Vidyaa – Optimization Made Easy

Nicolas Rüegg, Urs Fässler

November 25, 2012

Abstract

Vidyaa is a tool for the optimization of parameter values. What the individual parameters represent is of no interest to the program itself which allows to optimize a neural network as well as the hardware parameters of a certain machine or any other set of parameters.

To distinguish how good a set of parameter values is, the parameters are tested using an external, user-defined program. This simulation software then returns a so called *fitness value* which is used by *Vidyaa* to decide on its further steps.

The optimization algorithm used can be freely selected by the experimenter and unlike most other optimization tools, *Vidyaa* allows the combination of several algorithms for the optimization. For instance, it is possible to optimize some parameters with an evolutionary algorithm, and some other parameters with a simulated annealing algorithm.

The user defines the optimization task by configuring so called experiments. These experiments can be combined in a hierarchical structure which allows the optimization of parameters on different levels with different algorithms.

This documentation is split into two parts:

- The **User Manual** describes how *Vidyaa* is used to get the desired results. It also describes the basic design of the software as far as it is necessary to the usage of *Vidyaa*.

- The **Programmer Manual** describes the architecture of *Vidyaa* and how extensions are implemented. It gives an overview of the most important concepts of *Vidyaa*. Although it discusses some implementation issues, it is not a replacement for the detailed documentation of the individual classes and their members. The detailed documentation of the classes and their members can be found on *http://vidyaa.origo.ethz.ch/*

  It is strongly recommended to read the *User Manual* first. It is frequently referenced and some concepts already commented on are not further explained in the *Programmer Manual*.

Contents

Part I

User Manual

Chapter 1

System Requirements

Vidyaa is known to work on the following platforms:

- Hardware
  - x86 architecture, 32 / 64 Bit
  - 512 MB Ram
- Operating System
  - Ubuntu
  - Debian GNU/Linux
- Libraries
  - gsl
  - gslcblas
  - xml++-2.6
  - xml2
  - glibmm-2.4
  - gobject-2.0
  - sigc-2.0
  - glib-2.0
- Compiler
  - GNU C++ Compiler, Version 4

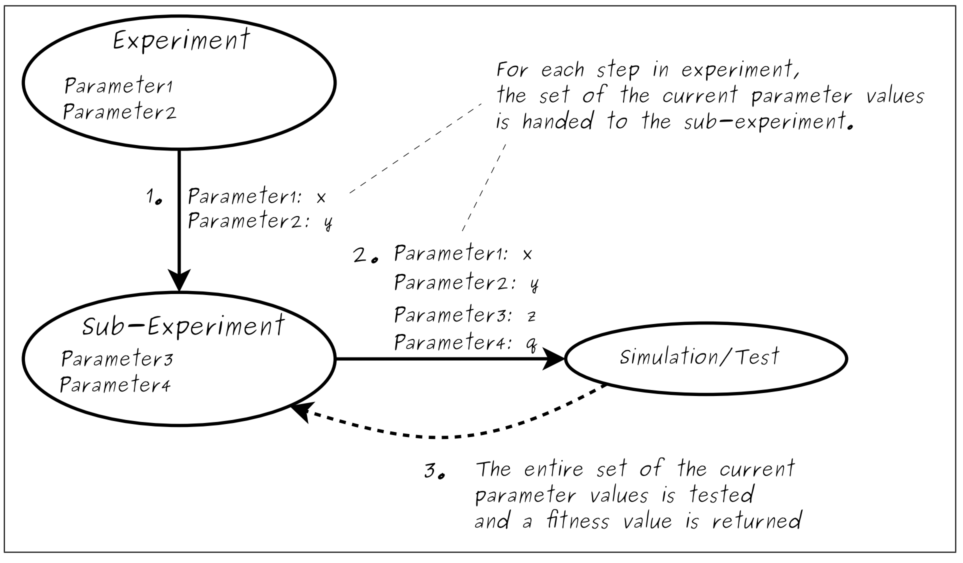
Chapter 2

Architecture Overview

This chapter describes the basic design of the software as far as it is necessary to the usage of *Vidyaa*. For a detailed documentation on the architecture, have a look at the *Programmer Manual*.

2.1 The Basic Idea

In *Vidyaa*, the user defines the optimization task by configuring so called experiments. An experiment contains the parameters to optimize and the algorithm to use for the optimization. Experiments can be combined in a hierarchical structure to allow the optimization of different parameters with different algorithms. From this follows that for every step of an experiment, the entire sub-experiment is conducted, too. This is illustrated in figure 2.1.



Experiment
Parameter1
Parameter2
For each step in experiment, the set of the current parameter values is handed to the sub-experiment.
1. Parameter1: x
Parameter2: y
2. Parameter1: x
Parameter2: y
Parameter3: z
Parameter4: q
Sub-Experiment
Parameter3
Parameter4
Simulation/Test
3. The entire set of the current parameter values is tested and a fitness value is returned

Figure 2.1: The optimization process

The experiment on the first level creates a set of parameter values and passes them to its sub-experiment. The sub-experiment then extends this set of parameter values and finally passes them to a simulator (or test program), which tests the current values and returns a fitness value. Based on the fitness value, the sub-experiment decides on its next step. When the sub-experiment is finished, it returns a fitness value to the experiment on the next higher level, which then also decides on its next step.

To put it differently, experiments pass their current parameter values to their sub-experiments. That is, a set of parameter values is passed down the hierarchy until it ends with a test of the parameter values. Every level in between can add or modify something.

This basic idea was implemented by using different types of experiments which are described in the next section.

2.2 The Experiment Types

In *Vidyaa*, the idea of an experiment usually is as follows:

First of all, by conducting an experiment, we usually optimize parameters of a given system. Whether that system is a robot or a neural network or any other optimizable set of parameters does not matter.

Second, we would like to be able to conduct optimization tasks within other optimization tasks. That is, we need an architecture where an experiment can contain other experiments and an experiment returns a value which can be used by the higher level experiments to make decisions. To achieve this, three basic types of experiments were defined: *Algorithm Experiments*, *Array Experiments* and *Simulator Experiments.*

In general, every type of experiment produces at least a fitness value as output. This fitness value is then used by the higher level experiment to make its decisions.

An *Algorithm Experiment* usually does the actual optimization. It expects a map of parameters with their range as input and produces a map of parameters with their values as output. The algorithm itself determines in which way the parameters are changed. So far, three algorithms were implemented: *Simulated Annealing*, an *Evolutionary Algorithm* and a *Systematic Algorithm.*

An experiment of type *Array* contains several experiments on the same hierachy level and defines how to handle the returned fitness values of the individual experiments.[1] For instance, it computes the average of the fitness values of the experiments it contains and returns this as its own fitness value.

Finally, an experiment of type *Simulation* is the actual test of the previously produced parameter values. The map of parameter values is written to a file and an external program is called which tests these values and returns at least a fitness value. This fitness value is then returned to the experiment on the next higher level. The *Simulation Experiment* is settled on the lowest level of the hierarchy and is necessary, to get a measurement for the quality of the current parameter values.

As an example, an *Algorithm Experiment* containing three other *Algorithm Experiments* of which each contains a *Simulation Experiment*, is represented in the hierarchy as shown in figure 2.2.

[1]Some say, *ExperimentContainer* would have been a more accurate name.

