FrEAK

Free Evolutionary Algorithm Kit

User's Guide

Project group 427

Patrick Briest	Dimo Brockhoff
Bastian Degener	Matthias Englert
Christian Gunia	Oliver Heering
Michael Leifhelm	Kai Plociennik
Heiko Röglin	Andrea Schweer
Dirk Sudholt	Stefan Tannenbaum

Thomas Jansen   Ingo Wegener

October 28, 2003

Contents

1 Introduction **4**
- 1.1 What is FrEAK? . . . 4
- 1.2 Evolutionary Algorithms . . . 4
- 1.3 Overview . . . 5
  - 1.3.1 Schedule Creation: Designing the Algorithm . . . 5
  - 1.3.2 Schedule Creation: Preparing the Simulation . . . 6
  - 1.3.3 Controlling Runs . . . 6

2 Creating a Schedule **7**
- 2.1 Some Words about Schedules . . . 7
  - 2.1.1 What is a Schedule? . . . 7
  - 2.1.2 What is a Batch? . . . 7
- 2.2 Overview and Navigation . . . 8
- 2.3 General Notes . . . 9
- 2.4 Designing the Algorithm . . . 10
  - 2.4.1 Step 1: Select a Search Space . . . 10
  - 2.4.2 Step 2: Select a Fitness Function . . . 11
  - 2.4.3 Step 3: Create an Algorithm Graph . . . 13
  - 2.4.4 Step 4: Select a Stopping Criterion . . . 14
  - 2.4.5 Step 5: Select Population Model and Initialization . . . 15
- 2.5 Preparing the Simulation . . . 16
  - 2.5.1 Step 6: Add Observers and Views . . . 16
  - 2.5.2 Step 7: Creating Batches of Runs . . . 17
- 2.6 Configuration of Modules . . . 18
  - 2.6.1 Configuration of Properties . . . 18
  - 2.6.2 Configuration of Event Sources . . . 19

3 Controlling Runs **21**
- 3.1 Getting Started . . . 21
- 3.2 Watching Replays . . . 23
- 3.3 Editing the Current Schedule . . . 24
- 3.4 Menu Items . . . 24
  - 3.4.1 File Menu . . . 24
  - 3.4.2 Control Menu . . . 25
  - 3.4.3 Views Menu . . . 25

3.4.4 Help Menu 25

4 Designing Algorithms **26**
4.1 Operator Graphs: Concepts 26
4.2 The Graph Editor 27
4.2.1 Adding and Removing Nodes and Edges 28
4.2.2 Changing Properties of Modules 29
4.3 Parameter Controllers 30
4.3.1 Adding Parameter Controllers 30
4.3.2 Assigning Event Sources and Configuring Parameter Controllers . . . 31
4.3.3 Assigning properties to parameters 32

A Modules Contained in FrEAK **34**
A.1 Search Spaces 34
A.2 Fitness Functions 34
A.2.1 Fitness Functions on Bit Strings 34
A.2.2 Fitness Functions on General Strings 37
A.2.3 Fitness Functions on Permutations 38
A.2.4 Fitness Functions on Cycles 38
A.2.5 Fitness Transformers 38
A.3 Operators 39
A.3.1 General Operators 39
A.3.2 Crossover 39
A.3.3 Initialization Operators 42
A.3.4 Mutation 42
A.3.5 Selection 43
A.3.6 Split 44
A.4 Population Models 44
A.5 Parameter Controllers 44
A.6 Stopping Criteria 45
A.7 Observers 45
A.8 Views 47

List of Figures

2.1 Creating a new schedule. . . . 8
2.2 The Schedule Creation dialog. . . . 8
2.3 Selecting a search space. . . . 10
2.4 Selecting a fitness function. . . . 11
2.5 Defining fitness transformers 12
2.6 Setup of fitness transformers. . . . 13
2.7 Creating an algorithm graph. . . . 13
2.8 Selecting a stopping criterion. . . . 14
2.9 Selecting population model and initialization. . . . 15
2.10 Adding observers and views. . . . 16
2.11 Creating batches of runs. . . . 17
2.12 Property dialog of view Boolean hypercube. . . . 18
2.13 Property dialog of observer All Individuals. . . . 19
2.14 Possible event sources for the observer All Individuals. . . . 20

