Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

Wydział Elektrotechniki, Automatyki, Informatyki i Elektroniki

KATEDRA AUTOMATYKI

PRACA INŻYNIERSKA

BARTOSZ WITKOWSKI

BUDOWA SOLVERA DLA ROZWIĄZYWANIA PROBLEMÓW OPTYMALIZACYJNYCH

PROMOTOR:
dr Adam Sędziwy

Kraków 2011

OŚWIADCZENIE AUTORA PRACY

OŚWIADCZAM, ŚWIADOMY ODPOWIEDZIALNOŚCI KARNEJ ZA POŚWIADCZENIE NIEPRAWDY, ŻE NINIEJSZĄ PRACĘ DYPLOMOWĄ WYKONAŁEM OSOBIŚCIE I SAMODZIELNIE, I NIE KORZYSTAŁEM ZE ŹRÓDEŁ INNYCH NIŻ WYMIENIONE W PRACY.

.....

PODPIS

AGH

University of Science and Technology in Krakow

Faculty of Electrical Engineering, Automatics, Computer Science and Electronics

DEPARTMENT OF AUTOMATICS

BACHELOR OF SCIENCE THESIS

BARTOSZ WITKOWSKI

BUILDING A SOLVER FOR OPTIMIZATION PROBLEMS

SUPERVISOR:
Adam Sędziwy Ph.D

Krakow 2011

Contents

Chapter 1

Introduction.

Almost any problem found in everyday life can be thought as an optimization problem, from choosing a car to finding an extrema of a function - we use the principle of finding a best solution to a problem.

This purpose of this paper is to introduce **jgol** a optimization library written in java. The library implements most well known stochastic optimization algorithms: Genetic Algorithms (**GA**s), Simulated Annealing (**SA**), and others more recent, or less popular; the implemented algorithms are later explored in detail.

In this work we will concentrate on stochastic, black box, heuristic methods of optimization; the difference between normal and black box optimization methods is that normal methods use some degree of knowledge about the problem (e.g. the derivative of a function, that a function is continuous etc.) and black box methods don't assume any outside knowledge. This additional knowledge makes traditional optimization methods more powerful but makes them less robust.

Stochastic methods have some degree of randomness when operating: a stochastic strategy of finding a peek of a hill could sound like: *"go a step in a random direction; if you're standing lower then you were before - go back - if no take another step"*.

Heuristic methods incorporate additional strategies or knowledge to their operation - the algorithms presented here use heuristics that are usually inspired by real-life processes: Genetic Algorithms are inspired by the process of evolution, Simulated Annealing is inspired by the process of annealing metals.

In this document we will also look closely at some possible optimization problems and the behavior of the implemented algorithms trying to solve them - examining the convergence of the methods used and their speed.

The result of this work include this paper, the optimization library and data collected while evaluating the test problems.

Chapter 2

Background.

2.1 Terminology.

In this document some terms, mostly taken from **genetic algorithms** and **genetic programming** will be used interchangeably with traditional terminology e.g. the term *Individual* will be used alongside a *variable*, *fitness function* will be used interchangeably with *objective function* .

This short list will demonstrate the conventions used in this document and the optimization library.

The function subject to optimization f will be called an **objective function** or an **fitness function**. An element in the domain of f will be called:

- **a variable, unknown** or a **parameter**
- **a point** - in the context of a search spaces
- **an individual** - particularly used when speaking of genetic algorithms

An individual is composed of **decision variables**, or **genes**. A **population** is a collection of individuals.

2.2 Additional definitions.

To help define optimization methods, some most common operators and conventions held throughout this document will be introduced:

We define:

- $r(a, b)$ as operator returning a random uniformly distributed value between a (inclusive), and b (exclusive), i.e: $a \leq r(a, b) < b$, and no value is more probable then another.
- $r_n(\mu, \sigma)$ as operator returning a random normal distributed value with the given expected value μ and the standard deviation σ

The nullary operators $r()$ and $r_n()$ return an uniform random value, or a normally distributed random value respectively, so that the values fit the constraints of current context.

2.3 Optimization.

Optimization is a process of finding a extrema of a function: given a set X and a function f defined on X we need to find a minimal (maximal) value of $f(x)$, $x : x \in X$.

We will call f an objective function, and the set X a search space:

$$f : X \to Y$$

where Y is any arbitrary set. The process of optimization is the problem of finding:

$$x^* = \min_{x \in X} f(x)$$

or alternatively:

$$x^* = \max_{x \in X} f(x)$$

Chapter 3

Optimization methods.

In this chapter the optimization methods implemented in the library will be summarized. The algorithms are stochastic in nature and most of them are based on some real life analogy, or are inspired by some real life process.

When describing the algorithms we assume maximization of the fitness function – if $f(a) > f(b)$ then a is more fit (better) then b.

3.1 Mutation operator

We begin by introducing a simple mutation operator used in most algorithms. $mut(x, \delta)$ is defined as follows:

Require: x be a vector of the size N
  $i \leftarrow r(1, N)$
  $x_i \leftarrow \pm r(0, 1) \cdot \delta$
  return x

Where the parameter δ is understood as the mutation strength.

3.2 Differential evolution

Differential evolution is a evolutionary algorithm created by Price and Stern intended to optimize real-parameter functions.

Differential evolution works on the analogy of a search space. The algorithm starts by randomly assigning positions in the search space to the population. On each iteration of the algorithm two "temporary" populations are produced - the mutant vectors and the trial vectors.

A single mutant vector is formed by a special differential operator. This operator differentiates the **DE** from other **EA**s. The vector is formed by taking three random positions from the population - the difference of two of them scaled by a scaling factor F is added to the remaining one: $v_i = x_{R_1} + F \cdot (x_{R_2} - x_{R_3})$. In terms of the search space: a *vector* was added to a *point*. Some additional restrictions are put on the indexes: $R_1 \neq R_2 \neq R_3 \neq i$.

The trial vectors are formed by crossing over the mutant vectors with individuals from the population. Additionally, at least one gene is taken from the mutant vector to ensure that some crossover took place.

Finally, if the trial vector is better then the corresponding individual in the population - it takes its place in the population.

Formal definition of the algorithm: let $f : \mathbb{R}^n \to \mathbb{R}$ be the fitness function subject to optimization - N being the dimensionality of the problem.

We define N_{pop} as the size of the population, and $x_1 \dots x_{N_{pop}} \in \mathbb{R}^N$ as the population to improve. $v'_1 \dots v'_{N_{pop}}$ - as "mutant vectors", and $u_1 \dots u_{N_{pop}}$ "trial vectors" – posible candidates for the next population.

Additional parameters: a scaling factor $F \in [0, 2]$, and a cutoff rate $CR \in (0, 1]$.

for $i = 1$ to N_{pop} **do**
  $x_i \leftarrow \underbrace{[r(), r(), \dots, r()]}_{N}$
end for
while not *terminationCriterion()* **do**
  for $i = 1$ to N_{pop} **do**
    $R_1 \leftarrow r(1, N_{pop})$
    $R_2 \leftarrow r(1, N_{pop})$
    $R_3 \leftarrow r(1, N_{pop})$
    $v_i \leftarrow x_{R_1} + F \cdot (x_{R_2} - x_{R_3})$
  end for
  for $i = 1$ to N_{pop} **do**
    for $j = 1$ to N **do**
      if $r(0, 1) \leq CR$ **then**
        $u_{i,j} \leftarrow v_{i,j}$
      else
        $u_{i,j} \leftarrow x_{i,j}$
      end if
    end for
  end for
  for $i = 1$ to N_{pop} **do**
    if $f(u_i) < f(x_i)$ **then**
      $x'_i \leftarrow u_i$
    else
      $x'_i \leftarrow x_i$
    end if
  end for
end while

For clarity, ensuring that $R_1 \neq R_2 \neq R_3 \neq i$ was ommited.

3.3 Random Optimization.

Random optimization is an iterative optimization method for single objective, numerical problems. The algorithm starts by assigning uniformly random values to an individual. In each iteration the individual is then changed by adding a normally distributed vector to it; the resulting individual stays changed only if it is better than the source individual.

Let $f : \mathbb{R}^N \to \mathbb{R}$ be the fitness function subject to optimization, N being the dimensionality of the problem. Let $x \in \mathbb{R}^N$ be a position in the search space. The algorithm can be described as:

$x \leftarrow \underbrace{[r(), r(), \dots, r()]}_{N}$
while not *terminationCriterion()* **do**

$x' \leftarrow x + \underbrace{[r_n(), r_n(), \ldots, r_n()]}_{N}$
if $f(x') > f(x)$ **then**
$x \leftarrow x'$
end if
end while

Because the values of an individual may be constrained, we will consider four strategies when dealing with situations where one or more genes don't fit in the constraints.

- Dropping the individual altogether - the algorithm leaves the old value of the individual.
- Dropping only the offending gene - the old gene is left.
- Trimming the gene to the constraint - trimming either to the left or right value of the constraint.
- Bounce back. If g_{min} and g_{max} are the minimum and maximum values of the gene then

while not *inConstraints(g)* **do**
if $g > g_{max}$ **then**
$g \leftarrow g_{max} - |g - g_{max}|$
else
$g \leftarrow g_{min} + |g_{min} - g|$
end if
end while

3.4 Harmony Search.

Harmony Search is a type of **Evolutionary Algorithm** inspired by the process of improvising *jazz* musicians.

Consider a band practicing improvisation over chord changes of a song. The overall available harmony can be imagined as a search space of some problem. Better sounding harmonies are **fitter** (in the sense of an *fitness function*) then bad ones.

Each musician represents a *decision variable*; at each and every practice session (an *iteration* of the algorithm) the musician generates some new note (a *value* of the decision variable).

When practicing, musicians either make something up as they play along (representing a *randomly created value*), or use/modify some note that they've found **good** in the previous practice session (the concept of the **Harmonic Search Memory**)

Notes are taken out of the memory with some given probability $P(chooseFromMemory)$ and can be modified according to some probability $P(pitchAdjust)$.

We additionally define:

- MS as the memory size
- *memory* as the harmony search memory – the *memory* should be at all times sorted so that

$$\forall i = 1 \ldots N - 1 : f(memory_i) \geq f(memory_{i+1})$$

- δ as the pitch adjustment strength

- N as the size of the decision variable
- $f()$ as the fitness function.
- $memory_{MS}$ as the worst element from *memory* (according to the fitness function).

for $i = 1$ to MS **do**
  $memory_i \leftarrow \underbrace{[r(), r(), \ldots, r()]}_{N}$
end for
while not *terminationCriterion()* **do**
  for $i = 1$ to N **do**
    if $r(0,1) \leq P(chooseFromMemory)$ **then**
      $x \leftarrow memory\,(r(0, N))$
      $x'_i \leftarrow x + r_n()$
      if $r(0,1) \leq P(pitchAdjust)$ **then**
        $x_i \leftarrow mut(x_i, \delta)$
      end if
    else
      $x'_i \leftarrow r()$
    end if
  end for
  if $f(x') > f(memory_{MS})$ **then**
    $memory_{MS} \leftarrow x'$
  end if
end while

In the **jgol** optimization library and when analyzing the algorithm memory is understood as the problems population, with population size equal to **MS**

3.5 Particle Swarm Optimization

Particle Swarm Optimization is a optimization algorithm that tries to simulate swarming/social behavior of agents (particles). A single particle in this scheme is a *solution candidate* "flying" through the search space with some velocity. The particle also remembers its best position and can learn the best position known to its neighbor particles.

While traveling the search space the particles adjust their speed (both direction and value) based on their personal experiences and on the knowledge of their neighborhood particles.

Various schemes of determining particle neighborhood can be imagined we will discern:

- the global neighborhood - all particles are neighbors to each other.
- the neighborhood determined by *euclidean* distance.
- the neighborhood determined by normalized *euclidean* distance.

The normalization process tries to adjust the size of the neighborhood so that the dimension of the search space is accounted for.

The algorithm can be described as:

$x_{0 \ldots N_{pop}} \leftarrow \underbrace{[r(), r(), \ldots r()]}_{N}$

$v_{0...N_{pop}} \leftarrow \underbrace{[r(), r(), \ldots r()]}_{N}$

while not *terminationCriterion()* **do**
  for $i = 1$ to N_{pop} **do**
    for $j = 1$ to N **do**
      $\phi_1 \leftarrow r(0, 1)$
      $\phi_2 \leftarrow r(0, 1)$
      $v'_{ij} \leftarrow \omega v_{ij} + \sigma\left(\phi_1 \cdot C_1 \cdot (best(p_i)_j - x_{ij}) + \phi_2 \cdot C_2 \cdot (best_{Nhood}(p_i)_j - x_{ij})\right)$
    end for
    if $v' > v_{max}$ **then**
      $v' \leftarrow v_{max}$
    end if
    if $v' < -v_{max}$ **then**
      $v' \leftarrow -v_{max}$
    end if
    $x'_i \leftarrow x_i + v'_i$
  end for
  $x \leftarrow x'$
  $v \leftarrow v'$
end while

Where:

- N_{pop} is the size of the population.
- N is the number of dimensions of the search space.
- x is a vector of particle positions.
- v is a vector of particle velocities (each velocity is a vector with N elements).
- v_{max} is a maximum posible velocity.
- ω is the *velocity-weight* or *inertia* factor - how much the particle will base its next velocity on its previous one.
- σ is the position weight determining the importance of the particles position to the next particle velocity.
- $\phi_{1,2}$ are a uniform random variables taken as an additional weight when determining the next particle velocity.
- C_1 is the **self confidence weight** (or self learning rate).
- C_2 is the **swarm confidence weight** (or neighborhood learning rate).
- $best_{Nhood}(p)$ is an operator that returns the best (most fit) position known to the particle and its neighborhood.
- $best(p)$ is an operator that returns the most fit position that the particle has visited.

3.6 Cross entropy optimization

The **Cross entropy optimization** consists of two steps:

- Generating a sample population from a distribution.
- Updating the parameters of the random mechanism to produce a better (fitter) sample in the next population.

This behavior conceptually substitutes the problem of finding a optimal individual to a problem of iteratively finding a random distribution that generates good individuals.

We use a random Gaussian distribution to produce variables; the distribution is improved by means of importance sampling and finding a new distribution from the best samples.

Let N be the number of dimensions of the search space, N_{size} be the sample size. $x_1 \dots x_N$ denotes the population (in terms of optimization not a statistical population) the size of N, we assume that the population is sorted so that

$$\forall i = 1 \dots N-1 : f(x_i) \geq f(x_{i+1})$$

We define:

- $mean(x, N_{sample}) = \frac{1}{N_{sample}} \sum_{i=0}^{N_{sample}} x_i$
- $stddev(x, N_{sample}) = \sqrt{\frac{1}{N_{sample}} \sum_{i=0}^{n} (x_i - mean(x, N_{sample}))^2}$

Because x is a vector of values $mean(x)$ and $stddev(x)$ work on a set of vectors and return a vectors of elements – a vector of means and standard deviations of columns of the given input matrix (population). Additional importance sampling with the size of N_{sample} was done to the given set.

The algorithm can be described as:

$x_{0 \dots N_{pop}} \leftarrow \underbrace{[r(), r(), \dots r()]}_{N}$
$\mu \leftarrow mean(x, N_{sample})$
$\sigma \leftarrow stddev(x, N_{sample})$
while not *terminationCriterion()* **do**
  for $i = 1$ to N_{pop} **do**
    $x_i \leftarrow \underbrace{[r_n(\mu_1, \sigma_1), r_n(\mu_2, \sigma_2), \dots, r_n(\mu_N, \sigma_N)]}_{N}$
  end for
  $\mu \leftarrow mean(x, N_{sample})$
  $\sigma \leftarrow stddev(x, N_{sample})$
end while

Additionally the values of μ and σ can be smoothed over time (from iteration to iteration) to improve the algorithm convergence.

3.7 Genetic algorithms

Genetic algorithms are nature inspired algorithms that simulate sexual reproduction of species and a "survival of the fittest" approach to evolution.

Elements of the search space X are encoded as strings of *genes*. The first **GA**s used binary strings to represent individuals; although, the particular implementation of genetic algorithms in the library is encoding agnostic, real valued genes were used for encoding.

Creation - the individuals are usually created by assigning random values to their genes, such a population can be imagined as a primordial soup. The population progresses iteratively by reproducing either asexually by **mutation** only or sexually by means of **crossover**.

Mutation - Individuals mutate if one (or more) of their genes changes randomly by either flipping a bit in binary encoded chromosomes, adding a random value to real/integer encoded genes, changing the order of genes in the individual etc – because the implicitly the individuals are encoded as vectors of real values we use the previously defined mutation operator $mut(x, \delta)$.

Crossover - crossover or recombination is a binary operator that tries to mimic biological reproduction and crossover. The operator swaps the genes from two individuals of the population (parents) producing new individuals (children). There are many schemes of crossover, some of theme are:

- **one point crossover** where a point is chosen at random in the parents gene-strings, genes up to that point are taken from the first parent, the rest from the second.
- **two point crossover** identical to one point crossover - only two points are chosen. Genes are taken from the first parent then the second then then first once again.
- **uniform crossover** where the likelihood that a gene will be taken from any parent is the same.

Traditionally crossover took two parent individuals and produced two children individuals, producing only one child is also acceptable. Crossover from many parents is possible with the uniform crossover operator.

Selection - the selection process in genetic algorithms determines which individuals survive and mate producing offspring (by mutation or crossover). Some selection operators:

- **Best selection** - N best individuals from the population are selected.
- **Random selection** - N random individuals from the population are selected.
- **Roulette selection** - also known as fitness proportional selection - in this scheme of more fit individuals are more likely to be selected.

  This is usually done by calculating the total fitness of individuals in the population $totalFit$, sorting the population, and assigning "spaces" on the roulette wheel so that the fittest individual "occupies" the space between 0 and $f(x_1)$, the second $f(x_1)$ to $f(x_1) + f(x_2)$ and so on.

  A number between 0 and $totalFit$ is generated - the number determines which individual is selected. This process can be imagined as spinning a roulette wheel with the size of the fields corresponding to the fitness of individuals of the population.

  This is done so that N individuals are selected.
- **Top percent selection** - N individuals are selected at random from the given top $t \in (0, 100)$ percent individuals of the population.
- **Tournament selection** - selects N individuals by holding a *tournament*.

  A tournament is done by taking t_{size} individuals from the population at random and *selecting* the best individual from the tournament.

Let N be the dimensionality of the problem, N_{pop} be the size of the population, N_g the number of genes n a individual, $x_1 \dots x_{N_{pop}}$ the populationl

$sel_N(x)$ is the operator representing the selection method - returning N selected individual, $cross(x_{i_1}, x_{i_2})$ the crossover method, and $mut(x_i, \delta)$ the mutation operator defined previously.

Parameters:

- $P(m)$ the probability that the offspring will be mutated.
- $P(c)$ the probability that the offspring will be crossed.

for $i = 1$ to N_{pop} **do**
  $x_i \leftarrow \underbrace{[r(), r(), \dots r()]}_{N}$
end for
while not *terminationCriterion()* **do**
  for $i = 1$ to N_{pop} **do**
    $parents \leftarrow sel_2(x)$
    if $r(0,1) \leq P(c)$ **then**
      $child \leftarrow cross(parents_1, parents_2)$
    else
      if $r(0,1) \leq 0.5$ **then**
        $child \leftarrow parents_1$
      else
        $child \leftarrow parents_2$
      end if
    end if
    if $r(0,1) \leq P(m)$ **then**
      $child \leftarrow mut(child, \delta)$
    end if
    $x'_i \leftarrow child$
  end for
  $x \leftarrow x'$
end while

The algorithm implicitly handles sorting; the details of the selection and crossover operators were omitted for brevity sake.

3.8 Simulated annealing

Simulated annealing is a probabilistic optimization method inspired by the physical process of annealing metals - where a metal is slowly cooled so that its structure is "frozen" in a minimal energy configuration. [2]

The algorithm itself is similar to hill climbing: an individual will iteratively go to a better neighbourhood position (picked at random), additionally an individual may go to a worse position with a probability proportional to its *temperature*.

The probability that a individual will go to a worse position is calculated by the Boltzmann probability factor $e^{-\frac{E(pos)}{k_B T}}$. Where $E(pos)$ is the energy at the new position (calculated as a difference of fitness two positions), k_B is the Botzmann constant ($1.380650524 \cdot 10^{-23} J/K$), and T is the temperature of the "solid".

The rate at which the solid is frozen is called a *cooling schedule*. We will explore two variants of cooling schedules:

- $$getTemperature(t) = T_{start} \cdot (a^t)$$

  Where T_{start} is the starting temperature, t is the current iteration, and a is a parameter of the cooling schedule. Additionally: $a > 0 \wedge a \approx 0$.

- $$getTemperature(t) = T_{start} \cdot (1 - \varepsilon)^{t/m}$$

  Where T_{start} is the starting temperature, t is the current iteration. ε and m are parameters of the cooling schedule. Additionally $0 < \varepsilon \leq 1.0$

The algorithm itself can be summarized as:

$x \leftarrow \underbrace{[r(), r(), \ldots r()]}_{N}$
$x_{best} \leftarrow x$
while not *terminationCriterion()* **do**
  $x' \leftarrow mut(x, \delta)$
  $\Delta E \leftarrow f(x) - f(x')$
  if $\Delta E \leq 0$ **then**
    $x \leftarrow x'$
    if $f(x) > f(x_{best})$ **then**
      $x_{best} \leftarrow x$
    end if
  else
    $T \leftarrow getTemperature(t)$
    if $r(0,1) < exp\left(\frac{\Delta E}{{k_B}^t}\right)$ **then**
      $x \leftarrow x'$
    end if
  end if
  $t \leftarrow t + 1$
end while

Where f is the objective function $f : \mathbb{R}^N \to \mathbb{R}$, and $mut(x, \delta)$ is the mutation operator, and δ the mutation strength parameter.

3.9 Taboo search

Taboo search is a search method that retains the memory of visited points in the search space called a **taboo list** which are not visited again.

We will consider only one individual, but the algorithm holds for entire populations (if applied to every individual in the population). Additional operators:

- $mut(x, \delta)$ denotes the mutation operator defined before, and δ the mutation strength parameter.
- **initializeTaboo**() initializes the taboo list.

- **isTaboo**(x) checks if a individual is taboo, we introduce two variants
  1. We compare individuals from the taboo list with the given individual gene by gene. We say that a individual is taboo if all genes of the given individual are the same as one of the individuals of the taboo list.
  2. We compare individuals using clustering - the given individual is taboo if it is close (in some measure of closeness - Euclidean distance) to any individual in the taboo list.
- **updateTaboo**(x) updates the taboo list with the given individual, and removes the oldest entry of the list when some capacity limit is encountered.

$x \leftarrow r()$
initializeTaboo()
while not *terminationCriterion()* **do**
  $x' \leftarrow mut(x, \delta)$
  if not *isTaboo*(x') **then**
    if $f(x') > f(x)$ **then**
      $x \leftarrow x'$
    end if
    updateTaboo(x')
  end if
end while

Chapter 4

Implementation.

The **Java Generic Optimization Library** or **jgol** implements the described optimization methods. The library is divided into packages and is may be distributed as a `jar` file, the library uses Jakarta Commons Chains – for the *,,Chain of responsibility"* pattern, and an Mersenne twister random generator implementation by Sean Luke, held in a separate package.

4.1 Package structure

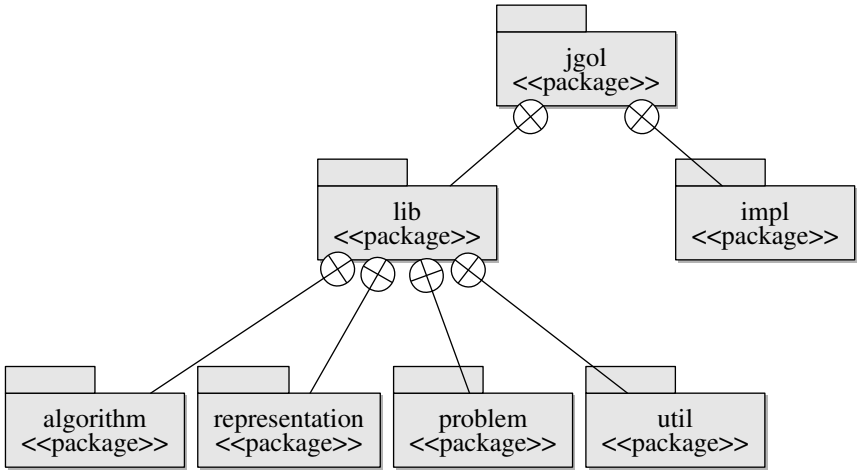


Figure 4.1: The library package structure.

The most important packages in the library are:

- `jgol.lib.algorithm` – holding the implementations of the algorithms, and the abstract base class `Algorithm`.
- `jgol.lib.problem` – which stores the problem specific classes – the abstract class `Fitness` that classes implementing a fitness function must extend, the abstract `Gene` class, an abstract `FitnessType` class – the fitness type returned by the fitness function, and most importantly the abstract `Problem` class which defines an optimization problem.
- `jgol.lib.representation` – storing the `Individual` and `Population` classes.

- `jgol.lib.util` – utility classes.

4.2 Concurrency

The library provides its own internal Producer–Consumer mode of concurrency with abstract `Work` classes done by the `WorkerThreads` – implementations of the Consumer pattern.

