Evolving a Nelder–Mead Algorithm for Optimization with Genetic Programming

Iztok Fajfar iztok.fajfar@fe.uni-lj.si
Faculty of Electrical Engineering, University of Ljubljana, Ljubljana, 1000, Slovenia

Janez Puhan janez.puhan@fe.uni-lj.si
Faculty of Electrical Engineering, University of Ljubljana, Ljubljana, 1000, Slovenia

Árpád Bűrmen arpad.burmen@fe.uni-lj.si
Faculty of Electrical Engineering, University of Ljubljana, Ljubljana, 1000, Slovenia

doi:10.1162/EVCO_a_00174

Abstract

We used genetic programming to evolve a direct search optimization algorithm, similar to that of the standard downhill simplex optimization method proposed by Nelder and Mead (1965). In the training process, we used several ten-dimensional quadratic functions with randomly displaced parameters and different randomly generated starting simplices. The genetically obtained optimization algorithm showed overall better performance than the original Nelder–Mead method on a standard set of test functions. We observed that many parts of the genetically produced algorithm were seldom or never executed, which allowed us to greatly simplify the algorithm by removing the redundant parts. The resulting algorithm turns out to be considerably simpler than the original Nelder–Mead method while still performing better than the original method.

Keywords

Derivative-free optimization, Nelder–Mead, Direct search methods, Downhill simplex method, Genetic programming, Meta-optimization, Hyper-heuristic.

1 Introduction

Optimization is a process of finding the best possible element from a set of alternatives, subject to certain criteria. The simplest optimization problem consists of a real function minimization (or maximization) where one searches input values—possibly with regard to some constraints—trying to find the one yielding the lowest (or highest) output value. In many practical applications, function values are the only information available for use by an optimization algorithm (a.k.a. solver), either because no derivative information is available or it is too expensive to obtain. An algorithm that uses no derivative information in its search is called a *derivative-free algorithm*. Derivative-free optimization has a long history and has experienced a revived interest over the past two decades, fed by a growing number of solvers for problems from diverse fields (Rios and Sahinidis, 2013; Conn et al., 2009). One of the first published derivative-free algorithms was a downhill simplex optimization algorithm originally proposed by Nelder and Mead (1965), which is still subject to many studies and improvements (Bűrmen et al., 2006; Gao and Han, 2012; Wright, 2012), and is no doubt one of the most used and cited methods in the field of unconstrained optimization.

Manuscript received: August 3, 2015; revised: December 22, 2015; accepted: January 7, 2016.

Genetic programming (GP) is a type of domain-independent automatic programming paradigm inspired by biological systems. The method is able to automatically generate a computer program using only a high-level description of a problem at hand, which is given in the form of an objective statement. In practice, however, one usually uses problem-dependent primitives to effectively limit the search space. Although the approach is much older, the method that is today known as *standard GP* was formally proposed by Koza (1989, 1992). GP starts with a population of randomly created computer programs—commonly formulated as syntax trees rather than as sequences of instructions—which it continuously breeds over several generations, applying the Darwinian principle of survival of the fittest. The method simulates population evolution by mimicking natural processes such as crossover and mutation.

Since the early days of GP, innumerable researchers have contributed promising results in many areas, such as image and signal processing, economic modeling and time series prediction, medicine, process control, computer games, and even arts (Poli et al., 2008; Koza, 2010). Until very recently, however, the application of GP to synthesize programs that can deal with classes of problems rather than solve one specific problem (a so-called *hyperheuristic*) has not been paid sufficient attention (White et al., 2013; O'Neill et al., 2010; Pappa et al., 2014). A book chapter by Burke et al. (2009) is a decent review of early attempts of using GP as a hyperheuristic. Typically, GP has been used as a hyperheuristic for combinatorial optimization and other combinatorial problems, either in its standard or in some of its modified forms (Burke et al., 2006; Koohestani and Poli, 2014; Nguyen et al., 2012; O'Neill et al., 2014; Martin and Tauritz, 2013; Nguyen et al., 2015; Sim et al., 2015). To the best of our knowledge, there has been a single important attempt at evolving a real function optimizer using GP. Dioşan and Oltean (2009) evolved several evolutionary algorithms for real function optimization using multi-expression programming, which is a linear GP variant. While their general idea of evolving evolutionary algorithms received some attention in the evolutionary computation community, no further attempts were made in the direction of real function optimization.

In this article, we employ standard GP to produce a deterministic rather than a stochastic real function optimization algorithm. Our aim is to breed a population of programs that will result in a deterministic downhill simplex optimization algorithm, similar to the original method proposed by Nelder and Mead (1965).

The following section is a brief review of the original Nelder–Mead (NM) optimization algorithm. In Section 3, we derive from the original NM algorithm a set of terminals and functions to be used as a primitive set for GP. We rewrite the original NM method using this primitive set and shape it into a tree-formed structure, just to see whether it can be done before actually running GP. The other preparatory steps and GP parameters that we used in our experiment, such as the fitness measure and the population size, are described in Section 4. In Section 5, we compare the general performances of the potentially useful solvers produced by our GP system with the performances of the original as well as of tree-formed NM using a standard set of test functions. A more detailed comparison of the solvers is presented in Section 6, where we take a closer look at the optimization results by individual test functions. We present an even more in-depth analysis of the best of the genetically obtained solvers in Section 7, where we also simplify the algorithm on the basis of findings from the analysis. Finally, in Section 8 we summarize the main conclusions and make suggestions for further work to be done.

2 The Nelder–Mead Algorithm

In this section, we summarize the standard Nelder–Mead algorithm as used by Lagarias et al. (1998) and Bűrmen et al. (2006), which is a slightly modified version of the originally published algorithm (Nelder and Mead, 1965). However, this modification only remedies some minor ambiguities present in the original algorithm and does not importantly influence the algorithm's behavior.

The aim of the NM algorithm is, given a real cost function (CF) of n real parameters $f : \mathbb{R}^n \to \mathbb{R}$, without constraints, to find a parameter vector $\boldsymbol{x}_{\min} \in \mathbb{R}^n$ such that

$$\exists \delta > 0 : f(\boldsymbol{x}_{\min}) \leq f(\boldsymbol{x}_{\min} + \boldsymbol{a})\ \forall \boldsymbol{a} \in \mathbb{R}^n,\ \|\boldsymbol{a}\| \leq \delta.$$

The $\boldsymbol{x}_{\min}$ vector is a local minimum of the cost function f in the neighborhood $\|\boldsymbol{a}\|$.

The NM algorithm uses an n-dimensional simplex defined by $(n+1)$ vertices $\boldsymbol{v}_0, \boldsymbol{v}_1, \ldots, \boldsymbol{v}_n \in \mathbb{R}^n$, which it manipulates by iteratively replacing the worst vertex by a better one. Before each iteration of the algorithm, vertices are relabeled according to the CF value they produce, so that $f(\boldsymbol{v}_0) \leq f(\boldsymbol{v}_1) \leq \ldots \leq f(\boldsymbol{v}_n)$. Three special vertices are labeled by the indices b, sw, and w, to represent the vertices with the best (lowest), second worst (second highest), and worst (highest) values of the CF, respectively:

$$\boldsymbol{v}_{\mathrm{b}} = \boldsymbol{v}_0,\ \boldsymbol{v}_{\mathrm{sw}} = \boldsymbol{v}_{n-1},\ \boldsymbol{v}_{\mathrm{w}} = \boldsymbol{v}_n. \tag{1}$$

The algorithm also uses the centroid (or geometric center) $\boldsymbol{c}$ of all vertices except the worst one:

$$\boldsymbol{c} = \frac{1}{n} \sum_{i \neq \mathrm{w}} \boldsymbol{v}_i. \tag{2}$$

Four different operations (moves) are used to compute a new vertex candidate to replace the worst one in each iteration. They are all expressed with the same equation, only with different values of γ:

$$\boldsymbol{v}_{\mathrm{new}} = \boldsymbol{c} + \gamma(\boldsymbol{c} - \boldsymbol{v}_{\mathrm{w}}). \tag{3}$$

Note that $\boldsymbol{v}_{\mathrm{new}}$ will always lie on the line joining $\boldsymbol{c}$ and $\boldsymbol{v}_{\mathrm{w}}$, only at different positions. Depending on whether $\boldsymbol{v}_{\mathrm{new}}$ lies on the far side or on the near side of $\boldsymbol{c}$ from $\boldsymbol{v}_{\mathrm{w}}$, and how far, we get different names for the same transformation given by (3). In this study we use the γ values originally proposed by Nelder and Mead (1965):

$$\boldsymbol{v}_{\mathrm{new}} = \boldsymbol{c} + 1(\boldsymbol{c} - \boldsymbol{v}_{\mathrm{w}}) = \boldsymbol{v}_{\mathrm{r}} = \mathrm{refl}(\boldsymbol{c}, \boldsymbol{v}_{\mathrm{w}}) \text{ (Reflection)}, \tag{3a}$$

$$\boldsymbol{v}_{\mathrm{new}} = \boldsymbol{c} + 2(\boldsymbol{c} - \boldsymbol{v}_{\mathrm{w}}) = \boldsymbol{v}_{\mathrm{e}} = \mathrm{exp}(\boldsymbol{c}, \boldsymbol{v}_{\mathrm{w}}) \text{ (Expansion)}, \tag{3b}$$

$$\boldsymbol{v}_{\mathrm{new}} = \boldsymbol{c} + 0.5(\boldsymbol{c} - \boldsymbol{v}_{\mathrm{w}}) = \boldsymbol{v}_{\mathrm{oc}} = \mathrm{outContr}(\boldsymbol{c}, \boldsymbol{v}_{\mathrm{w}}) \text{ (Outer contraction)}, \tag{3c}$$

$$\boldsymbol{v}_{\mathrm{new}} = \boldsymbol{c} - 0.5(\boldsymbol{c} - \boldsymbol{v}_{\mathrm{w}}) = \boldsymbol{v}_{\mathrm{ic}} = \mathrm{contr}(\boldsymbol{c}, \boldsymbol{v}_{\mathrm{w}}) \text{ (Inner Contraction)}. \tag{3d}$$

Occasionaly, the whole simplex is reduced toward the best vertex $\boldsymbol{v}_{\mathrm{b}}$ by replacing all but the best vertex according to the formula

$$\boldsymbol{v}_i = \boldsymbol{v}_{\mathrm{b}} - 0.5(\boldsymbol{v}_{\mathrm{b}} - \boldsymbol{v}_i)\ \forall i \in \{1, 2, \ldots, n\} \text{ (Reduction)}. \tag{4}$$

Finally, Algorithm 1 represents a single iteration of the NM method, which repeats until certain stopping criteria are met.