3.1 The window after starting FrEAK. . . . 21
3.2 The FrEAK window after creating a schedule. . . . 22
3.3 The FrEAK window during replay mode. . . . 23

4.1 Operators with Ports. . . . 26
4.2 The Graph Editor Dialog. . . . 27
4.3 Error Message: Graph Has Syntax Errors. . . . 28
4.4 The Property Inspector 29
4.5 Adding parameter controllers. . . . 31
4.6 The information panel for the selected parameter controller. . . . 32
4.7 Setting up properties. . . . 33
4.8 Selecting properties to control. . . . 33

A.1 Boolean Hypercube View 47
A.2 File Writer View 47
A.3 Individual Table View 48
A.4 Number Table View 48
A.5 Plotter View 48
A.6 Standard View 49

Chapter 1

Introduction

1.1 What is FrEAK?

FrEAK, the Free Evolutionary Algorithm Kit, is a free toolkit for the creation, simulation, and analysis of evolutionary algorithms within a graphical interface.

We created FrEAK as a comfortable, flexible and extendable platform to perform experimental analysis of evolutionary algorithms. Several components commonly used in evolutionary algorithms are provided with FrEAK and a developer's guide explains how to create your own components.

The graphical interface allows you to create your own algorithms and to watch your algorithm running. With the built–in replay function, you can step back anytime and watch past runs and generations. Observers may be used to show, e.g., different performance measures, the individuals of the current population or other data derived from the algorithm.

FrEAK uses the approved random number generator "Mersenne Twister"[1] to provide well–distributed pseudorandom numbers. The random seed is drawn from the current system time on schedule creation and preserved during load and save operations.

1.2 Evolutionary Algorithms

There are many views of evolutionary algorithms in the scientific community. In FrEAK, we consider evolutionary algorithms as randomized search heuristics used to optimize a specified *fitness function*. The fitness function is defined on a *search space* which represents the set of all search points. Search points are referred to as *individuals*. An individual contains a *genotype* representing the individual's gene data and some other attributes like, e.g., its date of birth.

[1] M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator", ACM Trans. on Modeling and Computer Simulation Vol. 8, No. 1, 1998, pp. 3–30.

Evolutionary algorithms in FrEAK are represented by *algorithm graphs*, also denoted as *operator graphs*. Algorithm graphs are acyclic flow graphs leading individuals through various nodes. The nodes represent *operators* like mutation, recombination, selection, merging, and several other operators. They process the incoming individuals and propagate the resulting individuals through their outgoing edges.

Every generation, the current population[2] is led through the algorithm graph from a start node towards a finish node where the new population is received. A graphical editor is provided that enables you to design your own algorithm graphs. For further information, read Chapter 4: Designing Algorithms.

1.3 Overview

FrEAK is based on *schedules* containing the algorithm and simulation options. A wizard guides you through the process of creating a new schedule, as described in Chapter 2. You may save the current schedule or load one, so you don't have to create the same schedule twice.

When a schedule has been created, lean back and start the simulation. FrEAK provides a replay mode allowing you to watch past runs and past generations in the current run. To learn more about controlling runs, read Chapter 3: Controlling Runs.

1.3.1 Schedule Creation: Designing the Algorithm

The algorithm consists of six components, which are called *modules*:

- a search space,
- a fitness function,
- an algorithm graph,
- a stopping criterion,
- a population model, and
- an initial population.

The search space represents the origin of all genotypes and it is thus the domain of the fitness function. The fitness function is the function to be optimized. While most of the modules provided with FrEAK are designed for maximization problems, this is not a fundamental restriction; you may write custom modules that work with minimization problems as well.