4.3 Algorithms

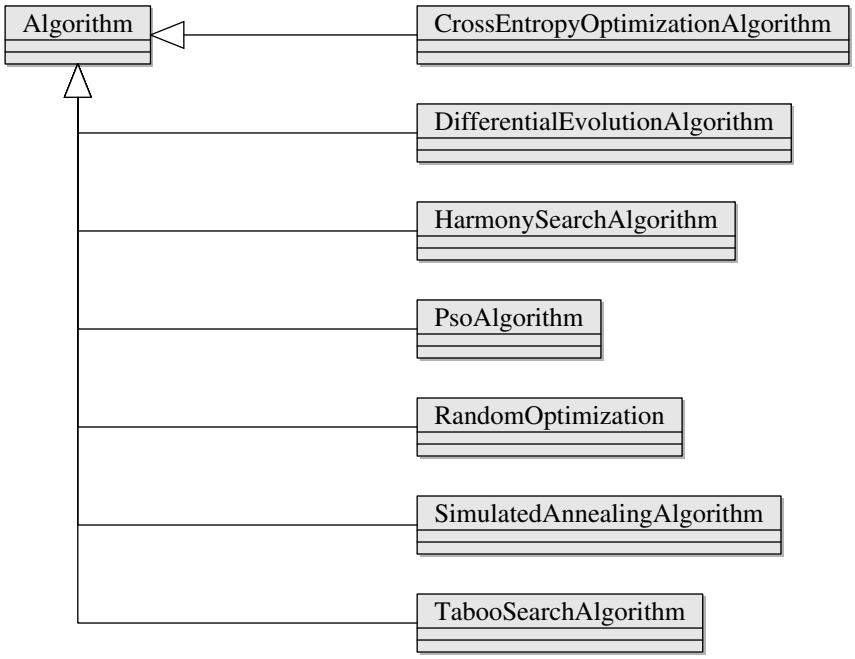


Figure 4.2: Algorithms.

Every algorithm must extend the abstract `Algorithm` class, and implement the `iterate()` method specific to some algorithm. For the sake of simplicity and ease of concurrency the implementations of this method are purely functional.

4.4 Solving optimization problems

To solve an optimization problem the user of the library must create an concrete instance of the `Problem` class, additionally setting an concrete `GeneFactory` – an implementation of the *Factory Method* pattern, and a `FitnessFunction` returning some `FitnessType` class.

The `Problem` class then creates an initial `Population` which is iterated by an instance of a `Algorithm`.

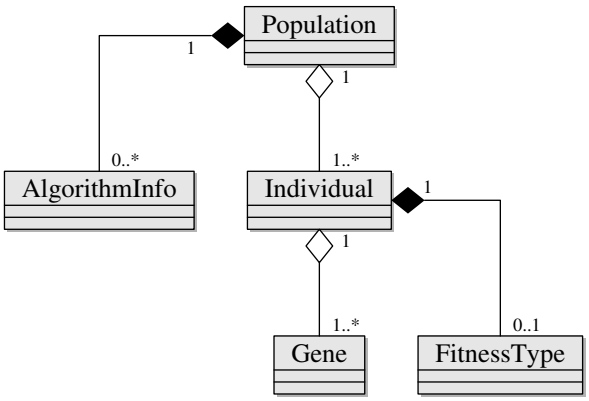


Figure 4.3: `Population` anatomy.

Chapter 5

Benchmarks.

According to the ,,No Free Lunch Theorem"[6] any two algorithms **A** and **B** *on average* perform identically; consequently, devising an test suite which determines which algorithm is better for any function is impossible.

Nevertheless, we can devise a benchmark which will evaluate problems judging them by some degree of performance; algorithms will be compared using the following criteria:

- How fast (in terms of real time) the implementation of an algorithm finds a known optima , or how fast it improves its best known solution in some particular optimization problem.
- How many evaluations of the fitness function it took to find the best solution.
- If the algorithms deteriorates when given problems with more dimensions.
- Whether the algorithms find the best solution .

5.1 Algorithms and parameters

The following algorithms will be evaluated by the test suite.

- **Differential Evolution Algorithm** for population size $PS \in \{5, 25, 100, 250, 500\}$, $f \in \{0.0, 0.2, 0.4\}$, cutoff $\in \{0.1, 0.3, 0.5, 0.7, 0.9\}$
- **Random Optimization** for $PS \in \{1\}$, $\mu \in \{-0.1, -0.01, 0.0, 0.01, 0.1\}$, $\sigma \in \{0.001, 0.01, 0.1, 0.5, 0.7, 1.0, 3.0\}$ and four constraint strategies.
- **Harmony Search** for $PS \in \{5, 25, 75, 100\}$, $P(chooseFromMemory) \in \{0.7, 0.845, 0.99\}$, $P(pitchAdjust) \in \{0.1, 0.3, 0.5\}$, $\delta \in \{0.1, 0.5, 1.0\}$
- **Particle Swarm Optimization** for $PS \in \{10, 30, 60\}$, $v \in \{-3.0, -1.5, 0.0, 1.5, 3.0\}$ **self learning rate** $\in \{0.0, 2.0, 4.0\}$ **neighborhood learning rate** $\in \{2.0, 4.0\}$ for **global**, and **local** neighborhoods; additionally, for local neighborhoods **neighborhood size** $\in \{0.1, 0.5\}$ (normalized and not). The position weight σ was set at 1.0, a maximum velocity v_{max} was set at 1.0.
- **Cross Entropy Optimization** for population size $PS \in \{50, 100, 250, 500, 750, 1000\}$ and with the importance sampling size $N_{sample} \in 0.1PS_i, 0.2PS_i, 0.5PS_i, 0.7PS_i$, where PS_i is the current population size.

Searching with $PS < 50$ was deemed not feasible - experiments showed that **CE** doesn't work well with small populations.

- **Genetic Algorithms** for $elisimSize = 0$ and $PS \in \{10, 25, 50, 100\}$, $selectionMethod = tournament$, $tournamentSize \in \{2, 6\}$, $parentCount \in \{2, 4\}$, $crossoverMethod \in \{$ *one point*, *uniform* $\}$

  Due to time constrains some combinations of parameters were not tested for $elisimSize = 2$.

- **Simulated Annealing** for **step** $\in \{0.1, 0.2, 0.4, 0.6, 1.0\}$, and two variants of cooling schedules:
  - **Geometric** for **Tstart** $\in \{2000, 5000, 10000, 20000\}$, **a** $\in \{0.99, 0.999, 0.9999, 0.99999\}$
  - **"VariantI"** for **Tstart** $\in \{2000, 5000, 10000, 20000\}$, **m** $\in \{5, 100\}$ $\varepsilon \in \{0.1, 0.001\}$

- **Taboo Search** for **step** $\in \{0.1, 0.2, 0.4, 0.6, 1.0\}$, **taboo size** $\in \{10, 100, 500, 1000, 10000\}$, **clustering** and **non-clustering** taboo, **cluster size** $\in \{0.001, 0.01, 0.1, 0.25, 0.5\}$

5.2 Test functions.

We will use the following convention for discerning test functions 1–5:

$$f_n^N : \mathbb{R}^N \to \mathbb{R}$$

Where N is the dimensionality of the problem, and n is a reference number.

1. **Minimization of the Sphere Function:**

$$f_1^N(x) = \sum_{i=1}^{N} x_i^2$$

   - Search domain: $|x_i| < 5.12$, $i = 1, 2, \ldots N$
   - Global minimum: $x^* = (0, \ldots, 0)$, $f(x^*) = 0$
   - No local minima besides the global minimum.

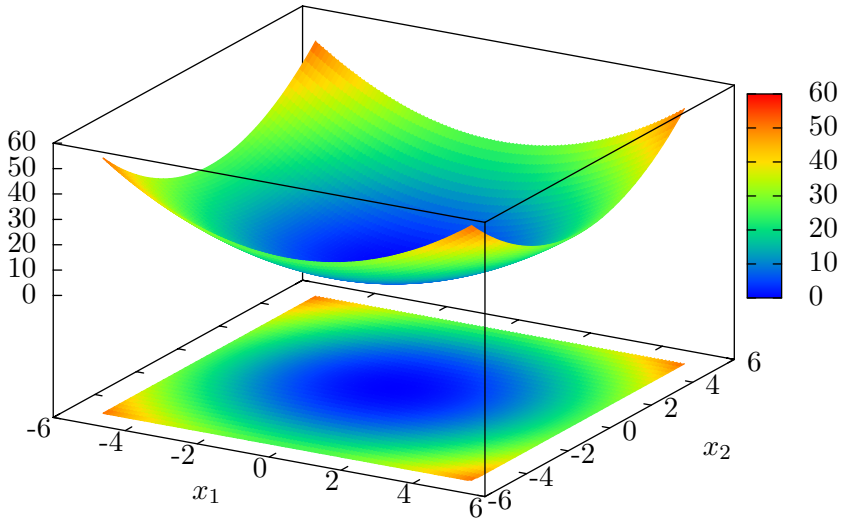


Figure 5.1: $f_1^2(x)$

2. **Minimization of the Rosenbrock Function:**

$$f_2^N(x) = \sum_{i=1}^{N-1} \left[100(x_i^2 - x_{i+1})^2 + (x_i - 1)^2\right]$$

- Search domain: $|x_i| < 5.12$, $i = 1, 2, \ldots N$
- Global minimum: $x^* = (1, \ldots, 1), f(x^*) = 0$
- Several local minima.

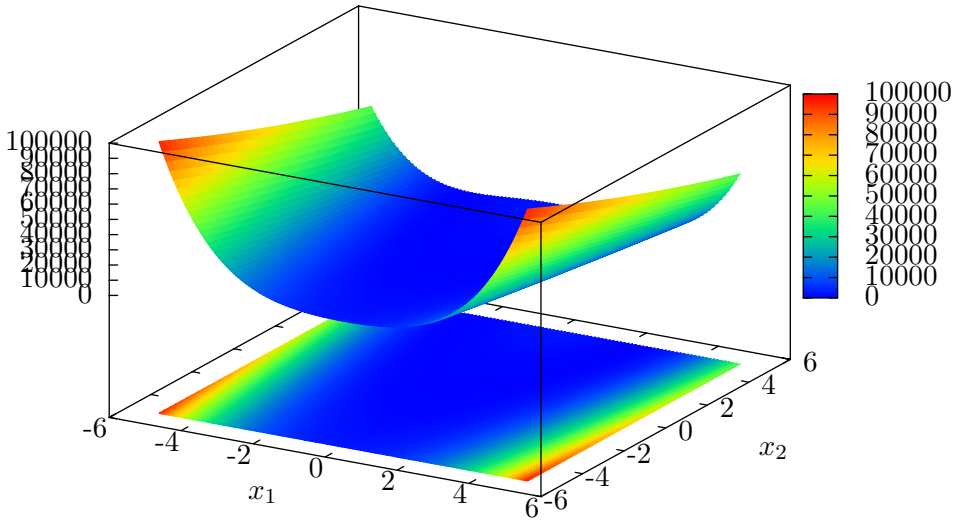


Figure 5.2: $f_2^2(x)$

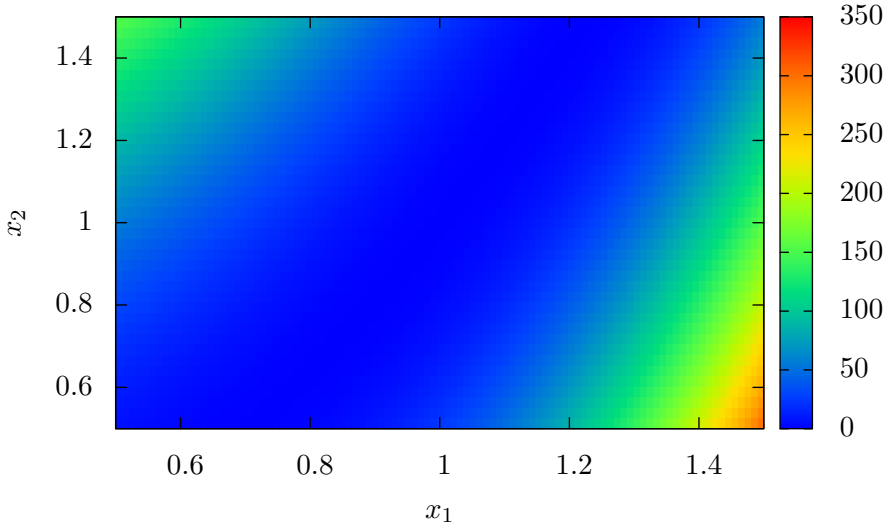


Figure 5.3: $f_2^2(x)$ near the global minimum $(1, 1)$

3. **Minimization of the Step Function:**

$$f_3^N(x) = \sum_{i=1}^{N} \lfloor x_i \rfloor$$

- Search domain: $|x_i| < 5.12,\ i = 1, 2, \ldots N$
- Global minima: $x^* : x_i \leq -5,\ f(x^*) = -6N,\ i = 1, 2, \ldots N$
- No local minima besides the global minimum.

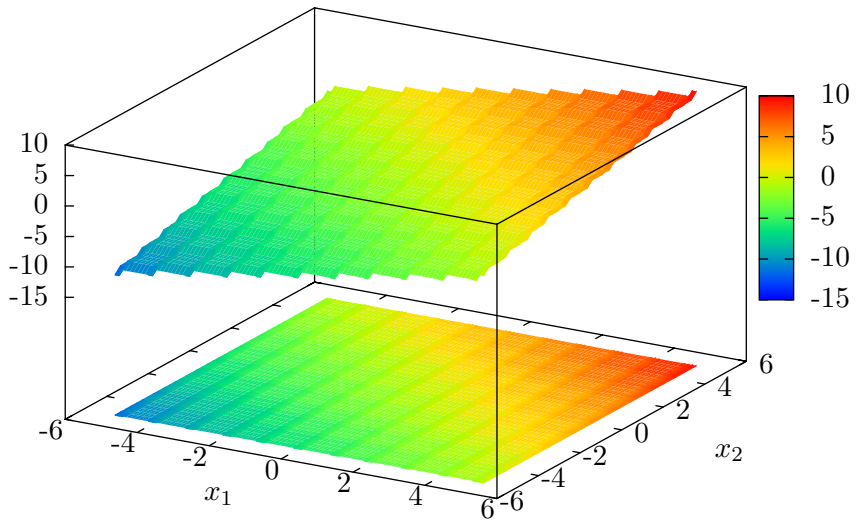


Figure 5.4: $f_3^2(x)$

4. **Minimization of the Rastrigin Function:**

$$f_4^N(x) = 10N + \sum_{i=1}^{N}(x_i^2 - 10\cos(2\pi x_i))$$

- Search domain: $|x_i| < 5.12,\ i = 1, 2, \ldots N$
- Global minimum: $x^* = (0, \ldots, 0), f(x^*) = 0$
- Several local minima.

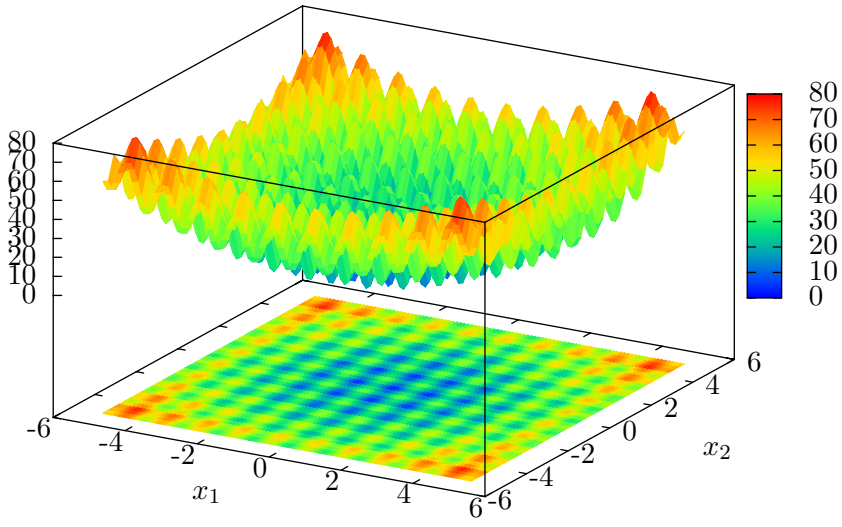


Figure 5.5: $f_4^2(x)$

5. **Minimization of the Griewank Function:**

$$f_5^N(x) = \sum_{i=1}^{N} \frac{x_i^2}{4000} - \prod_{i=1}^{N} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$$

- Search domain: $|x_i| \leq 600$, $i = 1, 2, \ldots N$.
- Global minimum: $x^* = (0, \ldots, 0)$, $f(x^*) = 0$.
- Several local minima.

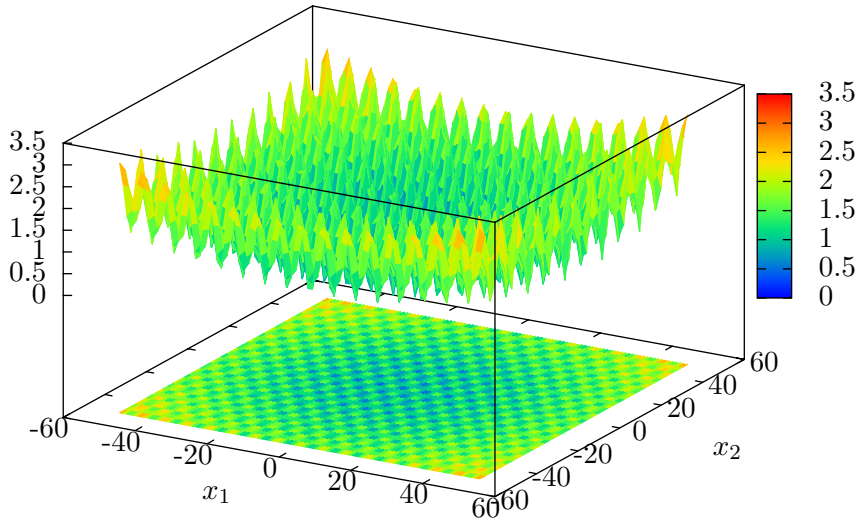


Figure 5.6: $f_5^2(x)$

6. **Minimization of the Easom Function:**

$$f_6(x_1, x_2) = -cos(x_1)cos(x_2)e^{-(x_1-\pi)^2-(x_2-\pi)^2}, x_i : |x_i| \leq 100$$

- Number of variables: 2
- Search domain: $|x_i| < 100$, $i = 1, 2$
- Global minimum: $x^* = (\pi, \pi), f(x^*) = -1$
- No local minima besides the global minimum.

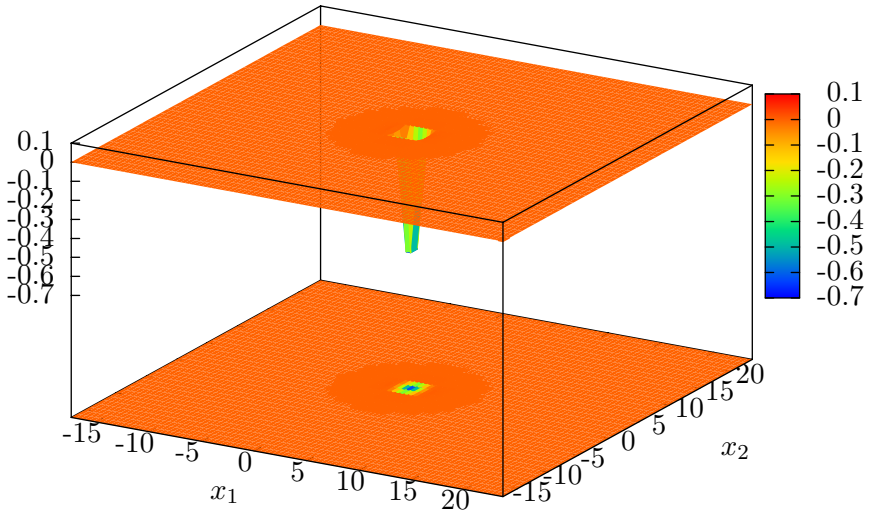


Figure 5.7: $f_6(x_1, x_2)$

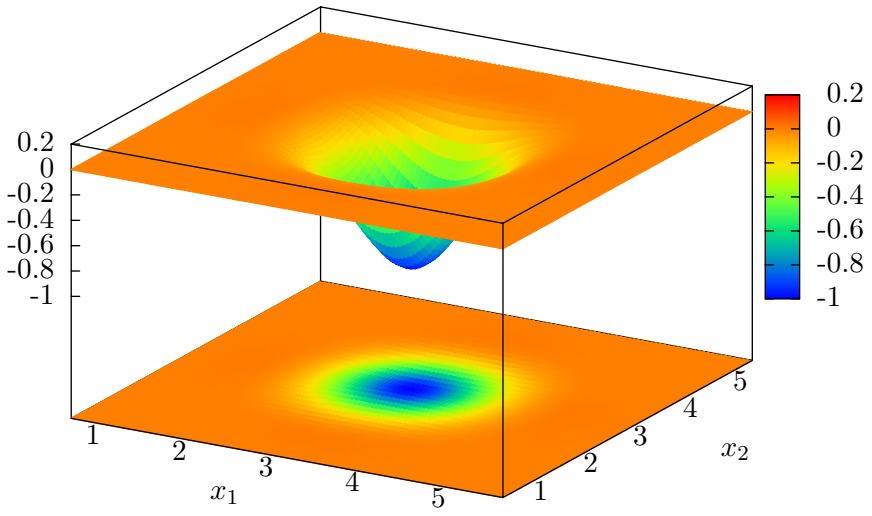


Figure 5.8: $f_6(x_1, x_2)$ near the global optimum (π, π)

7. **Image from polygons.**

   The problem of finding the best combination of N semitransparent colored D–gons ($100 \leq N \leq 1000$, $3 \leq D \leq 10$) that when rendered will produce an image I, so that the difference

between the rendered image and the given one is minimized.

The polygons are encoded so that each polygon is represented by $2D + 4$ real numbers $0 \leq x_i \leq 1$, and the first $2D$ values correspond to the positions of the points of the polygon. The last four values correspond the $RGBA$ (Red, Green, Blue and Alpha).

Given the encoding $\{0, 0, 0, 1, 1, 0, 1, 0, 0, 0.5\}$ and an input image I with the width of w_I and height h_I the encoded polygon will render to a fully red triangle with 50% alpha formed with by points $(0, 0)$, $(0, h_I)$, $(w_I, 0)$.

$$f_7^{N,D,I}(x) = \sum_{i=1}^{w_I} \sum_{j=1}^{h_I} (I(i,j) - I'(i,j))^2$$

- Large number of variables: 1000 – 14000
- Search domain: $0 \leq x_i < 1$, $i = 1, 2, \ldots N \cdot (2D + 4)$
- Unknown global minimum.
- Unknown quantity of local minima.

5.3 Test suite.

The test suite consists of:

- $f_i^N(x)$ for $i \in \{1, 2, 3, 4, 5\}$ $N \in \{2, 4, 8, 14, 20, 32, 44, 54, 76, 98, 120, 148\}$
- $f_6(x)$
- $f_7^{N,D,I_{1,2}}(x)$ for $N = 150$, $D = 6$

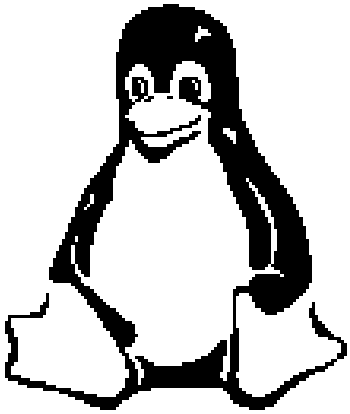


Figure 5.9: The tests images: I_1 and I_2

5.4 Benchmark

We will evaluate the optimization methods against the test suite, keeping track of:

- 1. **the amount of iterations**
  2. **time** – real time spent on optimizing

3. **amount of evaluations of the fitness function**

- For test functions 1 – 6: the distance to the know optimum.

5.5 Technical and implementation details.

5.5.1 System parameters

The tests were done on a Intel Core 2 Duo CPU T9400 running at 2.53GHz on Ubuntu Karmic Koala Linux with a 2.6.31-22-generic kernel on a `x86_64` architecture under normal system load; `java -version` reports:

```
Java(TM) SE Runtime Environment (build 1.6.0_20-b02)
Java HotSpot(TM) 64-Bit Server VM (build 16.3-b01, mixed mode)
```

5.5.2 Logging

The `Log` class created for is called every iteration of the algorithm and keeps track how much time has passed between its calls. Log entries are buffered and, as a compromise between efficiency and memory usage, are written to the hard drive from time to time; time spent on writing the log is taken into ignored so that it doesn't corrupt the results.

For the sake of hard drive space, repeated values of the solution are omitted – only **improvements** of the current solution are logged.

The stopping criteria for algorithms are as follows:

- Time spent on optimizing exceeded 5 minutes.

In problems 1–6:

- The best found solution x is within $\varepsilon = 10^{-4}$ of the know optimum.
- The algorithm hasn't improved the population for more then 1000 iterations.

For the "Image from polygons" problem:

- The algorithm hasn't improved the population for more then 100 iterations.

The population is not improved if:

Table 5.1: Improvement measures.

Algorithm	Improvement measure
Random Optimization Taboo Search Harmony Search Differential Evolution	No individual in the population was improved.
Cross–Entropy Optimization	The fitness of the mean individual was not improved
Particle Swarm Optimization Genetic Algorithm Simulated Annealing	The best found solution was not improved.

The test were repeated five times and averaged so that bad/good starting values may be taken into account.

Chapter 6

Results.

6.1 Influence of parameters and algorithm analysis

The conclusions and statistics presented here may only hold true for the given optimization problems, especially when ranking which algorithm is the best. Nevertheless, some insights and "rules of a thumb" may be gathered by looking at the performances of the algorithms and their settings; heuristics can be made about "good" values of the algorithm parameters.

Some parameters and settings that were not tested and could possibly influence the performance of the algorithms.

- Encoding methods (especially for genetic algorithms)
- Different selection schemes for genetic algorithms.
- Different stopping criteria.
- Different implementations of the mutation operator.
- Topologies in the **Particle Swarm Optimization** algorithm.
- Adaptive algorithms - especially adaptive cooling schedules for the **Simulated Annealing** algorithm.
- Code-optimization, profiling and the influence of parallelism on the algorithms.

