Chapter 12
Examples and applications

Human beings, who are almost unique in having the ability to learn from the experience of others, are also remarkable for their apparent disinclination to do so.
Douglas Adams

This chapter contains a number of examples to illustrate the use of μGP. In the following each example is described completely, so that each experiment can be easily repeated. Every section contains detailed information about the preparation of the settings file, of the population settings file and of the constraints file. In addition, it contains the code of the fitness evaluator. The results of the evolutionary runs are provided for each experience.

The examples are not chosen for their usefulness, but rather to bring out some difficulties that may arise in the use of any evolutionary tool, and to show how these can be solved using μGP. For instance, in some cases the user may need to design the constraints file and the fitness evaluator together to correctly describe individuals whose syntactic structure does not match trivially the structure of the constraints file. In other cases describing the individuals is not a problem, but their semantic may be, such as when it contemplates the possibility of endless loops. Sometimes the simplest possible fitness function may be too difficult to optimize, or it may lead to bloating. In such cases the user may have to modify the fitness function, or to add further fitness values.

At the same time, the results allow the reader to appreciate the way in which the different parameters influence the evolutionary process. The performance of the tool may be sensitive to different parameters when confronted with different problems. In this perspective, the following examples are meant to provide an insight on the evolutionary process.

12.1 Classical one-max

This problem is somewhat like a “hello world” of optimizing methods in general, and of evolutionary algorithms in particular. The problem is simple. Let’s call s a generic string of bit. Given the set S_N of all the possible strings of bits of length N, the goal is finding a string $s_m \in S_N$ so that the number of bits set to ’1’ in the string is maximum.

The problem has an immediate solution, represented by a string made of N '1' bits. Obviously, this problem does not require an optimizer, but can be used as a test to check that an optimizer actually works.

12.1.1 Fitness evaluator

Despite the simplicity of the problem there is stil ample choice for the implementation of the fitness evaluator. The simplest choice is a program that takes a string of '0' and '1' as input and produces as output the number of '1' characters in the string.

One source of problems with this approach is the ability of μGP to produce variable-size individuals. The result of the evolution would then be different from expectations, since the evoutionay core would be able to increase the fitness of individuals just by adding random bits to their genome. On average, one half of those bits would be '1', so the fitness could increase without bounds. There would be no selective pressure on the individuals for shedding the '0' bits.

There is a simple route for solving this problem. Just assign zero fitness to all individuals that have a number of bits different from N. This approach works, and eventually leads to the expected result.

The same result could be obtained by choosing which genetic operators are used during evolution. If all operators that could change the size of the individual are suppressed by setting their weight to 0, and the initial size of the individuals is set to N, then all individuals generated would be the right size, and then would be no need to check it.

What's wrong with this approach? Nothing, but the user should be careful, because it relies on *a priori* knowledge of the problem domain. In particular, it is known that a sequence of alteration mutations, local mutations and scan mutations can lead to the desired result, no matter what the starting point is. If such knowledge is available it is perfectly acceptable to use it. When this approach can be used, it may lead to substantial savings in optimization time, since it greatly restricts the search space.

For other problems, allowing the individuals to grow or shrink outside the problem domain may make alternative evolutionary routes available and allow the optimizer to reach an optimal solution.

The example provided with the tool uses the approach described above. Individuals are bound to be exactly N bits long, and every mutation that changes the size results in a failure. No individual longer or shorter than N is ever evaluated, so the evaluation script does not check the size.

Below is the evaluation script.

```
#!/usr/bin/perl -w-

# Starting from v3.1.2_1142 fitness scripts can use
# (again) environment variables:
#
# $UGP3_FITNESS_FILE : the file created by the
#                      evaluator
# $UGP3_OFFSPRING    : the individuals to be
#                      evaluated
#                      (space separated list)
# $UGP3_GENERATION   : generation number
# $UGP3_VERSION      : current ugp3 version.
#                      eg. 3.1.2_1142
# $UGP3_TAGLINE      : full ugp3 tagline.
#                      eg. ugp3 (MicroGP++)
#                      v3.1.2_1142 "Bluebell"

open OUT, ">$ENV{UGP3_FITNESS_FILE}"
  or die "Can't create $ENV{UGP3_FITNESS_FILE}: $!";
foreach $file (@ARGV) {
    open F, $file or die "Can't open $file: $!";
# read a single line from file F
    $_ = <F>;
# count the '1' characters
    $n = tr/1/1/;

# the comment string is the current time
    $time = localtime;
    $time =~ tr/ /_/;
    print OUT "$n $file\@$time\n";
}

close OUT;
```

The fitness script is written in PERL scripting language. A modified version, with additional comments, is reported here. First it opens the fitness file for writing, or aborts if it is unable to do that. In case the fitness script aborts without generating any file, the whole evolutionary process will be aborted.

Then the script parses its command line arguments. Every argument is the name of an individual to evaluate. Again, if opening the corresponding file is not possible, the evaluation is aborted.

Every file is expected to contain a single line, which is read and scanned to count the '1' in it. No attempt is made to count the total number of characters in the line.

The comment string is set to the name of the individual followed by the time of evaluation, with the spaces replaced by underscores.

This script is able to evaluate many individuals, as specified by the concurrent evaluations parameter in the population settings file, but evaluation is actually sequential. To achieve actual concurrent evaluation the script should be modified so that it sets up all the needed processes or threads. Even if evaluation is not parallel it may still pay off to evaluate many individuals together, since it will save several calls to the perl interpreter, with the corresponding initialization sequences.

12.1.2 Constraints

The constraints for one-max are conceptually simple. Every individual must be composed by a linear sequence of *N* character, either '0' or '1'. Global and section prologues and epilogues are not needed.

