GenetiCHR tutorial

Christian Theil Have
cth@ruc.dk

November 9, 2009

1 Introduction

GenetiCHR is a library for genetic computing written in CHR/Prolog. It is designed primarily to be simple to understand and easy to use, performance being a secondary concern.

Implementing a genetic algorithm in GenetiCHR is a simple matter of implementing a few callback functions that must implement crossover, mutation, fitness calculation and generation of the initial generation. The library contains a few common crossover and mutation functions for lists/strings and trees.

The library gives the user a number of choices with regard to parameters of the genetic algorithm such as generation size, fitness threshold and selection method (tournament or elitism).

The framework is available for download at,

`http://github.com/cth/GenetiCHR`

2 Overview of the framework

The core of the framework is in the file `genetichr.pl`. In the constraint a generation of individuals are maintained as `individual/4` constraints. These constraints are iteratively refined in each iteration of the algorithm. An iteration has three phases. The current phase is indicated by the `phase/1` constraint.

1. Mutation phase: A percentage of individuals in current generation is randomly selected for mutation. The mutation callback rule is called and the mutated individuals are inserted into the current generation.

2. Crossover phase: A percentage of pairs of individuals are selected for crossover. The crossover callback rule is called and two offspring individuals are inserted into the current generation. Note that also mutated individuals may be selected for crossover.

3. Selection phase: After mutation and crossover the number of individuals in the current generation m, exceeds the allowed population size, k. In

the selection phase k individuals are selected for survival and $m - k$ individuals are eliminated. The choice of which individuals should survive depends on the scheme indicated `selection_mode/1` - which may be either `elitism` or `tournament`. Both schemes uses the fitness callback rule to rank individuals.

After selection, there might be an individual with a fitness above the threshold or the generation threshold might have been reached. In both cases, the algorithm terminates and prints the individuals of the current generation. If neither is the case, the process loops, starting the next iteration with a new mutation phase.

2.1 Initialization of the framework

Before starting the evolutionary process certain parameters must be set by the user. This is done by adding certain constraints to the store:

- `population_size/1` - an integer which determines how many individuals survive each generation.
- `mutation_rate/1` - a floating point number [0..1] which is the probability with which individuals are selected for mutation.
- `crossover_rate/1` - a floating point number [0..1] which is the probability with which a particular pair of individuals are selected as parents in a crossover.
- `fitness_threshold/1` - an number float/integer. When the fitness of any individual reaches/exceeds this number, the algorithm terminates.
- `generation_threshold/1` - an integer number determines the maximal number of iterations that the algorithm will run for.
- `selection_mode/1` - see section 2.1.1.

2.1.1 Selection modes

A selection mode is set by adding a `selection_mode` constraint to the store. The two choices are `selection_mode(elitism)` and `selection_mode(tournament)`.

With elitism, the k best individuals of the generation (after mutation and crossover) are deterministically selected for survival, where k is the population size indicated with the `population_size/1` constraint.

In tournament selection k "tournaments" are created where the individuals are pitted against each other. Each individual is assigned to a random tournament. For each tournament, the individual with the highest fitness is selected for survival. Note that this kind of selection has a stochastic element and might therefore eliminate very fit individuals which would have survived in elitism. This can be an advantage as it may allow the algorithm to escape from local minima and explore the search space better.

The optimal choice of selection scheme depends on the nature of the problem.

2.2 Definition of callback rules

Whenever a new individual needs to be created, the fitness of an individual needs to be calculated, an individual needs to be mutated or a pair should be crossed over, callback rules defined by the user are used to accomplish the goal. To implement an genetic algorithm using the framework, four callback rules needs to be implemented.

- `cb_create_individual(-Genome)` : Called to create individuals in the initial generation. This will be called as many times as there are individuals in the population (indicated by `population_size/1`, see section 2.1). It is the job of the user to implement the rule such that `Genome` is unified to a representation of an individual.
- `cb_calculate_fitness(+Genome,-Fitness)` : Called in the selection phase to calculate the fitness of an individual. `Fitness` must unify to a number (either float or integer). Higher numbers indicate better fitness.
- `cb_mutate(+Genome, -MutatedGenome)` : Called in the mutation phase when individuals are selected for mutation. `MutatedGenome` must be unified to a mutated genome, but the mutation method is entirely up to the user. For a choice of common mutation rules, see section 2.3
- `cb_crossover(+Parent1,+Parent2,-Child1,-Child2)` : Called to crossover two individuals. The process yields two children. The crossover method up to the user. Some commonly used crossover rules are available as a convenience, see section 2.4.