Additionally, the tested parameters constitute only a small subset of the possible range of the available parameters.

6.1.1 Algorithm analysis

The tests provided a large data set – the algorithms were run on 63 problems, 61 of which were numerical optimization problems - and will be examined first.

Most algorithms faired well when solving most given optimization problems, the only problems for which no solution was found were the Rosenbrock function optimization problem for dimensions 144 and 120.

The algorithms will be compared by average proximity to a known optimum – this gives a good performance measure that scales well for different problems and varying number of dimensions. Additional performance measures will be introduced when analyzing parameter influence.

Variants of the **Differential evolution** algorithm scored best by a large degree, but on average were not better than all algorithms excluding **Particle Swarms** and **Random Optimization**.

The **DE**s were among the most robust – variants of the **Differential evolution** algorithm were more likely to find an optimum then any other algorithm.

algorithm_name	proximity	s.	time.percentage	n_eval
DA (f = 0.2, cutoff = 0.1) PS = 100	53.48	2609.48	26.52	184940273.00
DA (f = 0.2, cutoff = 0.1) PS = 250	54.49	3114.65	31.66	155036916.00
DA (f = 0.2, cutoff = 0.1) PS = 500	56.40	3815.92	38.79	116247759.00
DA (f = 0.4, cutoff = 0.9) PS = 250	60.45	2455.16	24.96	104865895.00
DA (f = 0.2, cutoff = 0.3) PS = 100	68.69	2214.95	22.51	138700727.00

Table 6.1: Global top 5 algorithms (by proximity).

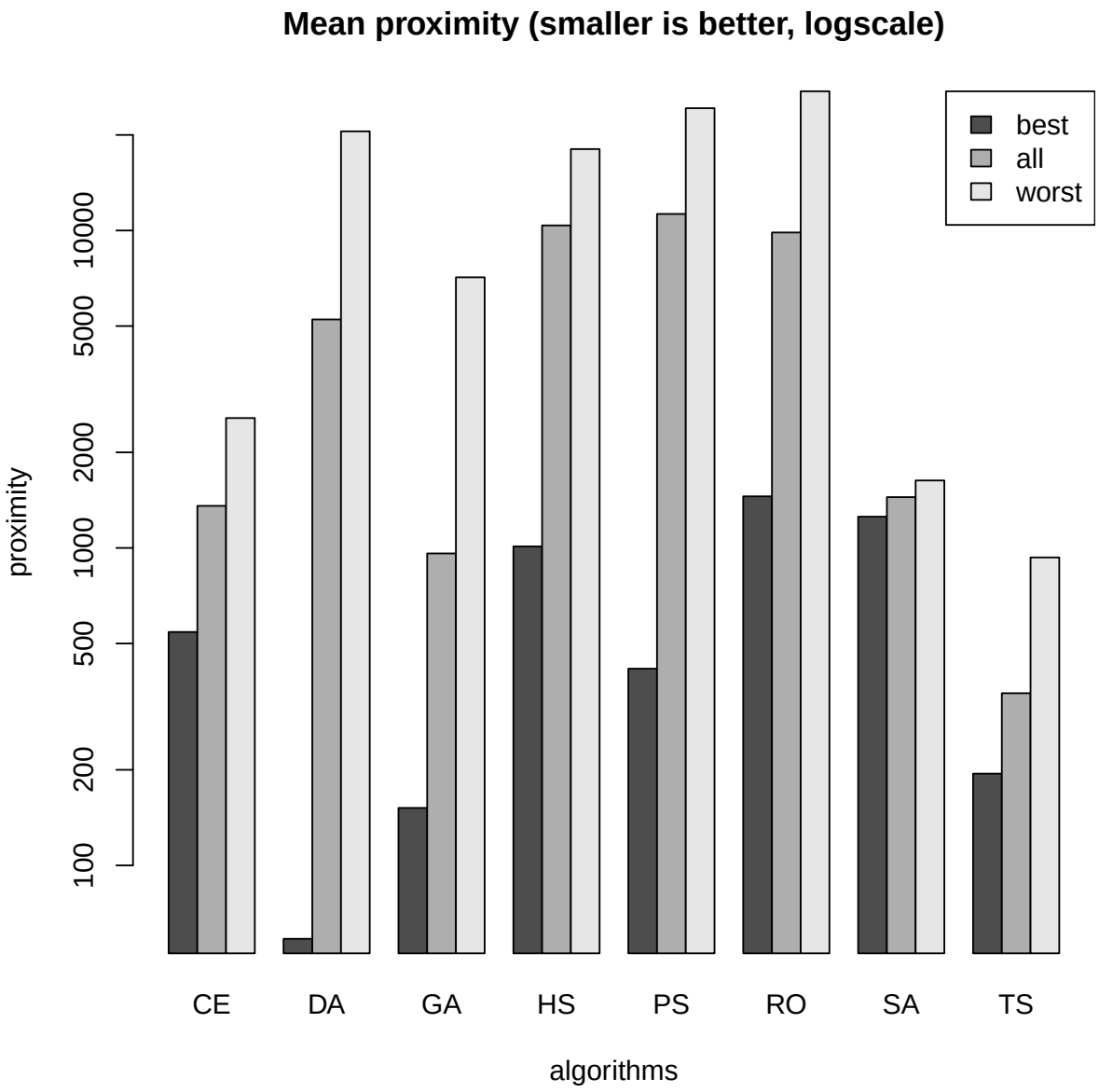


Figure 6.1: Algorithm performance – for every algorithm three sets were taken into account - five best variants of the algorithm, all variants of the algorithm and five worst variants. Proximities of those sets were then averaged.

Algorithm robustness

Robustness was measured by comparing the best found values of each algorithm variant to a know optimum. If the best found solution was withing $\varepsilon = 10^{-3}$ to a known optimum the algorithm found a solution. All variants (sets of parameters) of an algorithm were scored this way.

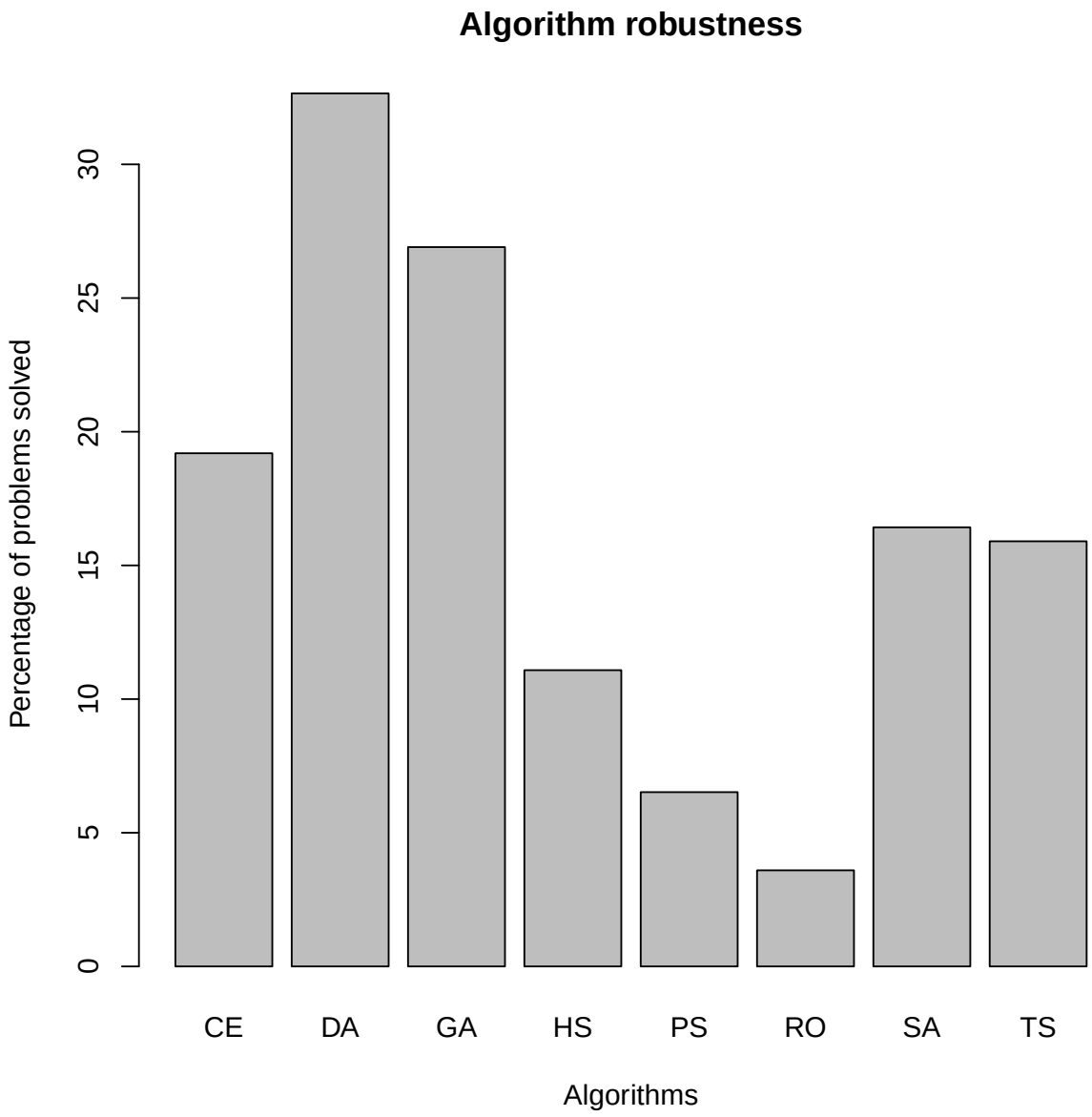


Figure 6.2: Algorithm robustness.

6.1.2 Parameter influence

When examining the influence of parameters three parameters will be compared:

1. Mean sum of proximities to the optimum of algorithms in numerical problems.
2. Mean sum of milliseconds that the algorithm has needed to stop.
3. Summed average (mean) count of fitness function evaluations.

The first parameter tells us about the mean convergence of a set of algorithms, the second and third tell about the speed of convergence.

Care must be taken to ensure that the second and third parameters are not taken into account by their own as they don't differentiate between fast run times due to failure and slow convergence to the optimum. Those parameters are helpful when examining two (or more) sets of parameters with similar proximity/convergence.

Where not stated explicitly it is implied that the influence of parameters is understood in terms of average performance (time, objective function evaluation count). Consequently, some combinations of parameters, although bad on average, may by among the best. Additionally, when speaking of algorithm speed we mean both real time spent optimizing, and the number of evaluations done, unless explicitly stated otherwise.

6.1.3 Differential Evolution

The Differential Evolution algorithm showed the best performance compared to other algorithms when considering only numerical optimization – globally the five best algorithms are **DE**s.

On average convergence improved when increasing the population size, with small differences in convergence speed (both real time, and the count of fitness function evaluations).

On average from the chosen scaling factor f seems to work best for $f = 0.4$, and worst for $f = 0.0$. There were slight variations in time for $f = 0.2$ and $f = 0.4$.

On average the smaller values of the *cutoff* factor worked best both in terms of convergence and convergence speed.

f	cutoff	population size	Proximity	Time [s]	Time [%]	n-eval
0.2	0.1	100	53.48	2609.48	26.52	184940273
0.2	0.1	250	54.49	3114.65	31.66	155036916
0.2	0.1	500	56.40	3815.92	38.79	116247759
0.4	0.9	250	60.45	2455.16	24.96	104865895
0.2	0.3	100	68.69	2214.95	22.51	138700727

Table 6.2: Best five settings (by proximity) for the **Differential Evolution** algorithm. Time [s] denotes time in ms spent on optimizing, and Time [%] is the time spent relative to the worst runtime. n-eval is the number of evaluations of the fitness function.

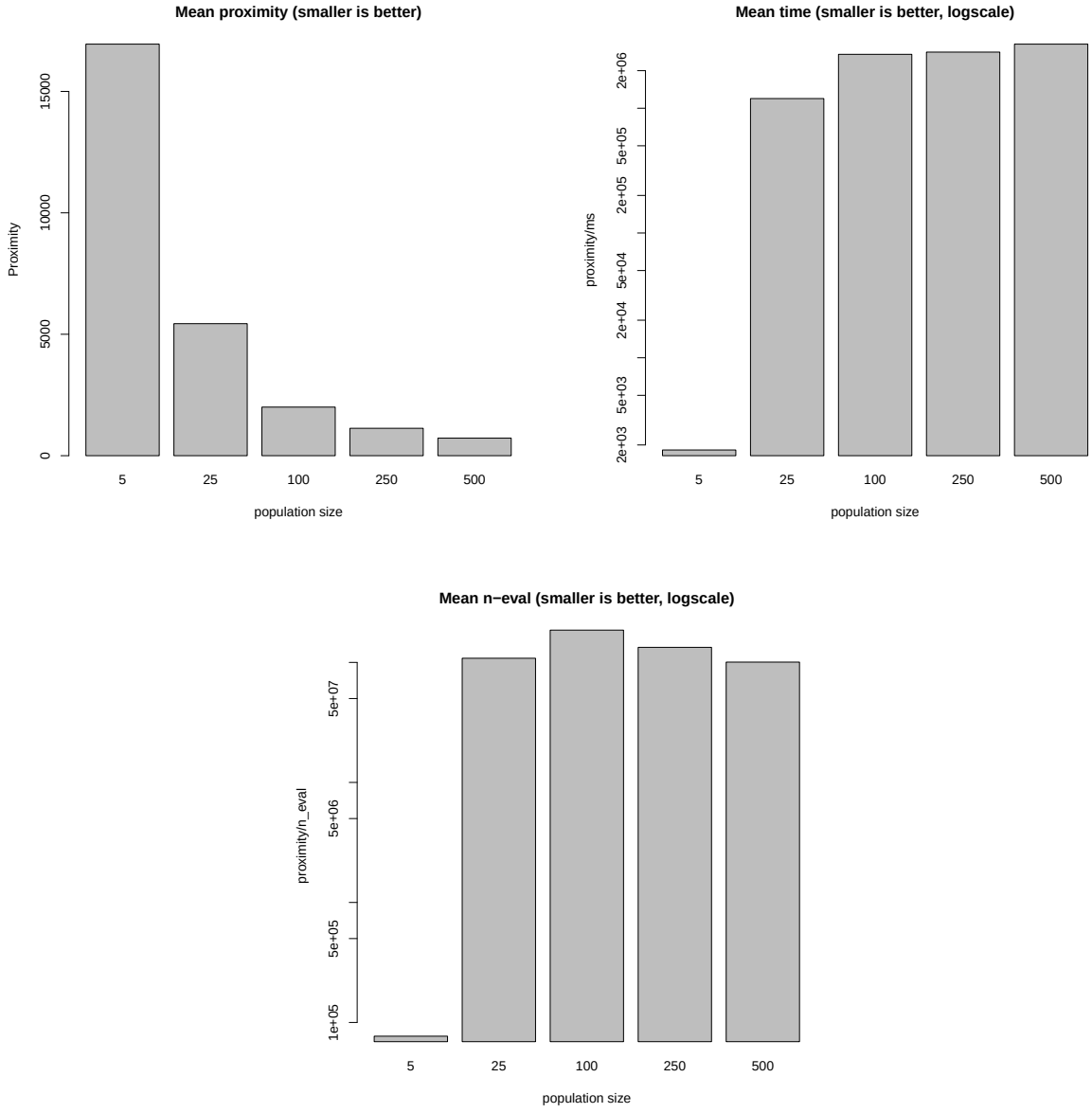


Figure 6.3: Parameter influence – population size

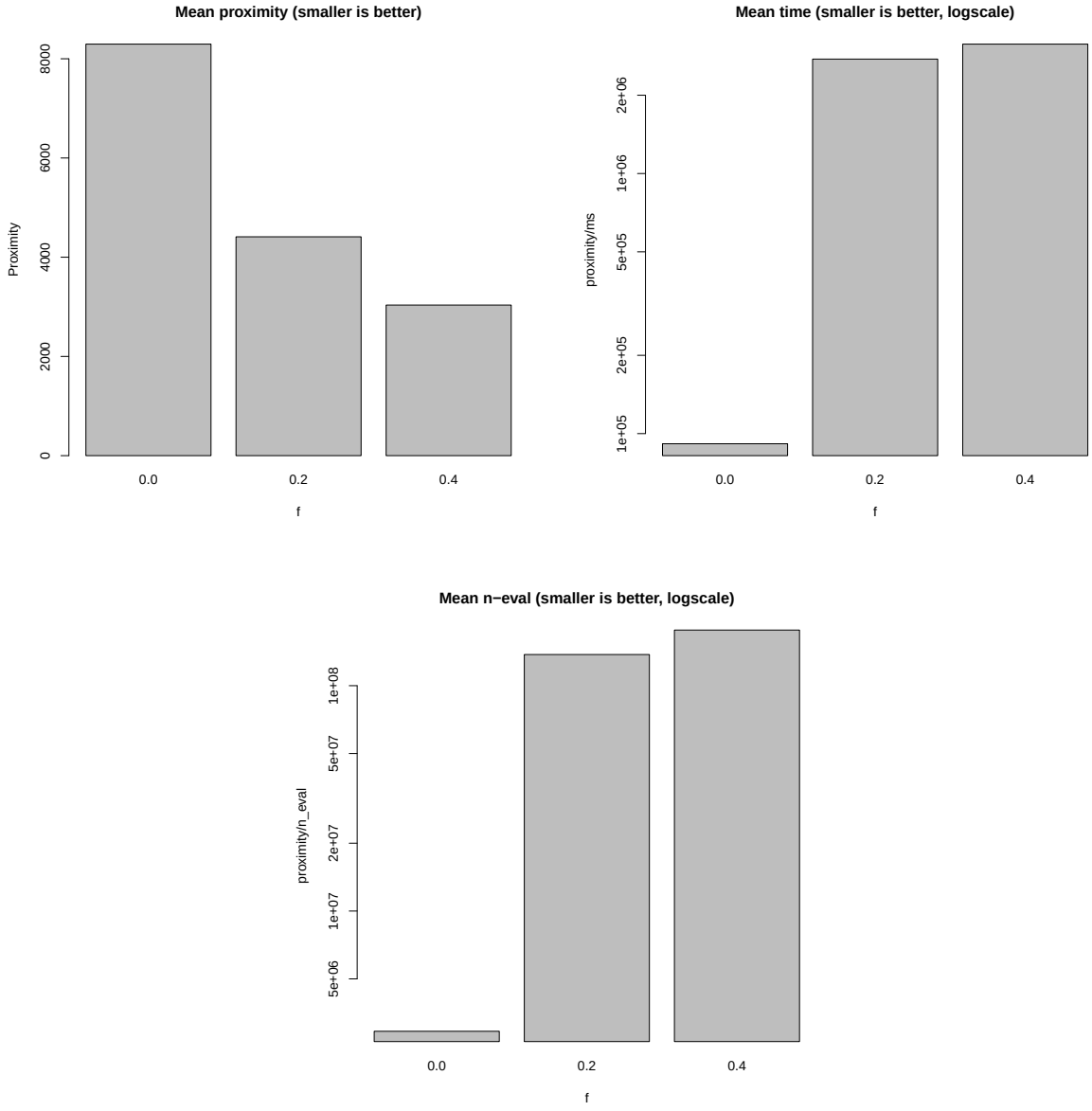


Figure 6.4: Parameter influence – the scaling factor f

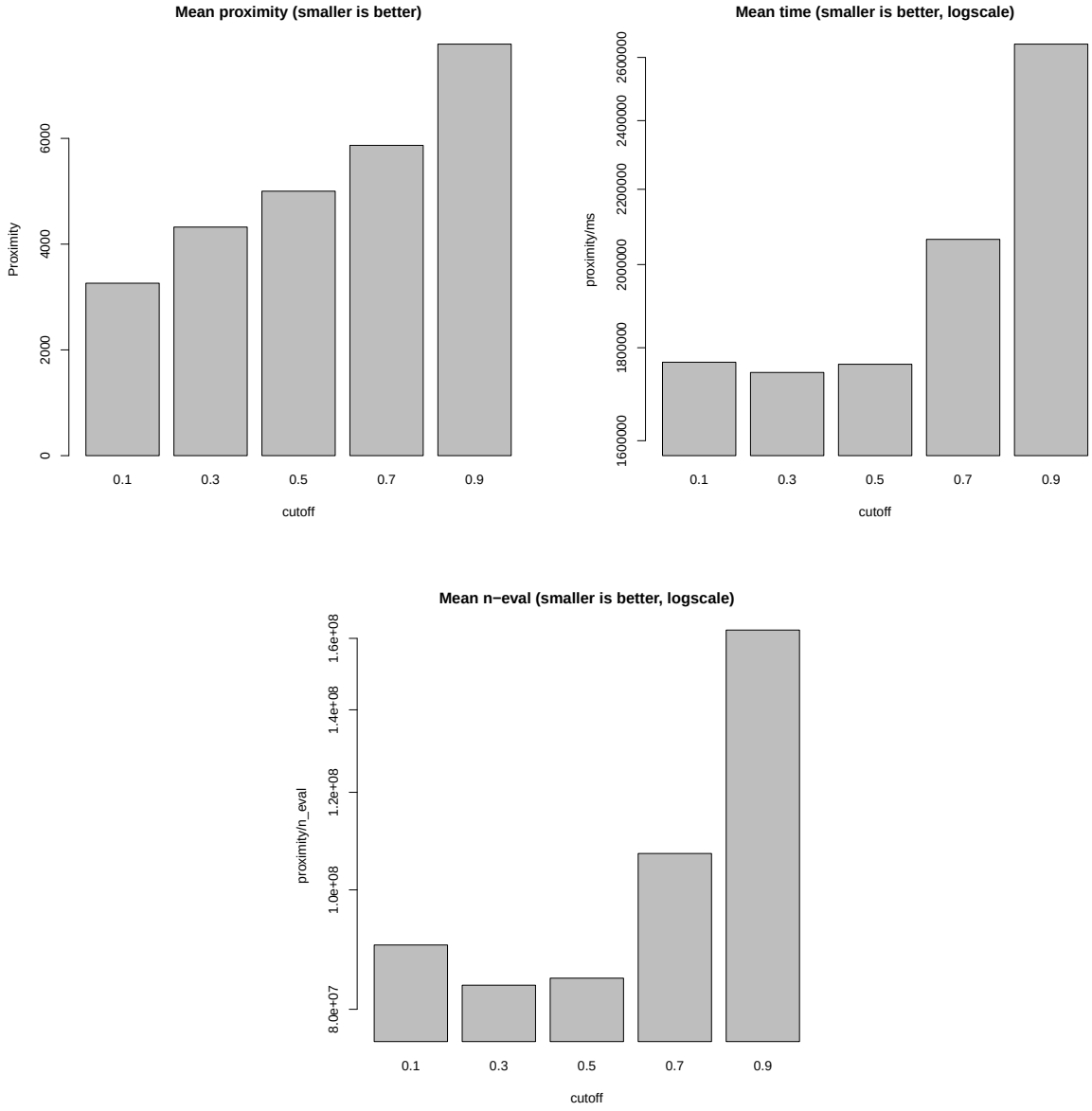


Figure 6.5: Parameter influence – cutoff

6.1.4 Random Optimization

For the given optimization problems and for the tested parameter values **Random Optimization** was among the worst algorithms tested. As figure 6.1 shows its mean performance is among the three worst, and the worst variants of **Random Optimization** behaved the worst globally. The algorithm also showed the lowest robustness.

Among the five best settings (6.3) the *stddev* was universally set to 3.0, and the *constraint strategy* used was either *trimming* or *dropping a genome*.

With the best set of parameters of the **Random Optimization** algorithm gave acceptable performance. On average, the constraint strategy used had no effect on convergence – the algorithm converged to a solution quicker (both real time and in terms of objective function evaluation counts)

when dropping the individual, the **bounce back** strategy was the second best in terms of speed.

The algorithm was influenced heavily by the set *mean* value – best performance was noted when $\mu = 0.0$.

Bigger values of the standard deviation parameter *stddev* were proportionally better in the range tested – both in terms of convergence and convergence speed.

μ	σ	constraint strategy	Proximity	Time [s]	Time [%]	n-eval
-0.1,	3.0	trim	1447.41	5.92	0.06	110601
0.1	3.0	trim	1447.46	6.35	0.06	115453
-0.01	3.0	trim	1447.94	5.92	0.06	112283
0.0	3.0	trim	1460.75	5.69	0.06	106811
-0.01	3.0	drop gene	1466.08	8.03	0.08	127311

Table 6.3: Best five settings (by proximity) for the **Random Optimization** algorithm. Time [s] denotes time in ms spent on optimizing, and Time [%] is the time spent relative to the worst runtime. n-eval is the number of evaluations of the fitness function.