Constraints are reported in a modified form to fit the page, splitting long lines in shorter ones. Splitting the lines, especially quoted strings, would raise an error in an actual run.

```
<?xml version="1.0" encoding="utf-8"?>
<?xml-stylesheet type="text/xsl"
  href="http://www.cad.polito.it/ugp3/transforms/
             constraintsScripted.xslt"?>
<constraints
  xmlns="http://www.cad.polito.it/ugp3/schemas/
              constraints"
  id="One-Max"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.cad.polito.it/ugp3/
                         schemas/constraints
                    http://www.cad.polito.it/ugp3/
                         schemas/constraints.xsd">
  <typeDefinitions>
    <item xsi:type="constant" name="bit_type">
      <value>0</value>
      <value>1</value>
    </item>
  </typeDefinitions>
  <commentFormat><value/></commentFormat>
  <identifierFormat>n<value /></identifierFormat>
  <labelFormat><value/>: </labelFormat>
  <uniqueTagFormat><value /></uniqueTagFormat>
  <prologue id="globalPrologue"/>
  <epilogue id="globalEpilogue"/>
  <sections>
    <section id="bitString"
             prologueEpilogueCompulsory="false">
      <prologue id="sectionPrologue"/>
      <epilogue id="sectionEpilogue">
```

```
        <expression></expression>
      </epilogue>
      <subSections>
        <subSection id="main" maxOccurs="1"
                    minOccurs="1" maxReferences="0">
          <prologue id="stringPrologue"/>
          <epilogue id="stringEpilogue"/>
          <macros maxOccurs="50" minOccurs="50"
                  averageOccurs="50" sigma="10">
            <macro id="bitString">
              <expression><param ref="bit"/>
              </expression>
              <parameters>
                <item xsi:type="definedType"
                      ref="bit_type" name="bit" />
              </parameters>
            </macro>
          </macros>
        </subSection>
      </subSections>
    </section>
  </sections>
</constraints>
```

The constraints contain an empty global prologue and epilogue, and a single section. In the section appear an empty prologue, an epilogue with an empty expression and a single subsection. It is interesting to note that the empty prologue and the epilogue with an empty expression produce the same effect, that is no string is produced at the beginning or at the end of an individual.

The subsection, named “main”, can occur exactly one time and cannot be referenced. In this specific case specifying the allowed number of references has no meaning, since there is no macro in the constraints with a reference.

Again, the subsection has empty prologue and epilogue. After that, the constraints specify that the subsection can contain from a minimum of 50 to a maximum of 50 macros, with the same average. This just means that the number of macros is fixed, and individuals with a different number of vertices in the corresponding subgraph are not valid and will not be evaluated. The non zero `sigma` parameter is actually meaningless, all individuals are generated with the specified number of vertices.

There is only one type of macro in the constraints, in which the expression is an unadorned parameter of type `bit_type`, defined at the start of the constraints as either ’0’ or ’1’.

12.1.3 Population settings

The population parameters for this example are reported below. All options, including optional ones, are used. As for the constraints, some elements are reported split on two or more lines to fit the page.

```
<?xml version="1.0" encoding="utf-8" ?>
<parameters type="enhanced">
  <cloneScalingFactor value="0"/>
  <eliteSize value="0"/>
  <maximumFitness value="50"/>
  <maximumSteadyStateGenerations value="20"/>
  <mu value="10"/>
  <nu value="10"/>
  <lambda value="10"/>
  <inertia value="0.9"/>
  <fitnessParameters value="1"/>
  <maximumAge value="10"/>
  <sigma value="0.9"/>
  <invalidateFitnessAfterGeneration value="0"/>
  <constraints value="onemax.constraints.xml"/>
  <maximumGenerations value="100"/>
  <maximumEvaluations value="1000"/>
  <selection type="tournamentWithFitnessHole" tau="1"
             tauMin="1" tauMax="1" fitnessHole="0" />
  <evaluation>
    <concurrentEvaluations value="4" />
    <removeTempFiles value="true" />
    <evaluatorPathName
        value="./onemax.fitness-script.pl" />
    <evaluatorInputPathName value="individual.in" />
    <evaluatorOutputPathName value="fitness.out" />
  </evaluation>
  <operatorsStatistics>
    <operator ref="onePointSafeCrossover">
      <weight current="1" minimum="0" maximum="1"/>
    </operator>
    <operator ref="onePointSafeSimpleCrossover">
      <weight current="1" minimum="0" maximum="1"/>
    </operator>
    <operator ref="twoPointSafeSimpleCrossover">
      <weight current="1" minimum="0" maximum="1"/>
    </operator>
    <operator ref="singleParameterAlterationMutation">
      <weight current="1" minimum="0" maximum="1"/>
    </operator>
    <operator ref="insertionMutation">
      <weight current="1" minimum="0" maximum="1"/>
    </operator>
    <operator ref="removalMutation">
      <weight current="1" minimum="0" maximum="1"/>
    </operator>
    <operator ref="replacementMutation">
      <weight current="1" minimum="0" maximum="1"/>
    </operator>
    <operator ref="alterationMutation">
      <weight current="1" minimum="0" maximum="1"/>
    </operator>
    <operator ref="subGraphInsertionMutation">
      <weight current="1" minimum="0" maximum="1"/>
    </operator>
    <operator ref="subGraphRemovalMutation">
      <weight current="1" minimum="0" maximum="1"/>
    </operator>
    <operator ref="scanMutation">
      <weight current="1" minimum="0" maximum="1"/>
    </operator>
    <operator ref="subGraphReplacementMutation">
      <weight current="1" minimum="0" maximum="1"/>
    </operator>
    <operator ref="randomWalkMutation">
      <weight current="1" minimum="0" maximum="1"/>
    </operator>
    <operator ref="localScanMutation">
      <weight current="1" minimum="0" maximum="1"/>
    </operator>
  </operatorsStatistics>
</parameters>
```

An enhanced population is used, with 10 individuals. At each generation, 10 genetic operators are applied. The entropy fitness hole is not used and the parameters of `tournamentSelection` are set to choose an individual only, thus performing a simple random selection on the individuals each time a genetic operator is applied.

All operator weigths are equal, exactly as they are generated using the `ugp3-population` tool.

The evolution is stopped as soon as it reaches the maximum possible fitness value, or if 20 generations are elapsed without any progress in the best fitness value. This event is extremely improbable, unless the population parameters are changed to extreme values.

12.1.4 µGP settings

Below are reported the general tool settings for one-max. As usual, some options are split on different lines to fit the page. We recommend that in actual use they are kept on single lines.

These are actually fairly standard settings. Apart from the reference to the population parameters file, nearly identical settings may be used for a wide range of problems.

```
<?xml version="1.0" encoding="utf-8" ?>
<settings>
  <context name="evolution">
    <option name ="populations">
      <population name="OneMax-Population"
           value="onemax.population.settings.xml" />
    </option>
    <option name="statisticsPathName"
            value="statistics.xml" />
  </context>
  <context name="recovery">
    <option name="recoveryInput" value="" />
    <option name="recoveryInputPopulations" value="" />
    <option name="recoveryOutput" value="status.xml" />
    <option name="recoveryOverwriteOutput"
            value="true" />
    <option name="recoveryDiscardFitness"
            value="true" />
  </context>
  <context name="logging">
    <option name="std::cout" value="info; brief" />
  </context>
</settings>
```

These settings specify a single population, described by the parameters in section 12.1.3. No recovery input is specified, and the recovery output file is overwritten at every generation. The settings also specify to discard the fitness values of the recovered status, but since there is no recovered status, this option has no effect.

