Distributed Genetic Programming Framework

The DGPF-Project
A presentation for interested students
Index

- What is the DGPF-project?
- Common Search API
- Genetic Algorithms and their distribution
- Genetic Programming
- The Teams and their Tasks
What is the DGPF-project?

- Genetic Programming Framework
- Target: Engineer Programs for small, interlinked devices like Sensor Nodes genetically
- The DGPF is a distributed application itself
- Open Source, LGPL-Licensed, Sourceforge-Project
- CVS-Repository: `ext:user-name@cvs.sourceforge.net:/cvsroot/dgpf`
The DGPF’s structure

- The DGPF-Framework employs a layered approach

```
org.dgpf.gp
org.dgpf.gp.netautomaton
org.dgpf.gp.auomaton
org.dgpf.ga
org.dgpf.search
org.sfc
```

Genetic Programming

Network Simulator

Genetic Algorithms

Simulated Annealing

Search API

SFC (Helper Classes)
SFC (Helper Classes)

- Event-Handling: `org.sfc.events`
- GUI: `org.sfc.gui`
- IO: `org.sfc.io`
- Mathematics / Statistics: `org.sfc.math`
- Client/Server and P2P Utils: `org.sfc.net.[cs, p2p]`
- Parallelization: `org.sfc.parallel`
- XML: DOM, SAX (SAX-Writer), XHTML, CSS, MathML, SVG: `org.sfc.xml.[dom, sax, formats]`
Common Search API

- The abstract base-class `SearchEngine` provides basic behavior (main loop, start/stop, event generation, checkpoints)

- Search Context: Executes search tasks, such as mutation and simulation. Reusable in arbitrary search algorithms.

- Utilities for the multi-dimensional, multi-objective search.

`org.dgpf.search`

**Search Control Loop**
Common Search API

- Adaptation utilities: All searches are able to adapt their parameters (automatically or manually)
- The search is divided in so-called “updates”. After each update (i.e. a generation in GA) the `SearchState` will be updated (i.e. with the fitness of the currently best individual).
- The `SearchParameters` are then updated with the new state. If set, they use the current `AdaptationStrategy` for automatic parameter adaptation.
- The new parameters are stored in the search state, which then influences the next search “update” (i.e. by defining the population size in GA)
Common Search API

- The simulation and reproduction of individuals is handled by instances of `SearchContext`.
- A `SearchContext` must only be used by one Thread, but many `SearchContext`s may exist in parallel.
- All search algorithms use `SearchContext`s.
- The simulation and reproduction routines of `SearchContext`s are overridden/provided for each problem domain.
- Search algorithms are now directly comparable.
Common Search API

- Searches are multi-dimensional / multi-objective: More than one parameter will be optimized (by more than one FitnessFunction).
- Therefore we have vectors of fitness values instead of single fitness values.
- How to compare vectors? Simple sums are inefficient.
- We use user-defined/predefined comparators, derivates of IndividualComparator.
- For example comparators see package org.dgpf.search.comparators.
Common Search API

- The evaluation of individuals is done by (multiple) `FitnessFunctions`.
- `FitnessFunctions` themselves are stateless and can be used by many `SearchContexts` in parallel.
- The state of `FitnessFunctions` is stored in `FitnessData` records. Each fitness function may provide an own implementation of `FitnessData`, allowing to store arbitrary data.
- Each `SearchContext` manages the `FitnessData` records for all `FitnessFunctions` it uses.
Common Search API - Individual Evaluation

- **SearchContext (c)**
  - `evaluate(individual i)`
  - `begin_simulation(i)`
  - `simulate(i, steps)`
  - `end_simulation(i)`

- **FitnessFunction [1..n]**
  - `should_simulate(i)`
  - `begin_simulation(fd, c)`
  - `inspect(fd, c)`
  - `end_simulation(fd, c)`

- **FitnessData (fd)**
  - `fill_in_fitness(i)`
  - `set_fitness(xx)`
  - `set_fitness(xx)`
  - `set_fitness(xx)`

The diagram illustrates the flow of evaluation steps in a distributed genetic programming framework, showing how individual evaluation, simulation, and fitness data are handled.
DISTRIBUTED SYSTEMS

Common Search API

- The individual reproduction can be separated into SearchTasks (SearchNew, SearchMutate, SearchCrossover) which can be stored in a SearchTaskQueue and from there extracted and executed by SearchTaskThreads.

- For all searches, there is one default distribution model: The search’s control loop runs locally, the work of individual reproduction and evaluation is done remotely in parallel by many servers. Therefore we call this approach Client/Server (also common: Master/Slave).

- SearchTasks can be distributed by a SearchClient to arbitrary many SearchServers.
Common Search API

A SearchWorkerThread executes tasks locally.

A SearchTaskThread can put them into a SearchClient, which distributes them.
The GA-package provides different distribution methods for arbitrary Genetic Algorithms.

Abstract Base Class: `GeneticEngine`
Genetic Algorithms - Local

- Simple problems can be solved best on one machine without distribution.
- `LocalGeneticEngine`

local machine
Genetic Algorithms – Client/Server

- The evaluation of individuals usually involves simulations. Simulations can become complicated and time consuming.
- The evaluation and replication phases can be distributed to servers (slaves) and performed in parallel.