Algorithm 1 One iteration of the original Nelder–Mead algorithm

```
Order the simplex vertices so that $f(\boldsymbol{v}_0) = f(\boldsymbol{v}_\mathrm{b}) \leq f(\boldsymbol{v}_1) \leq \ldots \leq f(\boldsymbol{v}_{n-1}) = f(\boldsymbol{v}_\mathrm{sw}) \leq f(\boldsymbol{v}_n) = f(\boldsymbol{v}_\mathrm{w})$
if $f(\boldsymbol{v}_\mathrm{r}) < f(\boldsymbol{v}_\mathrm{b})$ then            ▷ If the reflected vertex is better than the best
    if $f(\boldsymbol{v}_\mathrm{e}) < f(\boldsymbol{v}_\mathrm{r})$ then        ▷ If the expanded vertex is better than the reflected
        Replace $\boldsymbol{v}_\mathrm{w}$ with $\boldsymbol{v}_\mathrm{e}$.
    else
        Replace $\boldsymbol{v}_\mathrm{w}$ with $\boldsymbol{v}_\mathrm{r}$.
    end if
else if $f(\boldsymbol{v}_\mathrm{r}) < f(\boldsymbol{v}_\mathrm{sw})$ then     ▷ If the reflected vertex is better than the second worst
    Replace $\boldsymbol{v}_\mathrm{w}$ with $\boldsymbol{v}_\mathrm{r}$.
else if $f(\boldsymbol{v}_\mathrm{r}) < f(\boldsymbol{v}_\mathrm{w})$ then      ▷ If the reflected vertex is better than the worst
    if $f(\boldsymbol{v}_\mathrm{oc}) < f(\boldsymbol{v}_\mathrm{w})$ then      ▷ If the outer contracted vertex is better than the worst
        Replace $\boldsymbol{v}_\mathrm{w}$ with $\boldsymbol{v}_\mathrm{oc}$.
    else
        Reduction (4)
    end if
else if $f(\boldsymbol{v}_\mathrm{ic}) < f(\boldsymbol{v}_\mathrm{w})$ then     ▷ If the inner contracted vertex is better than the worst
    Replace $\boldsymbol{v}_\mathrm{w}$ with $\boldsymbol{v}_\mathrm{ic}$.
else
    Reduction (4)
end if
```

3 Primitive Set

The first preparatory step in constructing GP is selecting an appropriate set of terminals and functions, which together form a so-called *primitive set*. Terminals are in fact constant values and variables, which in our case all come in the form of n-dimensional real vectors (vertices).

In order to be able to compose a tree-based GP structure, we need to represent the optimization program as a syntax tree rather than as a linear sequence of instructions. It is therefore important that the elements of the primitive set are *type consistent*, which means that the function arguments and return values, as well as the terminal values, all have the same type. This is important, since it allows an arbitrary, randomly selected branch to be cut off from the tree and be swapped with another randomly selected branch, which enables us to execute crossover operations in GP. However, looking at Algorithm 1, we see that reduction (4)—which moves all vertices toward the best one and is performed when no other replacement can be made—does not have this property of type consistence. This is true because reduction does not return a single vertex as all other operations do, but it returns a list of vertices instead. Because of that, we approximate (4) with a transformation that moves only the worst vertex:

$$\boldsymbol{v}_\mathrm{new} = \boldsymbol{v}_\mathrm{b} - 0.5(\boldsymbol{v}_\mathrm{b} - \boldsymbol{v}_\mathrm{w}) = \mathrm{contr}(\boldsymbol{v}_\mathrm{b}, \boldsymbol{v}_\mathrm{w}) \text{ (Reduction of the worst vertex).} \tag{5}$$

Observe that (5) is in fact the inner contraction of the worst vertex toward the best vertex, which means that we actually do not need any special function to perform the reduction of the worst vertex. Rather, inner contraction can be used to do the job.

Moreover, we do not have to implement outer contraction as a separate function, either, because outer contraction can be constructed by a successive application of inner contraction and reflection:

$$\boldsymbol{v}_{\mathrm{oc}} = \mathrm{contr}(\boldsymbol{c}, \mathrm{refl}(\boldsymbol{c}, \boldsymbol{v}_{\mathrm{w}})) \tag{6}$$

Finally, although it is not the absolute minimum required for the job, this is the function set that we will use in our GP:

$$\mathrm{ifElse}(\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{d}): \text{ If } \boldsymbol{a} \text{ is better than } \boldsymbol{b}\text{, then it returns } \boldsymbol{c}\text{. Otherwise, it returns } \boldsymbol{d}. \tag{7a}$$

$$\mathrm{refl}(\boldsymbol{a}, \boldsymbol{b}): \text{ Performs reflection (3a) of } \boldsymbol{a} \text{ over } \boldsymbol{b}. \tag{7b}$$

$$\exp(\boldsymbol{a}, \boldsymbol{b}): \text{ Performs expansion (3b) of } \boldsymbol{a} \text{ over } \boldsymbol{b}. \tag{7c}$$

$$\mathrm{contr}(\boldsymbol{a}, \boldsymbol{b}): \text{ Performs inner contraction (3d) of } \boldsymbol{a} \text{ toward } \boldsymbol{b}. \tag{7d}$$

Here, $\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{c}$, and $\boldsymbol{d}$ are arbitrary n-dimensional real vectors. Note that ifElse (7a) is simply a wrapper around a standard if/else statement, which is implemented as a function that accepts four vector arguments and returns a single vector. This is necessary if we want to maintain type consistence within the primitive set.

Concerning the terminal (vector) set, we decided to add one more vector to those required by the original NM algorithm. Apart from vertices $\boldsymbol{v}_{\mathrm{b}}$, $\boldsymbol{v}_{\mathrm{w}}$, and $\boldsymbol{v}_{\mathrm{sw}}$ (1), and the centroid (2), we will use a vertex $\boldsymbol{v}_{\mathrm{sb}}$, defined as the vertex with the second best value of the CF (i.e., $\boldsymbol{v}_{\mathrm{sb}} = \boldsymbol{v}_1$). Altogether, five vectors acting as five terminals are used by our GP:

$$\boldsymbol{v}_{\mathrm{b}} = \boldsymbol{v}_0 \text{ (Best vertex),} \tag{8a}$$

$$\boldsymbol{v}_{\mathrm{w}} = \boldsymbol{v}_n \text{ (Worst vertex),} \tag{8b}$$

$$\boldsymbol{v}_{\mathrm{sb}} = \boldsymbol{v}_1 \text{ (Second best vertex),} \tag{8c}$$

$$\boldsymbol{v}_{\mathrm{sw}} = \boldsymbol{v}_{n-1} \text{ (Second worst vertex),} \tag{8d}$$

$$\boldsymbol{c} = \frac{1}{n} \sum_{i \neq \mathrm{w}} \boldsymbol{v}_i \text{ (Centroid).} \tag{8e}$$

Having identified a feasible primitive set of functions (7) and terminals (8), the next step is trying to manually construct the original NM algorithm using the same primitive set. The aim of this task is to find out whether or not it is possible to construct a tree-formed solver using the selected primitive set. We managed to compose a solution, which is listed under Algorithm 2. The solution is identical to Algorithm 1, with the exception of the reduction of all vertices toward the best one (4), which is replaced by the reduction of only the worst vertex (5).

We compared Algorithm 2 with the original NM method listed under Algorithm 1, and the tree-formed NM performed slightly better than the original NM, as will be shown in Figure 2 in Section 5. We speculate that this could be a consequence of an occasional premature convergence of the original NM due to too abrupt a shrinkage of the whole simplex caused by reduction (4).

4 Other Preparatory Steps and Parameters of GP

In the previous section we set up the primitive set to be used in our GP, but some more preparatory work must be done before we are able to run the GP. Table 1 summarizes all

Algorithm 2 One iteration of the modified Nelder–Mead algorithm written in a tree form using the primitive set of functions (7) and terminals (8)

```
Order the simplex vertices.
Replace $\boldsymbol{v}_\mathrm{w}$ with
ifElse(refl($\boldsymbol{c}, \boldsymbol{v}_\mathrm{w}$), $\boldsymbol{v}_\mathrm{b}$,
  ifElse(exp($\boldsymbol{c}, \boldsymbol{v}_\mathrm{w}$), refl($\boldsymbol{c}, \boldsymbol{v}_\mathrm{w}$), exp($\boldsymbol{c}, \boldsymbol{v}_\mathrm{w}$), refl($\boldsymbol{c}, \boldsymbol{v}_\mathrm{w}$)),
  ifElse(refl($\boldsymbol{c}, \boldsymbol{v}_\mathrm{w}$), $\boldsymbol{v}_\mathrm{sw}$, refl($\boldsymbol{c}, \boldsymbol{v}_\mathrm{w}$),
    ifElse(refl($\boldsymbol{c}, \boldsymbol{v}_\mathrm{w}$), $\boldsymbol{v}_\mathrm{w}$,
      ifElse(contr($\boldsymbol{c}$, refl($\boldsymbol{c}, \boldsymbol{v}_\mathrm{w}$)), $\boldsymbol{v}_\mathrm{w}$, contr($\boldsymbol{c}$, refl($\boldsymbol{c}, \boldsymbol{v}_\mathrm{w}$)), contr($\boldsymbol{v}_\mathrm{b}, \boldsymbol{v}_\mathrm{w}$)),
      ifElse(contr($\boldsymbol{c}, \boldsymbol{v}_\mathrm{w}$), $\boldsymbol{v}_\mathrm{w}$, contr($\boldsymbol{c}, \boldsymbol{v}_\mathrm{w}$), contr($\boldsymbol{v}_\mathrm{b}, \boldsymbol{v}_\mathrm{w}$))
    )
  )
)
```

(The reduction of the whole simplex (4) is approximated by the reduction of only the worst vertex (5), while outer contraction (3c) is replaced by consecutive applications of reflection and contraction (6).)

the settings that we used in our experiment, most of which were selected according to the findings from the literature (Koza, 1992; Poli et al., 2008; Ciesielski and Mawhinney, 2002; Vanneschi and Cuccu, 2009; Xie and Zhang, 2013).

One of the more important decisions was the selection of a problem (i.e., cost function) to be used for the training. Initially, we wanted to keep the number of parameters that could possibly influence the behavior of GP as low as possible, so as to keep things more transparent and manageable. In view of this decision, we speculated that only a single problem should be used for the training. As soon as we opted for a single problem, it was clear to us that the selected problem shouldn't be too complex, which could mislead GP to adapt to peculiarities of that single problem instead of searching for a more general solution. According to Törn et al. (1999), unimodal problems are the easiest to solve, under the assumption that no other information is used apart from cost function evaluations. We decided to use a plain ten-parameter quadratic function because it is not only unimodal but also radially symmetrical. However, it turned out that, using this function, the GP mistakenly interpreted the objective as 'Find the origin of the search space' instead of 'Find the minimum of the cost function', as the minimum happened to lie at the origin. Namely, we were getting algorithms that performed well but contained no if/else statements, and therefore no cost function evaluations. As a consequence, they also converged toward the origin on test functions whose optima were not at the origin.