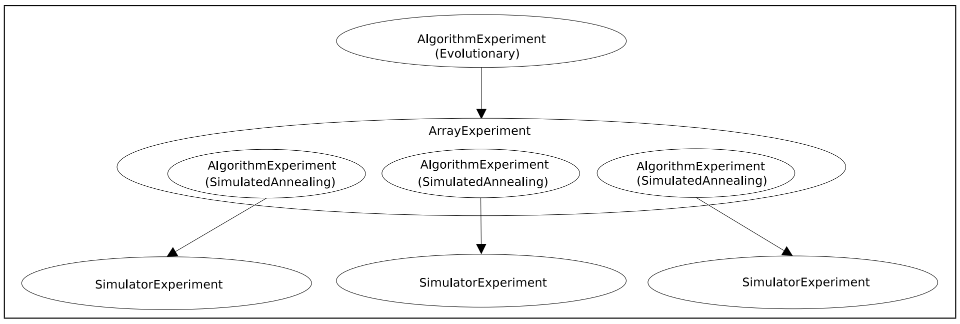


Figure 2.2: Hierarchy of the experiment types

At first glance, it might seem disturbing that these types are all called "experiment", although they mean different things. But if we refer to an optimization process as an experiment, it is indeed accurate.[2]

2.3 Experiment Configuration

The configuration of the experiment hierarchy is given by an XML configuration file. A typical configuration file which defines an *Algorithm Experiment* within another *Algorithm Experiment* is shown in listing 2.1.

[2]In case you disagree, just ignore it and accept the fact, that there are three different types of experiments.

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- the root node is always setting -->
<setting
  name="Exploration␣of␣the␣parameters"
  version="1.0"
  xmlns="http://ailab.ifi.uzh.ch/framework/2008/"
  xmlns:evol="http://ailab.ifi.uzh.ch/framework/2008/algorithm/evolution/"
  xmlns:siman="http://ailab.ifi.uzh.ch/framework/2008/algorithm/simulatedannealing/"
  xmlns:systematic="http://ailab.ifi.uzh.ch/framework/2008/algorithm/systematic/" >

 <!-- define the highest level experiment as an AlgorithmExperiment conducted with
      the evolutionary algorithm -->
 <algorithm type="Evolution">

        <!-- configuration of the evolutionary algorithm -->
        <evol:config
                changeprobability="0.50"
                distribution="gaussian"
                population="50"
                parents="5"
                generations="10"
                selection="leaders"
                elitism="on" />

        <!-- parameters to optimize in this experiment -->
        <parameters>
                <param name="FrontFemurLength" min="0.05" max="0.12"/>
                <param name="FrontTibiaLength" min="0.05" max="0.12"/>
                <param name="HindFemurLength" min="0.05" max="0.12"/>
                <param name="HindTibiaLength" min="0.05" max="0.12" />
        </parameters>

        <!-- a sub-experiment -->
        <algorithm name="sub-experiment" version="1.0" type="Simulated␣Annealing">

                <!-- configuration of the simulated annealing algorithm -->
                <siman:config
                        inittemperature="16"
                        terminatetemperature="2"
                        bestfitness="1.0"
                        cooling="0.95" />

                <!-- parameters to optimize in this sub-experiment -->
                <parameters>
                        <param name="C_Frequency" min="0.5" max="4"/>
                        <param name="C_Amp_F" min="-0.5" max="0.5"/>
                        <param name="C_Amp_H" min="-0.5" max="0.5"/>
                </parameters>

                <!-- the simulation software -->
                <simulation program="../../puppy_simulator/src/puppySimulator" />
        </algorithm>
 </algorithm>
</setting>
```

Listing 2.1: experiment.xml

As the listing demonstrates, the hierarchy of the experiments is also respected in the configuration file. More details on how to configure an experiment can be found in chapter 4.

2.4 Interface to the Simulator

Basically, the parameter values are tested in a completely serparated program which could also be executed as a standalone application. This program is called "the simulator". By definition, it takes an input file and returns an output file.

As indicated in figure 2.3, the simulation software is called by *Vidyaa* with the two arguments `INPUTFILE` and `OUTPUTFILE`. In this specific example, the input file is `parameters.txt` (see listing 2.2 for an example) and the output file is `results.txt` (see listing 2.3 for an example). These arguments specify from which file the parameters and their values are read and in which file the results are written by the simulation software. From *Vidyaa*'s point of view, this means the parameters are written into the `INPUTFILE` and the return values are read from the `OUTPUTFILE`.

Of course, the simulator needs to understand the parameters defined in the `INPUTFILE`. That is, if we specify a parameter named `PuppyWidth`, the simulator needs to know what to do with this parameter and how to apply its value internally.

```
PuppyLength 0.12
PuppyWidth 0.05
FrontFemurLength 0.0614236
FrontTibiaLength 0.0806978
HindFemurLength 0.119857
HindTibiaLength 0.0660368
C_Frequency 2.5
```

Listing 2.2: parameters.txt

Like the input file, the result file can contain strings and a corresponding value (for instance a parameter and its value). The only constraint is, that it must contain at least a parameter called `fitness` and the corresponding value. This is the fitness value which is returned to the next higher level.

