Implementation of the Electrostatic Potential Evolutionary Algorithm in Opt4J

Luis Gerhorst

Abstract—Meta-heuristic optimization allows the solution of diverse problems using generic algorithms. The author develops a modular implementation of such an optimizer described in "Obtaining Optimal Pareto Front Approximations using Scalarized Preference Information" by Braun et al. and proposes multiple changes in order to improve it's performance. In this context, we show that approximative approaches that trade numerical correctness for runtime can outperform strategies that do not allow for numerical errors. We also have an in-depth look at the implemented optimizer's structure and outline the differences with regard to the implementation provided by the original authors.

1 Introduction

Optimization with respect to certain objectives is a usual problem occurring in a wide range of engineering disciplines. Although there exist many mathematical approaches to the minimization or maximization of functions, the real world regularly does not fit into a simple formula that can be differentiated. Also, many problems have multiple conflicting goals (e.g. an engines fuel consumption and the power it delivers), thus not allowing us to name one individual best solution, but instead multiple solutions, each making certain trade offs with regard to the conflicting goals.

The naive approach to such a real-world problem would be to simply generate new random solutions until one of them fits our needs. This of course is only practicable for problems where the total number of solutions is small, for complex problems we instead need more intelligent, meta-heuristic, approaches. Those algorithms, to which the Electrostatic Potential Evolutionary Algorithm (ESPEA) [1] belongs, gradually improve a set of solutions (each representing a different preference with regard to the trade offs) until the solutions are, or are at least very close to, the optimal solutions. Also, since there exists an infinite number of trade off preferences when the objectives are continuous, we only supply a limited number of solutions to the user, from which one, matching the trade offs he's willing to make, can be chosen.

In the following we first get a high level understanding of the mechanics that make ESPEA work and then implement the algorithm using the Opt4J[1] Java[2] framework [2]. Finally, we analyze the performance of the implemented algorithm.

2 Algorithm

Optimization problems can be modeled as a function $f : X \to \mathbb{R}^n$ with n being the number of objectives. The goal of the optimizer is now to find the elements of the genotype space X, that minimize the components of the returned vector. If certain objectives are to be maximized, we first remap them to become a minimization problem. Since the number of solutions returned by the algorithm is limited, we only return solutions that make different trade offs regarding the conflicting objectives. When one solution has the same objective values as another solution except for one objective value, we only keep the solution obtaining the smaller value for that objective. However, if the solutions differ with regard to more than one objective, each reaching a smaller value than the other on one of the components, we can not simply choose one over the other. A multi-objective optimizer returns such a set of so-called Pareto optimal solutions.

In order to maximize the value the returned archive provides to the user, multiple criteria have to be met. First, since the exact trade offs the user is willing to make are not known (or may even change depending on the solutions returned), a wide range of solutions has to be explored. Second, although the exact preference is not known yet, users usually already have an idea of the solutions that are practical to them. Such information can be encapsulated in a function $W : \mathbb{R}^n \to \mathbb{R}_+$ mapping from a location in the objective space to a scalar value indicating the interest of the user in that area (the smaller the returned scalar, the greater is the interest). Provided such a function, the algorithm should put a focus on solutions from interesting areas.

In order to achieve both of these optimization goals, ESPEA uses concepts derived from Electrodynamics. In a closed physical system particles of the same charge naturally oppose each other, trying to minimize the electrostatic potential energy between them. This energy between two particles $a, b \in \mathbb{R}^n$ can be calculated using the formula

$$e(a,b) = C\frac{W(a) \cdot W(b)}{|a-b|}$$

where C is Coulomb's constant and $W : \mathbb{R}^n \to \mathbb{R}_+$ gives a particle's charge. Thus, the larger is the distance between two particles, the smaller is the electrostatic potential between them. The total energy of a system $A = \{y_1, \ldots, y_N\}$ containing multiple such particles is now simply the sum of the energies between all particles $\sum_{i=1}^{N}\sum_{j=1}^{i-1} e(y_i, y_j)$. Each member is said to introduce a certain energy into the system, being simply the sum of the energies between it and all other archive members

$$e(y_i) = \sum_{j=1, j\neq i}^{N} e(y_i, y_j)$$

Such systems, since they naturally try to minimize the encapsulated energy, seek towards an equal distribution when the available space is limited. Also, particles of lower charge move closer together in order to allow a greater distance towards particles of greater charge. These properties can be easily adapted for our purposes. Every solution is modeled as