To overcome this problem, we randomly displaced the ith parameter by a factor d_i, whose absolute value was limited to $|d_i| < 100$, thus obtaining the following cost function for the training:

$$f(\boldsymbol{x}) = \sum_{i=1}^{10} (x_i - d_i)^2. \tag{9}$$

We kept the randomly computed displacement values d_i fixed throughout the whole run of the GP. We also limited the simplex vertices to lie within a space limited by $|x_i - d_i| \leq 100, i = \{1, 2, \ldots, 10\}$, just to keep solvers from producing infinite values during the evolution. Cost function (9) was the only function that we used for training, but we

Table 1: Summary of all the parameters used for the genetic programming run.

Objective	Find an algorithm (i.e., solver) that locates the minimum of a ten-parameter quadratic function (9).
Terminal set	See (8).
Function set	See (7).
Initial population	Ramped half-and-half (tree depth of 5) (Koza, 1992).
Population size	200.
Fitness	An average cost value over ten runs, each time from a different initial simplex and with a different displacement vector. Punish lengths of more than 2,000 characters by multiplying the fitness value by 100.[a]
Selection	The best 90% of the population automatically pass to an intermediate population (elitist selection). Also, 20% of the population is selected using a tournament selection (tournament size of 2) (Poli et al., 2008) to act as parents in crossover operations.
Probability for a crossover to occur at a terminal	10%.
Tree depth limit	No.
Mutation	No.
Decimation	In 50th, 150th, and 350th generations. All the individuals whose fitness is less than 0.1% worse than that of the closest better individual are deleted from the population and replaced with new ones.
Edit	In every 30th generation, all parts that have no effect are removed from solvers in the current population.[a]
Termination	After 400 generations.

[a]See text for details.

used it ten times for each of the solvers in the population, each time from a different initial simplex, and each time using a different displacement vector $\boldsymbol{d} = (d_1, d_2, \ldots, d_{10})$. The ten initial simplices were randomly generated and kept fixed throughout the run of GP, as were the ten displacement vectors. In order to evaluate the fitness value of an individual solver in the population, we first produced Python code from the tree representation of the solver and inserted that code into a preprogrammed optimization loop. This loop was then run ten times using ten different initial simplex/displacement vector settings, each time for 5,000 iterations. Finally, we calculated the fitness value of the solver as the average of the ten CF values thus obtained.

To control bloat (i.e., the uncontrolled growth of the size of an individual), a common approach used in tree-based GP is limiting the maximal allowed tree depth, sometimes combined with punishing excess size of an individual (Luke and Panait, 2006). In fact, only limiting the size of an individual is often enough, as it implicitly controls the tree depth as well—more bushy trees will have a smaller depth, while sparse trees can be allowed to grow deeper. In our setting, we used the latter option of only limiting the size of an individual. We didn't operate directly on tree representations, but rather on the produced Python code. In order to limit the size of individuals, we simply punished those that produced code containing more than 2,000 nonspace characters by multiplying their fitness by 100. This is not a very accurate measure, but it nevertheless turned out to produce good results. Another approach could be to normalize the obtained fitness value with the average number of actual CF evaluations inside a single iteration, which would have a slightly different effect, because not all branches of generated solvers are always executed, and some of them are even not executed at all. While these might be important as hidden genetic material, which might reemerge later in the evolution, it is also true that solvers that are too long are generally not expected to yield competitive results.

We chose the population size to be 200 and selected the widely used ramped half-and-half method (Koza, 1992) for the population initialization, limiting the maximum tree depth to 5. The ramped half-and-half method combines the full method, where all the individual leafs of a randomly generated tree have the same depth, and the grow method, where leafs have arbitrary depths that are not larger than the specified maximum depth.

To produce the next generation, we performed two selection operations on the population simultaneously. First, we copied the best 90% of the individuals from the current generation to an intermediate population. Second, we selected 20% of the individuals from the whole (i.e., 100%) of the current generation to be parents in crossover operations. Parents were selected using tournament selection (Poli et al., 2008) with the tournament size of two. In tournament selection, the probability of an individual being selected is not proportional to its fitness value, which is different from fitness proportionate selection. Tournament selection simply selects better individuals with higher probability and does not pay attention to how much better they are. This effectively adjusts fitness values so that the selection pressure on the population stays constant. In the end, we selected the best 100% of the obtained 110% of candidates from the intermediate population to represent the next generation.

As originally proposed by Koza (1992), we adjusted the probability of a crossover to occur at a leaf (i.e., terminal) to be only 10%, and the probability of a crossover to occur at a function was the remaining 90%. This method is still popular because it is simple and allegedly lessens the number of potentially less productive situations in which two terminals are merely swapped by a crossover operation. Although there is little general evidence for or against the beneficial effects of this practice, it has been demonstrated that biasing node selection for crossover toward larger subtrees can indeed improve GP performance on simple standard problems (Helmuth et al., 2011).

While we limited the tree depth during the population initialization, we did not limit the depth during the run of GP. That said, the tree depth was implicitly limited by punishing longer solvers, as already described. Nor did we use mutation in our GP. Rather, we decided to decimate the population three times during the evolution process (in the 50th, 150th, and 350th generations) by deleting all individuals whose fitness was less than 0.1% worse than that of the next better individual that was kept in the population. We replaced all the deleted individuals by generating new ones using the same ramped half-and-half method we had used for initializing the whole population.

We noticed that, after some time, quite a few portions of the optimization algorithms appeared that performed meaningless operations like, for example, checking whether the best vertex was better than the worst, or contracting a vertex toward itself. That is why we introduced an edit operation, which automatically simplifies the solvers in every 30th generation by removing the parts that are not doing anything. In particular, the edit operation replaces any if/else statement whose condition invariably evaluates to either true or false with only the *if* or the *else* part of the statement, respectively. Also, it replaces any vector transformation function (i.e., reflection, expansion, or inner contraction (7b)–(7d)) that accepts two equal vertices as arguments—and therefore returns the same untransformed vertex—with a single vertex.

5 Results

We ran our GP 20 times for 400 generations,[1] using 20 2.66-GHz Core i5 (four cores per CPU) machines, which took approximately 12 hours to complete the runs. The GP process converged to an acceptable solution five times out of the 20 trials. We decided that a solution was acceptable if the fitness of the obtained solver was lower than 10^{-5}.

Although we only used a single quadratic function to train our algorithm, we were curious as to how well the obtained algorithm would perform on other test functions. We used the same set of test functions that had been used by Bűrmen et al. (2006), which is basically the set of functions originally proposed by Moré et al. (1981). Before analyzing how the algorithms behaved on individual functions, we first wanted to get an overall picture of their performance.

For this purpose, we used the normalized data profiles proposed by Moré and Wild (2009) to benchmark the obtained algorithms. The approach uses the convergence test

$$f(\boldsymbol{x}_0) - f(\boldsymbol{x}) \geq (1-\tau)(f(\boldsymbol{x}_0) - f_{\mathrm{L}}), \tag{10}$$

where $\tau > 0$ is a tolerance and $\boldsymbol{x}_0$ is the initial point for the problem. The test assumes that all the solvers whose performance one wishes to compare will be applied to the same test function, denoted by f in (10). Then, f_{L} denotes the smallest value of f computed by any of the solvers using a given fixed number of function evaluations, and a solver is said to solve the problem as soon as the found best point $\boldsymbol{x}$ satisfies (10). Data profiles are obtained using convergence test (10) on a set of solvers (optimization algorithms) $\mathcal{S}$, each applied to a set of problems (test functions) $\mathcal{P}$. Let $t_{p,s}$ be the number of function evaluations needed for a solver $s \in \mathcal{S}$ to satisfy (10) within a given tolerance τ when applied to a problem $p \in \mathcal{P}$. Then, the normalized data profile for a solver $s \in \mathcal{S}$ is defined as

$$d_s(\alpha) = \frac{1}{\mathrm{size}\{\mathcal{P}\}}\mathrm{size}\left\{p \in \mathcal{P} : \frac{t_{p,s}}{n_p + 1} \leq \alpha\right\}, \tag{11}$$

where n_p is the number of parameters in $p \in \mathcal{P}$ and α represents the number of cost function evaluations (NCF). Put differently, $d_s(\alpha)$ represents the percentage of problems that can be solved by a solver s with α simplex gradient estimates. Namely, $n_p + 1$ is the number of function evaluations required to compute a one-sided finite-difference approximation of the gradient.

Figure 1 shows the profiles of the original NM algorithm and five genetically evolved solvers at a precision of $\tau = 10^{-7}$. One can see that the convergence of the evolved solvers is generally slower than that of the original NM. However, after 3,500 or so simplex gradient estimates, only one of the solvers solves fewer problems than the original NM. Beyond 5,000 simplex gradient estimates, three of the solvers solve more problems than the original NM, and one solver solves exactly the same number of problems as the original NM. The good news is that the best of the evolved solvers is only slightly slower than the original NM, and it catches up already at the approximately 500th simplex gradient estimation, at the point where slightly more than 80% of the problems are solved by the original NM.

Having identified the best of the evolved solvers, we were also interested in how well it performed in comparison to the tree-formed NM. After all, considering the building blocks (i.e., primitives) that we used in GP, the evolved solver is closer to the

[1]The source code to reproduce the GP runs is freely available at fajfar.eu/genetic-programming/NMwithGP.rar.