```
fitness 1.079
speed 2.45
energy 12.348
```

Listing 2.3: results.txt

Since the simulation software is completely separated from *Vidyaa*, the simulation can do whatever it likes as long as it respects the interface.

Usually, the simulator has to be programmed by the user. Most of the time, the user will just write a small program to implement the interface to external applications such as a physics simulator or an application for the simulation of robots. Details on how to write a simulator can be found in section 6.3.

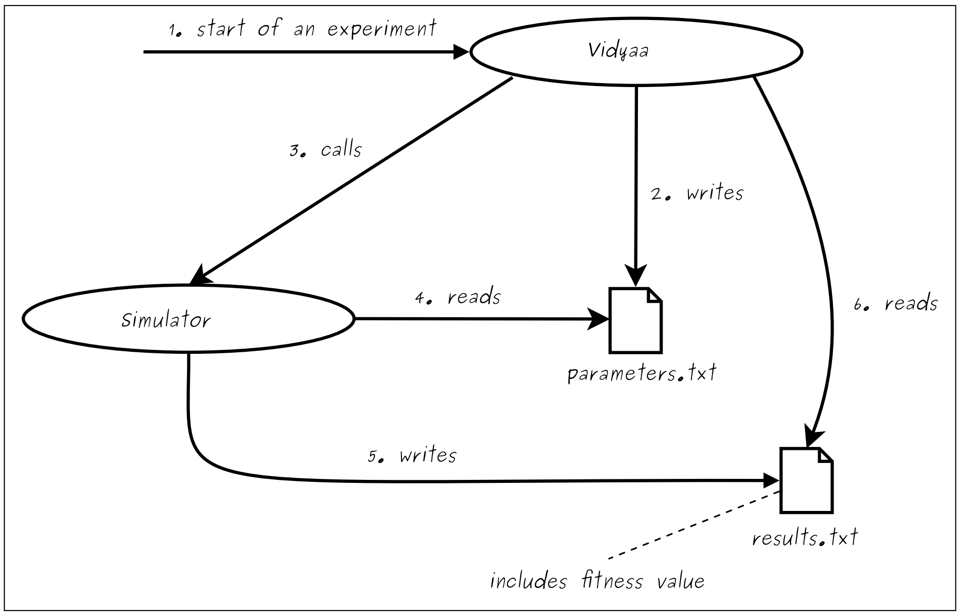


Figure 2.3: Interface between *Vidyaa* and the simulator

Chapter 3

Available Optimization Algorithms

3.1 Simulated Annealing

Simulated annealing is an algorithm to locate a good approximation to the global minimum of a given function.

The following definition is taken from wikipedia[1] and is a good description of the basic idea:

> Each step of the simulated annealing algorithm replaces the current solution by a random "nearby" solution, chosen with a probability that depends on the difference between the corresponding function values and on a global parameter T (called the temperature), that is gradually decreased during the process. The dependency is such that the current solution changes almost randomly when T is large, but increasingly "downhill" as T goes to zero. The allowance for "uphill" moves saves the method from becoming stuck at local minima – which are the bane of greedier methods.

Currently, there are two different versions of the simulated annealing algorithm implemented. The advanced version called "simulated annealing" is described in detail in [2].

The second version is simpler and thus called "textbook simulated annealing".

In this second version, an error function $E(p_k)$ of the parameters p_k is needed. It quantifies the error and is defined as follows:

$$E(p_k) = \quad 1 - \left(\tfrac{F(p_k)}{F_b}\right) \tag{3.1}$$

Whereas $F\left(p_k\right)$ is the fitness value of the parameters and F_b is the best possible fitness value.

The algorithm stops when the best fitness value or the termination temperature T_t is reached.

The aim of the algorithm is to reduce the error to zero. In each step, the new set of parameter values p'_k is calculated as follows:

$$r = T_k * (r_{\max} - r_{\min}) \tag{3.2}$$
$$R = \left[\max\left(p_k - \tfrac{r}{2}, r_{\min}\right), \min\left(p_k + \tfrac{r}{2}, r_{\max}\right)\right] \tag{3.3}$$
$$p'_k = x(R) \tag{3.4}$$

Described in words, this means r is the range of values of the parameter scaled with the current temperature. In equation 3.3, the interval R is defined. It is close to the previous value p_k and limited to the range of the parameter. Finally, the new parameter value p'_k is randomly selected from the interval R.

Subsequently, the change of the error is calculated:

$$\Delta E = E(p'_k) - E(p_k) \tag{3.5}$$

To decide whether the new parameter values are used or the old ones should be kept, a probability is calculated:

$$p(\Delta E) = \begin{cases} 1 & \text{for } \Delta E \leq 0 \\ \exp\left(-\frac{T_0}{T_k}\right) & \text{for } \Delta E > 0 \end{cases} \tag{3.6}$$

According to the following equation, the calculated probality is the probability, with which p'_k is chosen as the new parameter value:

$$p_{k+1} = \begin{cases} p'_k & \text{for } x\left([0,1)\right) < p(\Delta E) \\ p_k & \text{else} \end{cases} \tag{3.7}$$

Finally, the new temperature is calculated as $T_{k+1} = T_k * \gamma$ whereas γ is the cooling constant.

3.2 Evolutionary Algorithm

The evolutionary algorithm is based on the principles of the evolutionary theory. [1]

Every generation contains a certain number of individuals[2] of which some are selected as parents. Mutations (children) of the parents create the next generation.

The fitness of an individual is determined by simulating it with the program specified by the user. The higher its fitness, the higher its chance of becoming a parent.

There are two different algorithms to select the parents. With the *leaders* algorithm, the parents are randomly chosen from the fittest N individuals. If the *roulette* algorithm is used, the parents are randomly chosen from the entire generation. The normalized fitness value of every individual is the probability with which an individual is selected as a parent. Individuals with a high fitness value are more likely to become parents (even more than once) than individuals with a low fitness value.

[1] That is not entirely true: Unlike nature, this implementation just uses mutation and no cross-over.

[2] In our case an individual is a set of parameter values.

Additionally, the algorithm exhibits a so called *elitism* switch. If *elitism* is on, the very best individual of every generation is copied to the next generation without being mutated.

Each parameter of a parent has a preset chance of being mutated. If a parameter is mutated, it is calculated as follows:

Be p_p a parent and p_c a child then:

$$\mu = (r_{\max} - r_{\min}) * \alpha \tag{3.8}$$

$$p_c = p_p + x_d() * \mu \tag{3.9}$$

Whereas $r_{\min}$ and $r_{\max}$ are the minimum and maximum values for the current parameter. The function $x_d()$ is a random number generator with the distribution d. Equation 3.9 is repeated until $p_c \in [r_{\min}, r_{\max}]$.

The parameter values of the first generation are uniformly distributed over the entire search space.

3.3 Systematic Algorithm

Unlike the previously described algorithms, the systematic algorithm is not an optimization algorithm.

Testing every possible combination of parameters is done by systematically increasing the value of the free parameters until every combination has been tried. This is very time consuming with many free parameters and should be avoided for more than three free parameters.

Figure 3.1 shows the distribution of the fitness values of a system with two free parameters and 13 steps per parameter.

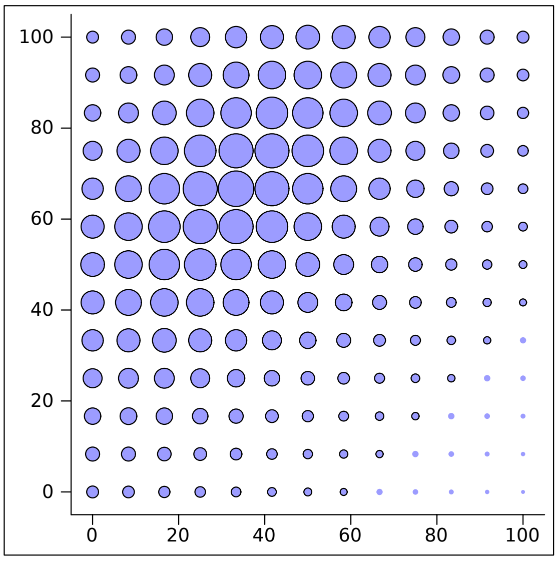


Figure 3.1: Analysis of a systematic experiment. The bigger the circle, the bigger the fitness value at this position.

Chapter 4

Using *Vidyaa*

4.1 Designing an Experiment