1. http://opt4j.sourceforge.net
2. http://www.oracle.com/technetwork/java/javase/overview/

a particle with it's location being the values of the objective vector $y = f(x)$ and it's charge being the value returned by the preference function $W(y)$. Now, whenever we have to decide whether a new Pareto optimal solution should replace an existing archive member, we check whether the replacement decreases the total energy of the system. If it does, we have increased the diversity and have made one step towards an equal point distribution, also accounting for preference information by providing a higher density in interesting areas. We can easily check for an decrease in the total energy of the system by comparing the energy the old member a introduced $e(a)$ with the energy the new solution b introduces if the old one is removed, $e_{-a}(b)$. Using those concepts we can now define our evolutionary algorithm:

while *iterations* $<$ *limit* **do**
  generate a new candidate;
  discard the candidate if it is dominated by an archive member;
  remove all individuals from the archive dominated by the candidate;
  if *archive size* $<$ *capacity* **then**
    add the candidate to the archive;
  else
    find the members introducing more energy into the archive than the candidate would;
    replace one of them by the candidate;
  end
end

The generation of new candidates is done using a staged approach. At the beginning, individuals are generated randomly giving us a initial set of genotypes. Then, while the archive capacity was not reached yet, we combine those genotypes using simulated binary crossover and finally, once we have reached the archive capacity, we use differential evolution to generate new genotypes. Also, since there may actually be multiple members whose replacement decreases the total energy, we need a way to choose one out of this set. Three strategies are possible here, we can either replace the member introducing the largest energy, maximize the decrease in total energy achieved by the replacement or choose the member to be replaced in such a way, that the new candidate introduces the least energy.

3 Implementation

Opt4J uses the Google Guice[3] framework to allow a great modularity of the individual components. Using a technique known as dependency injection, every class defines it's dependencies as a list of interfaces that have to be implemented. On startup, Guice then injects the actually bound classes into the components that require them. ESPEA greatly benefits from this concept, allowing both the reuse of existing components from the Opt4J system but also making potential future modifications to the algorithm very easy.

3.1 General Structure

ESPEA's central component is the class `ESPEA` implementing the `IterativeOptimizer` interface. When this class is bound, Opt4J calls it's `next()` method in each iteration, asking the algorithm to perform one step of the optimization process. ESPEA does this by first generating a configurable number of new `Individuals`, followed by their completion and finally, evaluation, determining whether they should become part of the `Archive`. The generation of the initial random population is done directly in `ESPEA`. Once the archive contains at least two individuals, dependencies implementing the `Mating` interface are used for further `Individual` generation. `ESPEA` requires two such dependencies, one being used while the archive has not reached it's capacity yet, the other afterwards. In the default module, `MatingCrossoverMutate` and `MatingDifferentialEvolution` fulfill those functions. The first one being actually a part of the Evolutionary Algorithm included in Opt4J was adapted for our needs by injecting a custom `Coupler` into it. This `CouplerDistincTournament` class creates pairs from a given set of parents using tournament selection based on the energies the individuals introduce into the archive. It should be noted that, in contrast to the original author's implementation [1], the coupler uses a mechanism that creates distinct couples as long as the number of parents provided is sufficient. Both the tournament size (being 2 by default) and the `Comparator` determining the tournament winner are configurable so this class may be easily reused by other optimizers. Unfortunately, such great reusability was not possible with the differential evolution component, making `MatingDifferentialEvolution` basically an adapted copy of Opt4J's `DifferentialEvolution` optimizer. Every iteration, the implementation generates one offspring for each archive member in order to guarantee equal chance. This is another difference to the original implementation, where parents were chosen randomly one at a time.

Once a new generation of individuals has been created, we have to decide whether any existing archive members have been superseded. To do that, all generated individuals are passed on to the `EnergyArchive`'s `update` method, leaving it to decide what should be done next. Being a subclass of `AbstractArchive` our archive implementation only receives individuals that are Pareto optimal with regard to all existing archive members (removal of dominated archive members is also handled by the parent class). Thus, for each generated individual we only have to find all archive members introducing more energy than the candidate would and replace one of them with respect to the configurable replacement strategy. Calculation of energies introduced by members and candidates is abstracted into the the class `EnergyCache` whose inner workings will be discussed in chapter 3.3.