2.3 Mutation rules

The library contains some rules which implements some basic approaches to mutations. These mutation rules assumes that the genome is represented as Prolog list. These rules are implemented in the file `mutation.pl`.

`single_point_mutate(-Genome,+MutatedGenome,-Alphabet)`
One random element of the `Genome` list is mutated to one of the elements of the `Alphabet` list, selected at random.

```
multi_point_mutate(-Genome,+MutatedGenome,
                   -Alphabet,-PointMutationProb)
```

Any element of the `Genome` list is mutated with a probability of `PointMutationProb`. The mutation outcomes are randomly selected from the `Alphabet` list with uniform probability.

2.4 Crossover rules

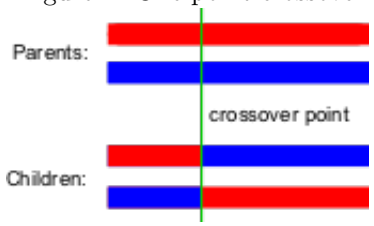
Some common crossover approached have been implemented as part of the framework. To use them, just call them from the user-defined `cb_crossover_callback` function.[1]

2.4.1 One-point crossover

`one_point_crossover(-MotherGenome,-FatherGenome,+Child1Genome,+Child2Genome)`

One point cross over assumes equal length of genomes. A single crossover point on both parents' organism strings is selected. All data beyond that point in either organism string is swapped between the two parent organisms. The resulting organisms are the children, see figure 1.

Figure 1: One-point crossover

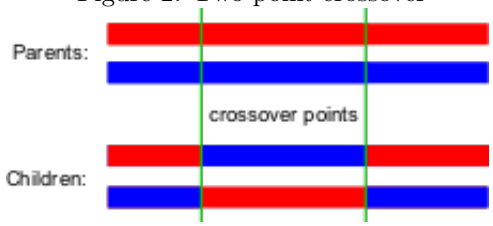


2.4.2 Two-point crossover

`two_point_crossover(-MotherGenome,-FatherGenome,+Child1Genome,+Child2Genome)`

Two-point crossover calls for two points to be selected on the parent organism strings. Everything between the two points is swapped between the parent organisms, rendering two child organisms. See illustration in figure 2

Figure 2: Two-point crossover



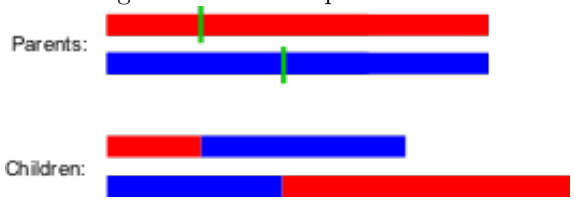
[1] Some of the contents of this section has been derived from wikipedia entry on crossover functions.

2.4.3 Cut-and-splice crossover

`cut_and_splice(-MotherGenome,-FatherGenome,+Child1Genome,+Child2Genome)`

The "cut and splice" approach, results in a change in length of the children strings. The reason for this difference is that each parent string has a separate choice of crossover point. Illustration is shown in figure 3.

Figure 3: Cut-and-splice crossover



2.4.4 Uniform crossover

```
uniform_crossover(-MotherGenome,-FatherGenome,
                  +Child1Genome,+Child2Genome,-SwapProbability)
```

In the uniform crossover scheme individual elements in the genome lists are compared between two parents. The elements are swapped with a fixed probability (`SwapProbability`). A typical choice for `SwapProbability` is `0.5`.

2.4.5 Binary tree crossover

`binary_tree_crossover(-Parent1,-Parent2,+Child1,+Child2)`

This method works on (binary) trees rather lists and is particularly useful for genetic programming in a lisp like setting. A random node (subtree) is selected in each of the parent trees. In the child trees the these subtrees are then swapped. See figure 4 for an illustration.

Figure 4: Binary tree crossover

```
Parent1: [mul,2,[plus,a,b]]
Parent2: [plus,[mul,a,b],a]
Child1: [mul, 2, b],
Child2: [plus, [mul, a, [plus, a, b]], a]
```

3 A simple example: Evolution of a string

This example is trivial and not a very interesting application of genetic algorithms. However, it illustrates the basic principles of developing a genetic algorithm using GenetiCHR. The example may be found in the GenetiCHR library in `examples/evolve_string/`.

The idea is to evolve a number of individual strings, represented as a list of character codes, into the string we are looking for. We define the target string to be,

```
target_string("cafebabe").
```

Also, we define the alphabet we wish to use for these strings. This delimits the search-space,

```
characters("abcdef").
```

3.0.6 Definition of callback rules

Four callback rules for creation of individuals, mutation, crossover and fitness calculation, needs to be defined. We go through each in turn.