Before we can use *Vidyaa* to optimize parameters, we need to set up the optimization task by designing the experiments. The design of the experiments should be done very carefully. Logical errors in the design of the experiments lead to useless results and possibly the loss of days of computing time.

The following sections lead the way to a complete run of an optimization task and should be considered as a guideline.

4.1.1 Specify the Parameters

First of all, we need to know which parameters we would like to optimize and we must specify a value range for each parameter (it is also possible to set a fixed value for a parameter).

Let us assume, we have a robot and we would like to optimize some body parameters like the length of the legs as well as some controller parameters like the amplitude of the leg movements. That is, what we actually want is a robot which moves as well[1] as possible. A possible list of parameters for this scenario is shown in table 4.1.

[1]"Well" stands for fast, stable or whatever is desired.

Parameter	*Range*
FrontLegLength	0.10 – 0.50
HindLegLength	0.10 – 0.50
BodyLength	0.50 – 0.75
BodyWidth	0.20 – 0.30
FrontLegAmplitude	0.0 – 1.0
HindLegAmplitude	0.0 – 1.0
LegFrequency	1.0 – 3.0
HindLegPhaseLag	0.5

Table 4.1: Parameters to optimize

4.1.2 Specify the Experiments

But defining a list of parameters to optimize is not sufficient yet. We also need to know in which way and which order these parameters should be optimized by *Vidyaa*. To put it differently, we need to define the hierarchy of the experiments.

The easiest way to optimize the set of parameters would be to define a single experiment, which optimizes the parameters with a specified algorithm. In this case, it might be an experiment using an evolutionary algorithm to optimize the given parameters.

A more natural design would be a two-stage experiment where a robot is born with certain body parameters and tries to develop a way to move as well as possible under the given circumstances. This could be achieved by defining an experiment within another experiment. The top-level experiment optimizes the body parameters using an evolutionary algorithm and calls its sub-experiment for every individual it creates. The sub-experiment then optimizes the controller parameters using a simulated annealing algorithm, for instance. Based on its sub-experiment's return values, the top-level experiment then decides which individuals survive and which do not. This design of the experiments is demonstrated in figure 4.1.

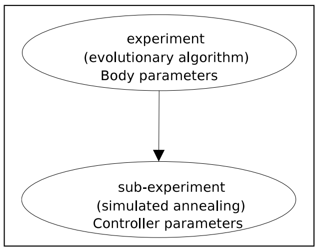


Figure 4.1: Experiment containing another experiment

One could also imagine a scenario, where an experiment does not just contain one sub-experiment, but several sub-experiments on the same level. This can be done in *Vidyaa*, too, and is implemented with the already introduced *Array Experiment*. An *Array Experiment* is illustrated in figure 4.2.

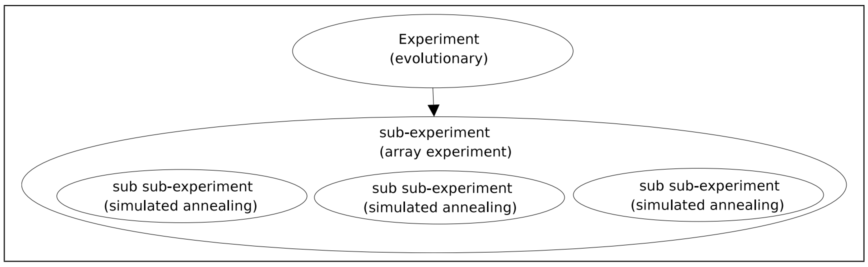


Figure 4.2: Experiment containing several experiments

Last but not least, we need to know how we would like to test our parameters. This is the lowest level of the experiment hierarchy and it is usually done in a separate program, which is automatically called by *Vidyaa* for every set of parameters. This is called *Simulator Experiment* and every series of experiments must have a *Simulator Experiment* on its lowest level. Otherwise, the parameters could never be tested.

4.2 Setting up an Experiment

As follows from the preceding sections, three types of experiments are available. They all have in common, that the experiments on the higher level pass them their current parameter values. That is, a set of parameter values is handed down the hierarchy until it ends with a *SimulationExperiment*. Every level in between can add or modify something.

Algorithm Experiment Modifies (e.g. optimizes) a set of parameters and passes it to its sub-experiment. This is done for each step of the used algorithm.

An *Algorithm Experiment* can contain any other type of experiment.

Array Experiment Contains several experiments on the same level. Defines how to handle their fitness values.

An *Array Experiment* can contain any other type of experiment.

Simulation Experiment Tests a set of parameters. It writes the parameters into a file and calls an external program. The return values from the external program are read from a file. A *SimulationExperiment* is settled on the lowest level of the hierarchy and does not have any sub-experiments.

Using these experiment types, we can set up the entire optimization process in an XML configuration file. Since examples are less abstract than formal descriptions we suggest to look at a few specific experiments before we go into the details.

4.2.1 One-stage Experiment

A one-stage experiment is an experiment, where all the parameters are optimized on the same stage using the same algorithm. An example is shown in figure 4.3.

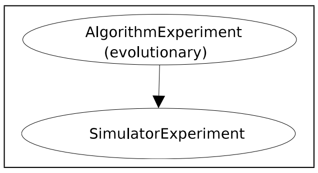


Figure 4.3: One-stage optimization

The corresponding configuration file is shown in listing 4.1.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <setting
4    name="A␣one-stage␣experiment"
5    version="1.0"
6    xmlns="http://ailab.ifi.uzh.ch/testframework/2008/"
7    xmlns:evol="http://ailab.ifi.uzh.ch/testframework/2008/algorithm/evolution/"
8    xmlns:siman="http://ailab.ifi.uzh.ch/testframework/2008/algorithm/
         simulatedannealing/"
9    xmlns:systematic="http://ailab.ifi.uzh.ch/testframework/2008/algorithm/systematic/
         "
10 >
11
12  <algorithm type="Evolution">
13         <evol:config
14                 changeprobability="0.50"
15                 distribution="gaussian"
16                 population="50"
17                 parents="5"
18                 generations="10"
19                 selection="leaders"
20                 elitism="on"
21                 parameterscaling="4" />
22
23         <parameters>
24                 <param name="FrontLegLength" min="0.05" max="0.12" />
25                 <param name="HindLegLength" min="0.05" max="0.12" />
26                 <param name="BodyLength" min="0.05" max="0.12" />
27                 <param name="BodyWidth" min="0.05" max="0.12" />
28         </parameters>
29
30         <simulation program="./dumsim" />
31
32  </algorithm>
33
34 </setting>
```