No seed for the random number generator is specified. Should the user desire to repeat a series of identical runs, the `randomSeed` element in the `evolution` context should be set to a specific value.

The logging is kept brief. Only the standard output is generated, and no file is written. Given the purpose of the example, logging should only be activated if the user suspects the presence of a software bug.

12.1.5 Running

Once the files described above are ready and placed in the same folder, the evolution can start by simply invoking the μGP executable, `ugp3`, without any command-line parameter. Evolution should end after a few seconds, finding the optimal solution.

The user can verify that the optimal solution has been reached by using the μGP extractor software. Provided that the μGP settings file is exactly as reported above, typing `ugp3-extractor status.xml` on the console will generate a text file containing the best individual, compute its fitness and display a brief report of the operations executed. We recommend the user to take note of these names, as they may be easily overlooked in a directory with many files inside. If a first run does not return the optimal solution, a new run most likely will. If the tool is regularly unable to find the best solution, chances are that the settings files are not conform to the samples reported above: and the user should thus check their correctness.

A first rule of thumb is that the maximum number of generations should be in linear relation with the size of the population. A safe bet is to allow at least $2n$ generations in a run with n individuals.

The user could also check that the genetic operators are not deactivated: any operator that does not appear in the population settings file or has a weight of zero, is not used. If several operators are missing from the settings or have a weight of zero, it might be impossible to perform the optimization at all.

On the other hand, choosing which operators are used may help speed up the evolution. Some operators do not change the size of the individual, some may or may not change it, some will certainly grow or shrink the individual. Removing the latter from the list of available operators should speed up evolution, simply because more valid individuals will be produced per generation.

A quick demonstration the user may perform consists of running the evolutionary process both with the settings described above and with modified population settings. The modified settings will prescribe a weight of zero for the insertion mutation, removal mutation, all subgraph mutation operators. These are all operators that are guaranteed to fail, since the individuals have a fixed size and only a single subgraph, as by constraint settings. The operators are deactivated by setting the `current` weight to 0. Operators with a current weight of zero will not be used, their success rate will not be positive, thus their weight will not change.

We tried the experiment repeating the run 1000 times for each setting, without controlling the seed of the random number generator. This means that the results are not exactly repeatable. Your mileage *will* vary.

With the first setup we obtained the optimal solution 999 times out of 1000, on average in about 26.9 generations. The second setup found the optimal solution in each run, with an average of generations 25 generations elapsed. If you choose to use self adaptation of μGP parameters, eventually genetic operators that fail too much will see their activation probability decrease to minimum levels. This procedure speeds up the evolution, *de facto* removing useless operators.

12.2 Values of parameters and their influence on the evolution: Arithmetic expressions

Arithmetic functions are often used as test benches to evaluate the convergence and performance of evolutionary algorithms. Some of them are well known in literature, for example all tests developed by Kenneth De Jong, but in principle every function with specific features (e.g. a great number of local optima and a single global optimum) can be used for this purpose.

In all the cases presented in this section, choosing the fitness function and the representation of the individuals is trivial: the fitness is obtained directly from the arithmetic function and each individual is an array of real numbers in a given interval.

More interesting are the variation of the performance of μGP obtained by tweaking the population parameters. It will be shown how each parameter influences a run and thus it should be set to an appropriate value in every experiment in order to maximize the performance of the evolutionary algorithm.

12.2.1 De Jong 3

The third function developed by De Jong is defined as follows:

$$f_{DJ3}(x) = 5 \cdot n + \sum_{i=1}^{n} \lfloor x \rfloor, \qquad x_i \in [-5.12,\ 5.12] \qquad (12.1)$$

The function is monomodal and not continouos, and it has an infinite number of local minima $f(x^*) = 0$ for $x_i^* \in [-5.12,\ -5]$, with $i = 1, 2, ..., n$.

For the following experiments $n = 5$, so each individual is composed by 5 real numbers, $x_{1,...,5} \in [-5.12, 5.12]$. Some parameters, summarized in Table 12.1 and Table 12.2 are not changed through all the experiments.

Parameter	Value
μ	1 000
ν	500
λ	250
σ	0.970, 0.980
`MaximumAge`	20
`maximumSteadyStateGenerations`	5 000
`maximumGenerations`	10 000
Seed	5987579

Table 12.1 Parameters of all the following experiments.

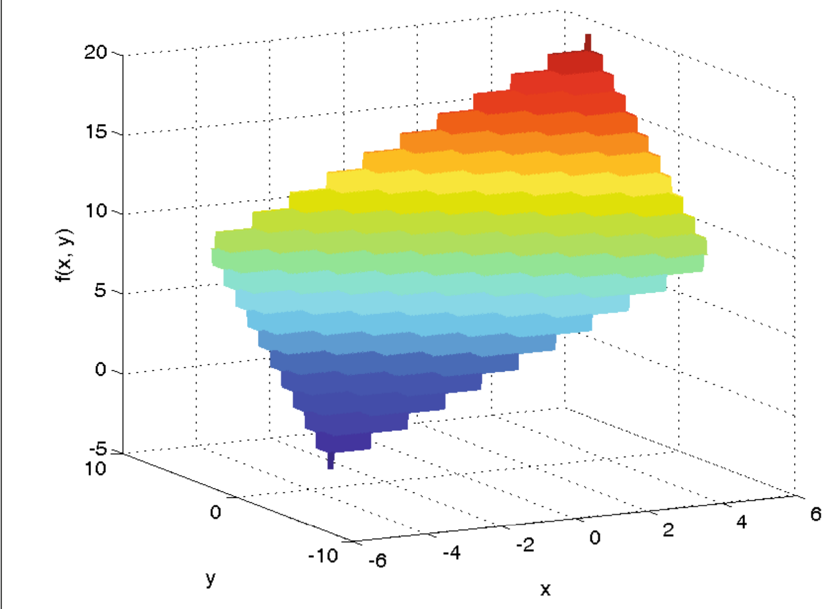


Fig. 12.1 De Jong's third function

Operator	Curr. Probability	Min. Probability	Max. Probability
`replacementMutation`	0.5	0	1
`singleParameterAlterationMutation`	0.5	0	1

Table 12.2 Genetic operators used in the experience

12.2.1.1 The τ parameter

The τ parameter describes the number of individuals that will be randomly selected to take part in the tournament selection to choose the parents for an activated genetic operator. Incresing τ makes the choice more deterministic, steadily rewarding individuals with better fitness values, while decreasing τ makes the choice of the parents more and more random.