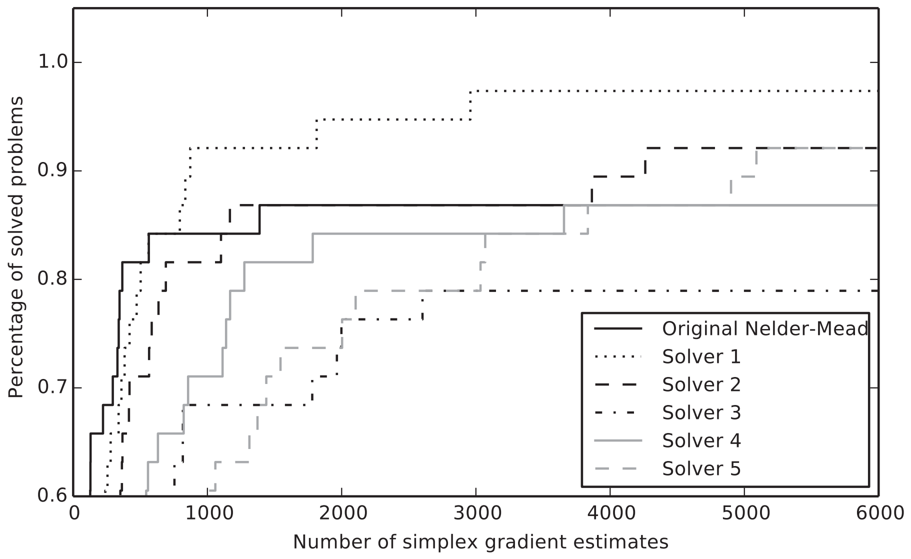
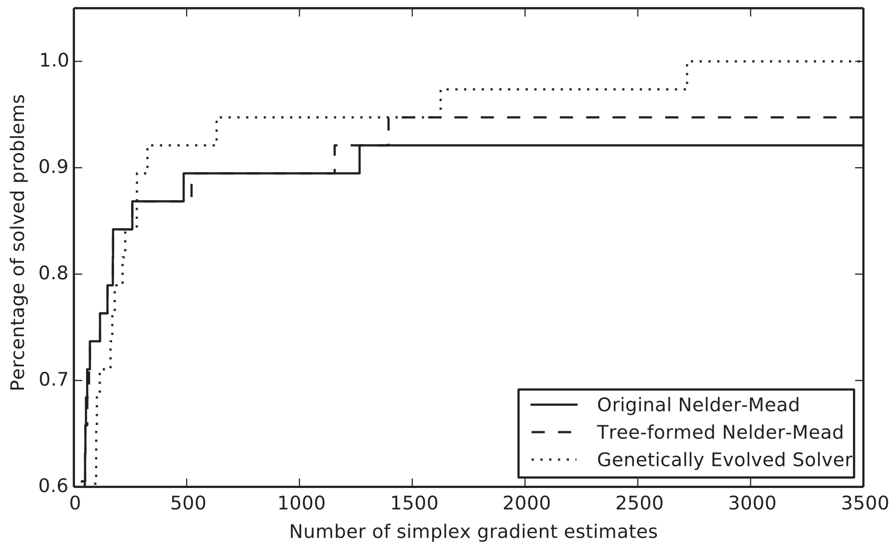


Figure 1: Data profiles for the original Nelder–Mead algorithm and five solvers obtained using GP, shown at a precision of $\tau = 10^{-7}$.

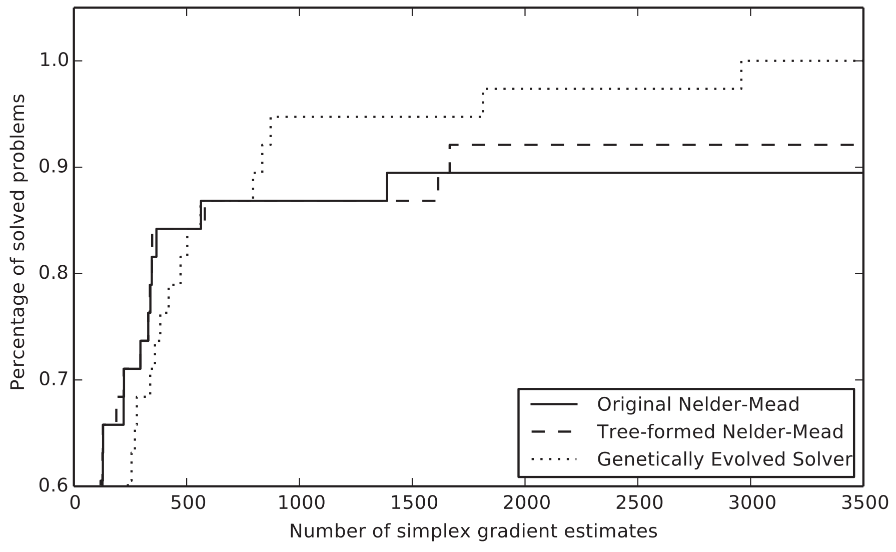
tree-formed than it is to the original NM algorithm. Apart from that, we wanted to see whether a required precision would have any significant influence on the profiles of the solvers. As can be seen in Figure 2, the tree-formed NM is just slightly better than the original NM algorithm. This might be due to the fact that the tree-formed algorithm never shrinks the whole simplex at once but does it gradually, which might allow it to find a solution that would otherwise have been overlooked by shrinking the simplex too rapidly. The important result for us is that the genetically produced solver exhibits an overall better performance than both NM algorithms. This is even more evident at a higher precision level ($\tau = 10^{-7}$) where, eventually, the percentage of solved problems is lower for both NM variants than it is at a lower precision level ($\tau = 10^{-3}$). Figure 2 also shows that the genetically produced solver is only inferior for up to about 270 simplex evaluations at a lower precision level and for up to about 500 simplex evaluations at a higher precision level. One general conclusion that we can infer from these data is that the GP-produced algorithm finds, on average, more accurate (i.e., more optimal) solutions at the expense of the slightly lower convergence speed. But as long as one has a large enough budget of cost function evaluations at his or her disposal, the GP-produced algorithm will generally yield better results.

6 Detailed Analysis of the Evolved Optimization Algorithms

Table 2 summarizes all the test functions that we used in the analysis and compares both NM algorithms with the best of the five genetically obtained algorithms. For each of the three algorithms, and for each of the test functions, one can see a computed minimum value and how many evaluations it took to get to this minimum. Apart from that, we summarize the known minima of the test functions in the last column of the table. The computed minimum values in bold match one of the known minima up to at least the double precision used in our computations. The results for quadratic functions are very interesting. We didn't test the solvers on a quadratic function with $n = 10$, which is the



(a) Data profiles for $\tau = 10^{-3}$.



(b) Data profiles for $\tau = 10^{-7}$.

Figure 2: Data profiles for the original and tree-formed Nelder–Mead algorithms and for the best of the solvers obtained using GP (i.e., solver 1 from Fig. 1). The latter solves the greatest proportion of problems, even more so when a higher precision ($\tau = 10^{-7}$) is required. However, at a higher precision the genetically evolved algorithm is the least successful for less than 500 simplex evaluations.

number of parameters used in training. Instead, we took quadratic functions with 4, 8, 16, and 24 parameters. The tree-formed NM algorithm failed to arrive at the exact minimum in all cases except $n = 4$, while the original NM was successful with quadratic

Table 2: A comparison between the original and tree-formed NM algorithms and the best of the five genetically obtained solvers by individual test functions. The table shows the dimensions of each test function (*n*), the minimum value that an algorithm arrived at, and the number of actual test function evaluations (NCF) before this minimum was reached. All the exactly computed minima (up to at least double precision) are written in bold. For reference, the last column shows the known minima of each test function.

	Original NM	Tree-Formed NM	Best GP Solver (Solver 1)	
Test Function (*n*)	Minimum (NCF)	Minimum (NCF)	Minimum (NCF)	Actual Minima
Rosenbrock (2)	$1.2325\cdots10^{-32}$ (313)	$4.9303\cdots10^{-32}$ (313)	$4.4373\cdots10^{-31}$ (867)	0.0
Freudenstein and Roth (2)	**48.9842...** (159)	**48.9842...** (218)	**48.9842...** (425)	0.0 48.9842...
Powell badly scaled (2)	**0.0** (804)	**0.0** (802)	**0.0** (1,957)	0.0
Brown badly scaled (2)	**0.0** (405)	**0.0** (411)	**0.0** (1,349)	0.0
Beale (2)	**0.0** (255)	**0.0** (259)	**0.0** (683)	0.0
Jennrich and Sampson (2)	**124.362...** (143)	**124.362...** (2,342)	**124.362...** (397)	124.362...
McKinnon (2)	**−0.25** (175)	**−0.25** (176)	**−0.25** (503)	-0.25
Helical valley (3)	**0.0** (3,566)	**0.0** (3,586)	**0.0** (10,679)	0.0
Bard (3)	$\mathbf{8.2148\cdots10^{-3}}$ (322)	$\mathbf{8.2148\cdots10^{-3}}$ (769)	$\mathbf{8.2148\cdots10^{-3}}$ (1,067)	$8.2148\cdots10^{-3}$
Gaussian (3)	$\mathbf{1.1279\cdots10^{-8}}$ (368)	$\mathbf{1.1279\cdots10^{-8}}$ (931)	$\mathbf{1.1279\cdots10^{-8}}$ (870)	$1.1279\cdots10^{-8}$
Meyer (3)	**87.9458...** (1,892)	**87.9458...** (2,268)	**87.9458...** (4,511)	87.9458
Gulf research (3)	$6.5000\cdots10^{-34}$ (3,948)	$3.0092\cdots10^{-34}$ (3,880)	$4.3122\cdots10^{-33}$ (16,324)	0.0
Box 3D (3)	$7.5588\cdots10^{-2}$ (496)	$7.5588\cdots10^{-2}$ (886)	**0.0** (2,430)	0.0
Powell singular (4)	$5.7442\cdots10^{-64}$ (2,291)	$2.1964\cdots10^{-62}$ (2,565)	$1.9509\cdots10^{-61}$ (4,871)	0.0
Wood (4)	$5.5910\cdots10^{-30}$ (907)	$2.0411\cdots10^{-30}$ (908)	$3.2183\cdots10^{-30}$ (2,779)	0.0
Kowalik and Osborne (4)	$\mathbf{3.0750\cdots10^{-4}}$ (427)	$\mathbf{3.0750\cdots10^{-4}}$ (9,332)	$\mathbf{3.0750\cdots10^{-4}}$ (1,206)	$3.0750\cdots10^{-4}$ $1.0273\cdots10^{-3}$
Brown and Dennis (4)	**85,822.2...** (426)	**85,822.2...** (1,137)	**85,822.2...** (1,288)	85,822.2...
Quadratic (4)	**0.0** (5,689)	**0.0** (5,623)	**0.0** (17,173)	0.0
Penalty I (4)	$\mathbf{2.2499\cdots10^{-5}}$ (1,365)	$2.8355\cdots10^{-5}$ (119)	$3.9053\cdots10^{-5}$ (289)	$2.2499\cdots10^{-5}$
Penalty II (4)	$\mathbf{9.3762\cdots10^{-6}}$ (3,709)	$\mathbf{9.3762\cdots10^{-6}}$ (5,940)	$\mathbf{9.3762\cdots10^{-6}}$ (5,322)	$9.3762\cdots10^{-6}$

Table 2: Continued.