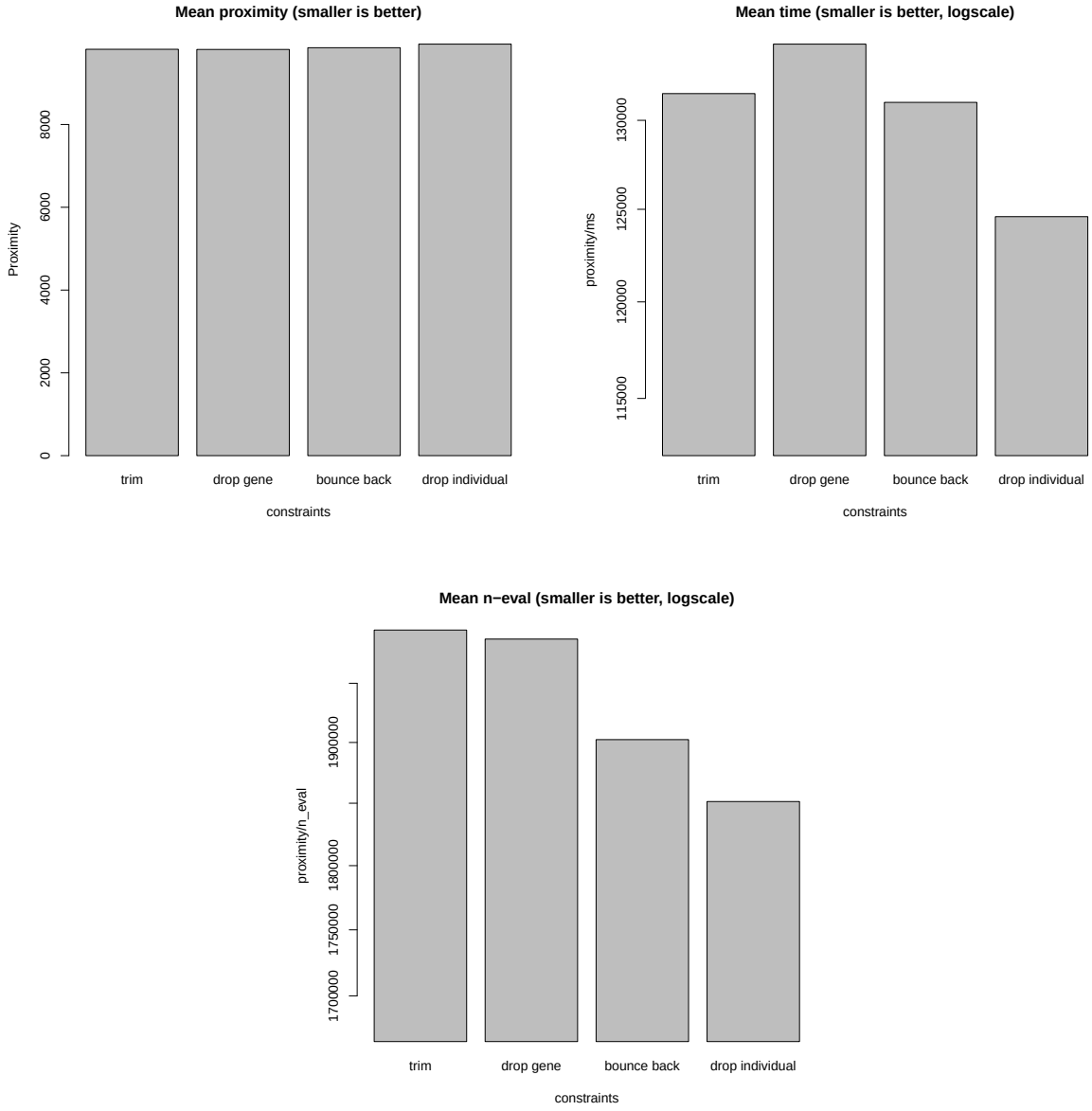


Figure 6.6: Parameter influence – constraint strategy

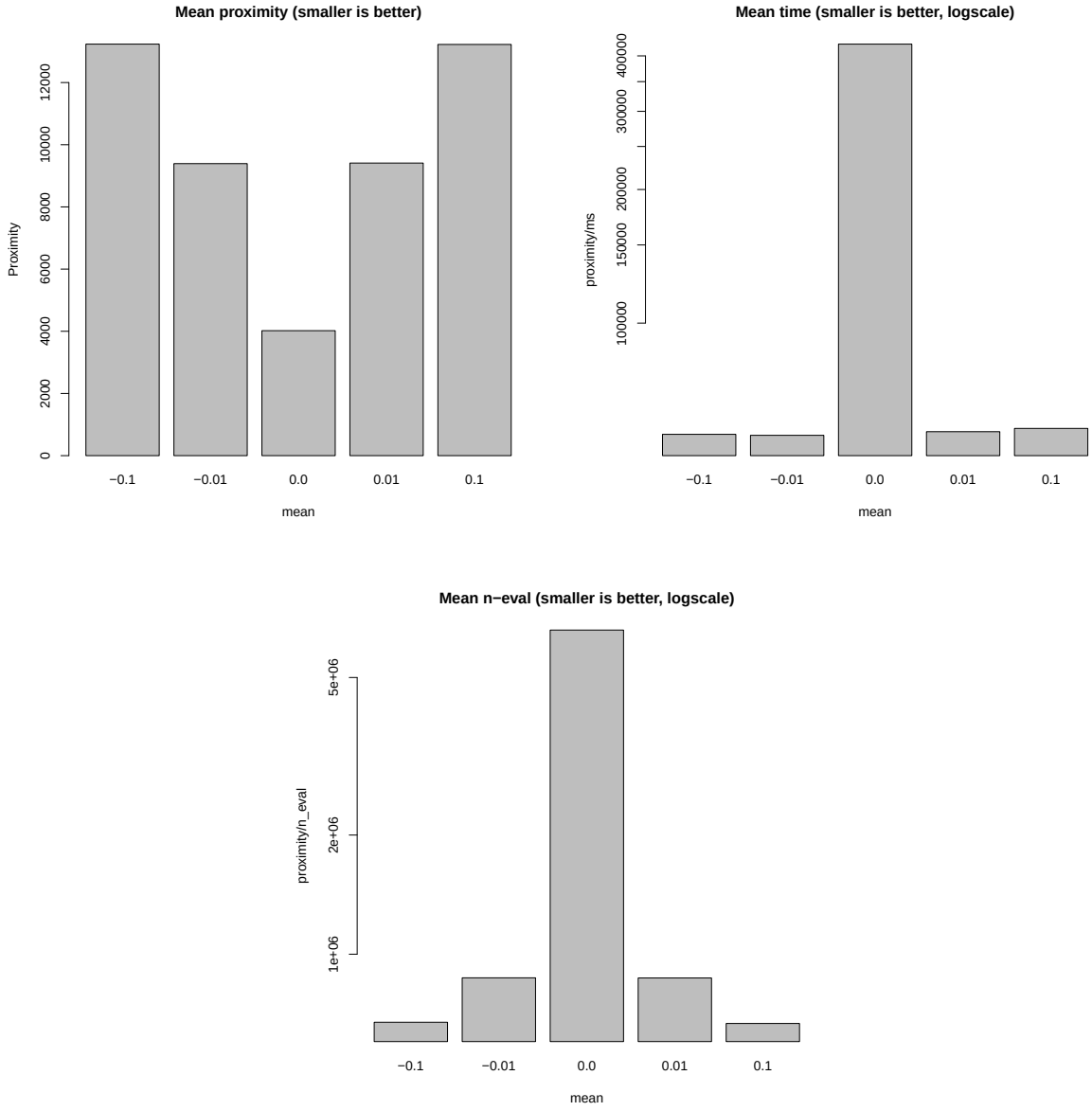


Figure 6.7: Parameter influence – mean

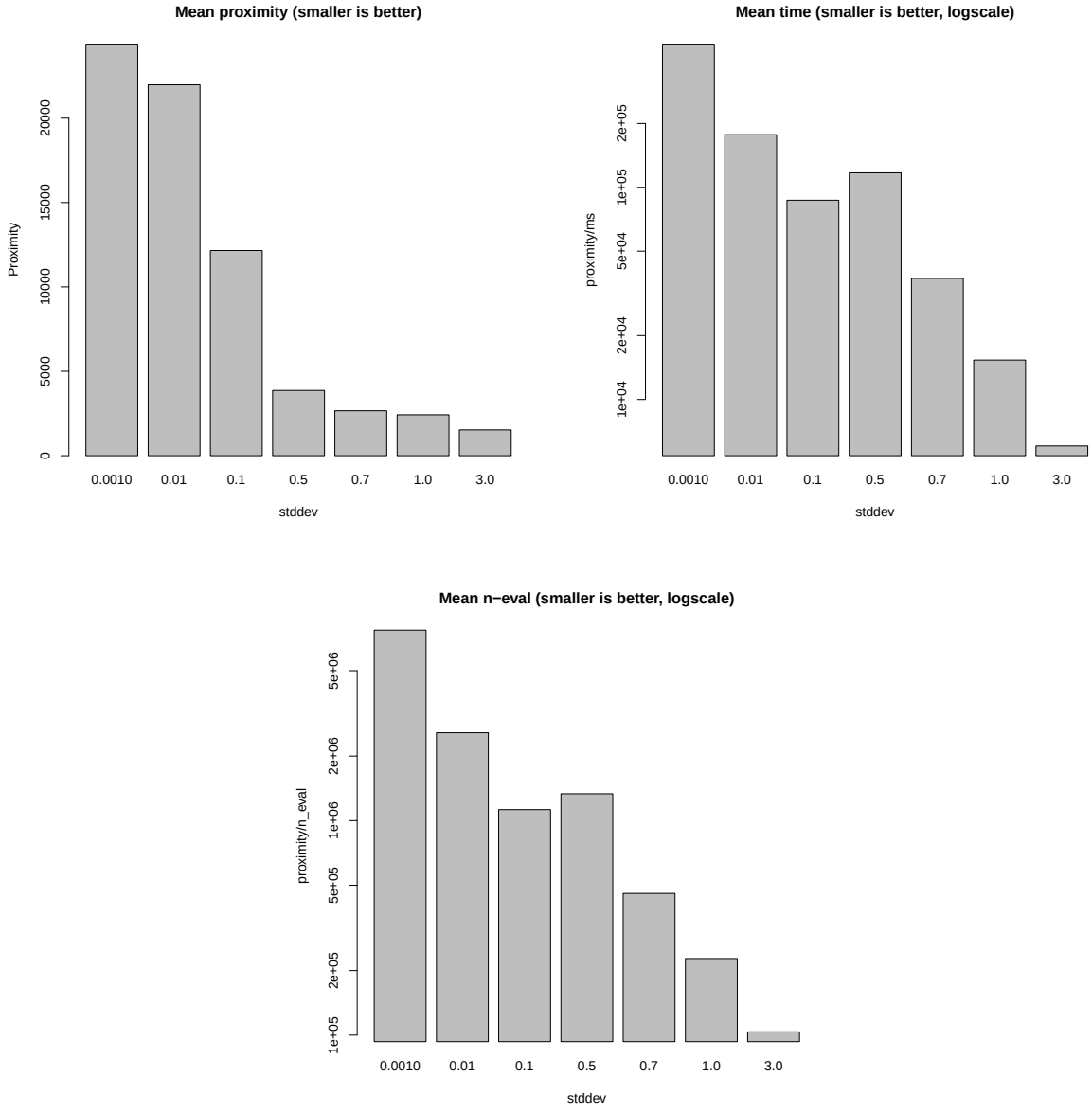


Figure 6.8: Parameter influence – stddev

6.1.5 Harmony Search

Among the tested algorithms **Harmony Search** ranks low both performance-wise and in terms of robustness. As table 6.4 shows the best settings for **HS** all had $P(chooseFromMemory) = 0.99$ a low *pitch adjustment probability* - either 0.1 or 0.3 and each had *adjustment strenght* $= 0.1$.

The best variants of the **Harmony Search** were only better then **Simulated Annealing** and **Random Optimization**; on average the performance of **HS** is on par with the average performances of **Particle Swarms** and **Random Optimization**

The algorithm behaves better with increasing population sizes but at a cost of convergence speed.

On average, higher values of the *P(chooseFromMemory)* parameter were proportionally better

in terms of convergence.

On average, lower values of the *P(adjustPitch)* parameter were proportionally better in terms of convergence, as is the case with the pitch adjustment strength *pitchAdjust* parameter.

algorithm_name	proximity	s.	time.percentage	n_eval
HS (chooseMemP = 0.99, adjustP = 0.1, adjustStr = 0.1) PS = 100	640.27	1778.35	18.08	6236538.00
HS (chooseMemP = 0.99, adjustP = 0.1, adjustStr = 0.1) PS = 75	671.16	1204.16	12.24	5039876.00
HS (chooseMemP = 0.99, adjustP = 0.1, adjustStr = 0.1) PS = 25	906.34	310.52	3.16	2141347.00
HS (chooseMemP = 0.99, adjustP = 0.1, adjustStr = 0.1) PS = 5	1256.81	78.89	0.80	753533.00
HS (chooseMemP = 0.99, adjustP = 0.3, adjustStr = 0.1) PS = 100	1580.20	1042.29	10.59	4196919.00

Table 6.4: Top 5 algorithms settings (by proximity).

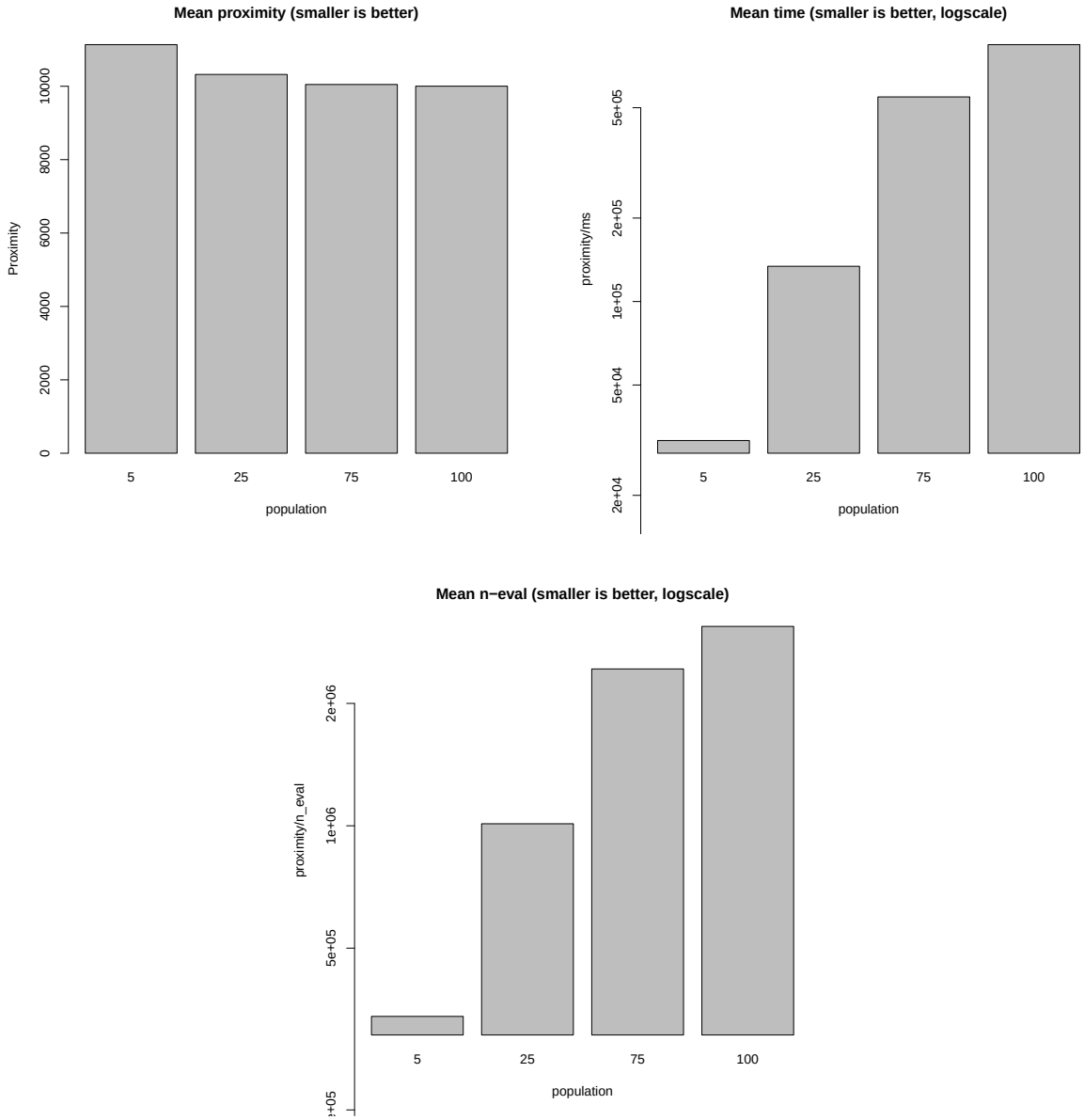


Figure 6.9: Parameter influence – population size

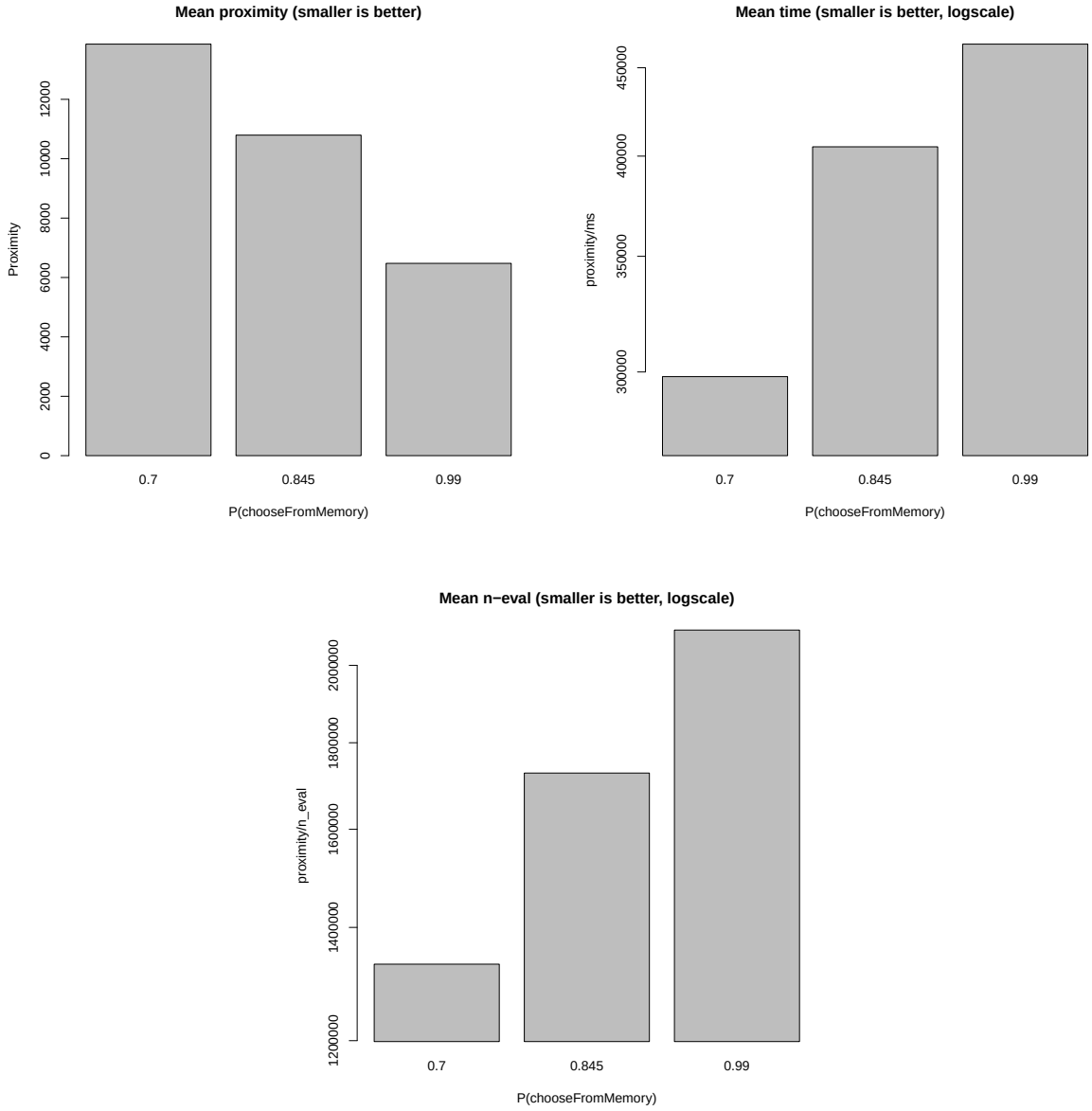


Figure 6.10: Parameter influence – $P(chooseFromMemory)$

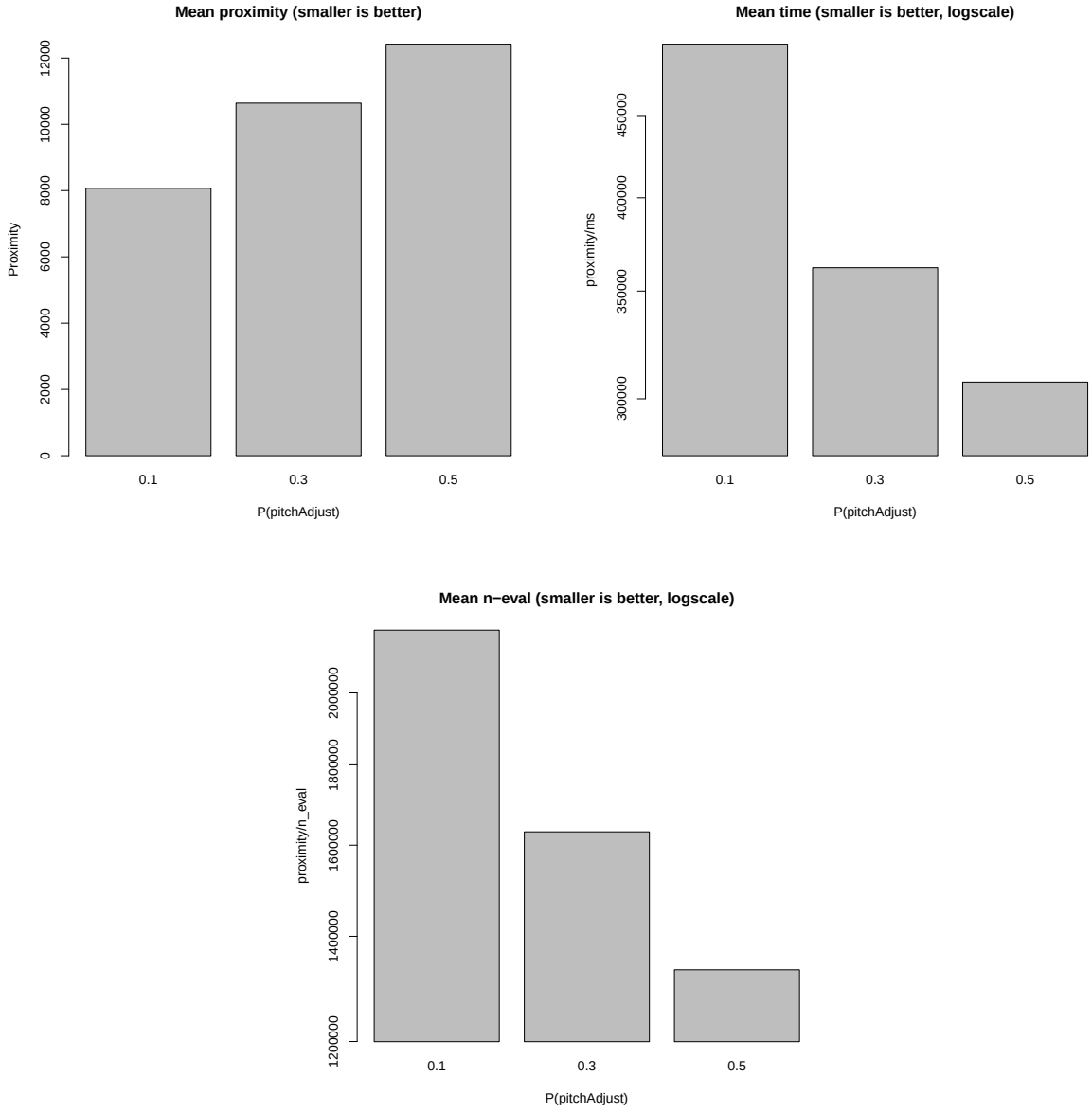
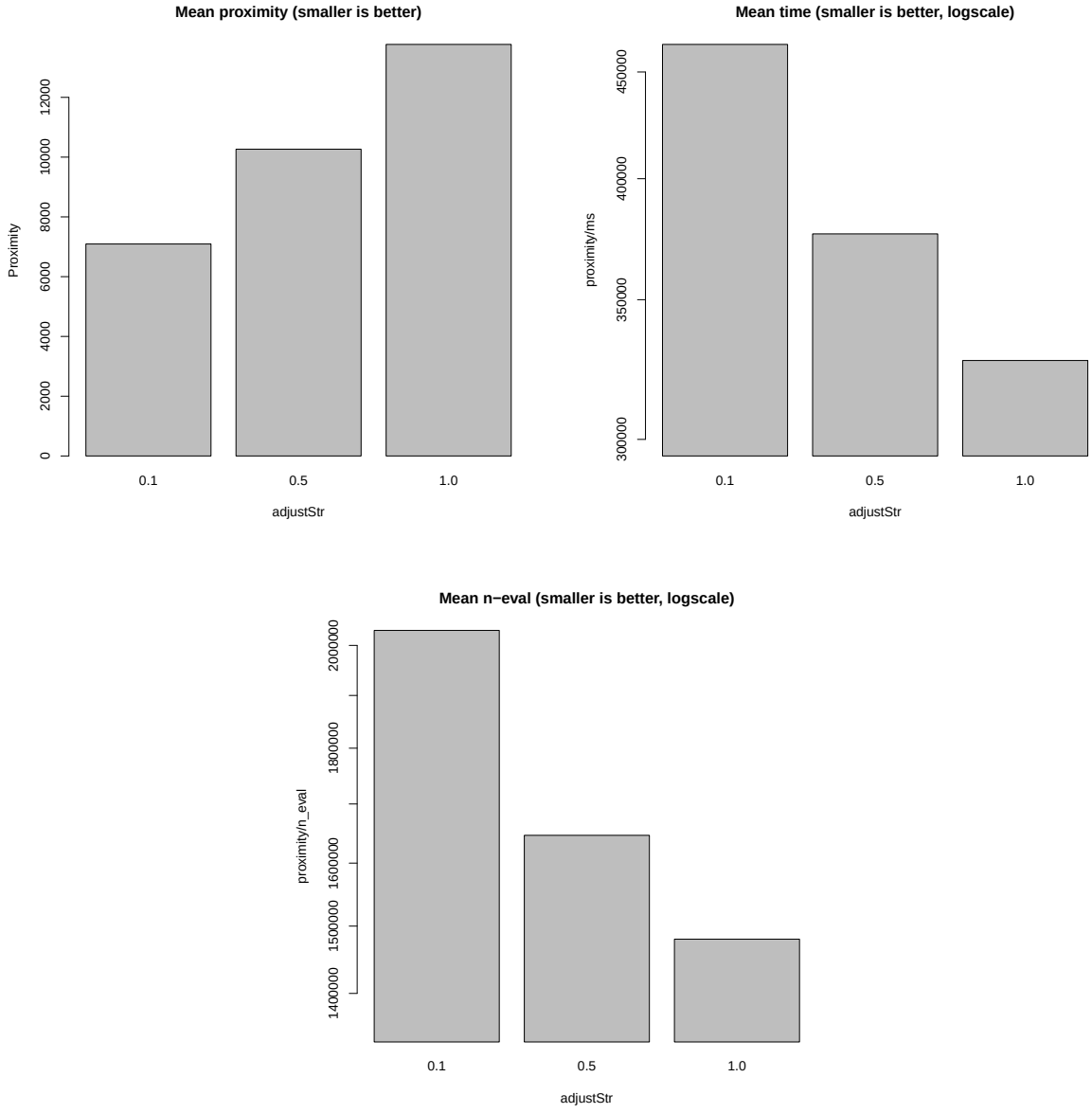


Figure 6.11: Parameter influence – $P(adjustPitch)$

Parameter influence – pitch adjustment strength δ



6.1.6 Particle Swarm Optimization

Globally the best versions of **Particle Swarms** ranked fourth best in terms of convergence, but the algorithm noted a very poor performance on average and in terms of robustness. The amount of parameters to tune to in **PSO** is staggering and finding better combinations may give better performance.