Thus, τ should be chosen wisely, depending on the function you are trying to optimize. As a rule of thumb, high values of τ are beneficial up to a certain point, where the trend becomes the opposite. In Table 12.3 to Table 12.8 we see how τ tweaking works for this particular arithmetic function, with different values of the `EliteSize` and `Inertia` parameters. For each combination of values, the number of steps μGP required to find the global optimum is listed.

In the third De Jong function, an increase in τ leads to a small improvement, up to $\tau = 128$, then the system has a small setback and there are no further improvements. Fig. 12.2 summarizes the results.

$\tau = 8$		16	32	64	128	256
`Inertia`	Steps	steps	steps	Steps	Steps	Steps
0.93	322	180	92	72	64	69
0.94	296	170	94	90	61	65
0.95	267	206	169	95	82	109
0.96	251	204	175	96	96	85
0.97	286	180	220	111	104	103
Average	284.4	188	150	92.8	81.4	86.2

Table 12.3 Results for `EliteSize` = 8

$\tau = 8$		16	32	64	128	256
`Inertia`	Steps	Steps	Steps	Steps	Steps	Steps
0.93	224	157	114	105	64	69
0.94	149	92	76	90	61	65
0.95	305	177	146	95	82	116
0.96	298	210	163	118	96	85
0.97	256	239	177	195	104	103
Average	246.4	175	135.2	120.6	81.4	87.6

Table 12.4 Results for `EliteSize` = 16

$\tau = 8$		16	32	64	128	256
`Inertia`	Steps	Steps	Steps	Steps	Steps	Steps
0.93	276	154	114	72	64	69
0.94	281	145	76	90	61	65
0.95	272	163	169	112	82	116
0.96	360	184	172	111	96	85
0.97	290	263	187	156	104	103
Average	295.8	181.8	143.6	108.2	81.4	87.6

Table 12.5 Results for `EliteSize` = 32

12.2.1.2 The ν parameter

The ν parameter describes the initial size of the first population, containing only randomly-generated individuals. In some experiments, exspecially those where the evaluation of a single individual takes some time, it can be useful to draw upon an initially larger quantity of genetic material. Aside from these specific cases, how-

$\tau = 8$		16	32	64	128	256
`Inertia`	Steps	Steps	Steps	Steps	Steps	Steps
0.93	268	160	110	72	64	69
0.94	269	150	76	90	61	65
0.95	186	188	134	112	82	116
0.96	289	175	145	111	96	85
0.97	227	255	182	174	104	103
Average	247.8	185.6	129.4	111.8	81.4	87.6

Table 12.6 Results for `EliteSize` = 64

$\tau = 8$		16	32	64	128	256
`Inertia`	Steps	Steps	Steps	Steps	Steps	Steps
0.93	229	171	110	72	64	69
0.94	251	150	76	90	61	65
0.95	282	186	134	112	82	116
0.96	303	201	145	111	96	85
0.97	354	226	174	174	104	103
Average	283.8	186.8	127.8	111.8	81.4	87.6

Table 12.7 Results for `EliteSize` = 128

$\tau = 8$		16	32	64	128	256
`Inertia`	Steps	Steps	Steps	Steps	Steps	Steps
0.93	229	160	110	72	64	69
0.94	255	150	76	90	61	65
0.95	236	176	134	112	82	116
0.96	284	224	145	111	96	85
0.97	228	163	174	174	104	103
Media	246.4	174.6	127.8	111.8	81.4	87.6

Table 12.8 Results for `EliteSize` = 256

ever, it is expected that any initial advantage ν can provide to the fitness values in the population will be lost as the generations go on.

For the third function of De Jong, the latter proves true: as shown in Fig. 12.3, any initial increase in the average fitness value is rapidly lost, and the graph assumes a similar shape no matter the value of ν in the experiment.

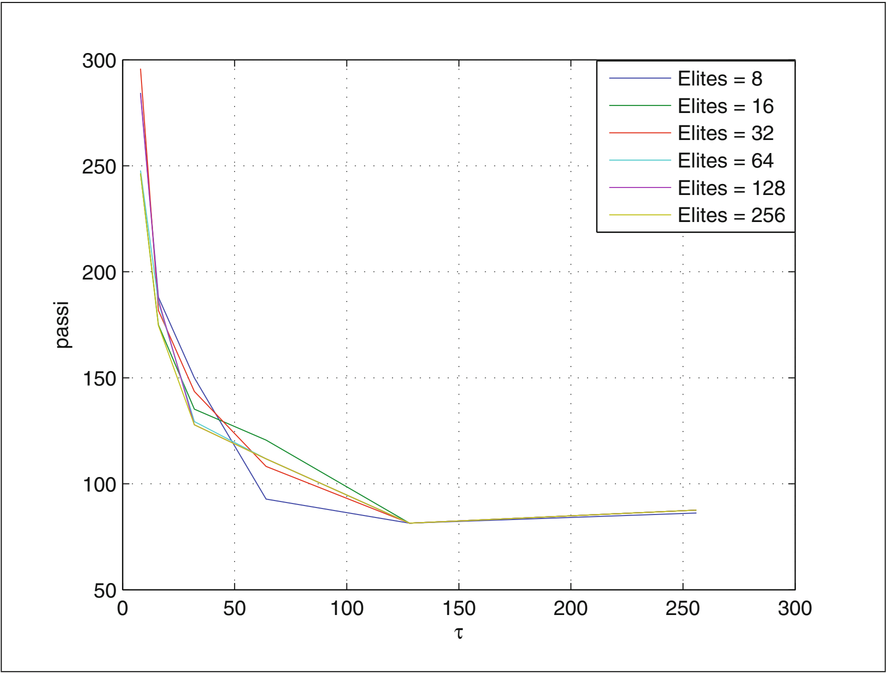


Fig. 12.2 Number of steps needed to find the optimal solution for different values of τ.

