Evolutionary Algorithms

Proposal for a programming project for INF431, Spring 2014

version 14-02-19+23:09

Benjamin Doerr, LIX, Ecole Polytechnique

Difficulty *—***

1 Synopsis

This project deals with the implementation and experimental evaluation of a very recently developed evolutionary algorithm. To this aim, (i) the basic evolutionary algorithms has to be implemented (which is very easy), (ii) it has to be adapted to several to problems we want to apply it to (maximum matching, finding Eulerian cycles, ...; this needs an understanding of the datastructures and algorithms discussed in INF431, in particular, with respect to graphs), and (iii) the parameters and self-adaptive parameter choices have to be optimized via experiments.

In its most basic form, this project is not difficult (owing to the generally easy applicability of evolutionary algorithms). When putting more effort in this project, results may be obtainable that are of high scientific interest (owing to the fact that this algorithm was so far analyzed only by mathematical means, but no experimental/practical experience exists so far). The project is fully in English.

2 Details of the Topic

On last year's *Genetic and Evolutionary Computation Conference* (GECCO 2013), one of the two big international conferences on evolutionary algorithms, a new genetic algorithm was presented and analyzed on a very simple test problem (the infamous onemax-test function $f : \{0,1\}^n \to \mathbb{Z}; x \mapsto \sum_{i=1}^{n} x_i$, which simply counts the number of ones in the bit-string). Still, this work for the first time presented an algorithm for this classic test problem where crossover (generating a new solution candidate from two parent solutions) provably gave a significant speed-up. Moreover, this was also the first time that a self-adaptive parameter choice was (mathematically) proven to be useful. For these reasons, this paper was well received by the community (it won the best-paper award of the Genetic Algorithms track of the conference). On the other hand, no experience exists how this algorithm performs on more interesting optimization problems. This is what we want to overcome in this project.

2.1 Algorithm and Implementation

The algorithm we are talking about is given as Algorithm 1. The description should be self-contained, but if you need further information, please refer to the original paper [1] (8 pages, only the algorithm details are important, not the mathematical proofs)[1]. The algorithm here is described to work on the search space $\Omega = \{0,1\}^n$ of bit-strings of length n, which is a common representation in evolutionary algorithms. For some combinatorial problems, you will need to adapt this suitably. Please implement the algorithm in a way that for $\lambda = 1$, you omit the (useless) crossover phase, so that your algorithm then becomes the classic $(1+1)$ EA. A main point of interest is when the new algorithm outperforms the $(1+1)$ EA.

For all evolutionary algorithms, the choice of parameters (here: population size λ, mutation probability p and crossover probability c) is important (see also the next section on the experimental work). Previous work indicates that taking $p = \lambda/n$ and $c = 1/\lambda$ is a useful choice. For the value of λ, moderate size constant λ like $\lambda = 10$ gave good results. Take these values for your first steps until you gain more experience.

The new algorithm can be used with a *self-adaptive parameter choice.* This means that we keep $p = \lambda/n$ and $c = 1/\lambda$, so we only have one parameter λ to discuss. For this, we do a self-adaptive choice inspired by the so-called one-fifth rule (a rule of thumb stemming from evolution strategies): After each selection step, we set $\lambda = \lambda/F$ after a successful iteration (finding a truly better solution) and $\lambda := \lambda * \sqrt[4]{F}$ otherwise (see also Section 4.3 of [1]).

I have no particular preference for the programming language you use to implement this algorithm. Using Java, the language of the course, is perfectly fine. Anything else is equally possible. For any implementation, the code must be easily readable and follow the common rules of good practice (should all be no problem for a short algorithm as this). The main algorithmic challenge is to adapt the algorithm to different optimization problems. I will tell you mostly which representation of the solution candidates to use (when bit-strings are not appropriate) and also what a good mutation and crossover operator could be for this representation, but you have to find a suitable way to implement all this. To this aim, you will need your current knowledge of datastructures, in particular, for graphs. Note that my description in natural language always sound easy, but a simple natural language statement like "take as fitness function the number of connected components" still needs some thinking to implement.

