol{x} \mapsto |\boldsymbol{x}|_1.$$

LeadingOnes The LeadingOnes [Rud97] benchmark is also a unimodal function, but its slope to the global maximum is narrower than in OneMax. In particular, bits only contribute to the fitness if they are preceded only by 1s. Hence, bits at the back of an individual do not contribute to the fitness for a long time, which can pose a problem to certain algorithms. Formally,

$$\text{LeadingOnes}\colon \boldsymbol{x} \mapsto \max\{i \in [0..n] \mid \forall j \in [i]\colon \boldsymbol{x}_j = 1\}.$$

Jump The Jump [JW02] benchmark utilizes a parameter $k \in [n]$, where higher values indicate a (typically) harder problem[2]. In essence, Jump resembles OneMax but with a *plateau* of local optima for solutions with exactly $n - k$ 1s. Afterward, the fitness drastically decreases, with the exception of the all-1s bit string, which remains the global optimum. Formally,

$$\text{Jump}_k : \boldsymbol{x} \mapsto \begin{cases} k + |\boldsymbol{x}|_1 & \text{if } |\boldsymbol{x}|_1 \in [0..n-k] \cup \{n\}, \\ n - |\boldsymbol{x}|_1 & \text{else.} \end{cases}$$

> **Task 5.** Implement the OneMax, LeadingOnes, and Jump benchmarks as well as a termination criterion that returns true if and only if an individual that an algorithm newly created is optimal or a user-specified budget of fitness evaluations is surpassed.

Evaluation

Last, we compare the three algorithms among the three benchmarks, making use of theoretical guarantees for the parameters of the sig-cGA [DK20] and the cGA [CDK25; SW19] (one of which was recently proven by a student from Ecole Polytechnique!).

Setup Since we aim for statistically strong results, run each configuration multiple times. Explain your choice of the number of independent repetitions in your report.

In order to follow best scientific practices surrounding randomized methods, please use seeds for your random-number generators such that the results are reproducible.

Choose different problem sizes for each combination of function and algorithm. Your problem sizes should be at the very least 100. Explain your choices for the various problem sizes in your report and how these choices allow you to properly assess the performance of an algorithm.

For the sig-cGA and the cGA, use different parameter values for ε and K, respectively. For the sig-cGA, we only have guarantees of $\varepsilon > 12$ for OneMax and LeadingOnes, and we have no guarantees for Jump. Treat 12 as an upper bound for ε and see how low you can go. For the cGA, for OneMax and Jump, include at least $K = \sqrt{n}\ln(n)$ in your choice for K. For LeadingOnes, include at least $K = n\ln^2(n)$. Like for the sig-cGA, treat these values (up to small constant factors) as upper bounds and see how low you can go.

For Jump, include at least the cases $k \in \{2, 3\}$ and see how far you can go. Please keep in mind that the run time can scale quite badly in k, so choose a reasonable maximum number of fitness evaluations for termination. Make sure to discuss this cut-off properly in your report and to handle it carefully when conducting a statistical analysis.

In general, when making different choices for parameters, make *reasonable* steps. We aim at getting a good picture for many different values without wasting too much computing time. Think in advance about which values you think should be tried. Explain all of your choices in the report, including why you believe that they give you a good overall picture.

2 Depending on which problem classes and algorithms are considered, all parametrizations of Jump are equally hard, or some with a higher value of k are easier, but such settings are rarely practical.

Task 6. Run all the experiments as described in the setup. Summarize and visualize your results meaningfully. Explain what we can *learn* (not just see). If you think that it is helpful, measure and report other quantities that may explain the observed behavior. Conclude your findings with a general recommendation.

What to Hand In?

Please hand in your (sufficiently well documented) code files as well as a PDF report. The report should complement your code. Ideally, it explains the most important details about how you solved each task. It should focus on the high-level ideas.

Note that you are sometimes also explicitly asked to justify certain choices or claims. This is always done in the report. When justifying run times, a formal proof is not necessary, but please be precise enough so that it is easy to follow why the claimed bound should hold.

When reporting on the results of the experiments, please also try to *interpret* them. That is, do not just write *what* you see, but also provide possible explanations for *why* you see it.

In case of any doubts, please send me an e-mail. Have fun!

References

[CDK25] Marcel Chwiałkowski, Benjamin Doerr, and Martin S. Krejca. "Runtime analysis of the compact genetic algorithm on the LeadingOnes benchmark". In: *IEEE Transactions on Evolutionary Computation* (2025). Early access. DOI: 10.1109/TEVC.2025.3549929 (see page 8).

[DK20] Benjamin Doerr and Martin S. Krejca. "Significance-based estimation-of-distribution algorithms". In: *IEEE Transactions on Evolutionary Computation* 24.6 (2020), pp. 1025–1034. DOI: 10.1109/TEVC.2019.2956633 (see pages 1, 2, 8).

[HLG99] Georges R. Harik, Fernando G. Lobo, and David E. Goldberg. "The compact genetic algorithm". In: *IEEE Transactions on Evolutionary Computation* 3.4 (1999), pp. 287–297. DOI: 10.1109/4235.797971 (see pages 2, 6).

[JW02] Thomas Jansen and Ingo Wegener. "The analysis of evolutionary algorithms – a proof that crossover really can help". In: *Algorithmica* 34.1 (2002), pp. 47–66. DOI: 10.1007/s00453-002-0940-2 (see page 8).

[PHL15] Martin Pelikan, Mark Hauschild, and Fernando G. Lobo. "Estimation of distribution algorithms". In: *Springer Handbook of Computational Intelligence*. Springer, 2015, pp. 899–928. DOI: 10.1007/978-3-662-43505-2_45 (see page 1).

[Rud97] Günter Rudolph. *Convergence Properties of Evolutionary Algorithms*. Verlag Dr. Kovăc, 1997 (see page 7).

[SW19] Dirk Sudholt and Carsten Witt. "On the choice of the update strength in estimation-of-distribution algorithms and ant colony optimization". In: *Algorithmica* 81.4 (2019), pp. 1450–1489. DOI: 10.1007/s00453-018-0480-z (see page 8).