12.2.1.3 The `FitnessHole` parameter

`FitnessHole` is a parameter that can be extremely useful in experiments where individuals have distinct blocks (e.g. individuals that describe several assembly functions), since certain values help to reward individuals with a pattern which is uncommon in the population at a given generation. That increments entropy in the system and preserves some potentially useful genetic material that could otherwise be wasted.

In functions where an individual is basically an array of numbers, however, the same values for this parameter can be a disadvantage: lacking distinct blocks, the only result of tweaking `FitnessHole` will be a random selection of individuals during a tournament.

As shown in Fig. 12.4, this is the case with the third De Jong function. `FitnessHole` should be altered from its default value of 0 only when the structure of a single individual is really complex.

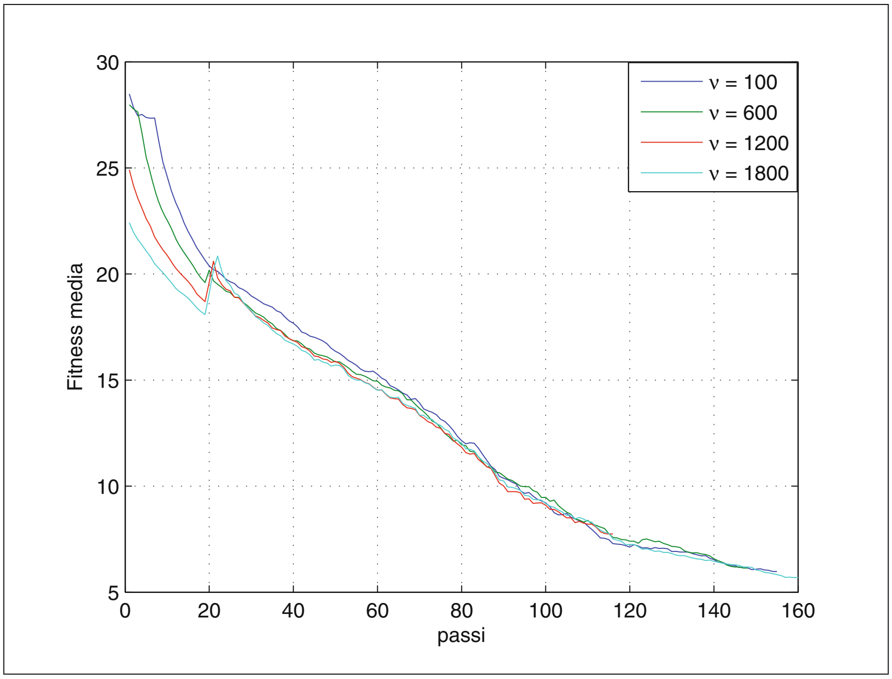


Fig. 12.3 Influence of ν on the average fitness for each generation.

12.2.2 De Jong 4 - Modified

The fourth function chosen by De Jong to be included in his *test suite* provides a good example for parameter tweaking. In the original function, the random variable η has a gaussian distribution $N(0,\ 1)$; in this experiment, however, the function has been slightly modified. η, in fact, is considered having a uniform distribution $\in [0,1)$, because in its original version, the fourth De Jong function does not have a global minimum. To increase the difficulty of the problem, η is included in the sum. Thus, μGP is requested to minimize 30 random variables while simultaneously searching for the minumum of a function with 30 variables.

$$f_{DJ4}(x) = \sum_{i=1}^{30} (i \cdot x_i^4 + \eta), \qquad x_i \in [-1.28,\ 1.28] \tag{12.2}$$

The global minimum is:

$$f_{DJ4}(x^*) = 0 \qquad for \quad x^* = (0, 0, ..., 0). \tag{12.3}$$

While locating the zone near the minimum is trival, finding the exact global optimum is difficult. In that zone the random component η is in the same order of magnitude of $\sum_{i=1}^{30} i \cdot x_i^4$, so fitness values oscillate in the range $(0,\ 1)$. Selection and

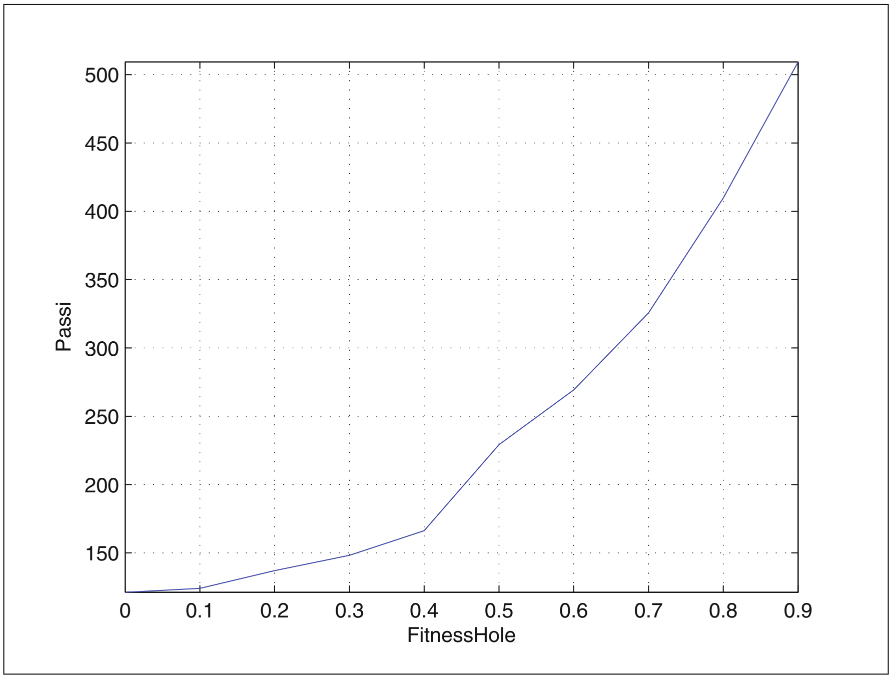


Fig. 12.4 Influence of `FitnessHole` on the number of steps needed to converge.

reproduction of individuals are thus strongly influenced by the constantly changing environment. An appropriate choice of μ and λ parameters can help the convergence even in this hard situation.

12.2.2.1 The μ and λ parameters

Finding the ideal value of μ and λ is a problem-specific issue. In the results of a series of experiments reported in Fig. 12.6, we can notice how the number of generation steps decreases rapidly for increasing values of λ. The number of evaluations, however, drops to a minimum and then slowly climbs up: the correct value for λ should always be chosen in conjuction with the value for μ.