Of course, you not only need to adapt the algorithm to the particular optimization problems, but you also need to find a way to experimentally evaluate the algorithm, that is, run the algorithm several times on suitable instances, store the raw data (good scientific practice), and compute aggregate data (e.g., average runtimes).

[1] All papers cited can be found at `www.mpi-inf.mpg.de/~doerr/X.pdf`, replace X by the paper number.

Algorithm 1: The $(1+(\lambda,\lambda))$ GA with offspring population size λ, mutation probability p, and crossover probability c. Mutation here means first generating a number ℓ according to a binomial distribution with parameters n and p and then generating each offspring independently by flipping exactly ℓ bits in the parent individual. Crossover consists of taking each bit independently with probability c from the second argument x', otherwise from the first argument x. Experience showed that $p = \lambda/n$ and $c = 1/\lambda$ lead to good results.

```
1  Initialization: Sample $x \in \{0,1\}^n$ uniformly at random and compute $f(x)$;
2  Optimization: for $t = 1, 2, 3, \dots$ do
3      Mutation phase: Sample $\ell$ from $\mathrm{Bin}(n,p)$;
4      for $i = 1, \dots, \lambda$ do
5          Sample $x^{(i)} \leftarrow \mathrm{mut}_\ell(x)$ and compute $f(x^{(i)})$;
6      Choose $x' \in \{x^{(1)}, \dots, x^{(\lambda)}\}$ randomly with $f(x') = \max\{f(x^{(1)}), \dots, f(x^{(\lambda)})\}$;
7      Crossover phase: for $i = 1, \dots, \lambda$ do
8          Sample $y^{(i)} \leftarrow \mathrm{cross}_c(x, x')$ and compute $f(y^{(i)})$;
9      Choose $y \in \{y^{(1)}, \dots, y^{(\lambda)}\}$ randomly with $f(y) = \max\{f(y^{(1)}), \dots, f(y^{(\lambda)})\}$;
10     Selection step: if $f(y) \geq f(x)$ then $x \leftarrow y$;
```

2.2 Optimization Problems to Regard

In this section, I describe some optimization problems that we want to regard. The first two are obligatory, then there is one optional one, and finally you can try own favorite problems.

2.2.1 OneMax (obligatory)

This is the classic test problem used in the paper [1]. As a first step into this topic, you shall implement the algorithm as described above and conduct a sequence of experiments with identical parameter settings as in the paper [1] (that is, redo the experiments giving Figure 3 in the paper, possibly only for smaller n to save computation time), and thus check that your implementation works.

To undertake at least one new experiment, try varying the self-adaption rule for the self-adaptive version of the algorithm. Currently, the so-called one-fifth success rule is proposed, that is, setting $\lambda = \lambda/F$ after a successful iteration (better solution found) and $\lambda := \lambda\sqrt[4]{F}$ otherwise (see Section 4.3 of [1]). Try some other settings for this rule, that is, replace the 4 (stemming from the one-fifth rule) by, e.g., 3, 6, or 9).

2.2.2 Maximum Matching (obligatory)

Given an undirected graph $G = (V, E)$, a matching is a set of edges that are pairwise disjoint. A maximum matching is a matching that has the maximal cardinality among all matchings. To compute a maximum matching (or one that has at least a large cardinality), we can use the bit-string representation. Let $m := |E|$. Enumerate the edges arbitrarily, that is, choose $e_1, e_2, ..., e_m \in E$ pairwise distinct (consequently, $E = \{e_1, \ldots, e_m\}$). Then we can uniquely describe a subset $M \subseteq E$ by a bit-string $x \in \{0,1\}^m$, namely by setting $M := \{e_i \mid x_i = 1\}$. Hence in the following, we talk about edge sets or bit-strings depending on what is more convenient. Having a bit-string representation, we can use mutation and crossover as in Algorithm 1. What is missing is the fitness function. A natural choice here is $f(M) := |M| - m \sum_{v \in V} \max\{0, \deg_M(v) - 1\}$, where $\deg_M(v)$ denotes the degree of the vertex v in (the subgraph consisting of the edges of) M. In simple words, our fitness is the number of edges in M minus a penalty of m for each excessive coverage of a vertex by an edge (one is OK, anything beyond counts as excess).