Creation of individuals,

```
cb_create_individual(Genome) :- create_random_string(Genome).
```

This rule is called every time a new individual should be created for inclusion in the initial generation. `population_size/1` determines how many times it will be invoked. It just calls `create_random_string/1` which creates a random list of character codes from the defined alphabet,

```
create_random_string([]) :-  random(Prob),  Prob < 0.1, !.  % Stop
create_random_string([Char|Rest]) :-
  create_random_character(Char),
  create_random_string(Rest).

create_random_character(Char) :-
  characters(Chars),
  random(P),
  length(Chars,NumChars),
  CharIndex is round(P * (NumChars-1)),
  nth0(CharIndex,Chars,Char).
```

Mutation uses the library `multi_point_mutate` rule (see section 2.3),

```
cb_mutate(Genome, MutatedGenome) :-
  characters(C),
  multi_point_mutate(Genome,MutatedGenome,C,0.1).
```

After a call to `cb_mutate`, `MutatedGenome` will have mutations in 10 percent of all its bases on average. Note that bases are only mutated to characters from the defined alphabet.

Crossover uses the library `cut_and_splice` rules, see section 2.4.3.

```
cb_crossover(GenomeFather,GenomeMother,ChildGenome1,Child2Genome) :-
  cut_and_splice(GenomeFather,GenomeMother,ChildGenome1,Child2Genome).
```

Note that we cannot easily use one-point crossover, which assumes that the parent genomes are of equal length, since in our initial generation individuals are generated with random lengths.

Fitness calculation is calculated as the negative Levenshtein/edit distance[2] to the target string,

```
cb_calculate_fitness(Genome,Fitness) :-
  target_string(Target),
  atom_codes(Atom1,Target),
  atom_codes(Atom2,Genome),
  levenshtein(Atom1,Atom2,Distance),
  Fitness is 0 - Distance.
```

Initialization To initialize the algorithm we create a rule which adds all the necessary constraints as indicated in section 2.1:

```
run_example :-
  population_size(50),
  mutation_rate(0.1),
  crossover_rate(0.2),
  fitness_threshold(0),
  generation_threshold(1000),
  selection_mode(elitism),
  phase(initialization).
```

Note, that the last constraint `phase(initialization)` effectively starts the algorithm with a first generation.

We can now start the process by loading the file and calling the goal `run_example`.

3.1 Running the example

The example runs in SWI prolog. It may be a good idea to load SWI with a large global and local stack, e.g.

```
swipl -G128M -L128M
```

[2]The Levenshtein distance is defined as the minimal number of character substitutions, insertions and deletions needed to transform one string into an other.

Once we have started SWI prolog, we can load the example,

```
?- [evolve_string].
```

We can now run the example by calling the goal `run_example`:

```
?- run_example.
cb_create_individual: debdbcfbfcece
cb_create_individual: deeeedbdc
....
cb_create_individual: ba
cb_create_individual: eec
Generation 0
  - mutation phase
  - cross over phase
  - selection phase
Evolution cycle done. Starting on generation 1
Best individual: individual(64, 1, [97, 101, 98, 98], -4)
Average fitness of generation: -6.26
  - mutation phase
  - cross over phase
  - selection phase
Evolution cycle done. Starting on generation 2
Best individual: individual(204, 2, [99, 99, 100, 101, 98, 98, 101], -3)
Average fitness of generation: -5.62
  - mutation phase
  - cross over phase
  - selection phase
...
...
Evolution cycle done. Starting on generation 27
Best individual: individual(2538, 26, [99, 97, 102, 97, 101, 98, 97, 98, 101], -1)
Average fitness of generation: -2.24
  - mutation phase
  - cross over phase
  - selection phase
Individual with fitness threshold 0 found. Terminating.
Best individual: individual(2720, 28, [99, 97, 102, 101, 98, 97, 98, 101], 0)
Average fitness of generation: -2.14
```

The average fitness of each generation is reported and the best individual in the generation is displayed in each iteration. Note that an `individual` constraint has four elements: The first is the unique id of the individual, the second is the generation in which the individual was created, the third is the genome of the individual and the fourth is its fitness.

Finally, in generation 28, a suitable individual is found, causing the process to terminate. You can verify that the sequence of character codes correspond to the string we were looking for using,

```
?- atom_codes(String,[99, 97, 102, 101, 98, 97, 98, 101]).
String = cafebabe.
```