	Original NM	Tree-Formed NM	Best GP Solver (Solver 1)	
Test Function (n)	Minimum (NCF)	Minimum (NCF)	Minimum (NCF)	Actual Minima
Osborne 1 (5)	$\mathbf{5.4648\cdots 10^{-5}}$ (1,273)	$\mathbf{5.4648\cdots 10^{-5}}$ (4,970)	$\mathbf{5.4648\cdots 10^{-5}}$ (2,790)	$5.4648\cdots 10^{-5}$
Brown almost linear (5)	$1.5777\cdots 10^{-30}$ (1,092)	**0.0** (1,146)	**0.0** (2,788)	0.0 1.0
Biggs EXP6 (6)	$\mathbf{5.6556\cdots 10^{-3}}$ (1,059)	$\mathbf{5.6556\cdots 10^{-3}}$ (4,450)	$5.3402\cdots 10^{-29}$ (8,068)	$5.6556\cdots 10^{-3}$ 0.0
Extended Rosenbrock (6)	$3.1554\cdots 10^{-30}$ (4,635)	$2.7240\cdots 10^{-30}$ (4,861)	$3.9443\cdots 10^{-31}$ (7,494)	0.0
Brown almost linear (7)	$9.6635\cdots 10^{-30}$ (2,311)	$6.5081\cdots 10^{-30}$ (2,541)	$7.8886\cdots 10^{-31}$ (4,104)	0.0 1.0
Quadratic (8)	**0.0** (18,964)	$1.9762\cdots 10^{-323}$ (18,814)	**0.0** (39,785)	0.0
Extended Rosenbrock (8)	$3.1914\cdots 10^{-28}$ (13,583)	$7.6420\cdots 10^{-30}$ (15,961)	$2.7523\cdots 10^{-29}$ (19,164)	0.0
Variably dimensioned (8)	**0.0** (4,682)	$9.2814\cdots 10^{-30}$ (4,856)	$1.0292\cdots 10^{-29}$ (7,160)	0.0
Extended Powell (8)	$4.4086\cdots 10^{-61}$ (11,251)	$9.8782\cdots 10^{-59}$ (12,894)	$9.7234\cdots 10^{-61}$ (20,353)	0.0
Watson (6)	$\mathbf{2.2876\cdots 10^{-3}}$ (2,963)	$\mathbf{2.2876\cdots 10^{-3}}$ (5,120)	$\mathbf{2.2876\cdots 10^{-3}}$ (5,151)	$2.2876\cdots 10^{-3}$
Extended Rosenbrock (10)	9.72337... (7,821)	$1.9748\cdots 10^{-28}$ (20,691)	$9.0484\cdots 10^{-29}$ (36,268)	0.0
Penalty I (10)	$7.5675\cdots 10^{-5}$ (5,427)	$9.1907\cdots 10^{-5}$ (1,419)	$9.4271\cdots 10^{-5}$ (2,100)	$7.0876\cdots 10^{-5}$
Penalty II (10)	$2.9778\cdots 10^{-4}$ (6,342)	$2.9778\cdots 10^{-4}$ (6,547)	$3.0000\cdots 10^{-4}$ (1,543)	$2.9366\cdots 10^{-4}$
Trigonometric (10)	$2.7950\cdots 10^{-5}$ (3,610)	$2.7950\cdots 10^{-5}$ (4,277)	$2.7950\cdots 10^{-5}$ (4,252)	0.0
Osborne 2 (11)	$\mathbf{4.0137\cdots 10^{-2}}$ (4,821)	$\mathbf{4.0137\cdots 10^{-2}}$ (14,517)	$\mathbf{4.0137\cdots 10^{-2}}$ (7,381)	$4.0137\cdots 10^{-2}$
Extended Powell (12)	$8.3858\cdots 10^{-54}$ (39,823)	$1.0430\cdots 10^{-47}$ (39,093)	$5.7700\cdots 10^{-58}$ (50,117)	0.0
Quadratic (16)	$2.2211\cdots 10^{-120}$ (39,965)	$3.2435\cdots 10^{-159}$ (52,675)	**0.0** (112,564)	0.0
Quadratic (24)	0.58675... (39,976)	0.58527... (48,904)	$8.0493\cdots 10^{-173}$ (158,849)	0.0

functions with $n = 4$ and $n = 8$. In the case of $n = 24$, the failure of both NM variants was very serious.

All in all, the genetically obtained solver outperformed both NM variants on quadratic functions with higher numbers of parameters ($n = 16$ and 24). It was somewhat to be expected that the solver would yield the best results on the function it was trained on. However, it obviously adapted better to a higher number of parameters, thus exhibiting inferior performance on the quadratic function with only four parameters, and a little less inferior performance on the quadratic function with eight parameters. It has already been reported (Gao and Han, 2012) that the standard NM performs poorly when applied to higher-dimensional problems. Gao and Han (2012) suggested that this is due to an excessive number of reflections being used. Indeed, they discovered that for the quadratic function, the fraction of steps that are reflections in standard NM steeply increases with the dimensionality of the problem. Already at $n = 20$, approximately 90% of the steps are reflections. On the other hand, our algorithm uses less than 80% reflections when used on the quadratic function with $n = 24$, as will be seen in Section 7.

Inspecting the results obtained with other test functions, we can confirm that the genetically produced method generally performs better than both NM variants with higher-dimensional problems, but with lower-dimensional problems both NM variants are mostly better in terms of the needed NCF. One notable exception among lower-dimensional functions is the two-dimensional Jennrich and Sampson function, for which the genetically produced method is almost six times better in terms of consumed NCF evaluations than the tree-formed NM, but still worse than the original NM. Another exception is the three-dimensional Gaussian function, which the GP solver solved slightly better (i.e., faster) than the tree-formed NM. The Box 3D function is also interesting, for which the GP-generated solver found an exact minimum, and the Biggs EXP6 function, for which the GP-generated solver came quite close to a different (and also better) minimum than both NM variants. In the first of these two cases, both NM variants performed quite poorly, while in the second one they both found a local minimum at $5.6556 \cdots 10^{-3}$. The Rosenbrock function is known to be difficult, as the global minimum lies inside a long and narrow, parabolic-shaped flat valley. The GP-generated solver came the closest to the solution on the Rosenbrock functions with $n = 6$ and $n = 10$, and the original NM failed altogether on the ten-dimensional Rosenbrock function.

The results of the other four genetically evolved optimizers are summarized in Table 3. The found minima are in general not as good as those found by the best genetically evolved optimizer, of course, but one can detect performances that are superior on certain test functions. For example, the first row in the table, which contains the two-dimensional Rosenbrock function, already shows that two of the solvers have found the exact minimum, which was missed by all three previously analyzed solvers. Also, at least one of the remaining four solvers outperformed all of the previous three solvers on the following test functions: Gulf research ($n = 3$), Wood ($n = 4$), Biggs EXP6 ($n = 6$), Brown almost linear ($n = 7$), Penalty I ($n = 10$), Penalty II ($n = 10$), and Trigonometric ($n = 10$). Among these, the exact minima were found on the Wood ($n = 4$) and Penalty I ($n = 10$) test functions, which were not found by any of the previous three methods. It can be speculated that some of these generally worse solvers contain genetic material that could later in the evolution process combine with better solvers to produce even better solutions.

7 A Closer Look at the Best Solver

The best of the five evolved solvers discussed in the previous section emerged in the 363rd generation of the seventh run of GP, with a fitness value of $3.6569 \cdots 10^{-11}$. The rest of the article is dedicated to this one solver, which we analyze more closely.

Because the obtained solver appears in a tree form, we have first reshaped the algorithm by hand into a more human-friendly linear form, listed under Algorithm 3. Comparing this algorithm to the original NM, we immediately notice one striking

Table 3: A comparison of the other four genetically obtained solvers by individual test functions. The table shows the dimensions of each test function (n), the minimum value that an algorithm arrived at, and the number of actual test function evaluations (NCF) before this minimum was reached. All the exactly computed minima (up to at least double precision) are written in bold. For the actual minima of individual test functions, refer to Table 2.

	Solver 2	Solver 3	Solver 4	Solver 5
Test Function (n)	Minimum (NCF)	Minimum (NCF)	Minimum (NCF)	Minimum (NCF)
Rosenbrock (2)	**0.0** (1,516)	**0.0** (1,522)	$1.1093\cdots10^{-31}$ (2,397)	$1.2325\cdots10^{-30}$ (2,193)
Freudenstein and Roth (2)	**48.9842**... (928)	**48.9842**... (2,498)	**48.9842**... (900)	**48.9842**... (1,499)
Powell badly scaled (2)	**0.0** (3,240)	**0.0** (4,366)	**0.0** (10,300)	**0.0** (10,054)
Brown badly scaled (2)	**0.0** (2,588)	**0.0** (3,706)	$1.9721\cdots10^{-31}$ (4,510)	**0.0** (6,669)
Beale (2)	$5.9164\cdots10^{-31}$ (830)	**0.0** (1,029)	**0.0** (1,977)	**0.0** (844)
Jennrich and Sampson (2)	**124.362**... (669)	**124.362**... (880)	**124.362**... (650)	**124.362**... (421)
McKinnon (2)	**−0.25** (575)	**−0.25** (1,410)	**−0.25** (1,179)	**−0.25** (1,382)
Helical valley (3)	**0.0** (7,287)	**0.0** (15,151)	**0.0** (11,093)	**0.0** (13,824)
Bard (3)	$\mathbf{8.2148\cdots10^{-3}}$ (1,020)	$\mathbf{8.2148\cdots10^{-3}}$ (2,354)	$\mathbf{8.2148\cdots10^{-3}}$ (1,063)	$\mathbf{8.2148\cdots10^{-3}}$ (3,235)
Gaussian (3)	$\mathbf{1.1279\cdots10^{-8}}$ (567)	$\mathbf{1.1279\cdots10^{-8}}$ (705)	$\mathbf{1.1279\cdots10^{-8}}$ (707)	$\mathbf{1.1279\cdots10^{-8}}$ (853)
Meyer (3)	**87.9458**... (9,325)	**87.9458**... (9,847)	**87.9458**... (10,226)	**87.9458**... (24,991)
Gulf research (3)	$2.4074\cdots10^{-35}$ (16,186)	$1.7453\cdots10^{-33}$ (15,258)	$3.0092\cdots10^{-33}$ (31,749)	$6.1438\cdots10^{-31}$ (83,423)
Box 3D (3)	0.07558... (1,357)	0.07558... (2,136)	$5.8548\cdots10^{-32}$ (3,275)	**0.0** (5,106)
Powell singular (4)	$2.5289\cdots10^{-60}$ (6,326)	$2.5839\cdots10^{-59}$ (9,429)	$2.2833\cdots10^{-59}$ (7,565)	$3.8611\cdots10^{-60}$ (14,562)
Wood (4)	$2.0411\cdots10^{-30}$ (2,771)	**0.0** (4,648)	$1.6180\cdots10^{-26}$ (5,194)	$6.4450\cdots10^{-28}$ (9,233)
Kowalik and Osborne (4)	$\mathbf{3.0750\cdots10^{-4}}$ (7,271)	$\mathbf{3.0750\cdots10^{-4}}$ (1,902)	$\mathbf{3.0750\cdots10^{-4}}$ (1,535)	$\mathbf{3.0750\cdots10^{-4}}$ (2,550)
Brown and Dennis (4)	**85,822.2**... (4,150)	**85,822.2**... (3,324)	**85,822.2**... (2,259)	**85,822.2**... (3,638)
Quadratic (4)	**0.0** (13,253)	**0.0** (24,151)	$4.9406\cdots10^{-324}$ (27,863)	**0.0** (21,977)
Penalty I (4)	$\mathbf{2.2499\cdots10^{-5}}$ (20,944)	$3.3070\cdots10^{-5}$ (784)	$3.1533\cdots10^{-5}$ (718)	$\mathbf{2.2499\cdots10^{-5}}$ (7,854)
Penalty II (4)	$\mathbf{9.3762\cdots10^{-6}}$ (15,465)	$\mathbf{9.3762\cdots10^{-6}}$ (14,046)	$\mathbf{9.3762\cdots10^{-6}}$ (6,376)	$\mathbf{9.3762\cdots10^{-6}}$ (19,463)
Osborne 1 (5)	$\mathbf{5.4648\cdots10^{-5}}$ (4,538)	$\mathbf{5.4648\cdots10^{-5}}$ (3,990)	$\mathbf{5.4648\cdots10^{-5}}$ (4,589)	$\mathbf{5.4648\cdots10^{-5}}$ (7,961)

Table 3: Continued.