Experiment with Algorithm 1 (fixed λ and self-adaptive λ) on this problem in two ways: (i) Take arbitrary problem instances (graphs), that is, either suitable random instances or real-world instances you find somewhere. As random instances, I suggest graphs constructed as follows: Take n vertices. Each of them chooses uniformly at random exactly 3 friends. We then add an edge $\{i, j\}$ to the graph if i chose j as friend or vice versa. Most likely, your algorithm will not find an optimal solution in reasonable time. Then it makes more sense to analyze how the solution quality increases over time. (ii) Take as problem instances rings of size n, that is, the graph having the vertex set $[n] := \{1, \ldots, n\}$ and the edge set $\{\{i, i+1\} \mid i \in [n-1]\} \cup \{\{1, n\}\}$. Here, the algorithm should find an optimal solution in reasonable time (theory predicts $\Theta(n^4)$ iterations). Hence here we can analyze the time needed to find the optimum.

2.2.3 Eulerian Cycles (recommended)

An Eulerian cycle in a graph, informally speaking, is a cyclic path in the graph that traverses each edge exactly one. More formally, in a simple connected graph $G = (V, E)$ an Eulerian cycle is a sequence $e_1, \ldots, e_m$ of edges such that (i) $E = \{e_1, \ldots, e_m\}$ and $|E| = m$ and (ii) $|e_i \cap e_{i+1}| = 1$ and $|e_i \cap e_{i+1} \cap e_{i+2}| = 0$ for all $i \in [m]$ (here and in the following, to ease notation, let us agree to write $e_{m+1} := e_1$ and $e_{m+2} := e_2$. It is well-known that a graph contains an Eulerian cycle if and only if each vertex has even degree. Consequently, we shall only regard such graph here.

To compute Eulerian cycles, a number of evolutionary approaches have been proposed. One difficulty is that there is no good representation of cycles via bit-strings. So we have to come up with an alternative representation. The currently best one (taken from [2]) encodes solutions as sets of edge-disjoint paths and cycles via matchings in the adjacency lists as follows. Note first that when we have an Eulerian cycle, then at each vertex this cycle induced a matching (pairing) among the edges incident with this vertex. Whenever the cycle enters the vertex via some edge e, then we call the next edge f on the cycle the partner of e. Formally, the Eulerian cycle $e_1, \ldots, e_m$ for each vertex v induced the matching $M_v := \{\{e_i, e_{i+1}\} \mid i \in [m], e_i \cap e_{i+1} = v\}$ on the edges $E(v)$ incident with v. Note that this is indeed a perfect matching, that is, all edges incident with v have a partner. Conversely, if we have a perfect matching on $E(v)$ for each vertex v, then this gives rise to a set of edge-disjoint cycles. If this is only one cycle, it is an Eulerian cycle. Since starting with a perfect matching on each $E(v)$ might feel like solving the problem halfway before the evolutionary algorithm has started, let us be more modest and take as individuals of the evolutionary algorithm all families of matchings (perfect or not) on the $E(v)$. Translated back in the graph, this means that our individuals are sets of edge-disjoint paths and cycles not necessarily containing all edges. This approach also has the advantage that there is a natural initial individual, namely taking empty matchings for each $E(v)$.