12.2.3 Carrom

The complex Carrom function shows how some genetic operators provided with μGP can significatively improve the behavior of the evolutionary algorithm. The function is also known as *Carrom table* and is defined by the subsequent equation:

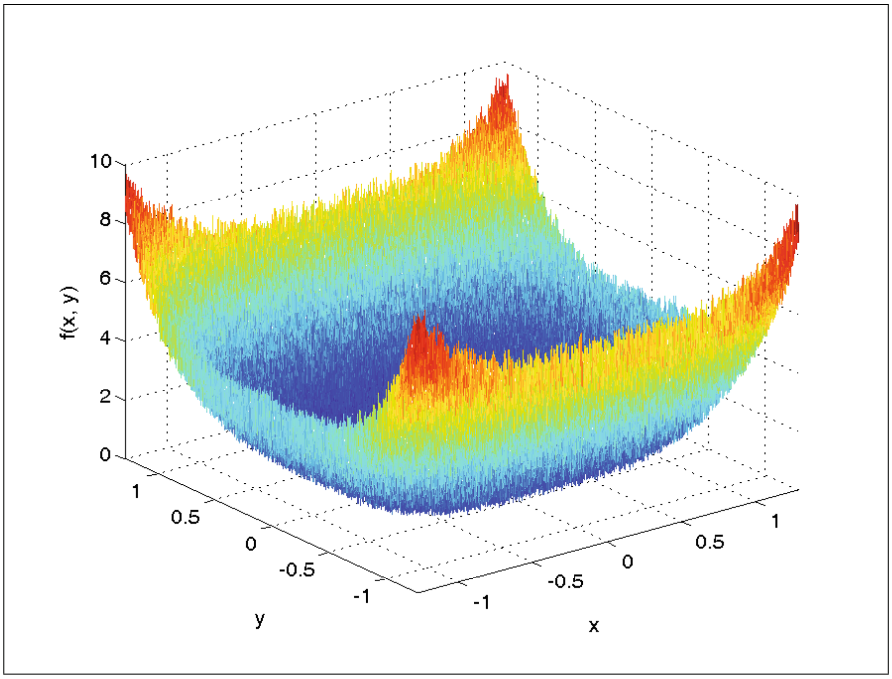


Fig. 12.5 Fourth De Jong function.

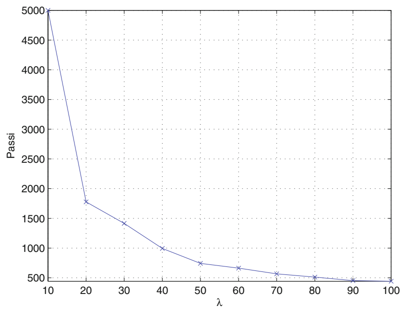
$$f(x) = -\frac{1}{30}\left\{\cos(x_1)\cos(x_2)\exp\left[\left|1 - \frac{(x_1^2 + x_2^2)^{\frac{1}{2}}}{\pi}\right|\right]\right\}^2 \tag{12.4}$$

The function is multimodal and it has four points of global minimum in the domain $x_i \in [-10, 10]$ with $i = 1, 2$ of value $f(x^*) \simeq -24.1568155$. The cartesian coordinates that identify the four points are $x^* = (x_1, x_2) \simeq (\pm 9.6463, \pm 9.6463)$.

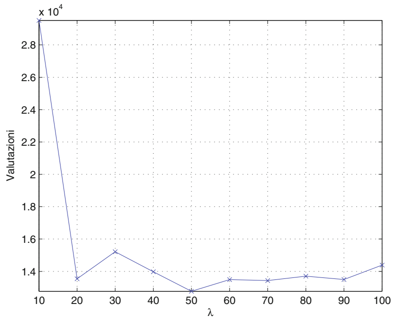
12.2.3.1 Preliminary run

In a preliminary series of runs, μGP is tested against the Carrom function with two different set of parameters, differing mainly for the population size. The objective is to provide the reader with statistics on the convergence when only two basic genetic operators are used. Parameters used can be found in Table 12.9 and Table 12.10. Table 12.11 reports the results of this first series of experiments.

The great number of evolutionary steps needed to find the solution are a clear indication of the complexity of the problem. Among the two sets of runs, experiment 2 presents slightly better results, probably because of the more favorable ratio $\frac{\tau}{\mu}$.



(a) Steps.



(b) Evaluations.

Fig. 12.6 Variation of parameter λ.

	Experiment 1	Experiment 2
Parameter	Value	Value
μ	100	300
ν	50	100
λ	33	33
σ	0.929	0.929
`maximumSteadyStateGenerations`	5 000	5 000
`MaximumAge`	20	20
`EliteSize`	2	2
$\tau = \tau_{min} = \tau_{max}$	64	64
ε	0.000001	0.000001
Seed	5987579	5987579

Table 12.9 Preliminary experiment. Population parameters.

12.2.3.2 Improvements with genetic operators

The behavior of μGP is then observed during the course of four different series of runs, each one with a different set of genetic operators. The initial activation probability of each operator is equal.

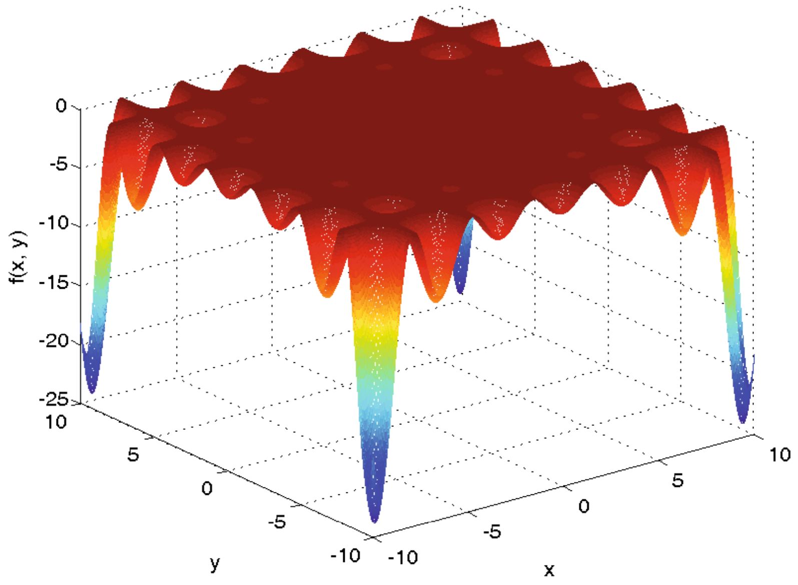


Fig. 12.7 Carrom function

Operator	Curr.	Min.	Max.
`replacementMutation`	0.5	0	1
`singleParameterAlterationMutation`	0.5	0	1

Table 12.10 Preliminary experiment. Genetic operators.