	Solver 2	Solver 3	Solver 4	Solver 5
Test Function (n)	Minimum (NCF)	Minimum (NCF)	Minimum (NCF)	Minimum (NCF)
Brown almost linear (5)	$1.2818\cdots10^{-29}$ (2,974)	$3.9936\cdots10^{-30}$ (4,710)	$1.8341\cdots10^{-29}$ (5,411)	$1.0862\cdots10^{-27}$ (4,813)
Biggs EXP6 (6)	$\mathbf{5.6556\cdots10^{-3}}$ (4,629)	$1.5191\cdots10^{-30}$ (16,185)	$\mathbf{5.6556\cdots10^{-3}}$ (6,071)	$\mathbf{5.6556\cdots10^{-3}}$ (10,623)
Extended Rosenbrock (6)	$2.4520\cdots10^{-28}$ (9,373)	$3.0080\cdots10^{-28}$ (21,912)	$2.8776\cdots10^{-26}$ (11,954)	$1.5225\cdots10^{-27}$ (31,037)
Brown almost linear (7)	$1.1694\cdots10^{-28}$ (4,900)	$4.4373\cdots10^{-31}$ (11,638)	$8.0799\cdots10^{-28}$ (9,846)	$4.9433\cdots10^{-28}$ (9,126)
Quadratic (8)	**0.0** (45,403)	$8.8931\cdots10^{-323}$ (94,405)	**0.0** (106,393)	$9.8813\cdots10^{-324}$ (126,241)
Extended Rosenbrock (8)	$1.9578\cdots10^{-28}$ (37,903)	$3.8246\cdots10^{-28}$ (87,494)	$5.3859\cdots10^{-28}$ (39,917)	$4.7260\cdots10^{-27}$ (52,144)
Variably dimensioned (8)	$8.0365\cdots10^{-30}$ (9,336)	1.58672... (26,717)	$5.2804\cdots10^{-29}$ (16,946)	$2.6537\cdots10^{-29}$ (29,425)
Extended Powell (8)	$5.5841\cdots10^{-54}$ (27,252)	$7.6763\cdots10^{-55}$ (45,264)	$1.8650\cdots10^{-52}$ (35,811)	$2.8897\cdots10^{-57}$ (86,725)
Watson (6)	$\mathbf{2.2876\cdots10^{-3}}$ (61,414)	$2.5917\cdots10^{-3}$ (18,861)	$\mathbf{2.2876\cdots10^{-3}}$ (9,303)	$\mathbf{2.2876\cdots10^{-3}}$ (25,864)
Extended Rosenbrock (10)	$3.8643\cdots10^{-27}$ (52,401)	$3.5135\cdots10^{-26}$ (149,656)	4.95928... (67,691)	1.83583... (179,463)
Penalty I (10)	$\mathbf{7.0876\cdots10^{-5}}$ (25,735)	$8.3055\cdots10^{-5}$ (59,472)	$9.1766\cdots10^{-5}$ (6,077)	$9.2389\cdots10^{-5}$ (6,169)
Penalty II (10)	$2.9411\cdots10^{-4}$ (51,485)	$2.9619\cdots10^{-4}$ (40,857)	$2.9719\cdots10^{-4}$ (87,523)	$2.9436\cdots10^{-4}$ (110,351)
Trigonometric (10)	$4.4735\cdots10^{-7}$ (7,253)	$2.7950\cdots10^{-5}$ (10,220)	$2.7950\cdots10^{-5}$ (7,428)	$2.7950\cdots10^{-5}$ (9,018)
Osborne 2 (11)	$\mathbf{4.0137\cdots10^{-2}}$ (10,266)	$\mathbf{4.0137\cdots10^{-2}}$ (29,677)	$\mathbf{4.0137\cdots10^{-2}}$ (14,435)	$\mathbf{4.0137\cdots10^{-2}}$ (21,610)
Extended Powell (12)	$1.9286\cdots10^{-49}$ (97,107)	$3.1426\cdots10^{-52}$ (212,337)	$6.4421\cdots10^{-29}$ (272,309)	$4.5749\cdots10^{-16}$ (177,159)
Quadratic (16)	$7.0333\cdots10^{-152}$ (196,200)	$1.6537\cdots10^{-12}$ (239,868)	0.05841... (273,342)	$2.8766\cdots10^{-73}$ (161,593)
Quadratic (24)	$6.8660\cdots10^{-6}$ (198,520)	0.31333... (239,876)	0.61552... (273,604)	$9.4729\cdots10^{-21}$ (160,893)

similarity between both, in that all transformations involve almost exclusively the worst vertex and centroid. In fact, in the genetically obtained algorithm, only two occurrences of vertices are *not* the worst vertex or centroid (found in lines 14 and 15), and even these are of marginal significance.

We should mention one more outstanding feature of Algorithm 3: Notice that the whole algorithm is composed of a single if/else statement, which in turn contains other nested statements. If one analyzes all the possible paths through the algorithm, one will notice that the *if* part of the outermost if/else statement invariably replaces the worst

Algorithm 3 One iteration of the best solver generated by GP

```
1: Order the simplex vertices.
2: if $f(\text{refl}(\boldsymbol{c}, \boldsymbol{v}_\text{w})) < f(\boldsymbol{v}_\text{w})$ then
3:     if $f(\exp(\boldsymbol{c}, \boldsymbol{v}_\text{w})) < f(\boldsymbol{c})$ then
4:         if $f(\text{refl}(\boldsymbol{c}, \boldsymbol{v}_\text{w})) < f(\boldsymbol{c})$ then
5:             $\boldsymbol{v}_\text{tmp} = \text{contr}(\text{refl}(\boldsymbol{c}, \boldsymbol{v}_\text{w}), \exp(\boldsymbol{c}, \boldsymbol{v}_\text{w}))$        ▷ L$_1$
6:         else
7:             $\boldsymbol{v}_\text{tmp} = \text{contr}(\text{refl}(\boldsymbol{c}, \text{refl}(\boldsymbol{c}, \boldsymbol{v}_\text{w})), \boldsymbol{c})$        ▷ L$_2$
8:         end if
9:     else
10:        $\boldsymbol{v}_\text{tmp} = \boldsymbol{c}$        ▷ L$_3$
11:    end if
12:    $\boldsymbol{v}_\text{tmp} = \text{contr}(\text{contr}(\boldsymbol{v}_\text{tmp}, \boldsymbol{c}), \exp(\boldsymbol{c}, \boldsymbol{v}_\text{w}))$
13: else
14:    if $f(\boldsymbol{v}_\text{sw}) < f(\boldsymbol{c})$ then
15:        $\boldsymbol{v}_\text{tmp} = \boldsymbol{v}_\text{b}$        ▷ L$_4$
16:    else
17:        $\boldsymbol{v}_\text{tmp} = \boldsymbol{c}$        ▷ L$_5$
18:    end if
19:    if $f(\text{contr}(\text{refl}(\boldsymbol{c}, \exp(\boldsymbol{c}, \boldsymbol{v}_\text{w})), \text{contr}(\boldsymbol{v}_\text{tmp}, \boldsymbol{c}))) < f(\boldsymbol{c})$ then
20:        $\boldsymbol{v}_\text{tmp} = \text{refl}(\boldsymbol{c}, \text{refl}(\boldsymbol{c}, \boldsymbol{v}_\text{w}))$        ▷ L$_6$
21:    else
22:        $\boldsymbol{v}_\text{tmp} = \boldsymbol{c}$        ▷ L$_7$
23:    end if
24:    if $f(\boldsymbol{v}_\text{tmp}) < f(\boldsymbol{c})$ then
25:        $\boldsymbol{v}_\text{tmp} = \text{refl}(\boldsymbol{c}, \boldsymbol{v}_\text{w})$        ▷ L$_8$
26:    else
27:        $\boldsymbol{v}_\text{tmp} = \text{contr}(\boldsymbol{c}, \boldsymbol{v}_\text{w})$        ▷ L$_9$
28:    end if
29:    $\boldsymbol{v}_\text{tmp} = \text{contr}(\boldsymbol{v}_\text{w}, \text{contr}(\boldsymbol{v}_\text{tmp}, \boldsymbol{c}))$
30: end if
31: Replace $\boldsymbol{v}_\text{w}$ with $\boldsymbol{v}_\text{tmp}$.
```

(We have reshaped the algorithm from the original tree form into a more readable one, for which purpose we used a temporary vertex $\boldsymbol{v}_\text{tmp}$ to hold intermediate vertex values. The marks L$_1$ to L$_9$ are used later in the article to refer to specific parts of the algorithm.)

vertex with a vertex positioned on the far side of $\boldsymbol{c}$ observed from $\boldsymbol{v}_\text{w}$. On the other hand, the *else* part of the outermost if/else statement invariably replaces the worst vertex with a vertex on the near side of $\boldsymbol{c}$.

Let us now dig a little deeper into the structure of the GP-obtained optimization algorithm. In order to estimate the importance of particular branches of the algorithm, we counted the number of times that each of the lines executed until the algorithm was stopped. We were interested in the nine lines marked from L$_1$ to L$_9$ in the pseudocode shown under Algorithm 3. Table 4 shows—in percentages of all iterations—how many times certain parts of the algorithm were executed. Surprisingly, nearly half of the code was almost never executed.