Now that we know how our individuals look like, we can define a mutation operator. An *elementary mutation* consists of the following steps: (i) Choose a random vertex v. (ii) Choose a random edge $e \in E(v)$. See if it is matched, that is, if there is an $f \in E(v)$ such that $\{e, f\} \in M_v$. (iii) Choose a second random edge $e' \in E(v)$. See if it is matched, that is, if there is an $f' \in E(v)$ such that $\{e', f'\} \in M_v$. (iv) Match e and e' and, if possible, their former parters. Formally, remove $\{e, f\}$ and $\{e', f'\}$ from M_v (if possible), add $\{e, e'\}$ to M_v, and (if f and f' are defined) add $\{f, f'\}$ to M_v. Consequently, an elementary mutation matches two random edges and clears up the mess created by this. As before, a full *mutation* step does not necessarily change each individual in only a single location. Hence again, we sample a random number ℓ and then create each offspring by performing ℓ elementary mutations on it. For historical reasons, ℓ should be obtained from taking a Poisson distribution with expectation 1 and then adding one (to ensure that we do not apply zero elementary mutations).

For the *crossover* operator, it makes sense to copy whole M_v's from either argument. Hence for two individuals x and x' (which are families $(M_v)_{v \in V}$ and $(M'_v)_{v \in V}$ of matchings), the outcome of $\text{cross}_c(x, x')$ should be an individual x'' such that for all $v \in V$ independently, we have $M''_v = M_v$ with probability $1 - c$ and $M''_v = M'_v$ with probability c.

As test instances for the Eulerian cycle problem, I again suggest certain random graphs. I suggest to take the graphs used also for the matching problem, however, we have to ensure that they have an Eulerian cycle. To this aim, simply check all vertices if they have an even degree. If not, connect them to the next vertex with odd degree you find.

2.2.4 Own Ideas (optional)

You are encouraged to try this evolutionary algorithm on one or two of your favorite algorithmic problems. This means, you have to find a suitable representation for the solution candidates, a fitness function, a suitable mutation operator, and a suitable crossover operator. You also have to find suitable test instances. Possibly discuss your ideas with me before you start implementing. In general, do not be over-ambitious. For example, maybe it could well be that your algorithm does not find an optimal solution for the problem in reasonable time. Hence rather than waiting for an optimal solution, in such a situation (as in the general matching problem) you should only track (over time) how good the solutions are that your algorithm produces.

3 Experimental Evaluation and Parameter Tuning

While the main aspect of this project is a good implementation of the discussed algorithms, it is in the nature of evolutionary algorithms that they do not “just work”, but for maximum efficiency need the parameters to be optimized. This is usually done with a mix of intuition and experimental guidance.

The general suggestion for this particular algorithm is to take the mutation probability $p = \lambda/n$ and the crossover probability $1/\lambda$. A first question is if the algorithm under consideration is superior to the simple hillclimber obtained from taking $\lambda = 1$ (and omitting the crossover phase). This algorithm is known under the name *(1+1) evolutionary algorithm.* So try first if you can find values for λ that achieve this. Then you may try other parameter variations. Ultimately, everything is allowed that gives good results.

The experiments serve two purposes, (i) you help you improve the algorithms by finding good parameter and self-adaption rules, but (ii) also to document that you did good work (i.e., in fact found good algorithms). For the first point, you can do whatever you feel most appropriate. For the second, you have to document the outcomes of your experiments. These need not to be super-beautiful, but understandable. That is, I need to be able to understand what problem instances you took, you should store the raw data coming out the experiments (heaps of numbers, readable with a text editor or Excel or whatever), and reasonably aggregated data (e.g., average runtimes). No worries, I do not want you to do perfect statistics, but you should find a way to convince the critical reader (me) that your implementation is correct and that your claims on the algorithm performance are valid.

One very important point at the end: What do we mean by performance? Usually not the actual runtime (measured in seconds), but the number of fitness evaluations. The general view is that this is the most costly part, so counting them tells us what we want to know. Hence focus on this performance measure, but do also give some actual runtimes (in seconds).

References

[1] B. Doerr, C. Doerr, and F. Ebel, “Lessons from the black-box: Fast crossover-based genetic algorithms,” in *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2013.* ACM, 2013, pp. 781–788.

[2] B. Doerr and D. Johannsen, “Adjacency list matchings: an ideal genotype for cycle covers,” in *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2007.* ACM, 2007, pp. 1203–1210.