**CSGeneticEngine**

org.dgpf.ga.cs
Some problems can only be solved using large populations. A single machine may be insufficient to hold all individuals needed.

Multiple Genetic Engines may be interlinked using a peer-to-peer network to form large virtual populations. (Individuals are exchanged during Selection phase)

**P2PGeneticEngine**

org.dgpf.ga.p2p
Genetic Algorithms – Hybrid Distribution

- A hybrid distribution creates large virtual populations which can be evaluated by complicated/time consuming simulations.

- **P2PCSGeneticEngine**
  
  *P2PCSGeneticEngine* can also interact with **P2PGeneticEngine**.
Genetic Programming

- Program-code evolved is Turing-complete
- Allows all possible algorithms to be evolved (in theory...)
- Reproduction uses optimization-cycle
- Reproduction uses knowledge about instructions and operator semantics
- Code is exchanged in a direct executable format
Genetic Programming

- Evaluation \(\equiv\) simulate program code and compare results with expectations
- Instantiate programs in automata (\(\equiv\) virtual hardware)
- Simulate hardware-ticks, perform introspections periodically

Program:

\[
\begin{align*}
@0: & \\
& \text{SendWord mem[0]} \\
& \text{SendMessage} \\
@1: & \\
& \text{mem[1]=ReceiveWord} \\
& \text{IfJump mem[1]<=mem[0], @1} \\
@2: & \\
& \text{mem[0]=mem[1]} \\
& \text{Goto @0}
\end{align*}
\]

org.dgpf.gp.atomaton
Genetic Programming

@0:
mem[0] = mem[0]
mem[1] = (mem[1] % mem[0])
mem[0] = (mem[0] - (mem[1] % mem[0]))
IfJump mem[1], @0
Sleep (mem[0] / mem[0])
IfJump mem[2], @0
mem[2] = (mem[0] / (0 / mem[0]))
Goto @0

@0:
mem[0] = mem[0]
mem[1] = (mem[1] % mem[0])
mem[0] = (mem[0] - mem[1])
Goto @0

@0:
mem[1] = 811
Goto @0

Fitness
Generation

0 5 10 15

0 5 10 15
Genetic Programming

- Normal GA and simple GP: continuous fitness functions
- Software Engineering: A program’s functionality might change dramatically if only one piece of code is modified slightly.
- Our GP: fitness function is not continuous, contains jump discontinuity
- Evolution is very complicated, often might not work at all
Genetic Programming – Network

- One Program used by multiple instantiated automata
- Network extension for virtual hardware
- State complemented by global network state

```
org.dgpf.gp.netautomaton
```

Network of Automata

Execution Status

```
<table>
<thead>
<tr>
<th>Data Memory</th>
<th>Execution Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>231</td>
<td>IP</td>
</tr>
<tr>
<td>-3</td>
<td>Lost</td>
</tr>
<tr>
<td>5532</td>
<td>Received</td>
</tr>
<tr>
<td>-1</td>
<td>Sent</td>
</tr>
<tr>
<td>39342</td>
<td>Lost</td>
</tr>
<tr>
<td>-234</td>
<td>Received</td>
</tr>
<tr>
<td>120</td>
<td>Sent</td>
</tr>
</tbody>
</table>
```

Network Status

```
MsgCount
Collisions
Lost
Time
```
Genetic Programming – Network

- The automata run at approximately the same speed (which cannot be regarded as constant)
- The system of automata runs asynchronously
- The automata will be started at different times
- The links between the nodes are randomly created (but no network partitions exist)
- Messages are simple sequences of memory words
- Messages are sent like radio broadcasts and will be received by every node in a specific distance
- Messages can get lost without special cause
- Transmissions may take a random time
- Collisions lead to message losses
Genetic Programming – Network

Fitness function

create random network

simulate the generated code on that network

compute the results the specified behavior would yield

compare both results

Repeat this $n$ times and calculate the average/minimum fitness.
DGPF – Examples

- The `examples` package holds examples of all things we just talked about.

- `examples.ga`: Examples for Genetic Algorithms. The functionality of the different GA distribution models is demonstrated by single-objective and multi-objective numeric approximations.

- `examples.gp.automaton`: An example is given that tries to find the Euclidian Algorithm for computing the ggd.

- `examples.gp.netautomaton`: Examples for network applications that compute the distributed maximum or try to find a list of all nodes in the network are shown.
Genetic Programming – Tasks

- Team 1: Create a GUI that allows the use to remotely control search engines / Genetic Algorithms / Genetic Programming and that helps evaluating their results.

- Provide a small web surface.

- Package: `org.dgpf.gui, org.dgpf.search.net`
Genetic Programming – Tasks

- Team 2: Find optimal adaptation strategies for GA, evaluate their performance using the GP examples (and standard benchmark functions).

- Implement other search algorithms using the Search API and compare the results.

- Packages: examples, org.dgpf.search.net, org.dgpf.xxx.adaption, org.dgpf.yyy
Genetic Programming – Tasks

- Meeting: Every two weeks.
- Structured Approach to work.