3.2 Scalarization Function

The implementation of new scalarization functions, denoted W in previous sections, is a task users of ESPEA are likely to encounter since it defines the user-specific trade offs they are willing to make. All scalarization functions have to be a subclass of the abstract `ScalarizationFunction`, implementing the function `double calculate(Objectives objectives)`. This method should calculate an individuals *charge* in a pure manner, that is, a call with objectives that compare `equal` should return the same value regardless of the archive state or any other mutable variable. Note that some scalarization methods, including the Nash Bargaining Solution and Proper Utility proposed in the original paper, explicitly require the Pareto front to calculate an individual's charge. The implementation provided by the original authors[4] uses the current archive to approximate the Pareto front, this however has the drawback of creating a circular dependency causing the

3. https://github.com/google/guice

4. https://sourceforge.net/projects/jmetalbymarlonso/

algorithm not to converge in some cases. Instead, the author recommends using the result of a previous optimizer run when such an approximation is required.

3.3 Energy Cache

The calculation of the electrostatic potential between individuals and the resulting total energy introduced by archive members is a central part of ESPEA. Implemented in a naive way, the algorithm spends a huge amount of it's runtime calculating the energies between individuals, making an optimization here highly worthwhile. `EnergyCache` relies both on the symmetry and the purity of the function used to calculate energies between two individuals. Purity allows us to cache the energies introduced by archive members in an array $\vec{s}$ where $s_i = e(y_i) = \sum_{j=1,j\neq i}^{N} e(y_i, y_j)$ and reuse them every time we need to check whether a candidate introduces less energy than any archive member. Unfortunately, since every archive member makes up a summand in the energies introduced by every other member, all energy sums calculated are no longer valid when we add or remove a member. Updating the energy sums and also initially calculating them efficiently leads us to a second, now $N \times N$, array E where entry $e_{ij} = e(y_i, y_j)$. The resulting structure can be outlined as follows

$$E = \begin{array}{c} \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_N \end{array} \begin{array}{c} \begin{array}{ccccc} y_1 & y_2 & y_3 & \cdots & y_N \end{array} \\ \begin{pmatrix} - & * & * & \cdots & * \\ e_{21} & - & * & \cdots & * \\ e_{31} & e_{32} & - & \cdots & * \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ e_{N1} & e_{N2} & e_{N3} & \cdots & - \end{pmatrix} \end{array} \Rightarrow \begin{pmatrix} s_1 \\ s_2 \\ s_3 \\ \vdots \\ s_N \end{pmatrix} = \vec{s}$$

Here we already make use of symmetrical nature of energies between particles, allowing us to half the storage usage and wipe out the entries above the diagonal. The components of $\vec{s}$ can easily be imagined as the entries of the corresponding row summed up (including the mirrored entries above the diagonal). Addition or removal of members is now simply a matter of inserting the corresponding row and column into E and applying the appropriate difference to $\vec{s}$. Unfortunately, this easy concept is flawed by the loss of precision faced when floating-point numbers become very large. The smaller is the distance between two particles, the greater is the electrostatic potential between them and thus also the corresponding summand in E, resulting in a, possibly huge, precision drop. Subtracting such a large summand from one of sums then results in an error that may be small in relation to the removed summand, but large with regard to the vector's other components. The alternative to using the inverse (INV), recalculation by adding up all remaining row entries (REC), does not seem very feasible since it raises the complexity of archive mutation from $\Theta(N)$ to $\Theta(N^2)$. Also, several mechanics exist that lower the influence of the resulting errors in $\vec{s}$:

1) The archive naturally converges towards a state where all members are equally far apart, also making them introduce similar amounts of energy. This limits the occurrence of large summands mostly to the early archive.
2) Since all energies are calculated for normalized `Objectives` (the largest known value being mapped to 1.0, the smallest to 0.0), we regularly have to invalidate the cache when new maxima or minima are discovered and rebuild both E and $\vec{s}$. Thought being very uncommon with regard to the total number of individuals discovered, this is especially common at the beginning which is also the time when the occurrence of large summands is usual.

Whether or not these mechanics together with the reduced complexity outweigh the negative consequences of incorrect sums will be further analyzed in chapter 4.2. Both INV and REC are implemented in ESPEA.