In order to find out whether the parts of the algorithm that are rarely executed matter at all, we simplified the genetically bred Algorithm 3 in such a way that we

Table 4: Relative numbers of executions of individual lines of Algorithm 3. Nearly half of the code is hardly ever executed. Note that either exactly one of the lines from L_1 to L_3 or exactly one of each of the pairs of the last six lines (L_4, L_5), (L_6, L_7), and (L_8, L_9) is executed during each iteration. Because of that, the first five columns add up to 100%. It is also true that the last three pairs of columns all add up to equal percentages.

	Lines (% of executions)								
Test Function (n)	L_1	L_2	L_3	L_4	L_5	L_6	L_7	L_8	L_9
Rosenbrock (2)	12.69	0.00	28.08	0.00	59.23	0.00	59.23	0.00	59.23
Freudenstein and Roth (2)	20.97	0.00	23.39	0.00	55.65	0.00	55.65	0.00	55.65
Powell badly scaled (2)	22.42	0.00	38.79	0.00	38.79	0.00	38.79	0.00	38.79
Brown badly scaled (2)	20.95	0.00	30.42	0.00	48.63	0.00	48.63	0.00	48.63
Beale (2)	5.39	0.00	29.90	0.00	64.71	0.00	64.71	0.00	64.71
Jennrich and Sampson (2)	8.40	0.00	23.66	12.98	54.96	2.29	65.65	1.53	66.41
McKinnon (2)	23.94	0.00	27.46	0.00	48.59	0.00	47.89	0.00	47.89
Helical valley (3)	2.28	0.00	25.91	0.00	71.81	0.00	71.81	0.00	71.81
Bard (3)	14.42	0.92	29.45	11.96	43.25	3.37	51.84	3.37	51.84
Gaussian (3)	7.49	1.87	27.34	14.98	48.31	7.12	55.81	5.62	57.30
Meyer (3)	28.21	0.08	42.35	1.61	27.75	0.84	28.44	0.61	28.67
Gulf research (3)	31.16	0.00	44.05	0.04	24.75	0.04	24.75	0.04	24.75
Box 3D (3)	29.20	0.00	20.94	0.00	49.85	0.15	49.71	0.00	49.85
Powell singular (4)	18.39	0.00	34.80	0.00	46.81	0.00	46.81	0.00	46.81
Wood (4)	12.73	0.00	42.78	0.00	44.49	0.00	44.49	0.00	44.49
Kowalik and Osborne (4)	13.76	0.00	35.47	0.00	50.76	0.00	50.76	0.00	50.76
Brown and Dennis (4)	19.88	0.00	29.97	0.00	50.15	0.00	50.15	0.00	50.15
Quadratic (4)	0.93	0.00	31.18	0.00	67.89	0.00	67.89	0.00	67.89
Penalty I (4)	47.95	0.00	39.73	0.00	12.33	0.00	12.33	0.00	12.33
Penalty II (4)	25.33	0.35	39.99	6.42	27.91	2.65	31.61	2.09	32.17
Osborne 1 (5)	16.08	0.00	51.33	0.00	32.59	0.00	32.59	0.00	32.59
Brown almost linear (5)	10.45	0.00	37.88	0.00	51.67	0.00	51.67	0.00	51.67
Biggs EXP6 (6)	19.43	0.00	52.15	0.00	28.42	0.00	28.42	0.00	28.42
Extended Rosenbrock (6)	19.79	0.00	47.61	0.00	32.60	0.00	32.60	0.00	32.60
Brown almost linear (7)	10.20	0.00	40.79	0.37	48.64	0.28	48.74	0.28	48.74
Quadratic (8)	0.86	0.00	37.33	0.00	61.82	0.00	61.82	0.00	61.82
Extended Rosenbrock (8)	23.05	0.00	50.66	0.00	26.29	0.00	26.29	0.00	26.29
Variably dimensioned (8)	21.10	0.00	36.92	0.11	41.87	0.11	41.87	0.11	41.87
Extended Powell (8)	15.74	0.00	57.08	0.00	27.18	0.00	27.16	0.00	27.16
Watson (6)	28.91	0.76	31.59	10.65	28.09	2.68	36.06	1.99	36.74
Extended Rosenbrock (10)	22.65	0.00	56.96	0.00	20.39	0.00	20.39	0.00	20.39
Penalty I (10)	28.65	0.00	56.55	0.00	14.80	0.00	14.80	0.00	14.80
Penalty II (10)	22.96	0.00	58.93	0.51	17.60	0.51	17.60	0.51	17.60
Trigonometric (10)	9.09	0.00	57.79	0.00	33.12	0.00	33.12	0.00	33.12
Osborne 2 (11)	8.36	0.36	61.01	4.05	26.22	1.56	28.66	1.14	29.08
Extended Powell (12)	17.93	0.00	61.84	0.00	20.23	0.00	20.23	0.00	20.23
Quadratic (16)	1.81	0.00	57.80	0.00	40.39	0.00	40.39	0.00	40.39
Quadratic (24)	3.03	0.00	75.38	0.00	21.59	0.00	21.59	0.00	21.59

removed the parts marked by L_2, L_4, L_6, and L_8. We also simplified the remaining vertex transformations so that, mathematically, they perform equivalent moves to those of Algorithm 3. The result was a much simpler algorithm, shown under Algorithm 4.

Algorithm 4 The algorithm obtained by removing branches L_2, L_4, L_6, and L_8 from Algorithm 3

```
Order the simplex vertices.
if $f(\boldsymbol{c} + 1(\boldsymbol{c} - \boldsymbol{v}_{\mathrm{w}})) < f(\boldsymbol{v}_{\mathrm{w}})$ then
    if $f(\boldsymbol{c} + 2(\boldsymbol{c} - \boldsymbol{v}_{\mathrm{w}})) < f(\boldsymbol{c})$ then
        $\boldsymbol{v}_{\mathrm{new}} = \boldsymbol{c} + 1.375(\boldsymbol{c} - \boldsymbol{v}_{\mathrm{w}})$
    else
        $\boldsymbol{v}_{\mathrm{new}} = \boldsymbol{c} + 1(\boldsymbol{c} - \boldsymbol{v}_{\mathrm{w}})$
    end if
else
    $\boldsymbol{v}_{\mathrm{new}} = \boldsymbol{c} - 0.625(\boldsymbol{c} - \boldsymbol{v}_{\mathrm{w}})$
end if
Replace $\boldsymbol{v}_{\mathrm{w}}$ with $\boldsymbol{v}_{\mathrm{new}}$.
```

The resulting algorithm has now become stunningly simple. It first checks whether the reflected vertex is better than the worst one. If it is not, then the algorithm automatically performs inner contraction with the multiplication factor of 0.625, without actually checking whether the contracted vertex is any better than the worst one. In part, this move might play the role of the shrinkage of the simplex in the original NM method when everything else fails.

If, however, the reflected vertex is better than the worst one, then the expanded vertex is compared to the centroid. If it is not better than the centroid, then reflection is performed, which is an obvious thing to do, since the reflected vertex was proven to be better than the worst one. The interesting part is the combination of the comparison of the expanded vertex with the centroid and the expansion of the worst vertex by a factor of 1.375. The fact that the expanded vertex is better than the centroid does not guarantee that it is also better than the worst vertex, of course. However, it obviously works in most cases to assume that, as soon as the expanded vertex is better than the centroid, an expansion by a factor of 1.375 will yield a vertex that is better than the worst one. Interestingly enough, a similar factor is reported by Bűrmen et al. (2006), who discovered that an expansion by a factor of 1.2 works better than the more usual expansion by a factor of 2.

In the previous section we mentioned that too many reflections in proportion to other performed transformations is responsible for the poor performance of the standard NM at higher dimensions. The fraction of performed reflections has been reported to be about 90% for the 20-dimensional quadratic function (Gao and Han, 2012). Comparing Algorithms 3 and 4, one can see that—in problems where the number of executions of the even-numbered lines L_i equals zero—the execution of line L_3 in fact leads to reflection, while the execution of line L_1 leads to expansion by a factor of 1.375, and the execution of lines L_5, L_7, and L_9 leads to inner contraction by a factor of 0.625. Indeed, as can be seen in Table 4, the fraction of reflections used for the quadratic function increases with increasing problem dimensionality, but reaches only a good 75% at $n = 24$, as opposed to the 90% at $n = 20$ observed in the original NM method. This finding is in agreement with the before-observed better performance of the genetically produced algorithm at higher dimensions.

The first result that we should expect from running the simplified Algorithm 4 is that the average NCF will be somewhat smaller. Namely, in Algorithm 3 a number of function evaluations check conditions that are never or rarely true. Indeed, on average,

Table 5: Differences between Algorithms 3 and 4. Only those functions are listed for which both algorithms found different minimum cost function values. In half of the listed cases (marked with an asterisk (*)), both algorithms are mathematically identical, since the removed branches from Algorithm 3 were actually never executed. In the remaining half of the cases, the number of executions of the removed branches equals a half percent or less of all iterations. All the exactly computed minima (up to at least double precision) are written in bold.

	Algorithm 3		Algorithm 4	
Test Function (n)	NCF	Minimum	NCF	Minimum
Rosenbrock* (2)	867	$4.4373\cdots 10^{-31}$	692	$1.2325\cdots 10^{-30}$
Gulf research (3)	16324	$4.3122\cdots 10^{-33}$	15119	$1.3240\cdots 10^{-34}$
Brown almost linear* (5)	2788	**0.0**	2129	$8.0118\cdots 10^{-31}$
Brown almost linear (7)	4104	$7.8886\cdots 10^{-31}$	3249	$1.9721\cdots 10^{-31}$
Variably dimensioned (8)	7160	$1.0292\cdots 10^{-29}$	5816	$9.3184\cdots 10^{-30}$
Penalty II (10)	1543	$3.0000\cdots 10^{-4}$	30721	$\mathbf{2.9366\cdots 10^{-4}}$
Quadratic* (16)	112564	**0.0**	48917	$1.6304\cdots 10^{-174}$
Quadratic* (24)	158849	$8.0493\cdots 10^{-173}$	48918	$1.5467\cdots 10^{-53}$

Algorithm 4 needs 18.5% fewer evaluations than Algorithm 3 to complete the same task. One notable exception is the ten-dimensional Penalty II function, for which the simplified algorithm takes almost 20 times as many runs as Algorithm 3 (see Table 5). However, Algorithm 4 succeeds in finding the exact minimum at $2.9366\cdots 10^{-4}$, which Algorithm 3 fails to do. Incidentally, this minimum wasn't found by any of the two NM variants, either. We can hypothesize that the parts of the algorithm that we deleted are in fact counterproductive. In spite of constituting a mere 0.51% of executions, they cause the algorithm to fail to converge to the actual minimum. However, by analyzing other test functions that also yielded different results, we discovered that this is not the only reason for the observed difference, if it is a reason at all.