As table 6.5 shows the best settings for **PSO** all had the biggest *population size* = 60, *self learning rate* = *neighborhood learning rate* = 4.0, the *velocity weight* was also universally = 1.5.

In terms of convergence, bigger population sizes behave slightly better – but at a huge cost of convergence speed.

Among the tested values of the *velocity weight* parameter two values 1.5, and 3.0 are better by

a big margin from the rest when comparing convergence. Convergence speed between those two parameters is comparable with a faster convergence with *velocity weight* = 3.0.

The *self learning rate* parameter made very small difference in terms of convergence, although, convergence speed was faster for *self learning rate* = 2.0 in terms of real time. For *self learning rate* = 4.0 the algorithm made substantially less evaluations of the fitness function.

On average, the *neighborhood learning rate* also had almost no influence on convergence; for *neighborhood learning rate* = 4.0 the algorithm converged faster (both in terms of speed, and function evaluation count).

On average, the algorithm preformed substantially better with *global* **neighborhood type** in terms of convergence and real time convergence speed.

For the local neighborhood **neighborhood size** had almost no influence on convergence, convergence speed however was fastest for the 0.1 unnormalized neighborhood size.

type	ω	δ	v_{max}	C_1	C_2	neighborhood size	population size	Proximity	Time [s]	Time [%]	n-eval
local	1.5	1.0	1.0	4.0	4.0	0.1 NORM	60	406.49	595.19	6.05	6206750
global	0.0	1.0	1.0	4.0	4.0	N/A	60	409.24	64.86	0.66	2810075
local	1.5	1.0	1.0	4.0	4.0	0.5 NORM	60	411.63	575.08	5.85	6053185
local	1.5	1.0	1.0	4.0	4.0	0.5	60	424.08	574.93	5.84	6019809
local	1.5	1.0	1.0	4.0	4.0	0.1	60	430.71	582.53	5.92	6266220

Table 6.5: Best five settings (by proximity) for the **Particle Swarm Optimization** algorithm. Time [s] denotes time in ms spent on optimizing, and Time [%] is the time spent relative to the worst runtime. n-eval is the number of evaluations of the fitness function.

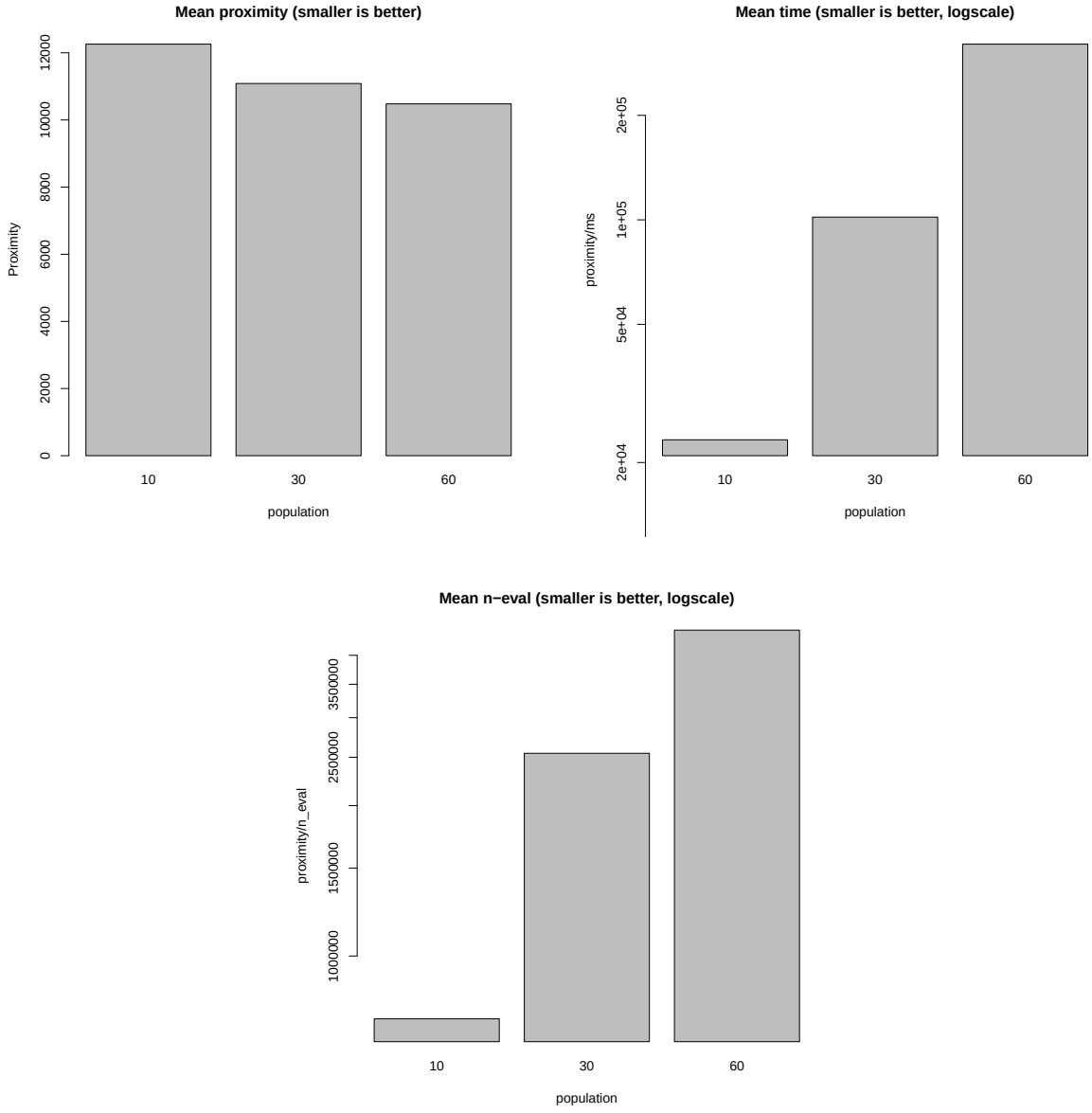


Figure 6.12: Parameter influence – population size

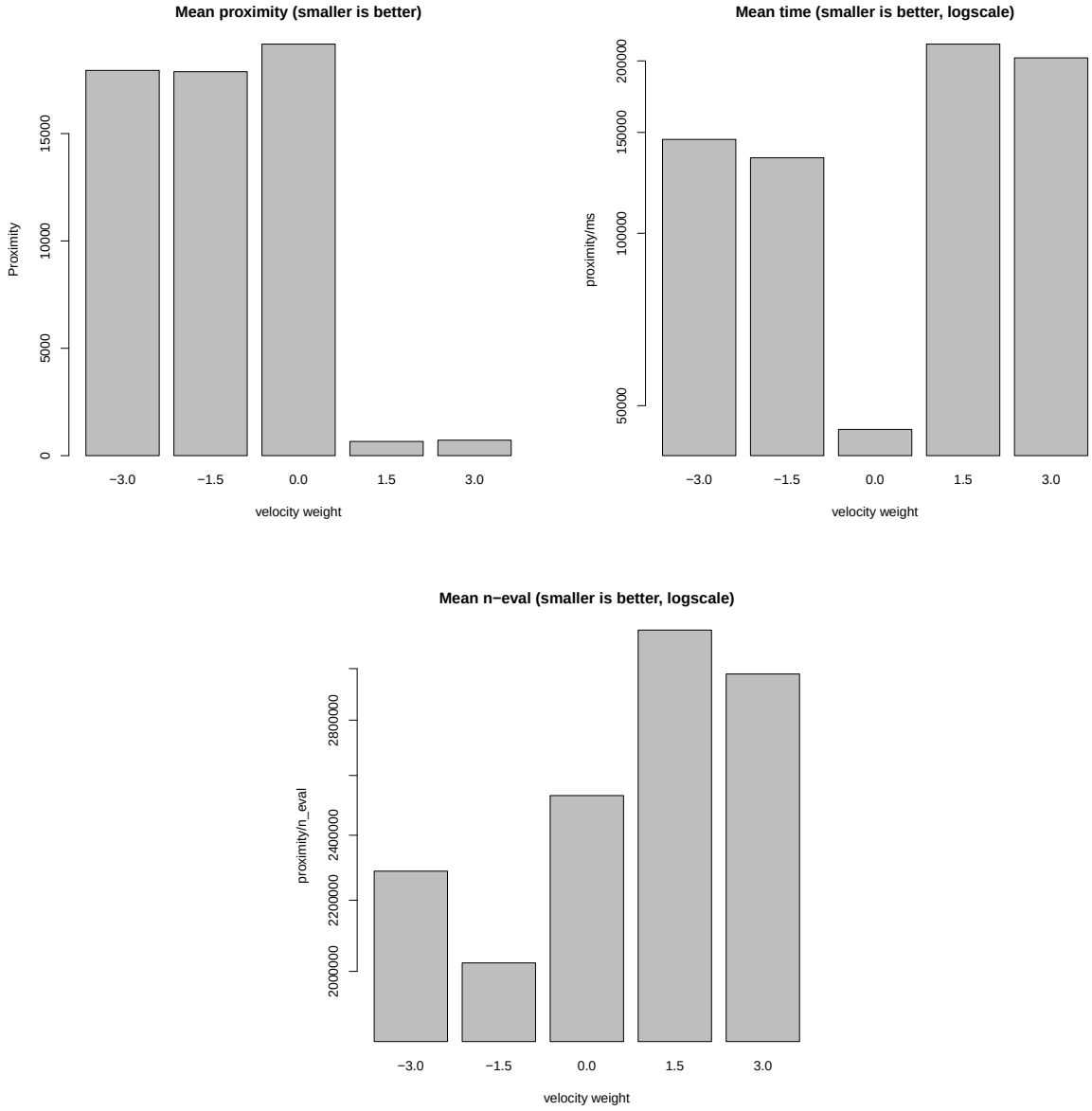


Figure 6.13: Parameter influence – velocity weight

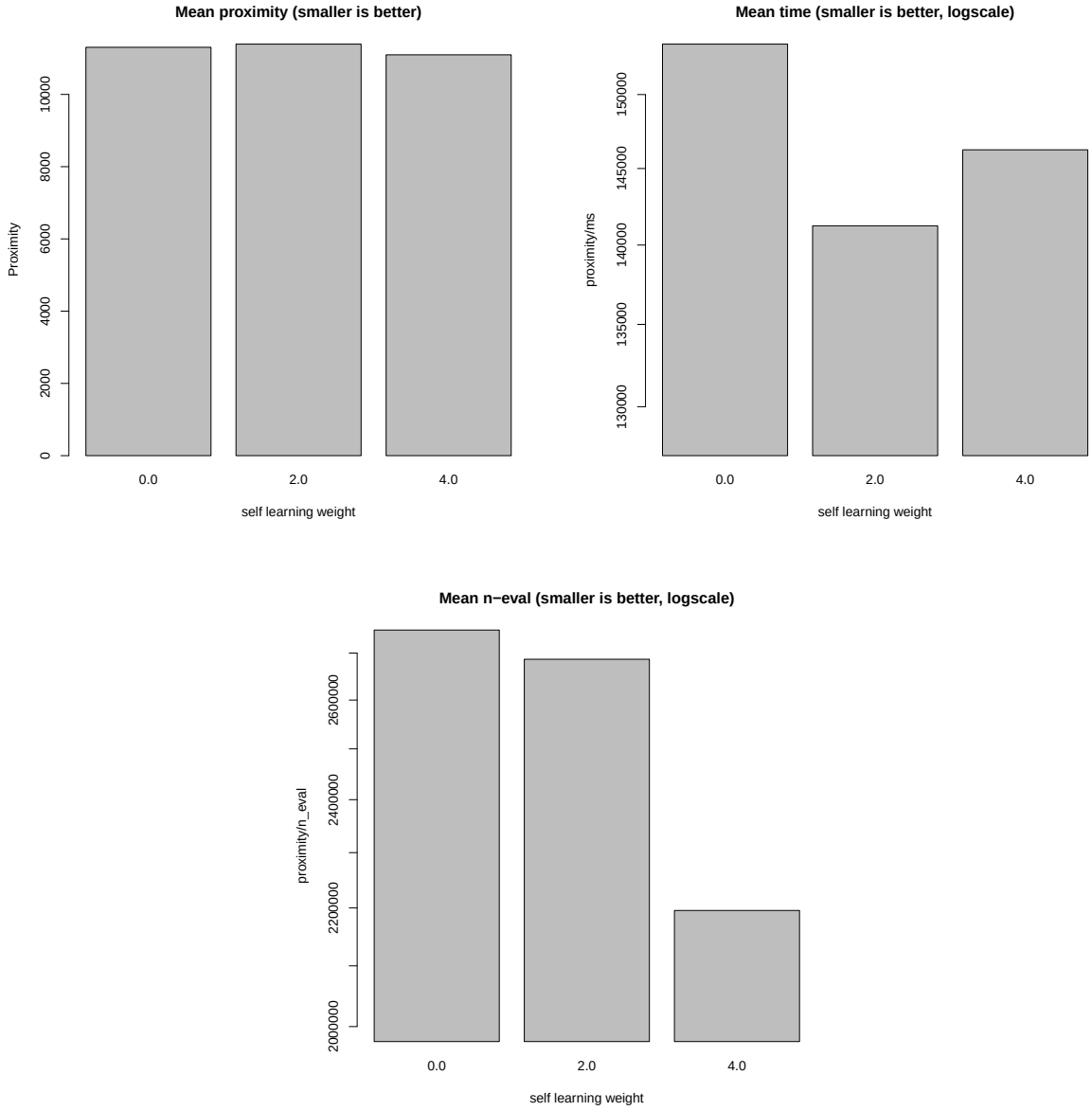


Figure 6.14: Parameter influence – self learning rate

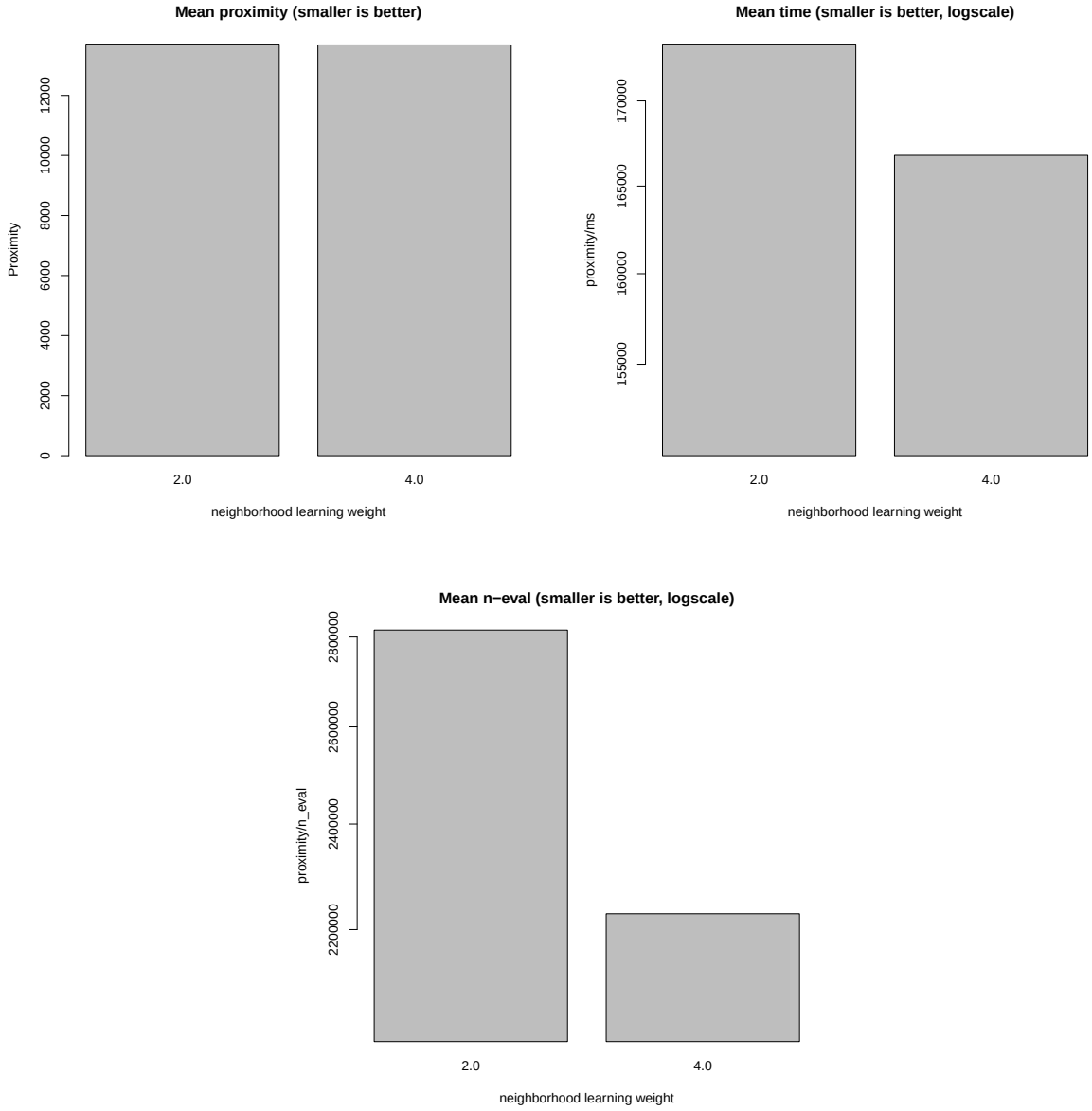


Figure 6.15: Parameter influence – neighborhood learning rate

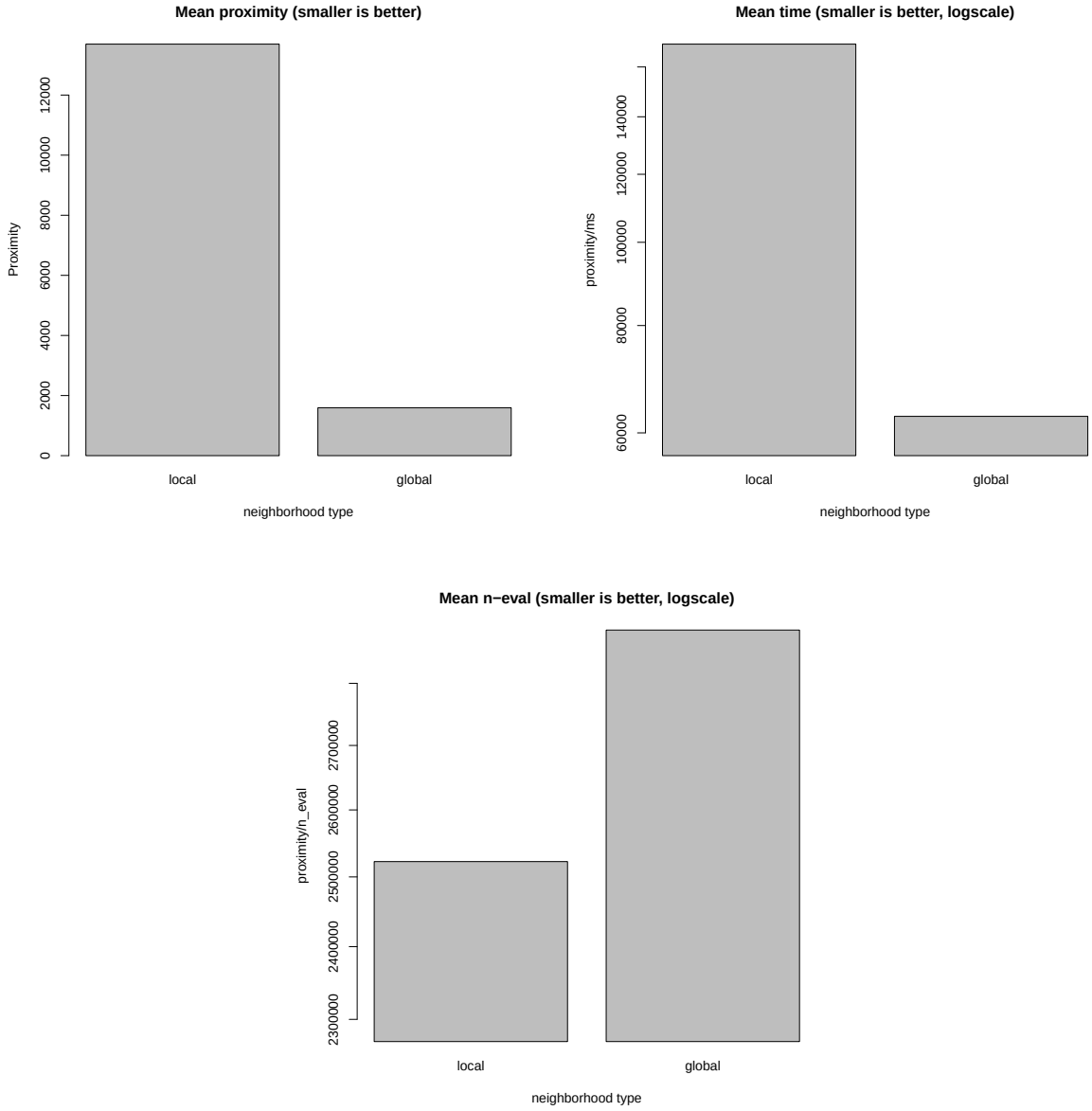


Figure 6.16: Parameter influence – neighborhood type

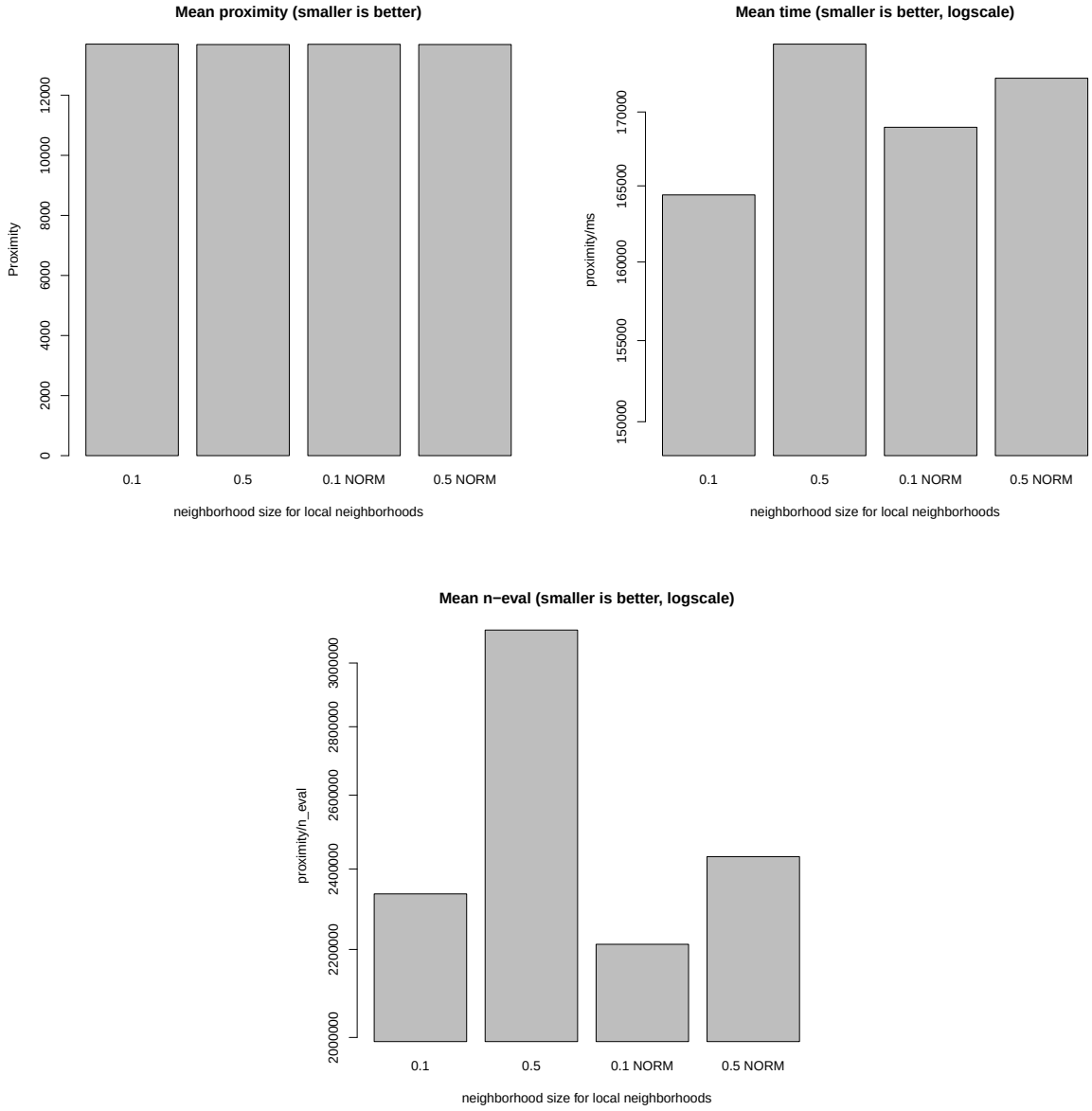


Figure 6.17: Parameter influence – neighborhood size (for local neighborhoods)

6.1.7 Cross Entropy Optimization

Cross entropy optimization turned out to have very high robustness, but a less then perfect performance. The best five settings (6.6) show mostly low importance sampling size (10% and 20%) and big population sizes - more then 500.