4 Simulation Results

To test the developed algorithm both the Black Box Challenge Test Server[5] and a local setup were used. The script for local testing and all generated data is included in the repository's `benchmark` directory. 200 repetitions were used to make results reproducible across runs. Also, a warm-up of 5 repetitions was used to make runtimes, measured in CPU seconds, more predictable. The machine used for testing contained an Intel Core i7 2.2GHz processor and 16GB of RAM, the exact specifications are available online.[6] The runtime environment included Opt4J version 3.1.4, Java version 1.8.0 and Mac OS X 10.12.6. The hypervolume metric, calculated using the `hv` program by Fonseca et al. [3], was chosen to compare the quality of returned Pareto fronts. To allow for a comparison between problems, hypervolumes were measured in percentages relative to the maximum sample value attained for each problem. This allows us to compute an average score that captures the overall performance of an optimizer across all tested problems, without giving greater weight to problems where larger hypervolumes are usual. The original hypervolumes are also included in the repository. If not noted otherwise, an archive capacity of 100 with 100,000 evaluations was used for all problems. Only continuous problems included in Opt4J were tested, namely DTLZ, WFG and ZDT.

4.1 Stateless Generation and Parallel Completion

For reference, two ideas that did not lead to an improvement in the quality of the returned Pareto fronts will also be commented on here. First, an earlier version of the algorithm did automatically switch back to the early offspring generation mechanism when the archive size dropped below the archive capacity. A comparison with the original approach run on the test server showed, that the new mechanism scored worse by 34 points for continuous problems. Thus, only the original approach got implemented in the final version.

A second idea was that by generating multiple individuals on each iteration, one could benefit from multi-core processors by completing the individuals in parallel. Opt4J already provides a `ParallelIndividualCompleter` that could simply be bound for this purpose. In practice however, simply binding the parallel completer, at least for the used generation size of 100, does not shorten the runtime. Coincidentally, it was observed that the optimizer's CPU usage only slightly exceeded 100%. It is suggested that for such small generation sizes and the analyzed problems, the parallelization overhead exceeds the benefits provided.

4.2 Energy Cache Mutation

In the following, we will compare the possibles approaches to update the energy cache outlined in chapter 3.3. Figure

5. https://www.cs12.tf.fau.de/lehre/lehrveranstaltungen/seminare/black-box-challenge-meta-heuristic-optimization-for-arbitrary-problems/

6. http://www.everymac.com/systems/apple/macbook_pro/specs/macbook-pro-core-i7-2.2-15-iris-only-mid-2014-retina-display-specs.html

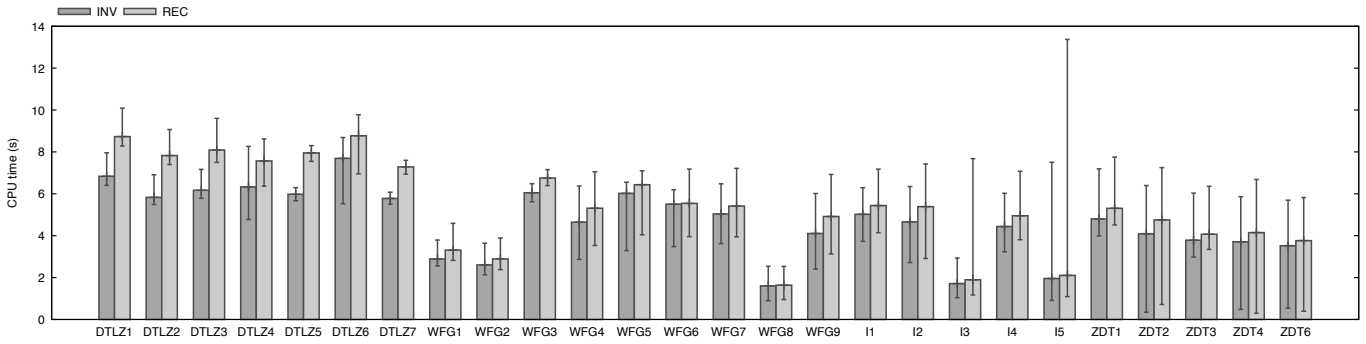

Figure 1. Runtime of ESPEA using INV and REC to evaluate 100,000 individuals against an archive of capacity 100