	Experiment 1			Experiment 2		
`Inertia`	Steps	Evaluations	Best Individual	Steps	Evaluations	Best
0.900	8783	145988	-24.156808	7339	135021	-24.156758
0.910	9922	166216	-24.156815	10420	174602	-24.156805
0.920	9972	155432	-24.156813	7338	130522	-24.156811
0.930	8915	145645	-24.156814	6839	124357	-24.156808
0.940	10990	176858	-24.156814	10398	171328	-24.156795
Average	9716.4	158027.8	-24.156813	8466.8	147166	-24.156795

Table 12.11 Preliminary experiment. Results.

The first experiment introduces two *crossover* genetic operators, *onePointSafeSimpleCrossover* and *twoPointSafeSimpleCrossover*. The complete list of genetic operators used is in Table 12.12. The results in Table 12.13 show a clear improvement: the duration of a single run decreases by about 30%, while the number of individuals evaluated drops to a 50%.

Operator	Curr.	Min.	Max.
`replacementMutation`	0.25	0	1
`singleParameterAlterationMutation`	0.25	0	1
`onePointSafeSimpleCrossover`	0.25	0	1
`twoPointSafeSimpleCrossover`	0.25	0	1

Table 12.12 Improvements with genetic operators, experiment 1. Genetic operators.

`Inertia`	Steps	Evaluations	Best
0.900	5014	55948	-24.156815
0.910	6778	110320	-24.156793
0.920	5955	67093	-24.156815
0.930	6239	105589	-24.156768
0.940	8474	127544	-24.156799
Average	6492	93298.8	-24.156798

Table 12.13 Improvements with genetic operators, experiment 1. Results.

12.2.3.3 The `alterationMutation` genetic operator

Not all genetic operators, however, are beneficial to the convergence a specific problem: the user should always take care when choosing the operators to include. Some of them may actually worsen the performance of μGP, because they are not able to work on the structure of the individual. For example, *alterationMutation* changes a number of different nodes in the individual with random values: but it fails if the node has no parameters. In the structure of our individual, both the prologue and the epilogue are empty macros with no parameters, so the operators fails often. The great number of failures of *alterationMutation* has direct repercussion on the steps needed to reach an optimal solution. Operators used in this test are presented in Table 12.14, while the results are in Table 12.15.

Operatore	Curr.	Min.	Max.
`replacementMutation`	0.2	0	1
`singleParameterAlterationMutation`	0.2	0	1
`alterationMutation`	0.2	0	1
`onePointSafeSimpleCrossover`	0.2	0	1
`twoPointSafeSimpleCrossover`	0.2	0	1

Table 12.14 *alterationMutation*, experiment 2. Genetic operators.

`Inertia`	Steps	Evaluations	Best
0.900	10182	148224	-24.156811
0.910	10203	145705	-24.156807
0.920	5780	98916	-24.156739
0.930	10492	148785	-24.156815
0.940	7708	118161	-24.156814
Average	8873	131958.2	-24.1567972

Table 12.15 *alterationMutation*, experiment 2. Results.

12.2.3.4 The `randomWalk` genetic operator

randomWalk is one of the genetic operators unique to μGP: basically, it is used to introduce more determinsm into individual creation, thus helping the algorithm to reach an optimal solution in all problems where at least a part of the fitness landscape can be explored effectively by a hill-climber algorithm. *randomWalk* proves its efficacy in this problem, cutting the time needed to reach the optimal solution by an order of magnitude, as shown in Table 12.17. In Table 12.16 all the operators used in this test.

Operatore	Curr.	Min.	Max.
`replacementMutation`	0.166	0	1
`singleParameterAlterationMutation`	0.166	0	1
`alterationMutation`	0.166	0	1
`onePointSafeSimpleCrossover`	0.166	0	1
`twoPointSafeSimpleCrossover`	0.166	0	1
`randomWalkMutation`	0.166	0	1

Table 12.16 *randomWalk*, experiment 3. Genetic operators.

`Inertia`	Steps	Evaluations	Best
0.900	127	4185	-24.156816
0.910	135	4206	-24.156815
0.920	147	4474	-24.156815
0.930	161	4939	-24.156815
0.940	134	4216	-24.156815
Average	140.8	4404	-24.1568152

Table 12.17 *randomWalk*, experiment 3. Results.

scanMutation is the equivalent of *randomWalk* for integer parameters. It has the same behavior, but it should be used in problems where the macros composing an individual present a great number of integer parameters.

12.3 Complex individuals' structures and evaluation: Bit-counting in Assembly

While arithmetic functions are useful to quickly identify the long-term and short-term effects of population parameters on the evolution, the examples reported in 12.2 share a common, trivial structure of the individuals. To explore some of the expressive potential of μGP constraints file, nothing is better than code generation, a task which the first version of the evolutionary algorithm was developed to tackle.

The aim of this experiment is to make μGP evolve an assembly function able to correctly count the number of bits set to 1 in the integer passed as an argument to the function. Not only the representation of individuals is not as intuitive as it would be in an arithmetic function, but the evaluator is not provided and needs to be conceived with care to effectively solve the problem.

12.3.1 Assembly individuals representation

The first thing that comes to mind when thinking about assembly are the single instructions. μGP provides a powerful mean to define a type of parameter that assumes a finite number of fixed values, `TypeDefinitions`.

```
<typeDefinitions>
  <item xsi:type="constant" name="register">
     <value>%eax</value>
    <value>%ebx</value>
    <value>%ecx</value>
    <value>%edx</value>
  </item>
  <item xsi:type="constant" name="instruction">
    <value>addl</value>
    <value>subl</value>
    <value>movl</value>
    <value>andl</value>
    <value>orl</value>
    <value>xorl</value>
    <value>test</value>
    <value>cmp</value>
  </item>
  <item xsi:type="constant" name="branch">
    <value>ja</value>
    <value>jz</value>
    <value>jnz</value>
    <value>je</value>
    <value>jne</value>
```

```
        <value>jc</value>
        <value>jnc</value>
        <value>jo</value>
        <value>jno</value>
      </item>
    </typeDefinitions>
```

It can be easily noticed in code above that not all the instructions have been included in the same `item`. This is a choice, to easily define different macros with different kinds or number of arguments (e.g., all instructions in `item` *instruction* have two parameters, while all instructions in *branch* only take one parameter). It is also useful to create an `item` for register names, since they will be extensively used in the macros.