Table 5 shows the test functions for which Algorithms 3 and 4 found different minimum values. With the functions Gulf research, Brown almost linear ($n = 7$), and Variably dimensioned, Algorithm 4 yielded just slightly better results, while it succeeded in finding the exact minimum on the Penalty II function. On the other hand, Algorithm 3 found two exact minima (with the Brown almost linear ($n = 5$) and quadratic ($n = 16$) functions) and exhibited more or less better performance with the Rosenbrock and quadratic ($n = 24$) functions.

An interesting fact is that only half of these eight cases actually use the parts of Algorithm 3 that were removed in Algorithm 4. Note that for all the problems that don't use the parts removed from Algorithm 3 at all, both algorithms are mathematically identical, and there is no reason why they should yield different results. The only possible explanation for the differences is numerical noise, which propagates differently through the two algorithms, because of quite different computing sequences.

For instance, the two consecutive expressions

$$\boldsymbol{v}_{\text{tmp}} = \text{contr}(\text{refl}(\boldsymbol{c}, \boldsymbol{v}_{\text{w}}), \exp(\boldsymbol{c}, \boldsymbol{v}_{\text{w}}))$$
$$\boldsymbol{v}_{\text{tmp}} = \text{contr}(\text{contr}(\boldsymbol{v}_{\text{tmp}}, \boldsymbol{c}), \exp(\boldsymbol{c}, \boldsymbol{v}_{\text{w}}))$$

are mathematically identical to a single simplified expression,

$$\boldsymbol{v}_{\text{tmp}} = \boldsymbol{c} + 1.375(\boldsymbol{c} - \boldsymbol{v}_{\text{w}}).$$

However, they cannot always give the same numerical results, due to rounding errors.

Because the observed influence of the removed parts of the code is no bigger than that of numerical noise alone, we can speculate that the parts of genetically produced algorithms that are seldom executed can be considered noise as well. This is even more the case due to the fact that one cannot rule out the influence of numerical noise even in the four parts that are not mathematically identical.

8 Conclusions

Our initial hypothesis was that it is possible to produce a deterministic derivative-free optimization algorithm using genetic programming. We used the popular Nelder–Mead method as a role model from which to select a primitive set used for breeding the population of optimization algorithms. Initially, we didn't want to have too many parameters involved in a GP system. Hence we decided on the most basic GP approach, first formally proposed by Koza (1992), adding only a few features found elsewhere in the literature. We selected a single ten-parameter quadratic function as a training problem. Among 20 GP runs, we observed convergence to an acceptable solution in five cases and selected the best of the five produced solvers for further analysis. We found out that it was possible to algebraically simplify the solver on the basis of the observation that many parts of the solver were quite seldom executed or not executed at all. The simplified solver was surprisingly plain and still performed significantly better than the original Nelder–Mead algorithm. We discovered that some parameter values and exhibited behavior of the evolved optimization algorithm were up to a certain extent in agreement with those proposed or reported in the literature.

Analyzing the differences between the solver that was originally produced by GP and the one simplified by hand, we discovered that the parts of code that were rarely or never executed had an influence comparable to that of numerical noise. Such parts of the code are probably a relic of the evolution history, but it is not clear whether they contain any important information for future GP generations.

All in all, the results of our research are extremely promising, even more so because we didn't do any fine-tuning of the GP at all. Instead we used parameters that we felt were the best for the problem at hand, based on findings from the literature. We believe that experimenting with the parameter values of the GP system could produce even better results. A lot of attention should also be paid to the selection of problems used for training, and even the choice of a primitive set may play a decisive role in obtaining better results. In our future work, we shall incorporate (into the training process) test functions on which the obtained solver showed the worst performance. It will be interesting to observe whether and how this will influence the structure of the solution and the performance of the solver on test functions for which it was now successful.

The results of our research show that GP can serve as a robust meta-optimizer to fine-tune an existing deterministic optimization method, provided that one is able to extract an appropriate primitive set from it. The more challenging step for the future will be to discover to what extent GP is capable of producing a new method that would be qualitatively altogether different from any humanly produced method. A big question remains regarding how to select a primitive set that wouldn't implicitly contain a solution, but rather be capable of producing something altogether new. We saw that, even though we gave GP complete freedom with regard to how to combine the functions and terminals, the system ended up with a solution that used transformations involving exclusively the centroid and the worst vertex, which is also true for the original Nelder–Mead algorithm. One exception is the contraction of the simplex toward the best vertex, which was not implemented by GP, but it also doesn't seem to matter in the final solution, as we observed in the case of a tree-formed modification of the original Nelder–Mead algorithm.

Acknowledgments

This work was supported by the Ministry of Education, Science and Sport of the Republic of Slovenia under Research Program P2-0246—Algorithms and Optimization Methods in Telecommunications.

References

Burke, E. K., Hyde, M. R., and Kendall, G. (2006). Evolving bin packing heuristics with genetic programming. In *Proceedings of the 9th International Conference on Parallel Problem Solving from Nature*, pp. 860–869.

Burke, E. K., Hyde, M. R., Kendall, G., Ochoa, G., Ozcan, E., and Woodward, J. R. (2009). Exploring hyper-heuristic methodologies with genetic programming. In C. L. Mumford and L. C. Jain (Eds.), *Computational intelligence*, pp. 177–201. Intelligent Systems Reference Library, Vol. 1. New York: Springer.

Bűrmen, A., Puhan, J., and Tuma, T. (2006). Grid restrained Nelder–Mead algorithm. *Computational Optimization and Applications*, 34(3): 359–375.

Ciesielski, V., and Mawhinney, D. (2002). Prevention of early convergence in genetic programming by replacement of similar programs. In *Proceedings of the 2002 Congress of Evolutionary Computation (CEC '02)*, pp. 67–72.

Conn, A. R., Scheinberg, K., and Vicente, L. N. (2009). *Introduction to derivative-free optimization*. Philadelphia, PA: Society for Industrial and Applied Mathematics.

Dioşan, L., and Oltean, M. (2009). Evolutionary design of evolutionary algorithms. *Genetic Programming and Evolvable Machines*, 10(3): 263–306.

Gao, F., and Han, L. (2012). Implementing the Nelder–Mead simplex algorithm with adaptive parameters. *Computational Optimization and Applications*, 51(1): 259–277.

Helmuth, T., Spector, L., and Martin, B. (2011). Size-based tournaments for node selection. In *Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO)*, pp. 799–802.

Koohestani, B., and Poli, R. (2014). Evolving an improved algorithm for envelope reduction using a hyper-heuristic approach. *IEEE Transactions on Evolutionary Computation*, 18(4): 543–558.

Koza, J. R. (1989). Hierarchical genetic algorithms operating on populations of computer programs. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, Vol. 1, pp. 768–774.

Koza, J. R. (1992). *Genetic programming: On the programming of computers by means of natural selection*. Cambridge, MA: MIT Press.

Koza, J. R. (2010). Human-competitive results produced by genetic programming. *Genetic Programming and Evolvable Machines*, 11(3/4): 251–284.

Lagarias, J. C., Reeds, J. A., Wright, M. H., and Wright, P. E. (1998). Convergence properties of the Nelder–Mead simplex method in low dimensions. *SIAM Journal on Optimization*, 9(1): 112–147.

Luke, S., and Panait, L. (2006). A comparison of bloat control methods for genetic programming. *Evolutionary Computation*, 14(3): 309–344.

Martin, M. A., and Tauritz, D. R. (2013). Evolving black-box search algorithms employing genetic programming. In *Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO)*, pp. 1497–1504.

Moré, J. J., Garbow, B. S., and Hillstrom, K. E. (1981). Testing unconstrained optimization software. *ACM Transactions on Mathematical Software*, 7(1): 17–41.

Moré, J. J., and Wild, S. M. (2009). Benchmarking derivative-free optimization algorithms. *SIAM Journal on Optimization*, 20(1): 172–191.

Nelder, J. A., and Mead, R. (1965). A simplex method for function minimization. *Computer Journal*, 7(4): 308–313.

Nguyen, S., Zhang, M., Johnston, M., and Tan, K. C. (2012). Automatic discovery of optimisation search heuristics for two dimensional strip packing using genetic programming. In *Proceedings of the 9th International Conference on Simulated Evolution and Learning*, pp. 341–350.

Nguyen, S., Zhang, M., Johnston, M., and Tan, K. C. (2015). Automatic programming via iterated local search for dynamic job shop scheduling. *IEEE Transactions on Cybernetics*, 45(1): 1–14.

O'Neill, M., Nicolau, M., and Agapitos, A. (2014). Experiments in program synthesis with grammatical evolution: A focus on integer sorting. In *Proceedings of the IEEE Congress on Evolutionary Computation*, pp. 1504–1511.

O'Neill, M., Vanneschi, L., Gustafson, S., and Banzhaf, W. (2010). Open issues in genetic programming. *Genetic Programming and Evolvable Machines*, 11(3/4): 339–363.

Pappa, G. L., Ochoa, G., Hyde, M. R., Freitas, A. A., Woodward, J. R., and Swan, J. (2014). Contrasting meta-learning and hyper-heuristic research: The role of evolutionary algorithms. *Genetic Programming and Evolvable Machines*, 15(1): 3–35.

Poli, R., Langdon, W. B., and McPhee, N. F. (2008). *A field guide to genetic programming* (with contributions by J. R. Koza). Retrieved from www.gp-field-guide.org.uk

Rios, L. M., and Sahinidis, N. V. (2013). Derivative-free optimization: A review of algorithms and comparison of software implementations. *Journal of Global Optimization*, 56(3): 1247–1293.

Sim, K., Hart, E., and Paechter, B. (2015). A lifelong learning hyper-heuristic method for bin packing. *Evolutionary Computation*, 23(1): 37–67.

Törn, A., Ali, M. M., and Viitanen, S. (1999). Stohastic global optimization: Problem classes and solution techniques. *Journal of Global Optimization*, 14(4): 437–447.

Vanneschi, L., and Cuccu, G. (2009). A study of genetic programming variable population size for dynamic optimization problems. In *Proceedings of the International Joint Conference on Computational Intelligence*, pp. 119–126.

White, D. R., McDermott, J., Castelli, M., Manzoni, L., Goldman, B. W., Kronberger, G., et al. (2013). Better gp benchmarks: Community survey results and proposals. *Genetic Programming and Evolvable Machines*, 14(1): 3–29.

Wright, M. H. (2012). Nelder, Mead, and the other simplex method. *Documenta Mathematica*, Extra volume ("Optimization Stories"):271–276.

Xie, H., and Zhang, M. (2013). Parent selection pressure auto-tuning for tournament selection in genetic programming. *IEEE Transactions on Evolutionary Computation*, 17(1): 1–19.