On average the convergence of the **Cross Entropy** method increased with the size of the population, although with diminishing returns.

The best value for the *importance sampling size* parameter was 0.5.

N_{sample}	population size	Proximity	Time [s]	Time [%]	n-eval
100 (10%)	1000	429.74	995.78	10.12	37714318
75 (10%)	750	456.66	737.72	7.50	28922034
50 (10%)	500	536.78	502.20	5.10	19439294
200 (20%)	1000	632.96	784.56	7.97	34206016
150 (10%)	750	661.56	591.07	6.01	25073451

Table 6.6: Best five settings (by proximity) for the **Cross Entropy Optimization** algorithm. Time [s] denotes time in ms spent on optimizing, and Time [%] is the time spent relative to the worst runtime. n-eval is the number of evaluations of the fitness function.

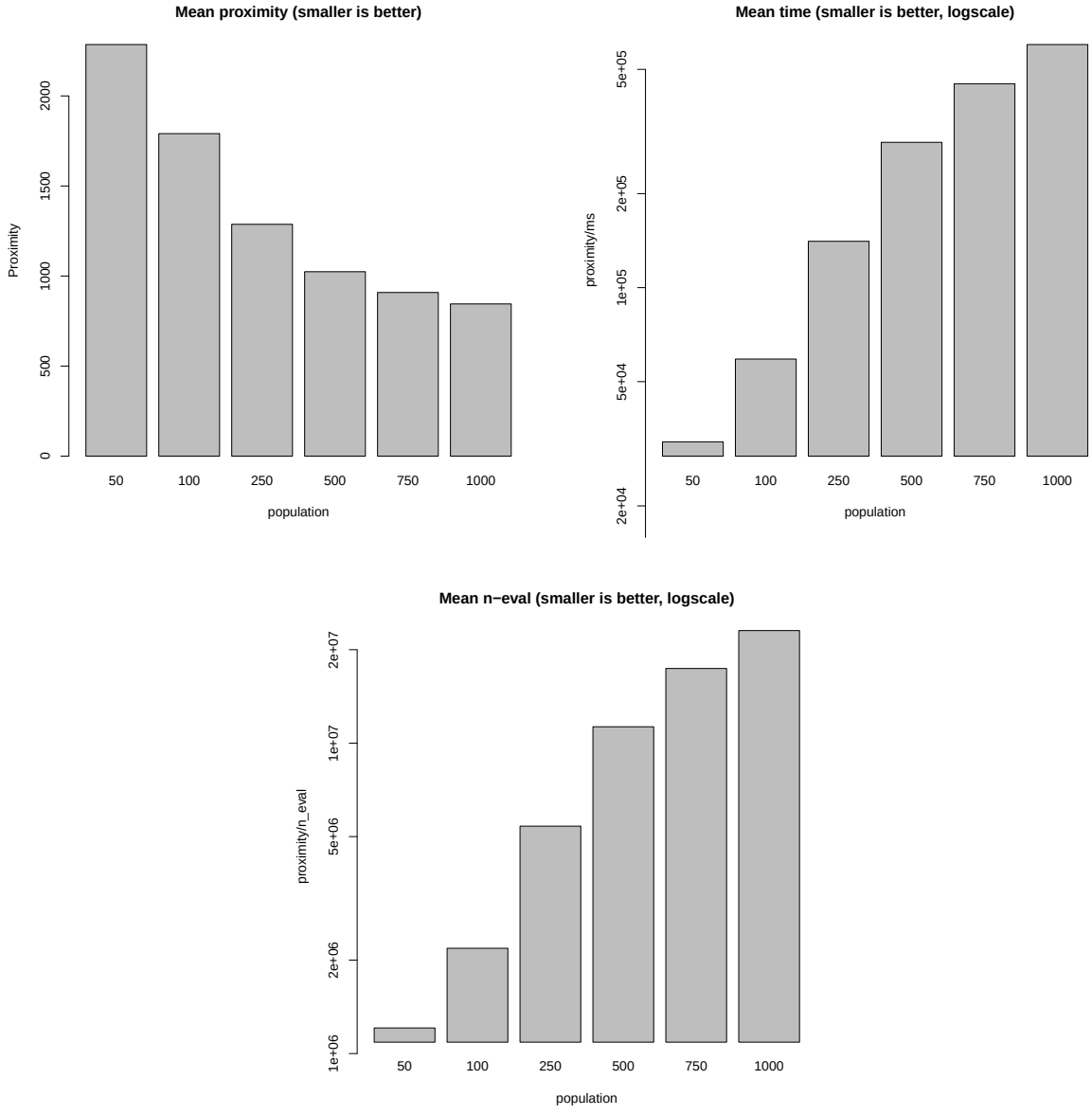


Figure 6.18: Parameter influence – population size

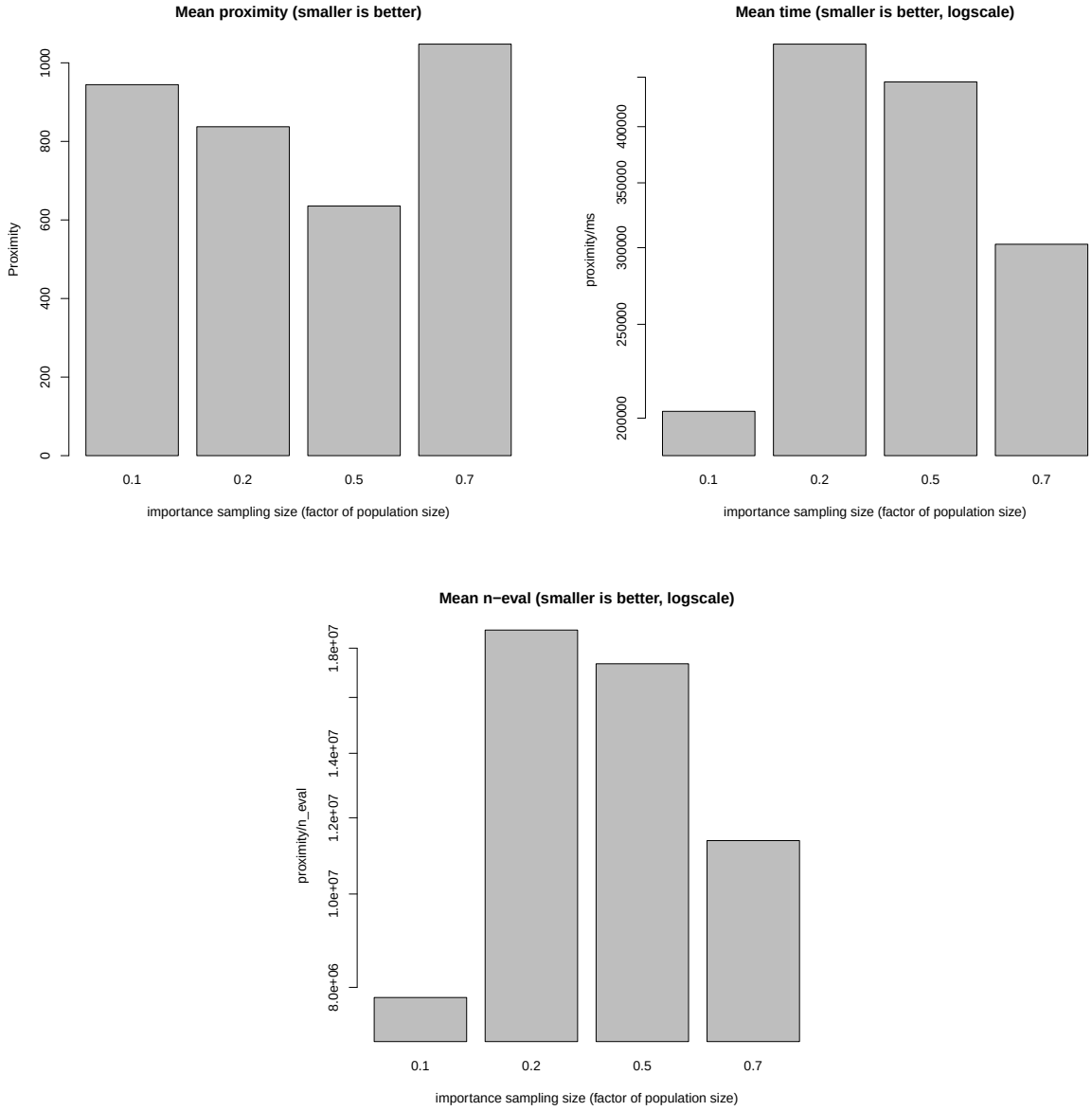


Figure 6.19: Parameter influence – importance sampling size

6.1.8 Genetic algorithms

Globally, **Genetic Algorithms** ranked second, **GA** proved to be robust, had very good mean performance (second to **Taboo Search**) and with a mediocre worst-case performance. As table 6.7 shows the best parameter settings used the biggest possible *population size* = 100, a hight *mutation size* = 1.0 and a low *mutation rate* = 0.1. When using *uniform selection* 6 parents were preferred.

On average, **Genetic Algorithms** improved as the population size increased, though with diminishing returns. On average convergence was better with *elitism size* = 2, and with a higher population size. **GA**s behaved better with a lower *mutation rate* and *mutation size*.

The crossover method had almost no influence on convergence, although convergence speed was faster in terms of real time spent for *1-Point* crossover; in terms of number objective function

invocations the *Uniform* crossover was substantially faster.

For uniform crossover the traditional – *parent count* = 2 worked slightly better but with a slower convergence.

elitism size	selection type	parent count	tournament size	crossover type	crossover rate	mutation size	mutation rate	population size	Proximity	Time [s]	Time [%]	n-eval
0	Tournament	2	6	Uniform	1.0	1.0	0.1	100	143.87	1345.87	13.68	93640410.00
2	Tournament	2	6	1-Point	1.0	1.0	0.1	100	151.26	970.70	9.87	97679415.00
2	Tournament	2	2	1-Point	1.0	1.0	0.1	100	153.64	992.40	10.09	98365619.00
2	Tournament	2	2	Uniform	1.0	1.0	0.1	100	154.38	1390.46	14.13	98081846.00
0	Tournament	2	6	1-Point	1.0	1.0	0.1	100	155.28	942.75	9.58	94673075.00

Table 6.7: Top 5 algorithms settings (by proximity).