The prologue of our individual will be a series of fixed assembly commands common to all functions: values stored in some registers are saved, then the integer passed to the function is put into register `%eax`. The same holds true for the epilogue: a series of fixed commands that return the result to the calling function and restore the initial values in the registers.

```
       <prologue id="sectionPrologue">
          <expression>
.globl foo
        .type   foo, @function
foo:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $4, %esp
        movl    8(%ebp), %eax
        movb    %al, -4(%ebp)
        pushl   %ebx
        pushl   %ecx
        pushl   %edx
        </expression>
      </prologue>
      <epilogue id="sectionEpilogue">
        <expression>
        popl    %edx
        popl    %ecx
        popl    %ebx
        leave
        ret
        .size   foo, .-foo
        </expression>
      </epilogue>
```

The macros, on the other hand, need to express all possible assembly instructions. The macros chosen are reported in the code below. `instrDirectDirect` represents all the instructions with two register as parameters, while `instrConstDirect` expresses those instructions which have a register and an integer as parameters. `branchCond` describes all instruction that jump to a label. Notice that the argument here is a label that refers to a specific line in the individual, and will be automatically created by μGP when individuals are instantiated.

```
<macros maxOccurs="infinity" minOccurs="1"
                 averageOccurs="70" sigma="60">
  <macro id="instrDirectDirect">
    <expression>  <param ref="ins"/>
                  <param ref="sreg"/>,
                  <param ref="dreg"/>
    </expression>
    <parameters>
      <item xsi:type="definedType" ref="instruction"
            name="ins" />
      <item xsi:type="definedType" ref="register"
            name="sreg" />
      <item xsi:type="definedType" ref="register"
            name="dreg" />
    </parameters>
  </macro>
  <macro id="instrConstDirect">
    <expression>  <param ref="ins"/>
                  $<param ref="scon"/>,
                  <param ref="dreg"/>
    </expression>
    <parameters>
      <item xsi:type="definedType" ref="instruction"
            name="ins" />
      <item xsi:type="integer" base="dec" minimum="0"
            maximum="255" name="scon" />
      <item xsi:type="definedType" ref="register"
            name="dreg" />
    </parameters>
  </macro>
  <macro id="branchCond">
    <expression>  <param ref="ins"/>
                  <param ref="target"/>
    </expression>
    <parameters>
      <item xsi:type="definedType" ref="branch"
            name="ins" />
      <item xsi:type="innerGenericLabel" name="target"
            itself="true" prologue="true"
            epilogue="true"/>
    </parameters>
  </macro>
</macros>
```

12.3.2 Evaluator

The evaluator for this particular experiment needs to be designed with the final objective in mind. It is important to provide μGP with a fitness landscape as smooth as possible. The evaluator is composed by two parts:

- A `main` function written in C;
- A script that compiles an assembly file with the `main` and runs it.

```
FITNESS\_FILE=fitness.output
rm -f foo fitness.error.log \$FITNESS\_FILE
gcc -o foo main.o \$1 -lm 2\>fitness.error.log
./foo 2\>fitness.error.log \>\$FITNESS\_FILE
if [ -s fitness.error.log ]; then
    echo \"0\" \> \$FITNESS\_FILE
    date \>\> error.log
    more fitness.error.log \>\> error.log
fi
```

Each individual is passed to the evaluator as an argument: the evaluator compiles it with the `main` in C and runs the resulting executable. As it is clear by the code reported above, the program calls the assembly function generated by μGP with a range of different integers, rewarding the individual if it computes correctly the number of bit set to 1 in the binary representation of the integer used as an argument.

It is important to notice that the fitness function has been conceived to be smooth: an individual is rewarded even if it comes near to the number of bits set to 1 for a particular integer. The presence of a slope towards the optimum helps the evolutionary algorithm to reach the best solution.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stdlib.h>
#include <setjmp.h>
#include <signal.h>
#include <sys/time.h>

char            foo(char a);
```

```
static void     timeout(int);

int             bits[] = {
    0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4,
    1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
    1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
    2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
    1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
    2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
    2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
    3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,
    1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
    2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
    2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
    3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,
    2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
    3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,
    3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,
    4, 5, 5, 6, 5, 6, 6, 7, 5, 6, 6, 7, 6, 7, 7, 8
};

static jmp_buf  wakeup_place;
static struct itimerval new;

int main(int argc, char *argv[])
{
    int             t;
    int             diff;
    float           fitness;

    if (!setjmp(wakeup_place)) {
        signal(SIGVTALRM, timeout);
        new.it_interval.tv_sec = 0;     /* next value, seconds */
        new.it_interval.tv_usec = 0;    /* next value, milliseconds */
        new.it_value.tv_sec = 0;        /* current value, seconds */
        new.it_value.tv_usec = 500;     /* current value, milliseconds */
        setitimer(ITIMER_VIRTUAL, &new, NULL);
        fitness = 0.0;
        t = 0;
    } else {
        t = 256;
    }
    while (t < 256) {
        diff = bits[t] - foo((char)t);
        if(diff > 0)
```

```
            diff = -diff;
        if (!diff) {
            fitness += 100.0;
        } else {
            fitness += exp(diff);
        }
        ++t;
    }
    printf("%f\n", fitness);

    return 0;
}

static void timeout(int foo)
{
    longjmp(wakeup_place, 1);
}
```

The C code for the `main` function that calls the assembly individuals implements also a timeout, since an individual could potentially fall into an infinite loop, given the `branchCond` macro and the fact that the initial population and subsequent mutations are randomly determined. If an individual triggers the timeout, it will have the fitness computed up to that moment. The presence of infinite loops should be always taken into account when considering code generation by evolutionary means.

12.3.3 Running

This experiment is quite complex. Depending on the hardware at your disposal, it could take hours to converge: μ and λ of the population should be dimensioned accordingly. In our experience, on a home desktop computer, it takes about 50 minutes to reach an optimal solution on 3 bits, with $\mu = 70$ and $\lambda = 35$. Finding an assembly program able to correctly evaluate all integers represented by more bits would require a bigger population and it would surely take more time.