Listing 4.1: one_stage_experiment.xml

As the listing demonstrates, the root node is `<setting>`, which configures global settings like the name of the experiment series, namespaces and so on. Every configuration file has a `<setting>` node as the root node. It can usually be transferred to other configuration files without any modifications except the name of the experiment series.

On line 12, the top-level experiment is defined as an *Algorithm Experiment* which uses the evolutionary algorithm to optimize the parameters. `<evol:config>` contains the configuration of the evolutionary algorithm. The attributes it can take differ from algorithm to algorithm and are described later on. Common to every *Algorithm Experiment* is the definition of the parameters it optimizes within the `<parameters>`-element (line 23 – 28).

Finally, on line 30, the program used for the simulation of these parameters is defined (according to the previous definitions, this is a so called *Simulator Experiment*). So, strictly speaking, we describe a two-stage experiment here (*Simulator Experiment* within an *Algorithm Experiment*), but we usually do not count the *Simulator Experiment* as a stage on its own.

4.2.2 Two-stage Experiment

A two-stage experiment is an experiment where the parameters are optimized on two different stages, possibly by different algorithms. An example is shown in figure 4.4.

The corresponding configuration file is shown in listing 4.2.

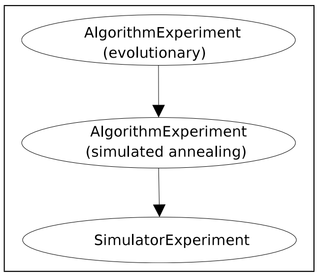


Figure 4.4: Two-stage optimization

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2
 3 <setting
 4    name="Two-stage␣experiment"
 5    version="1.0"
 6    xmlns="http://ailab.ifi.uzh.ch/testframework/2008/"
 7    xmlns:evol="http://ailab.ifi.uzh.ch/testframework/2008/algorithm/evolution/"
 8    xmlns:siman="http://ailab.ifi.uzh.ch/testframework/2008/algorithm/
          simulatedannealing/"
 9    xmlns:systematic="http://ailab.ifi.uzh.ch/testframework/2008/algorithm/systematic/
          "
10 >
11
12  <algorithm type="Evolution">
13          <evol:config
14                  changeprobability="0.50"
15                  distribution="gaussian"
16                  population="50"
17                  parents="5"
18                  generations="10"
19                  selection="leaders"
20                  elitism="on"
21                  parameterscaling="4" />
22
23          <parameters>
24                  <param name="FrontLegLength" min="0.05" max="0.12"/>
25                  <param name="HindLegLength" min="0.05" max="0.12"/>
26          </parameters>
27
28          <algorithm name="sub␣experiment" version="1.0" type="Simulated␣Annealing">
29                  <siman:config
30                          inittemperature="16"
31                          terminatetemperature="2"
32                          bestfitness="1.0"
33                          cooling="0.95" />
34
35                  <parameters>
36                          <param name="LegFrequency" min="0.5" max="4"/>
37                          <param name="HindLegAmplitude" min="0.0" max="0.5"/>
38                          <param name="FrontLegAmplitude" min="0.0" max="0.5"/>
39                  </parameters>
40
41                  <simulation program="./dumsim" />
42          </algorithm>
43  </algorithm>
44 </setting>
```

Listing 4.2: two_stage_experiment.xml

What is new in this example is, that the *Algorithm Experiment* contains another *Algorithm Experiment* (line 28). This sub-experiment uses the simulated annealing algorithm and optimizes the parameters defined on line 35 – 39.

This experiment, too, defines a *Simulator Experiment* on the lowest level (line 41). The attribute value of `program` is the path to the program which is called to test the set of parameters.

Vidyaa conducts this series of experiments as follows:

1. The evolutionary algorithm produces a generation of parameter values for *FrontLegLength* and *HindLegLength*.

2. For each individual of this generation, the set of parameters is given to the simulated annealing experiment which extends it with the *LegFrequency*,

the *HindLegAmplitude* and the *FrontLegAmplitude*. This means, the current set of parameters now contains values for *FrontLegLength HindLegLength*, *LegFrequency*, *HindLegAmplitude* and *FrontLegAmplitude*. To decide how good the chosen values are, the simulated annealing algorithm calls the *Simulation Experiment* after every step.

3. The *simulation experiment* returns a fitness value for every simulated set of parameters. This fitness value is used by the simulated annealing algorithm for its decisions.

4. When the simulated annealing algorithm is finished, it returns a fitness value to the evolutionary algorithm. With this fitness value, the evolutionary algorithm knows, how good the parameter values (*FrontLegLength* and *HindLegLength*) it gave to the sub-experiment were.

5. After performing these steps for the entire generation, the evolutionary algorithm uses the fitness values to generate the individuals for the next generation.

As this list implies, an algorithm does not know about the parameters optimized by sub-experiments. This means, an experiment just knows what "potential" the values it passed on to the next experiment have.

4.2.3 Array Experiment

In an *Array Experiment*, the parameters are optimized on the same level, but in different experiments. An example is shown in figure 4.5.

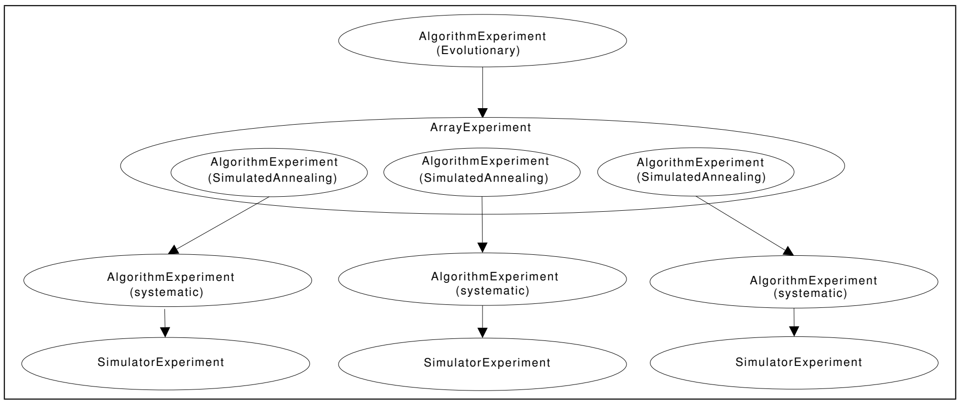

AlgorithmExperiment (Evolutionary)
ArrayExperiment
AlgorithmExperiment (SimulatedAnnealing)
AlgorithmExperiment (SimulatedAnnealing)
AlgorithmExperiment (SimulatedAnnealing)
AlgorithmExperiment (systematic)
AlgorithmExperiment (systematic)
AlgorithmExperiment (systematic)
SimulatorExperiment
SimulatorExperiment
SimulatorExperiment

Figure 4.5: Array experiment

The corresponding configuration file is shown in listing 4.3.

```
1   <?xml version="1.0" encoding="UTF-8"?>
2
3   <setting
4     name="Development_of_Gaits_for_a_Quadruped"
5     version="1.0"
6     xmlns="http://ailab.ifi.uzh.ch/testframework/2008/"
7     xmlns:evol="http://ailab.ifi.uzh.ch/testframework/2008/algorithm/evolution/"
8     xmlns:siman="http://ailab.ifi.uzh.ch/testframework/2008/algorithm/
          simulatedannealing/"
9     xmlns:systematic="http://ailab.ifi.uzh.ch/testframework/2008/algorithm/systematic/
          "
10  >
11
12    <algorithm type="Evolution" >
13      <evol:config
14          changeprobability="0.1"
15          distribution="gaussian"
16          population="80"
17          parents="8"
18          generations="250"
19          selection="roulette"
20          elitism="off"
21          parameterscaling="4" />
22
23      <parameters>
24        <param name="FrontLegLength" min="0.05" max="0.12"/>
25        <param name="HindLegLength" min="0.05" max="0.12"/>
26      </parameters>
27
28      <array fitness="average">
29        <algorithm name="Sub_experiment_walking_gait" version="1.0" type="Simulated_
              Annealing">
30          <siman:config
31          inittemperature="16"
32          terminatetemperature="2"
33          bestfitness="1.0"
34          cooling="0.95" />
35
36          <parameters>
37            <param name="PhaseLagLateral" min="0.45" max="0.55"/>
38            <param name="PhaseLagAnteriorPosterior" min="0.45" max="0.55"/>
39          </parameters>
40
41          <algorithm name="Test_stability" version="1.0" type="Systematic" >
42            <systematic:config
43            stepsperparameter="3" />
44
45            <parameters>
46              <param name="C_Frequency" min="1.0" max="4.0" />
47            </parameters>
48
49            <simulation program="./dumsim" />
50          </algorithm>
51        </algorithm>
52
53        <algorithm name="Sub_experiment_trotting_gait" version="1.0" type="Simulated_
              Annealing">
54          <!-- configuration similar as in "sub_experiment_walking_gait" -->
55        </algorithm>
56
57        <algorithm name="Sub_experiment_bounding_gait" version="1.0" type="Simulated_
              Annealing">
58          <!-- configuration similar as in "sub_experiment_walking_gait" -->
59        </algorithm>
60
61      </array>
62    </algorithm>
63  </setting>
```

Listing 4.3: array_experiment.xml

As shown in figure 4.5 and in the configuration file on line 28, the *Array Experiment* was used to implement several sub-experiments on the same level. The attribute `fitness` defines how the fitness values of the individual experiments are processed. `fitness="average"` means that the average of the fitness values is returned to the next higher level.

4.2.4 Settings to the Algorithms

As we have seen in the previous examples, an *Algorithm Experiment* is set up with `<algorithm type="[type]">`. The attribute `type` selects the actual optimization algorithm to use. Of course, these optimization algorithms can be further configured individually. The possible settings to the standard algorithms can be taken from the following sections.

Evolutionary Algorithm

Usage of the evolutionary algorithm is defined by the type `Evolution`. The basic configuration of the algorithm is done in `config` in the namespace of `http://ailab.ifi.uzh.ch/testframework/2008/algorithm/evolution/`. Allowed attributes and their values are listed in table 4.2.

Attribute	*Value*	*Explanation*
changeprobability	*double*	Probability that a parameter changes from one to another generation
distribution	gaussian	Distribution of the changes to the parent's values
generations	*integer*	Number of generations.
population	*integer*	Size of one generation.
selection	leaders, roulette	Parent selection algorithm.
parents	*integer*	Number of parents (if "leaders" is used)
elitism	on, off	Elitism guarantees that the best individual survives
parameterscaling	*double*	Stretches the gaussian curve

Table 4.2: Settings for the evolutionary algorithm

Simulated Annealing

Usage of the simulated annealing algorithm is defined by the type `Simulated Annealing`. The basic configuration of the algorithm is done in `config` in the namespace `http://ailab.ifi.uzh.ch/testframework/2008/algorithm/simulatedannealing/`. Allowed attributes and their values are listed in table 4.3.

Attribute	*Value*	*Explanation*
inittemperature	*double*	Init temperature
terminatetemperature	*double*	Temparature at which the algorithm terminates
bestfitness	*double*	Maximal fitness value
cooling	*double*	Cooling constant (recommended eg. 0.95).
boltzmannconstant	*double*	default = 1, a lower value increases the probability to accept less good parameters
parameterscaling	*double*	default = 1, increases probability for big changes

Table 4.3: Settings for the simulated annealing algorithm

Textbook Simulated Annealing

Usage of the textbook simulated annealing algorithm is defined by the type `Textbook Simulated Annealing`. The basic configuration of the algorithm is done in `config` in the namespace `http://vidyaa.origo.ethz.ch/2009/algorithm/textbooksa/`. Allowed attributes and their values are listed in table 4.4.

Attribute	*Value*	*Explanation*
bestfitness	*double*	Maximal fitness value (> 0)
cooling	*double*	Cooling constant ($> 0, < 1$)
terminatetemperature	*double*	Temparature at which the algorithm terminates (> 0)
inittemperature	*double*	Init temperature ($\geq$ terminatetemperature)

Table 4.4: Settings for the textbook simulated annealing algorithm

Systematic Algorithm

Usage of the systematic algorithm is defined by the type `Systematic`. The basic configuration of the algorithm is done in `config` in the namespace `http://ailab.ifi.uzh.ch/ testframework/2008/algorithm/systematic/`. Allowed attributes and their values are listed in table 4.5.

Attribute	*Value*	*Explanation*
stepsperparameter	*integer*	Default number of steps per parameter

Table 4.5: Settings for the systematic algorithm

The stepsize can be changed for every parameter by the attribute `steps` in the same namespace as the basic configuration.

The parameters-Element

The `<parameters>`-element contains a list of parameters the algorithm optimizes. These parameters are defined by using a `<param>`-Element with the mandatory attributes `name="[name]"`, `min="[value]"` and `max="[value]"`.

It is also possible to set an additional attribute `mode="relative"`, which causes the algorithm to interpret `min` and `max` as relative values to the value given by the experiment on the higher level. For instance, if the value for parameter *Weight* has been set to 15 by the experiment on the higher level,