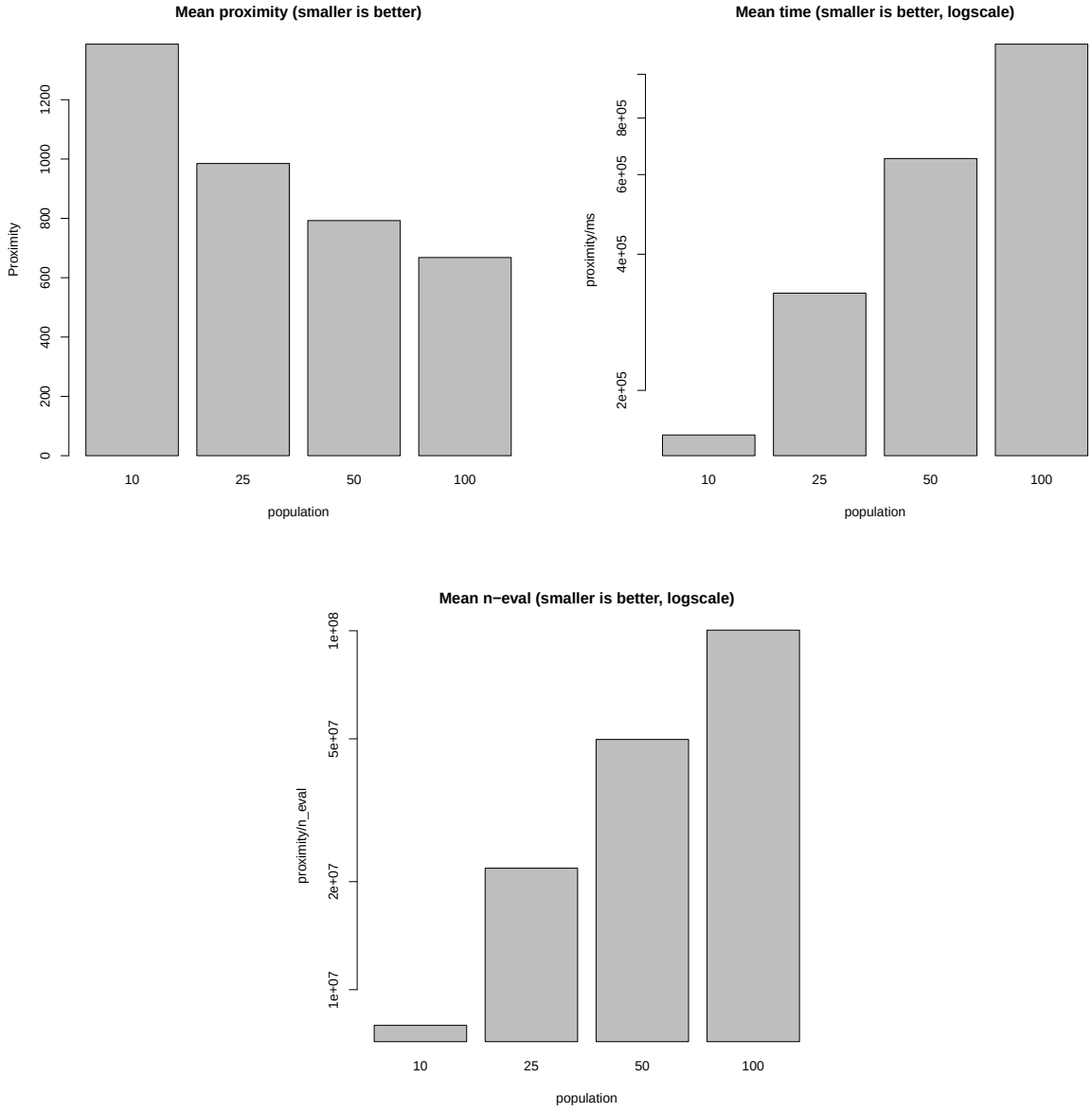


Figure 6.20: Parameter influence – population size

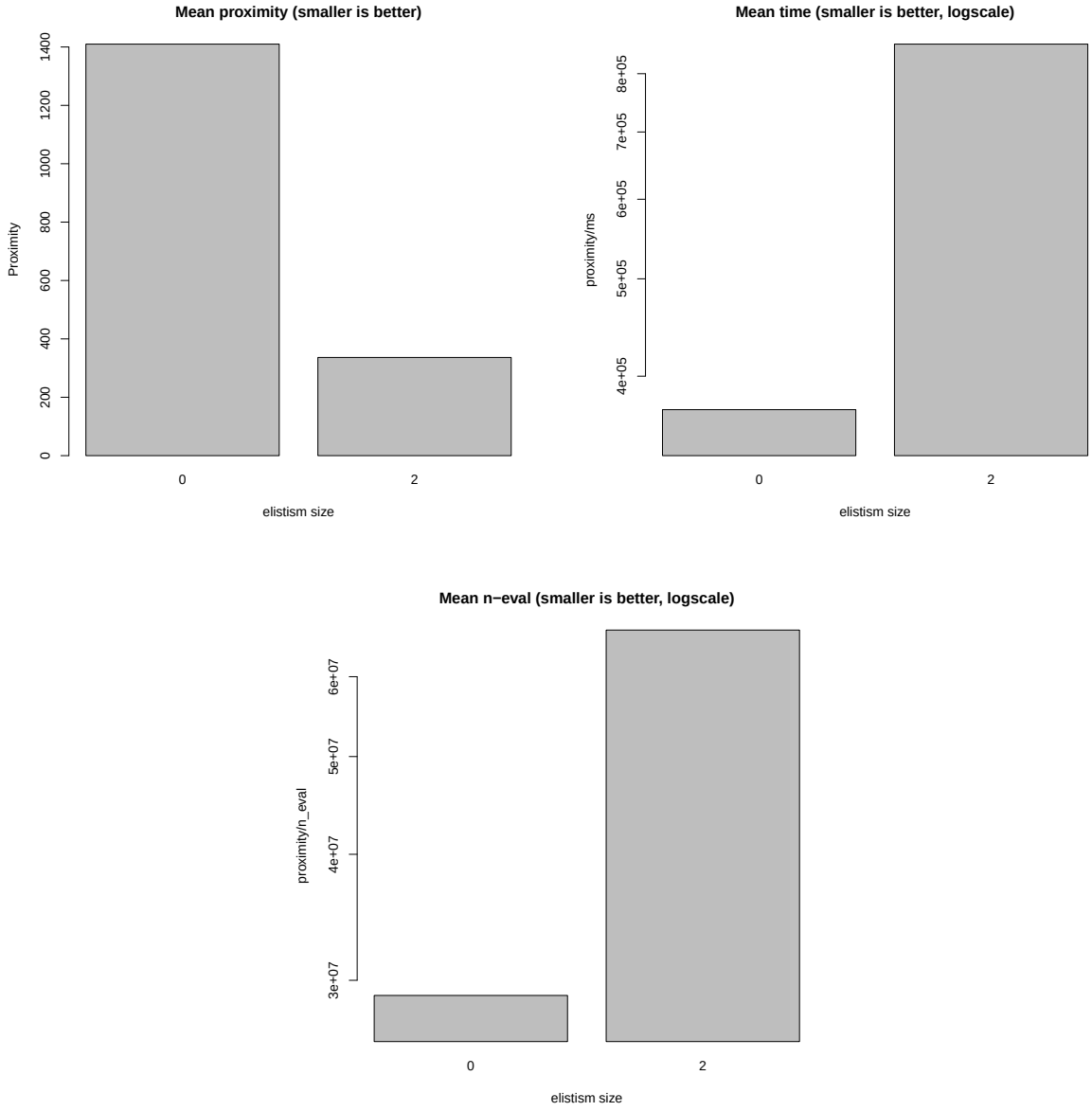


Figure 6.21: Parameter influence – elitism

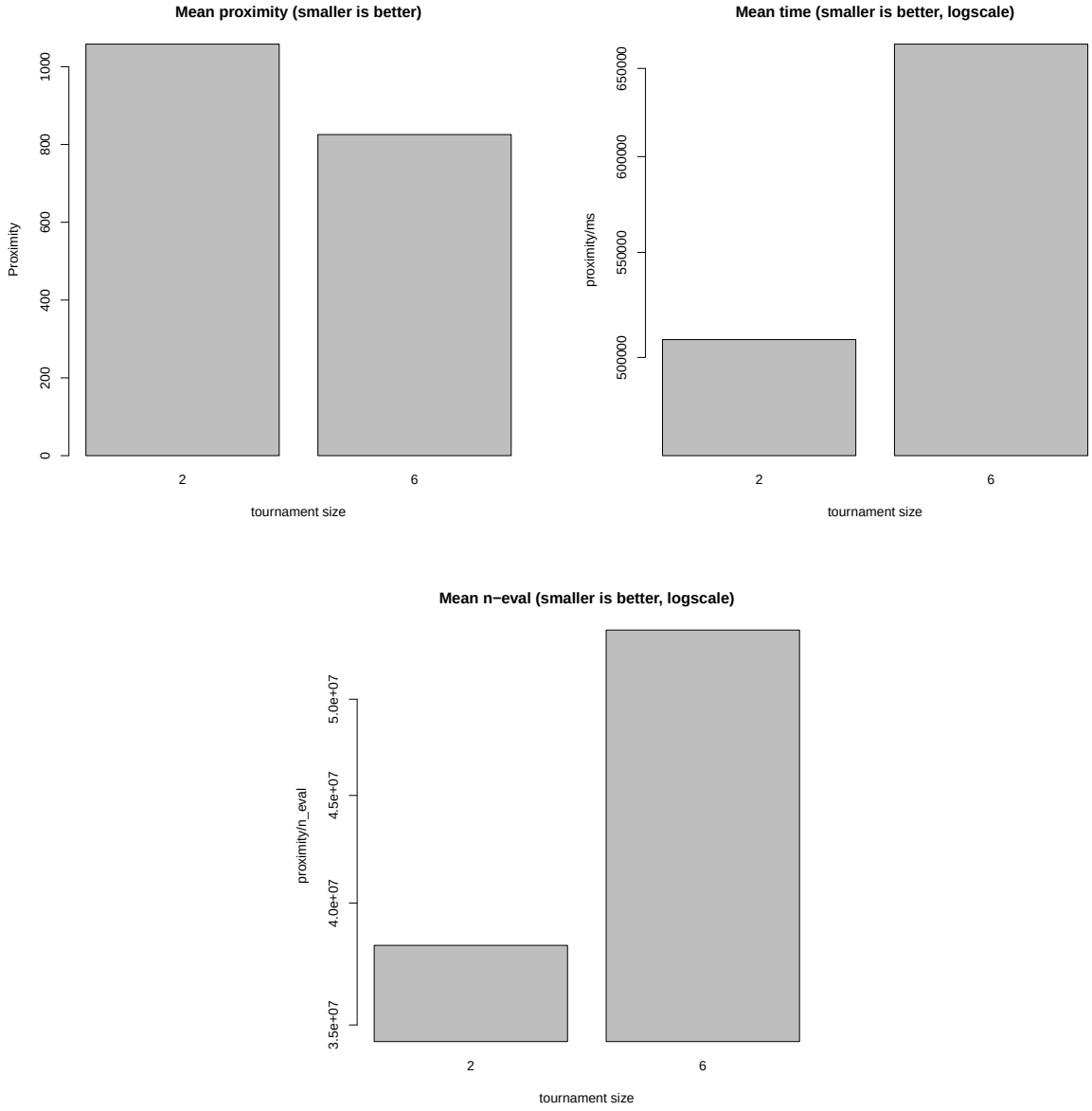


Figure 6.22: Parameter influence – tournament size

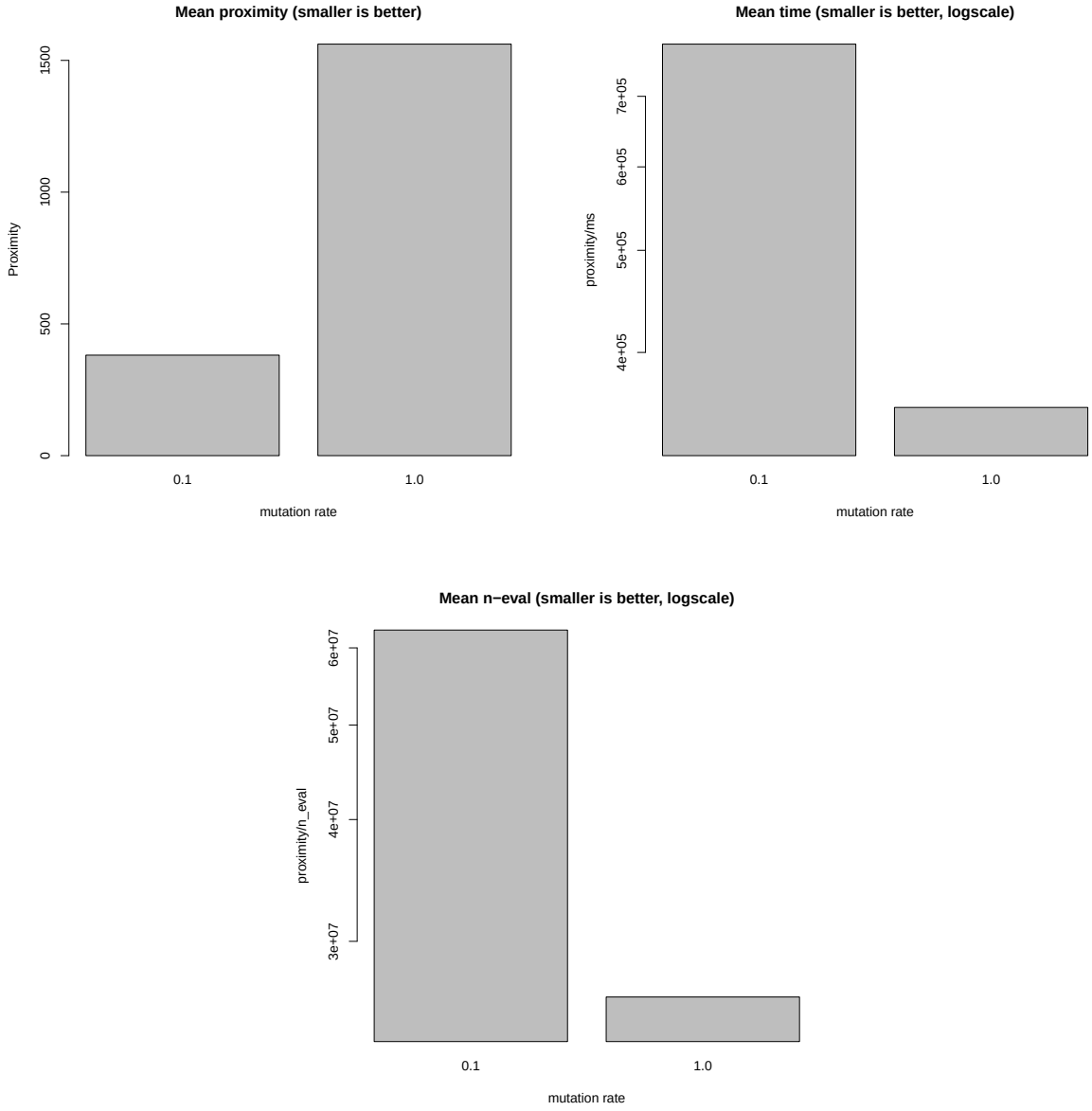


Figure 6.23: Parameter influence – mutation rate

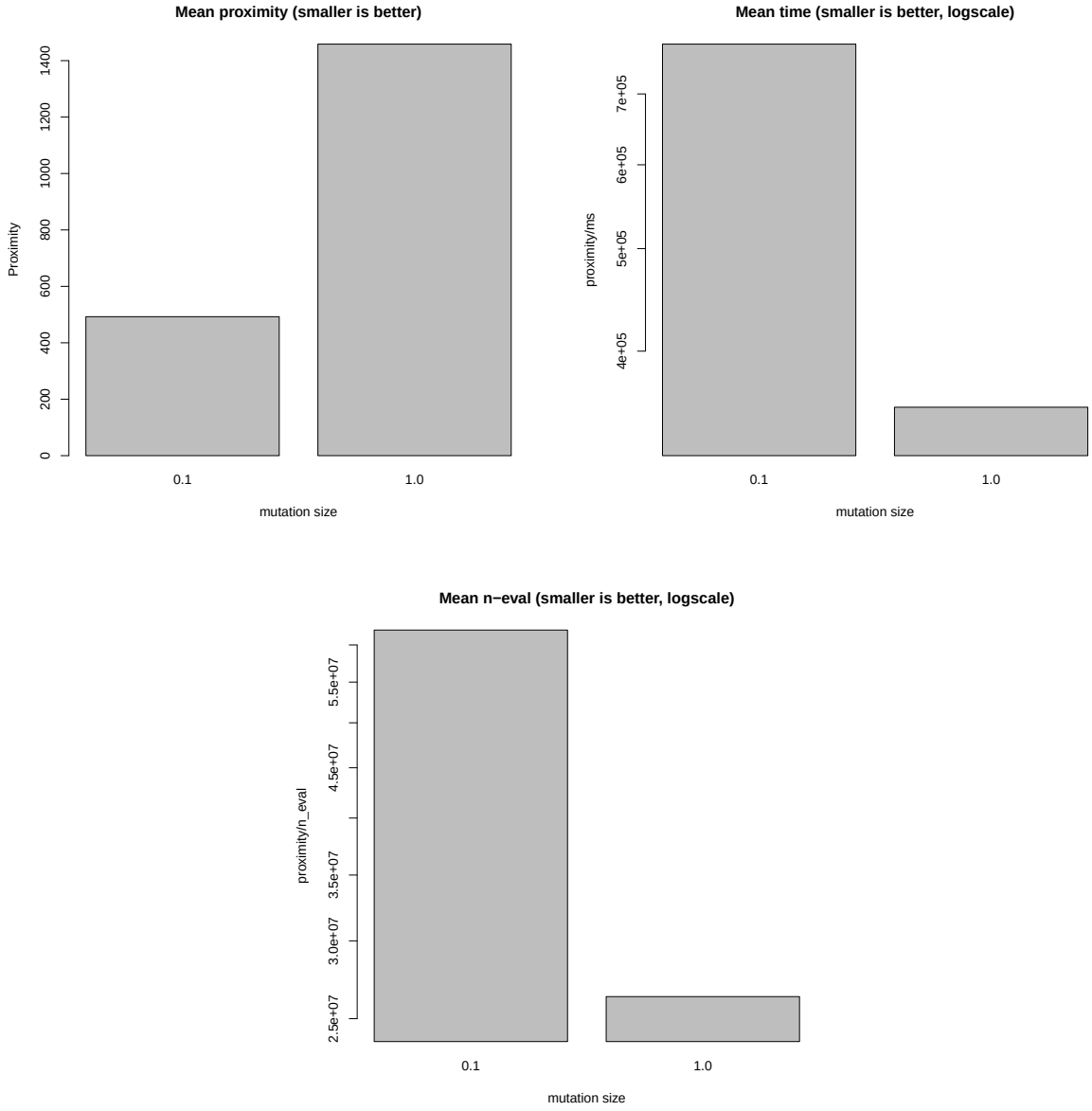


Figure 6.24: Parameter influence – mutation rate

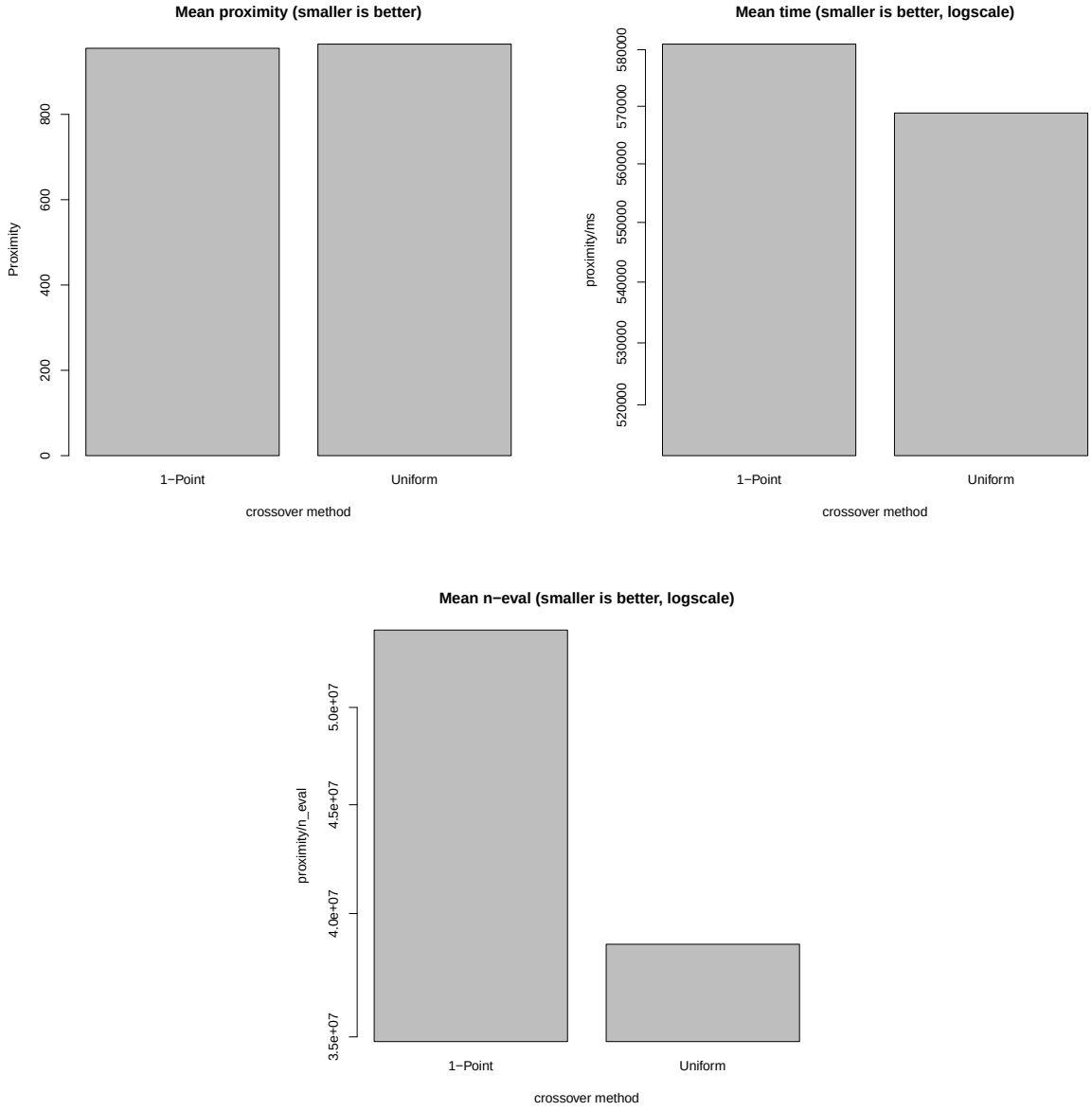


Figure 6.25: Parameter influence – crossover method

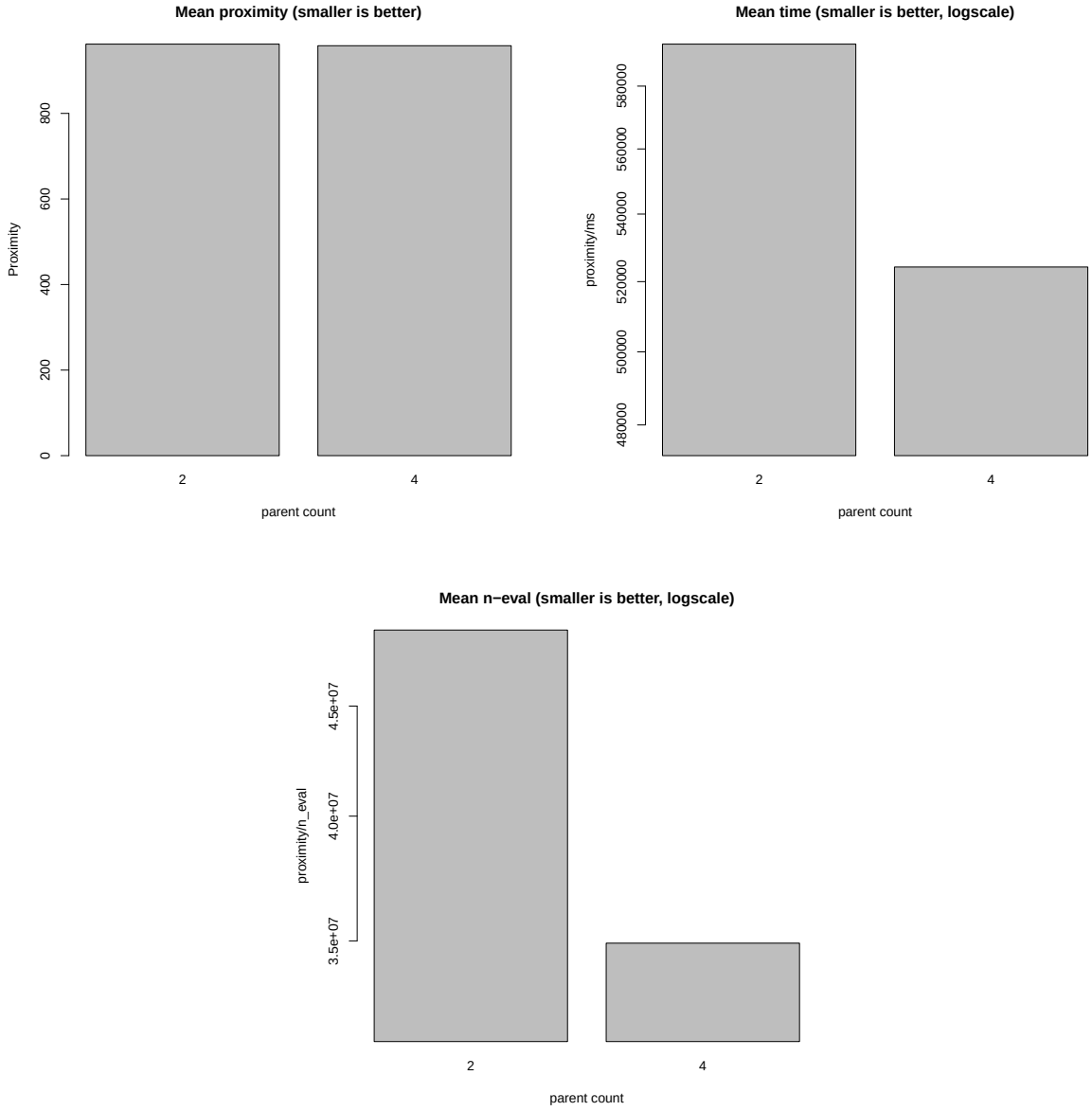


Figure 6.26: Parameter influence – parent count (with uniform crossover)

6.1.9 Simulated annealing

Simulated Annealing showed very stable performance with worst case convergence close to best case convergence - for the given problems it was also fourth best in terms of robustness. Nevertheless the best case performance was surprisingly low - second worst. In the five best algorithm settings (table 6.8) no cooling schedule seemed to dominate, but the *mutation step* was universally equal to 0.2.

On average the best performance was for **mutation step** $= 0.2$, the fastest rate of convergence was observed for **mutation step** $= 0.4$.

From the two cooling schedules both gave equal performance with the *Geometric* cooling schedule having a slightly faster real time speed of convergence.

The starting time T_{start} also had almost no influence on the convergence with a faster convergence rate for $T_{start} = 5000.0$

mutation size	cooling schedule	Proximity	Time [s]	Time [%]	n-eval
0.2	Geometric(Tstart = 20000.0, a = 0.999)	1245.23	9.65	0.10	379681.00
0.2	VariantI (Tstart = 5000.0, m = 5, eps = 0.0010)	1247.30	9.58	0.10	376341.00
0.2	VariantI (Tstart = 10000.0, m = 100, eps = 0.0010)	1257.54	9.47	0.10	369863.00
0.2	Geometric(Tstart = 10000.0, a = 0.99999)	1258.24	9.45	0.10	373114.00
0.2	VariantI (Tstart = 2000.0, m = 5, eps = 0.0010)	1263.64	9.70	0.10	380673.00

Table 6.8: Best five settings (by proximity) for the **Simulated Annealing** algorithm. Time [s] denotes time in ms spent on optimizing, and Time [%] is the time spent relative to the worst runtime. n-eval is the number of evaluations of the fitness function.

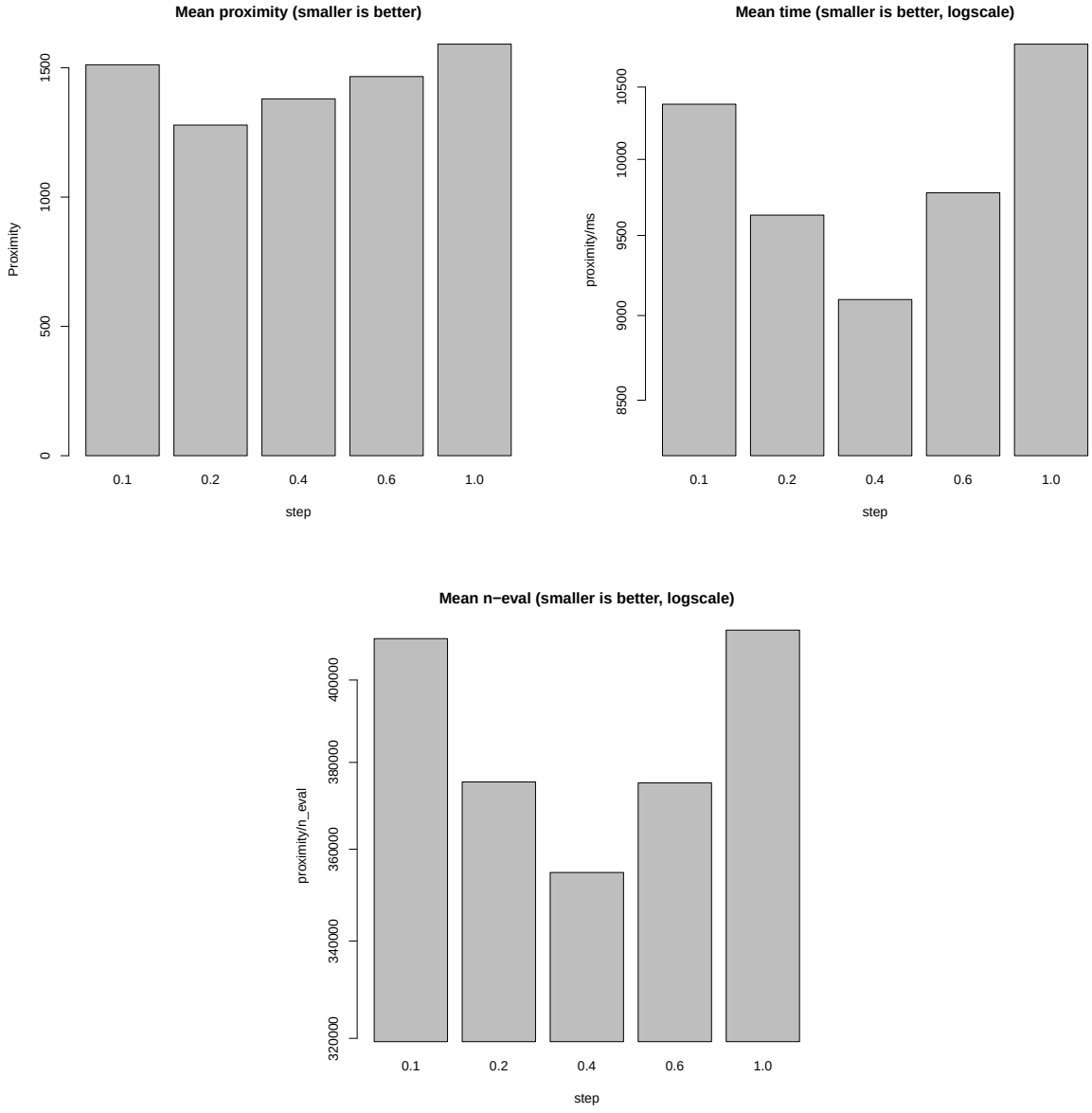


Figure 6.27: Parameter influence – mutation step size

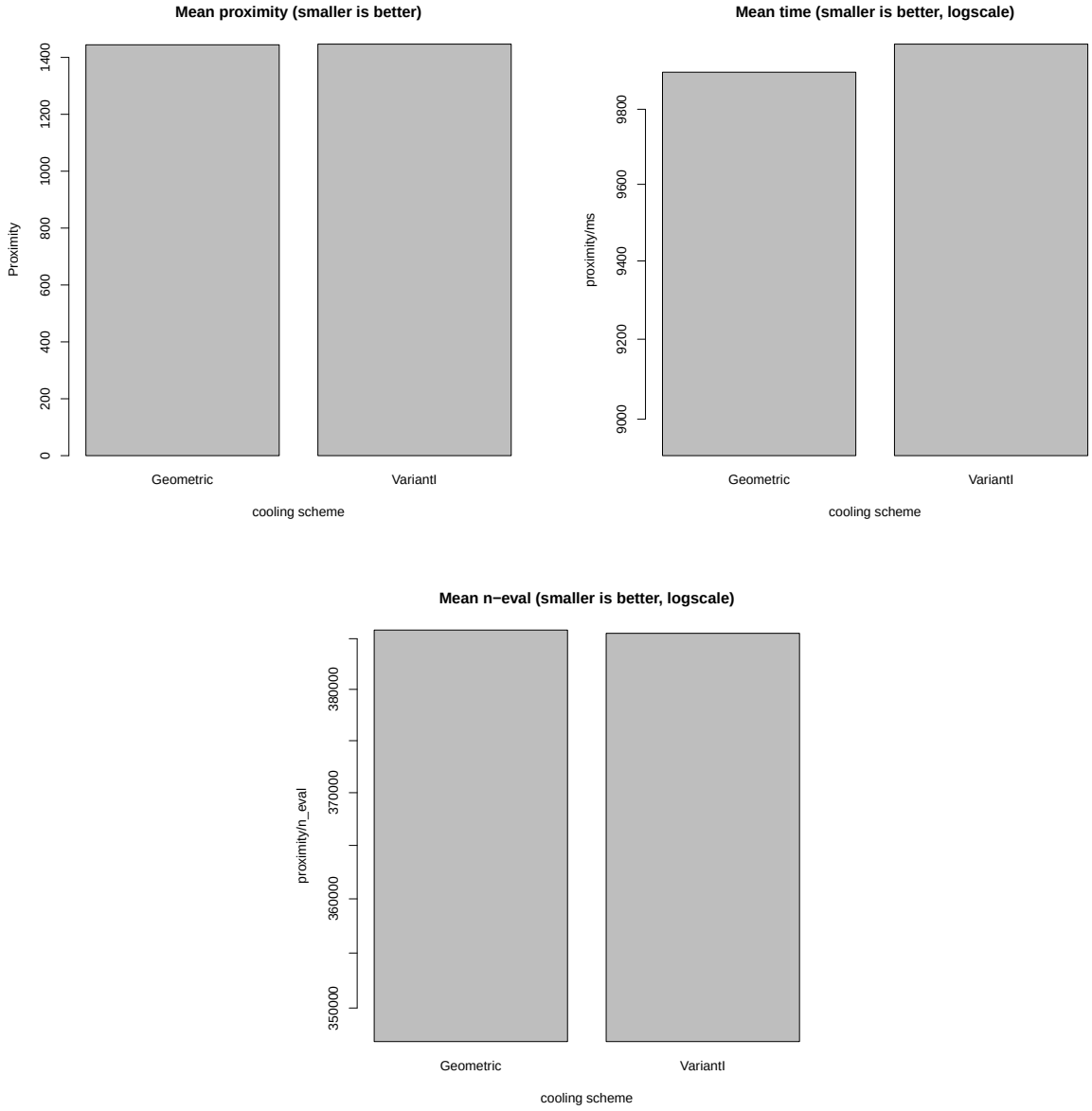


Figure 6.28: Parameter influence – cooling schedule

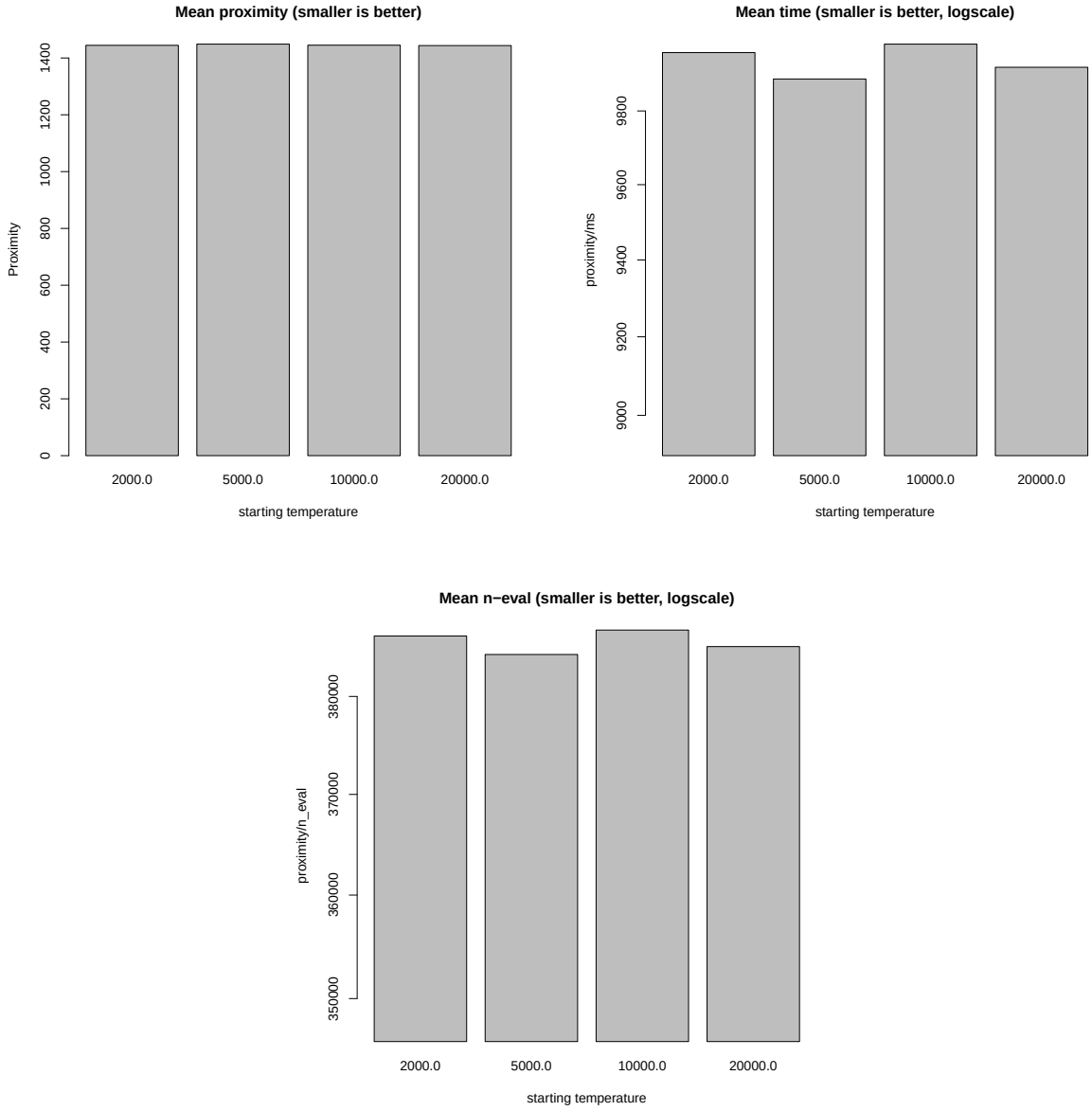


Figure 6.29: Parameter influence – starting temperature T_{start}

6.1.10 Taboo search

The very simple **Taboo Search** method overall ranked third in best-case performance and ranked first in terms of both mean and worst case performance, it's robustness was on par with the **Simulated Annealing** method. As the five best settings show 6.9 no other value then 0.2 for *mutation step* was noted, and there seems to be no preferred taboo size.