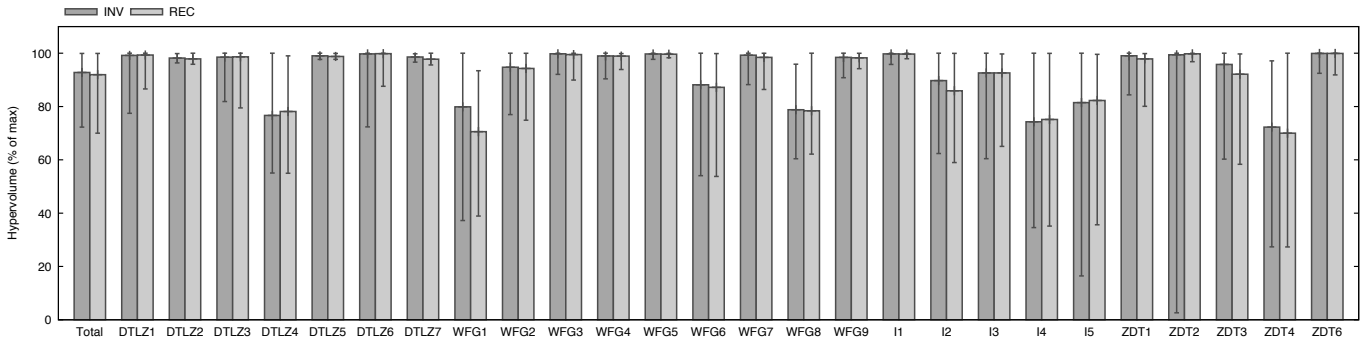

Figure 2. Hypervolumes attained by ESPEA with INV and REC after 1s for an archive of capacity 100

n	Hypervolume Difference
1	$8.58 \pm 0.50‰$
2	$11.02‰$
3	$12.74‰$

Figure 3. Average difference (INV minus REC) between the hypervolumes attained by INV and REC for varying configurations across all tested problems, runtime limit was n seconds, archive capacity $100 \times n$ and evaluation limit $100,000 \times n$

1 shows that INV consistently reduces the required runtime just as expected. The hypervolumes however did not differ significantly, putting REC in the lead with only $2.26 \pm 1.12‰$. By limiting the runtime and scoring the archives obtained by both strategies until they exceed the limit, we can determine which strategy uses it's runtime more efficiently. Figure 3 shows the average difference in the hypervolumes attained by both strategies for different runtime limits and archive sizes. As the complexity suggests, the larger the archive capacity the more INV is in the lead. Figure 2 compares the hypervolumes with INV and REC for a runtime of 1s on a per-problem basis. Note that although there was great variance in the returned results, the overall averages remained relatively consistent across runs. In conclusion it can be said that INV should be used in all cases since the negative effects of incorrect energies are easily canceled out by the reduced runtime and the resulting ability to perform more iterations in the same time frame.

5 Summary and Outlook

In the implementation of the algorithm a large amount of attention was given in order to make the code understandable, modular and thus reusable both by other components of the Op4J system but also persons seeking to improve the existing implementation. We developed an efficient mechanism to update the archive and showed that it, although being vulnerable to numerical errors, outperforms more defensive approaches that don't have such problems. Also, we outruled two possible changes to the algorithm and showed that they did not lead to a quality or performance improvement.

Future work may analyze whether parallel completion, as discussed in chapter 4.1, is able to provide a benefit for problems that take longer to complete than the ones tested here. Further optimization may also be possible by using a divide and conquer approach to extract the Pareto optimal solutions from a newly generated set of individuals.

List of Abbreviations

ESPEA the Electrostatic Potential Evolutionary Algorithm
INV inverse, updating a sum by subtracting the removed summand
REC recalculation, updating a sum by adding up all remaining summands

References

[1] M. Braun, P. K. Shukla, and H. Schmeck, "Obtaining Optimal Paretor Front Approximations using Scalarized Preference Information", in *Proceedings of the Annual Conference on Genetic and Evolutionary Computation*, 2015.

[2] M. Lukasiewycz, M. Glaß, F. Reimann, and J. Teich, "Opt4J - A Modular Framework for Meta-heuristic Optimization", in *Proceedings of the Genetic and Evolutionary Computing Conference (GECCO 2011)*, Dublin, Ireland, 2011, pp. 1723–1730.

[3] C. M. Fonseca, M. López-Ibáñez, L. Paquete, and A. P. Guerreiro. (2010). Computation of the hypervolume indicator, [Online]. Available: http : / / lopez - ibanez . eu / hypervolume (visited on 08/13/2017).