```
<param name="Weight" min="-1" max="1" mode="relative" />
```

means that the possible value range for *Weight* is 14 to 16.

But what if you want a parameter fixed to a certain value and not optimized at all? Fortunately, there is an element for this, too. Just use

```
<fixed name="[name]" value="[value]" />
```

instead of the `<param>`-element.

The simulation-Element

`<simulation>` is used to set up a *Simulation Experiment*. Or, to put it simpler, it just defines which program is used to test the current parameter values.

The path to the program is given with the attribute `program="[path]"`. The optional attribute `arguments="[args]"` defines additional arguments to the simulation program. As an example, the entry

```
<simulation program="/bin/simulator" attributes="-q -w" />
```

causes *Vidyaa* to call `/bin/simulator` with the arguments `-q` and `-w`.

4.3 Running an Experiment

After finishing the preparations, we are finally ready to run the optimization process. By calling `vidyaa --help` we get a list of the arguments it accepts.

Usually, *Vidyaa* is called with the arguments `-l [logfile]`, `-e [directory]` and the configuration file to use. `-l` defines the logfile to which the log messages and results are written while `-e` defines the path to which the simulated parameter lists and their results are written.

For detailed information, read the manpage (`man vidyaa`).

4.3.1 Examples

Usage:

```
vidyaa --usage
```

Display the list of available arguments:

```
vidyaa --help
```

A standard call is:

```
vidyaa -l experiment.log -e experiment/ experiment.xml
```

This call defines the logfile, the directory for the parameter lists and the configuration file.

Per default, all messages are written into the logfile. We can influence this behaviour by using one or more of the flags `--errors`, `--warnings`, `--infos`, `--results` and `--nothing`. For instance

```
vidyaa -l experiment.log -e experiment/ \
   --errors --warnings --results experiment.xml
```

defines, that *errors*, *warnings* and *results* are written to the logfile, but no *info* messages.

4.4 Evaluating the Results of an Experiment

The last step of an experiment is the evaluation or collection of the results. To do so, we usually interpret the values in the logfile. Listing 4.4 shows a common logfile.

```
INFO;Using generated seed: 1240922345
INFO;simulating file: ./puppy2/evolution/generation1/individual0.parameter
RESULT;C_Frequency;1.17545;RobotCenterOfMass;-0.661094;Energy;0.501866;
        TraveledDistance;0.0748375;fitness;0.00748375;
INFO;simulating file: ./puppy2/evolution/generation1/individual1.parameter
RESULT;C_Frequency;1.28929;RobotCenterOfMass;-0.454526;Energy;0.425771;
        TraveledDistance;0.0164073;fitness;0.00164073;
[...]
```

Listing 4.4: log.txt

The first field indicates of which type a message is. Possible types are ERROR, WARN, INFO and RESULT. When we evaluate the results, we are particularly interested in the RESULT lines. These lines print the parameter values which were given to the simulation, as well as the parameters which were returned by the simulation.

Since we have all the necessary information in the RESULT line, we can process these entries of the logfile with the tool of our choice. For instance, we could illustrate the behaviour of the fitness value using *gnuplot*. [2]

In addition to the logfile, *Vidyaa* creates a directory structure which contains all the input files to the simulation and the result files the simulation program returned. Where these directories are located, was defined by `-e` when we called *Vidyaa*. These files can be used, to start a simulation of a specific set of parameters again.

[2]A tip: It is often useful to pre-process the logfile with command line tools like *grep* and *cut*.

Part II

Programmer Manual

Chapter 5

Architecture Overview

5.1 Basic Concepts

When *Vidyaa* was designed, the main objectives were extensibility, flexibility and ease of use. The implemented architecture directly originates from these objectives and many characteristics are quite easy to understand when this is kept in mind.

One of the most basic principles is the separation of the actual optimization from the testing of the parameters. This increases the flexibility as well as the extensibility of the software. An in-depth description of this concept is provided in chapter 6.

Furthermore, the task of setting up experiments was made easier for the user by choosing XML as the format for a central configuration file. At the same time, existing XML parsers made it easy for a programmer to interprete the configuration file.

Concerning the internals of *Vidyaa*, it has been tried to keep it as simple as possible, without cutting back on flexibility or functionality.

Basically, an optimization process is represented by a hierarchy of experiments. Three different types of experiments are distinguished. *Algorithm Experiments* optimize parameters and contain another experiment. *Simulation Experiments* test the parameter values and *Array Experiments* contain one or more experiments on the same hierarchy level. Figure 5.1 shows a simplified class diagram.

The fact that experiments can contain other experiments is reflected by the classes *Algorithm* and *ArrayExperiment*, which both contain an *Experiment*.

DataLogger is the class responsible for log entries.

5.2 Programming Language and Libraries

Vidyaa is programmed in C++. It makes use of the following libraries:

- *gsl*
- *gslcblas*
- *xml++-2.6*

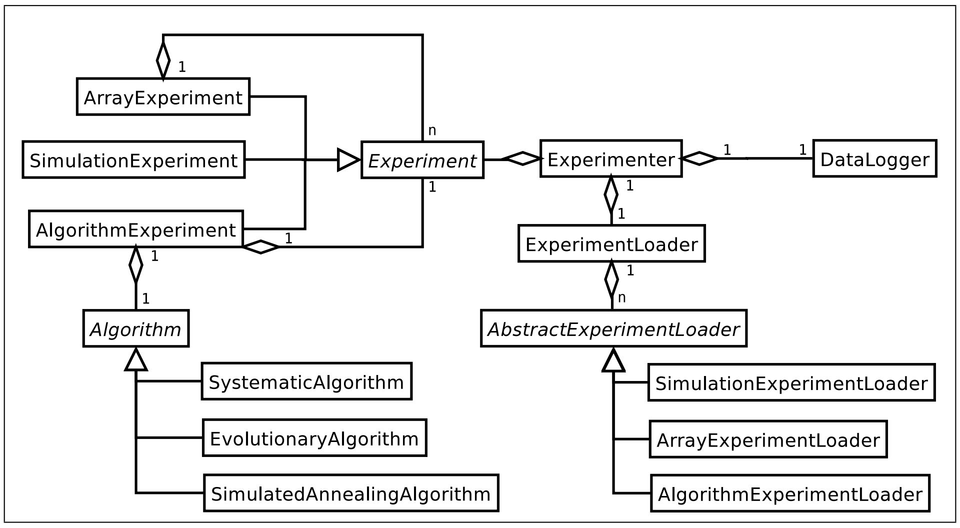


Figure 5.1: Simplified Class Diagram

- *xml2*
- *glibmm-2.4*
- *gobject-2.0*
- *sigc-2.0*
- *glib-2.0*

5.3 Source Directory Structure