On average, the best convergence was observed for **mutation step** sizes 0.2 and 0.4 with a faster real time convergence speed for the value 0.2. For **mutation step** = 0.2 the amount of evaluations was also lower but not significantly.

Increasing the **taboo size** slightly reduced the convergence of the algorithm, the real time speed of convergence was proportionally slower for bigger sizes.

Bigger taboo sizes decreased the fitness function evaluation count, although there was almost no significant difference between **taboo size** 10 and 100, or **taboo size** 500 and 1000.

Using clustering decreased the convergence, and slowed convergence speed, when using clustering cluster sizes $\{0.001, 0.01, 0.1\}$ behaved had similarly in terms of convergence with convergence speed proportionally slower as the cluster size increased.

taboo size	mutation size	cluster size	Proximity	Time [s]	Time [%]	n-eval
100	0.2	0.001	186.92	35.13	0.36	930640
500	0.2	N/A	193.24	83.62	0.85	986095
10	0.2	0.01	195.54	17.56	0.18	712367
10000	0.2	N/A	198.17	2612.56	26.55	1023093
1000	0.2	N/A	198.71	140.22	1.43	996131

Table 6.9: Best five settings (by proximity) for the **Taboo Search** algorithm. Time [s] denotes time in ms spent on optimizing, and Time [%] is the time spent relative to the worst runtime. n-eval is the number of evaluations of the fitness function.

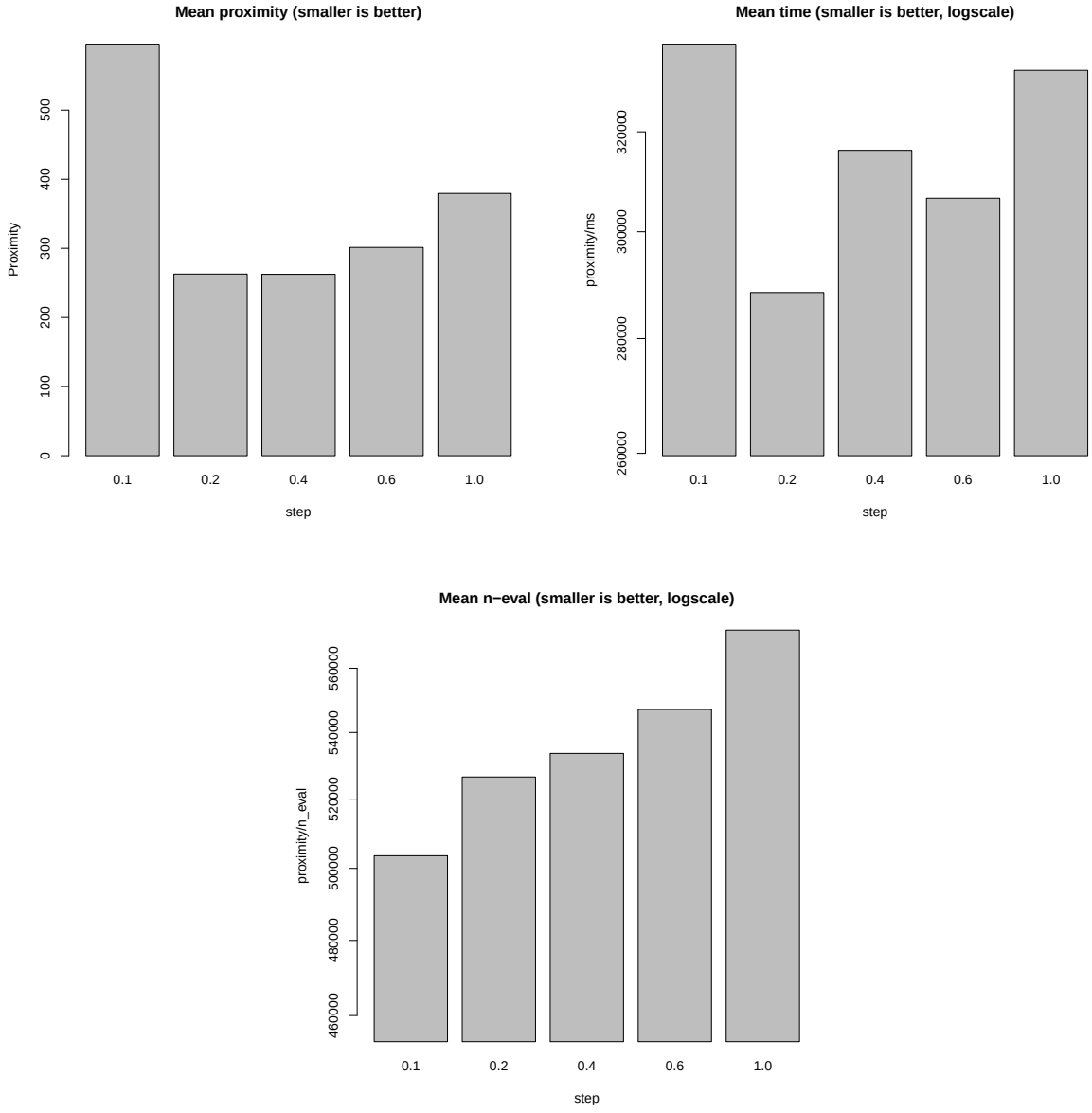


Figure 6.30: Parameter influence – mutation step size

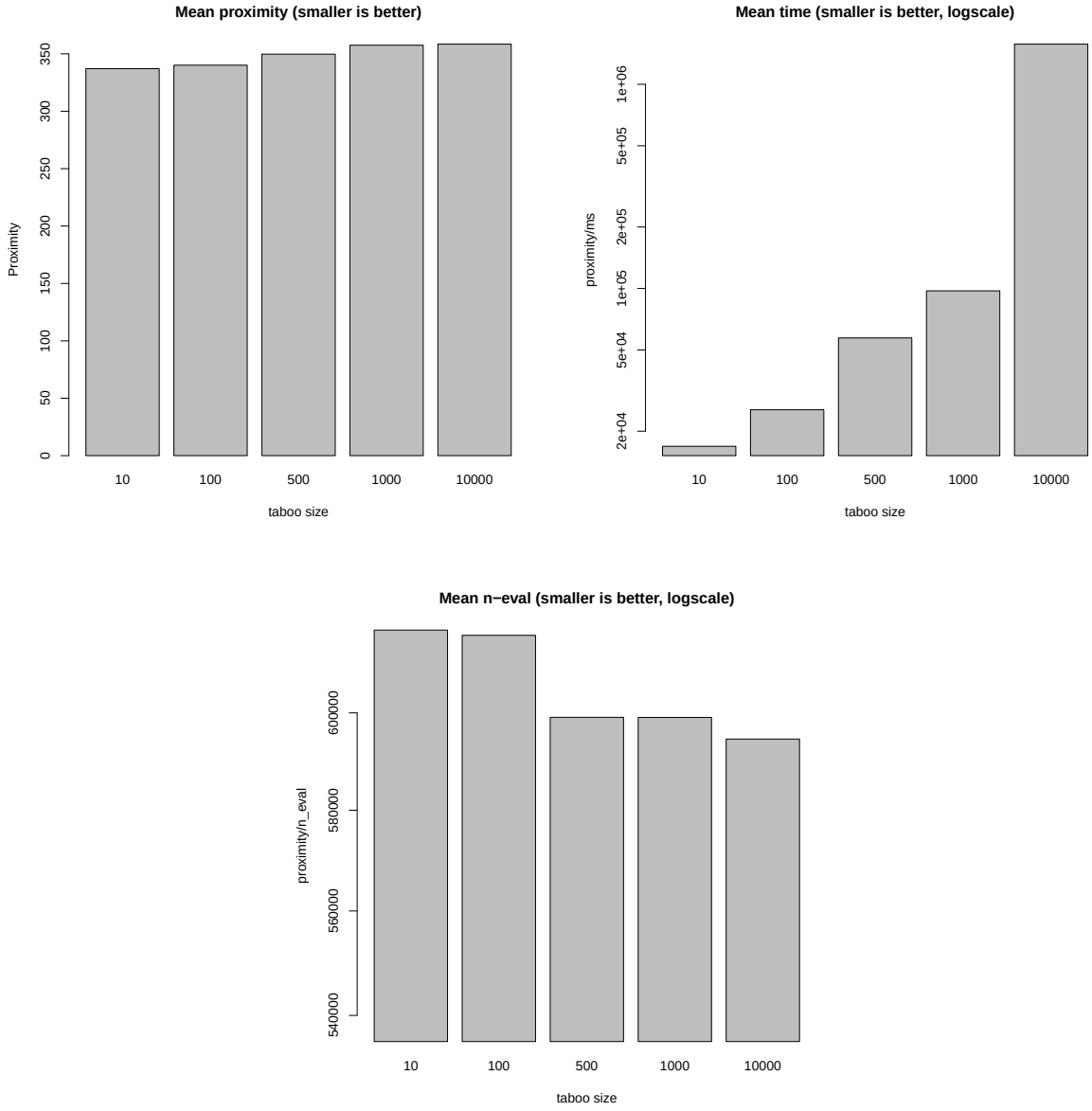


Figure 6.31: Parameter influence – taboo size

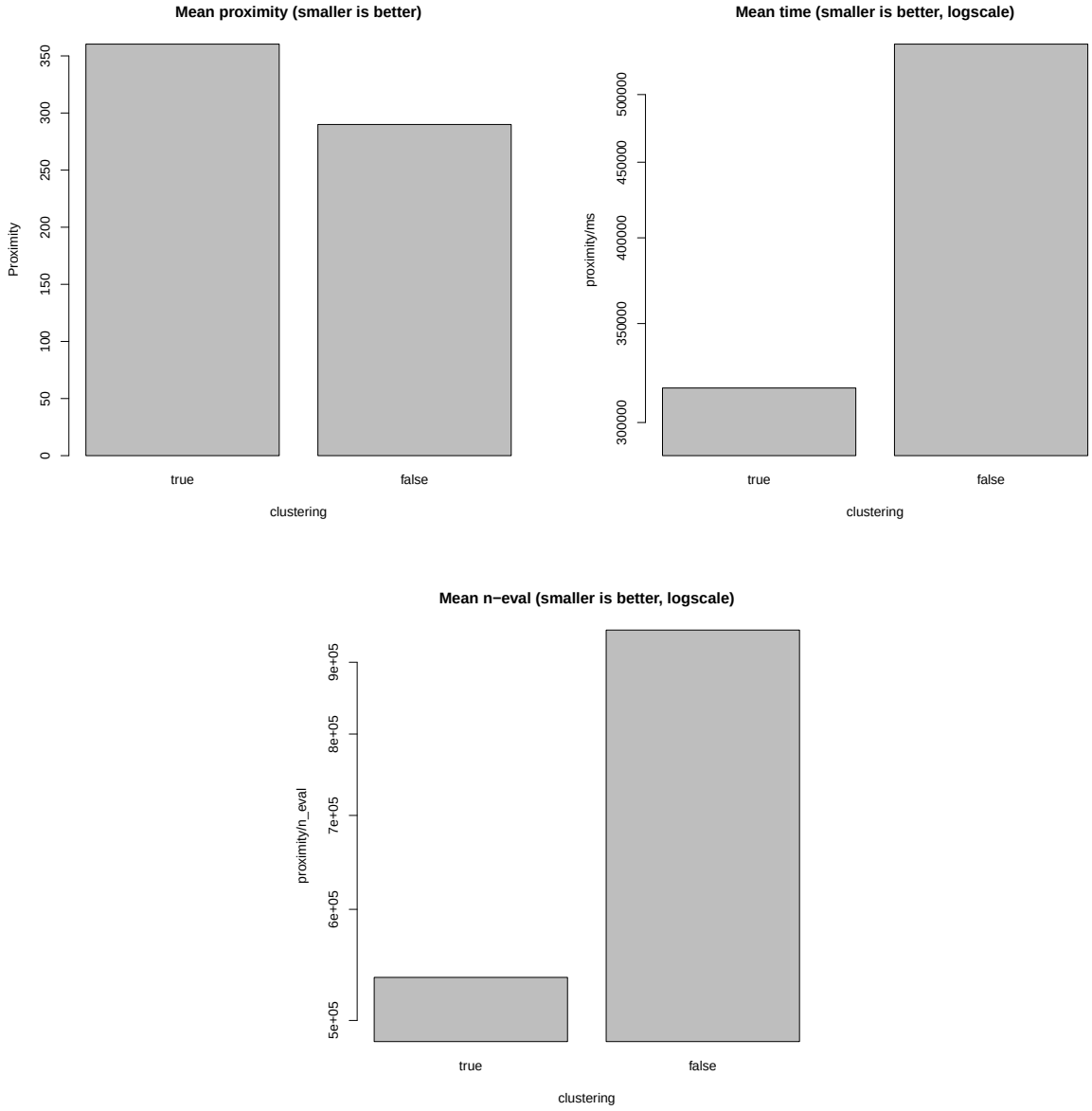
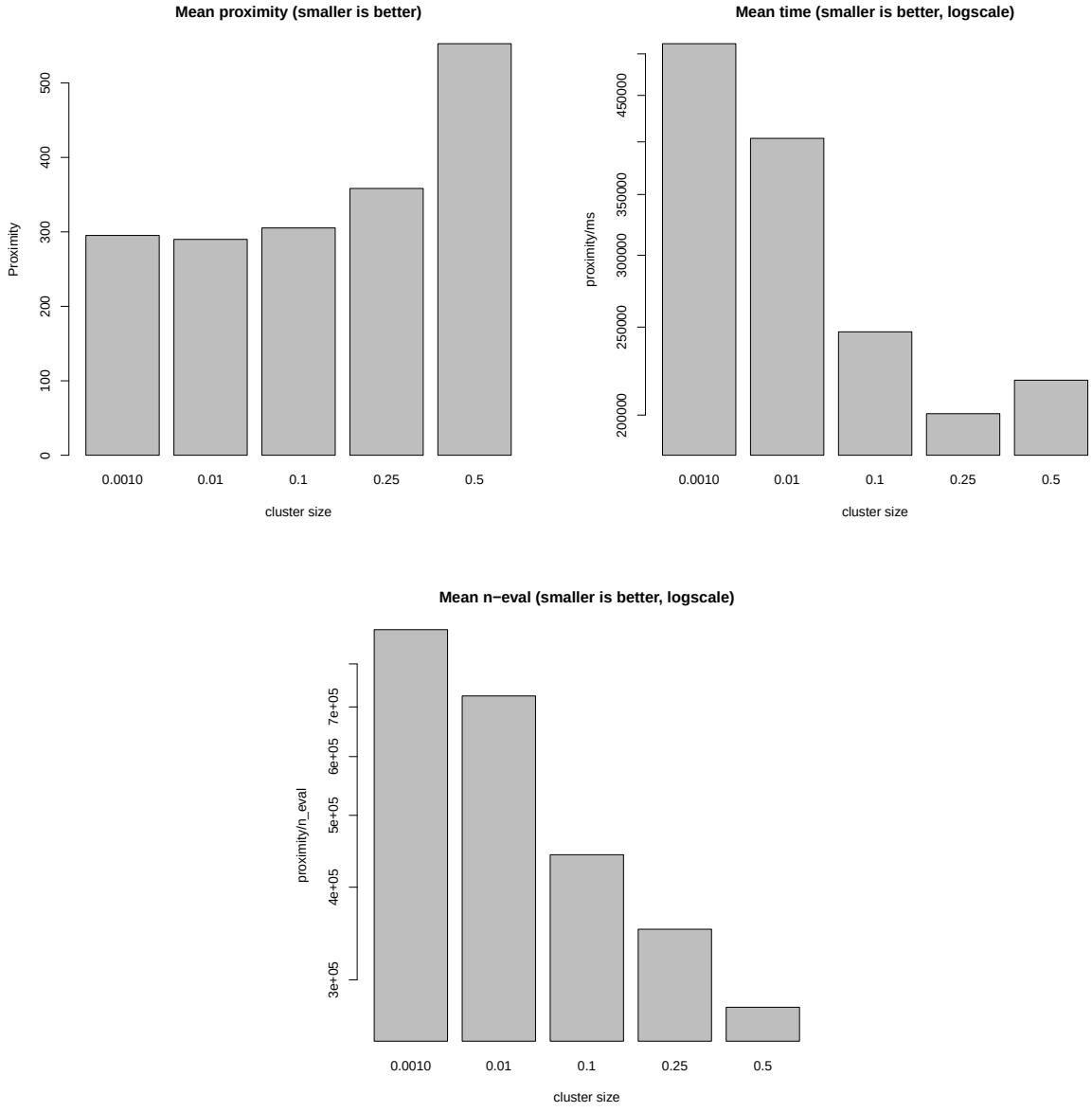


Figure 6.32: Parameter influence – clustering

Parameter influence – cluster size



6.2 The Image From Polygons Problem

The results for the "Image From Polygons" toy problem were a little different then the results in the numerical problems – and the differences may hint how the algorithms deteriorate with an extremely large dimension count but the results cannot be conclusive.

- The **Genetic Algorithms** scored best with **Differential Evolution** second.
- Performance of the **Cross Entropy Optimization** suffered having the worst best case performance of all algorithms.

- The **Simulated Annealing** behaves more gracefully with a third best performance globally with **Taboo Search** coming fourth.
- Relative performance of the **Harmony Search** algorithm was lower – second worst with **Random Optimization** taking third worst.

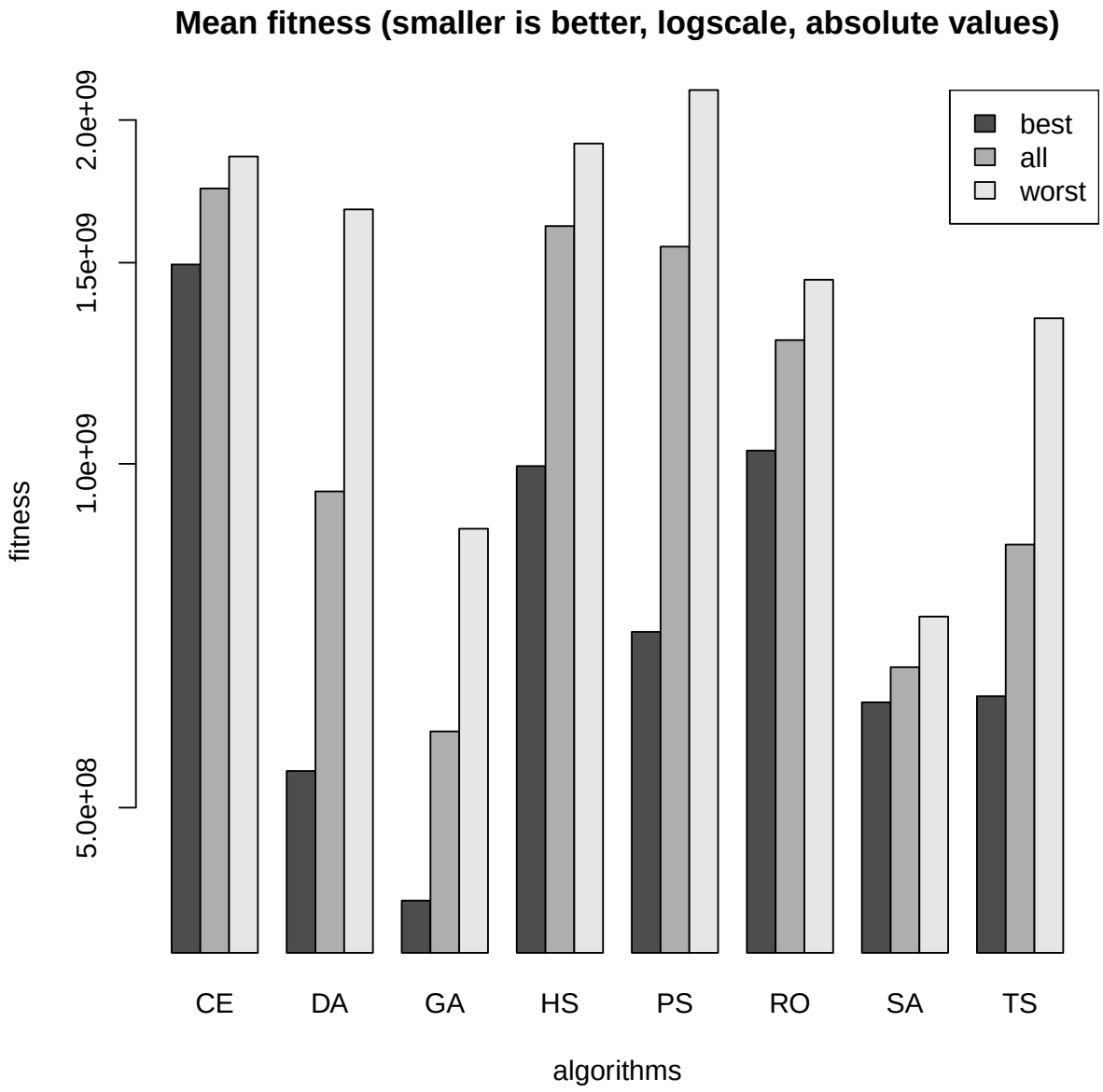


Figure 6.33: Algorithm ranking for the image from polygons problem.

Chapter 7

Conclustions.

7.1 Results

The **Differential Evolution** algorithm proved to be the best when solving the given numerical and toy problems, with the **Genetic Algorithms** as a second best. The situation reversed itself in the image from polygons problem.

The **DA** seems especially elegant in its simplicity with only two parameters to tune, as is the **Cross Entropy Method** which also did comparatively well in numerical problems (although its performance dropped substantially in the images from polygons problem).

In its basic form **Taboo Search** was third best, and very good results were also observed for **Particle Swarms** and **Cross Entropy Optimization**.

Simulated Annealing deemed a very stable algorithm with a worst case performance comparative to the worst case one. The algorithm didn't fare well in the numerical problems but flourished in the image generation problem.

7.2 Further Studies

Although some basic insights were gained on choice of good parameters for algorithms that should give a good starting point for future researchers, they may be inconclusive for tougher/real life problems and further inquiry/statistical analysis may confirm or refute these "rules of a thumb'.

Meta-optimization could be used to find the best set of parameters to specialized real-life applications or more general good values of parameters.

With regard to the optimization library - more algorithms could be incorporated and analyzed throughly - such as the **Firefly Algorithm** similar to **Particle Swarms**, the **Ant Colony Optimization**, various **memetic** and **Culture Based** algorithms. With a standard base for developing algorithms **hybrid** approaches to optimization can be developed.

Further inquiry about parallelization of implemented algorithms can be also sought, as can be the use of distributed systems for dividing the work load. One can speculate about the hybrid approaches to distributed systems with *populations* or *swarms* broadcasting best found solutions – exploring independently but sharing work-load and knowledge.

Bibliography

[1] Jonathan E. Rowe Colin R. Reeves. *Genetic Algorithms: Principles and Perspectives. A Guide to GA Theory.* Klutwer Academic Publishers, 2009.

[2] John Tsitsiklis Dimitris Bertsimas. Simulated annealing. *Statistical Science*, 8(1):10–15, 1993.

[3] Dr. Zong Woo Geem. Harmony search. `http://www.hydroteq.com`, September 2010.

[4] Abdel Rahman Hedar. Global optimization test problems. `http://www-optima.amp.i.kyoto-u.ac.jp/member/student/hedar/Hedar_files/TestGO.htm`, October 2010.

[5] Thomas Weise. *Global Optimization Algorithms - Theory and Application.* Self-Published, second edition, June 26, 2009. Available online at http://www.it-weise.de/.

[6] D.H. Wolpert and W.G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, April 1997.

Appendix – included files

Files included with this work are organized in a hierarchy:

```
/output.csv
/jgol/
```

This includes the log output and the contents of the jgol optimization library.

7.3 Log output

The `/output.csv` file contains comma separated values and an additional header describing the field names:

```
"algorithm_name","problem_name","iterations","ms","n_eval","fitness","proximity"
```

The file was generated from last entries of multiple log files produced by the test **problems** project; additionally the values of five run times were averaged.

The `algorithm_name` field contains a string describing the algorithm in the format:

```
${PREFIX} (${parameter 1} = ${value 1}, ... ) PS = ${POPULATION_SIZE}
```

Additionally some parameters may be boolean only; for example:

```
GA (elitism, sel = Tournament (count = 2, size = 2), cross = 1-Point (rate = 1.0), mut
= (mut-size = 1.0, mut-rate = 0.1) PS = 25
```

7.4 The jgol optimization library

The `jgol` optimization library is organized as a standard **Netbeans** project:

```
jgol/
|+build/
|+dist/
|+lib/
|+nbproject/
|+problems/
|+src/
|-build.xml
|-CHANGELOG
|-LICENSE
‘-TODO
```

The `lib/` directory contains libraries used by the optimization library or the **problems** project. One library used by the **problems** project, the java OpenGL wrapper (jogl) works correctly only on a `Linux x86_64` system (as it uses native binaries).

The `problems/` directory contains the test project utilizing the **jgol** optimization library, and implementing the optimization problems described in this work.

By default running the **problems** project will try solve the test benchmarks with every combination of algorithm and algorithm parameters that was described in this document. Be wary, because this may take a very long time (about 30 days of real time work for the system described previously in section 5.5.1).

The log files generated by the **problems** project are similar to the `output.csv` file, but lacking the header, and with one additional field:

```
"algorithm_name","problem_name","run","iterations","ms","n_eval","fitness","proximity"
```

The `run` field is a sequence number 1–5 discerning which run does the value belong to.

Additionally, the files logged, are more verbose logging every registered improvement of the fitness function.