```
src
|-- DataLogger
|   |-- DataLogger.cpp
|   `-- DataLogger.h
|-- Experimenter.cpp
|-- Experimenter.h
|-- Experiments
|   |-- Algorithms
|   |   |-- Algorithm.cpp
|   |   |-- Algorithm.h
|   |   |-- AlgorithmExperiment.cpp
|   |   |-- AlgorithmExperiment.h
|   |   |-- AlgorithmExperimentLoader.cpp
|   |   |-- AlgorithmExperimentLoader.h
|   |   |-- AlgorithmLoader.cpp
|   |   |-- AlgorithmLoader.h
|   |   |-- Evolutionary
|   |   |   |-- EvolutionaryAlgorithm.cpp
|   |   |   |-- EvolutionaryAlgorithm.h
|   |   |   |-- EvolutionaryAlgorithmLoader.cpp
|   |   |   `-- EvolutionaryAlgorithmLoader.h
|   |   |-- SimulatedAnnealing
|   |   |   |-- SimulatedAnnealingAlgorithm.cpp
|   |   |   |-- SimulatedAnnealingAlgorithm.h
|   |   |   |-- SimulatedAnnealingAlgorithmLoader.cpp
|   |   |   `-- SimulatedAnnealingAlgorithmLoader.h
|   |   |-- Systematic
|   |   |   |-- SystematicAlgorithm.cpp
|   |   |   |-- SystematicAlgorithm.h
|   |   |   |-- SystematicAlgorithmLoader.cpp
|   |   |   `-- SystematicAlgorithmLoader.h
|   |   `-- TextbookSa
|   |       |-- TextbookSaAlgorithm.cpp
|   |       |-- TextbookSaAlgorithm.h
|   |       |-- TextbookSaAlgorithmLoader.cpp
|   |       `-- TextbookSaAlgorithmLoader.h
|   |-- Array
|   |   |-- ArrayExperiment.cpp
|   |   |-- ArrayExperiment.h
|   |   |-- ArrayExperimentLoader.cpp
|   |   `-- ArrayExperimentLoader.h
|   |-- Experiment.cpp
|   |-- Experiment.h
|   |-- ExperimentLoader.cpp
|   |-- ExperimentLoader.h
|   |-- LoaderException.cpp
|   |-- LoaderException.h
|   `-- Simulators
|       |-- SimulationExperiment.cpp
|       |-- SimulationExperiment.h
|       |-- SimulationExperimentLoader.cpp
|       `-- SimulationExperimentLoader.h
|-- common
|   |-- Directory.cpp
|   |-- Directory.h
|   |-- convert.cpp
|   |-- convert.h
|   |-- include
|   |   `-- types.h
|   |-- paramutils.cpp
|   `-- paramutils.h
`-- main.cpp

11 directories, 50 files
```

Chapter 6

Interface to the Simulator

Vidyaa was designed with flexibility and extensibility in mind. To achieve this, knowledge about the parameters was delegated to a simulation program, which simulates the parameters and returns a value to *Vidyaa* to let it know, how good the parameters are. *Vidyaa* itself has no idea what the parameters it is optimizing represent.

The interface between *Vidyaa* and the simulation software is also discussed in the *User Manual*. Since the basic principles are described in the *User Manual*, it is advised to read it before you read this chapter.

6.1 Example ***PuppySimulator***

PuppySimulator is a software which simulates a *Puppy* robot using *Webots*[1]. The robot's parameters are read from a parameter file specified by the argument `-i [infile]` and the results are written into a file specified by `-o [outfile]`. The `-i` and `-o` arguments are necessary for every simulation program since *Vidyaa* calls the simulator with these arguments to specify an input and output file. The input file contains the parameters and their values, whereas the output file contains results of the simulation, in particular a fitness value. This fitness value is the measurement for the quality of the parameter values.

As figure 6.1 illustrates, the actual simulation program is completely independent of *Vidyaa*, as long as it respects the interface. This way, it is also possible to call the simulation program as a standalone application. You will like that in particular when you want to simulate a specific set of parameters again.

6.2 File Format

The format of the input and output file is easy to understand. There is one parameter per line and a line consists of the parameter's name followed by a space and its value. An input file could look as follows:

```
Length 10.5
Height 2.25
```

[1] *http://www.cyberbotics.com*

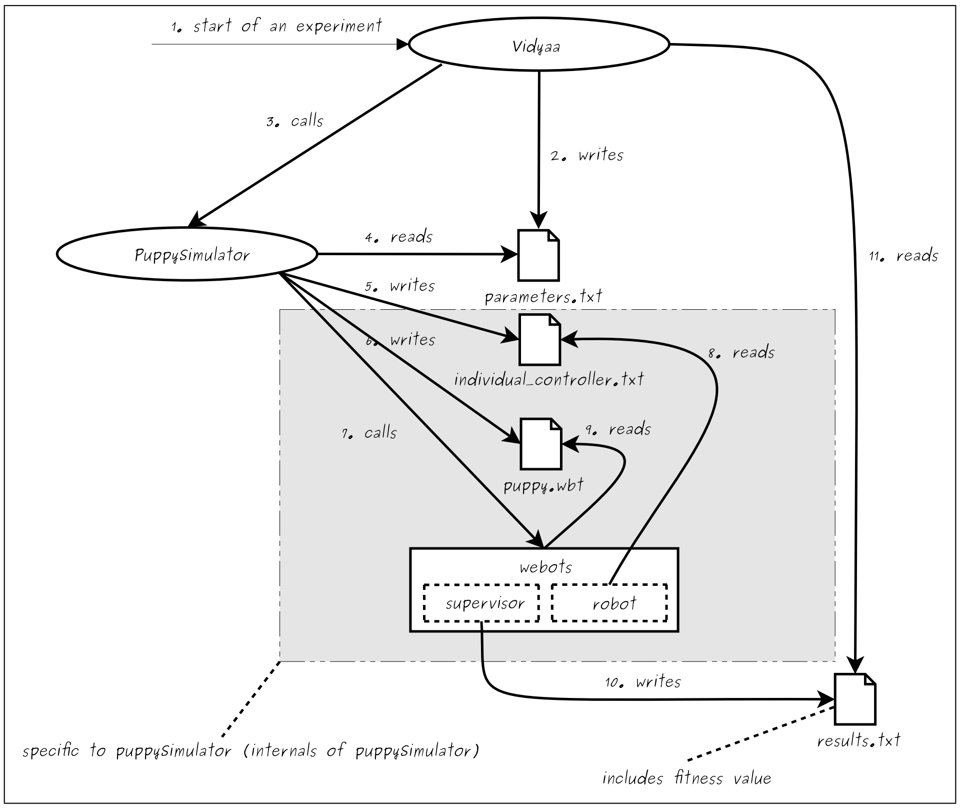


Figure 6.1: Interface between the framework and the PuppySimulator

The output file's format is exactly the same. The only constraint is, an output file has to contain a parameter called `fitness` with a value. Additional parameters and values are optional (all returned parameters are written into the log).

6.3 Writing a Simulator

As a programmer, you probably wonder, when we eventually get to the point and tell you how to write a simulator. Sorry about the delay, here we go:

1. Select a programming language of your choice.
2. Make sure your simulation program understands the arguments `-i [inputfile]` and `-o [outputfile]`.
3. Read the parameters and their values from the input file and simulate them in the way you like.
4. Write a fitness value into the output file (`fitness [value]`).
5. Write additional results of your simulation into the output file.
6. Close the open files, clean up and return.

Yes, it's that easy. Well, maybe one more thing: You should check whether the parameters you get in the input file are valid. If they are not, print an error message.

Chapter 7

Extending *Vidyaa*

This chapter deals with the extension of *Vidyaa* by additional algorithms and experiment types.

As already illustrated in the class diagram in figure 5.1, the experiment types and the actual optimization algorithms are separated. Thus, the introduction of a new optimization algorithm is relatively easy and can be done without having to touch the experiment types.

In case the available experiment types *Algorithm*, *Array* and *Simulation* shouldn't be enough for your needs, it's also possible to add your own experiment type.

7.1 Implementation of an Optimization Algorithm

A closer look at the interesting parts of the class diagram in figure 7.1 reveals the details of the architecture.

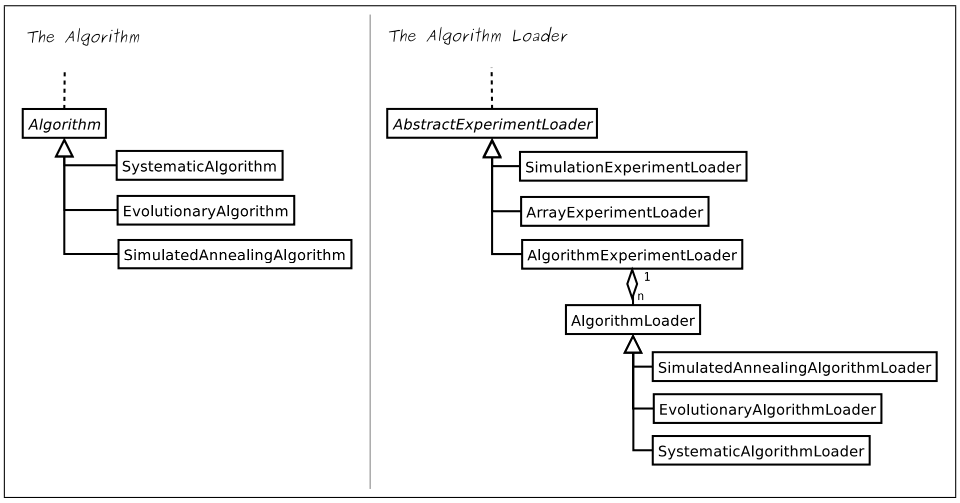

The Algorithm
Algorithm
SystematicAlgorithm
EvolutionaryAlgorithm
SimulatedAnnealingAlgorithm
The Algorithm Loader
AbstractExperimentLoader
SimulationExperimentLoader
ArrayExperimentLoader
AlgorithmExperimentLoader
1
n
AlgorithmLoader
SimulatedAnnealingAlgorithmLoader
EvolutionaryAlgorithmLoader
SystematicAlgorithmLoader

Figure 7.1: Detailed excerpt of the class diagram (algorithms)

The actual implementation of the algorithm is done in a class derived from the abstract class *Algorithm*.

Additionally, every algorithm needs its own loader which is derived from *AlgorithmLoader* and is responsible for the basic configuration of the algorithm.

Altogether, the following tasks have to be done if you want to introduce your own algorithm:

1. Create a new class called *MyAlgorithm*, derived from *Algorithm*.
2. Implement the methods *initialize(...)*, *hasNext()*, *next()*, *extendDirectory(...)*, *getName()* and *fitnessFeedback(...)*.
3. Create the loader called *MyAlgorithmLoader* by deriving *AlgorithmLoader*.
4. Implement its methods *getNamespace()* and *load(...)*.
5. Register the loader in the class *Experimenter*. This is done by instantiating the loader in *initLoader()*, choosing a name and adding it to the *AlgorithmExperimentLoader*. The chosen name is the name by which the algorithm can be selected in the configuration file.

Of course, this list is more of a to-do-list than an actual implementation guide. Before you implement your own algorithm, you should have a look at the source code of an existing algorithm.

7.2 Implementation of an Experiment Type

Modifying or extending the experiment types is done in a similar way as the introduction of a new algorithm. The relevant excerpts of the class diagram are shown in figure 7.2.

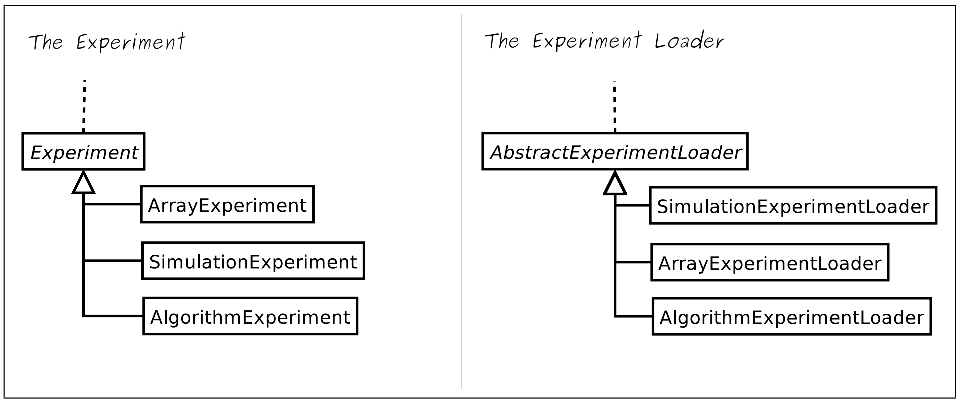


Figure 7.2: Detailed excerpt of the class diagram (experiment types)

Basically, the following tasks must be done:

1. Create a new class *MyExperiment*, derived from *Experiment*.
2. Implement its *run()*-method.

3. Create the loader called *MyExperimentLoader* by deriving *AbstractExperimentLoader*.
4. Implement its *createExperiment(...)*-method.
5. Register the loader in the class *Experimenter*. This is done by instantiating the loader in *initLoader()*, chosing a name and adding it to *m_experimentLoader*. The chosen name is the name of the XML node which will be used in the configuration file to create an experiment of this type.

As with the implementation of a new algorithm, this list does not contain all the information necessary to implement a new experiment type. You should have a look at an existing experiment type before you start writing your own.

Bibliography

[1] Simulated annealing. `http://en.wikipedia.org/wiki/Simulated_annealing`, 05 2009.

[2] Max Lungarella. *Exploring Principles Towards a Developmental Theory of Embodied Artificial Intelligence.* PhD thesis, University of Zurich, Switzerland, 2004.