To specify the algorithm graph, you may create your own algorithm with the graphical editor or load an algorithm. Several common algorithms are provided with FrEAK and can be picked from a ready–made list.

[2] or subdivisions of it, see Section 2.4.5 on population models

Stopping criteria tell the algorithm to stop the current run if a specified condition is fulfilled. The population model is an optional choice and allows you to maintain subdivisions inside the population. The last step here is to specify the initial population by choosing an *initialization* module that creates the individuals forming the first generation. To learn more about the modules provided with FrEAK, see Appendix A: Modules Contained in FrEAK. If you want to write your own module, refer to the *Module Developer's Guide*.

1.3.2 Schedule Creation: Preparing the Simulation

Preparing the simulation covers the following two steps:

- add observers and views and
- create batches of runs.

Observers are modules collecting and computing data that is then displayed by *views*. E.g., observers can compute measures like the fitness variance within the population or simply collect all individuals of the current population. A view displays the data observed by an observer if it is able to handle the type of data the observer provides. The two most commonly used data types are *individuals* and *numbers*. Observers may have an arbitrary number of views.

Last but not least, you may create *batches* of runs. A batch is a collection of runs with the same configuration for the fitness function and the initialization module. The modules selected and configured in the previous steps form the first batch which is created by default. You may then eventually add other batches with your fitness function or initialization module configured slightly different.

1.3.3 Controlling Runs

When the schedule creation is finished, you see a control panel to the left and the created views to the right. For details, see Chapter 3: Controlling Runs.

You may want to set a speed limit first. Then hit the **Start** button to start the simulation. When the stopping criterion is fulfilled, the run ends and the next run starts, if multiple runs or batches have been created.

Please note that several options and commands are disabled when the algorithm is running. To pause the algorithm and enable editing commands, hit the **Suspend** button to suspend the current run. For further information about editing the current schedule, see Section 3.3: Editing the Current Schedule.

In the control panel, the current run and the current generation are shown. Modify these fields and enter a past generation or a past run to switch to replay mode. In replay mode, you will see a slider at the bottom of the screen that can be used to navigate through the displayed run. To switch back to run mode, hit the **Start** button and when the replay is finished, FrEAK continues in run mode creating new generations.

Chapter 2

Creating a Schedule

2.1 Some Words about Schedules

The *schedule* is the central object you deal with in FrEAK, so it is necessary to explain it a little bit more in detail.

2.1.1 What is a Schedule?

A schedule is the collection of all data that is needed to simulate one or several runs of your algorithm. It contains the algorithm you want to simulate with all its components and the preferences for the simulation itself.

2.1.2 What is a Batch?

The runs that are to be simulated consist of one or multiple *batches*. A batch is a collection of settings for the fitness function and the initial population bundled with the information how often you want to repeat a run with these settings. A schedule will contain at least one batch with at least one run with the settings defined using the **Schedule Creation** dialog but you are free to either increase the number of runs or to append new batches to your schedule.

It is important to realize that the modules of the schedule stay the same throughout all runs and batches and that a batch simply gets new configuration settings for these modules.

A batch consists of the following three components:

- the number of runs,
- configuration settings for the selected fitness function, and
- configuration settings for the initialization module that creates the initial population.

Note:
Although the runs of a batch have all exactly the same settings that doesn't mean the components of the schedule will all be configured with the same values. If a configuration-setting is set to "random", it will create a random configuration every run, even if it's in the same batch.

2.2 Overview and Navigation

Most certainly, you now want to create your own schedule to see some algorithms in action, so let's begin right away.

First of all, select **New** from the **File** menu:

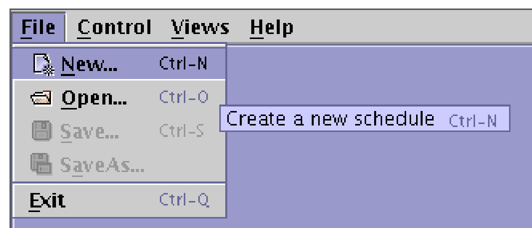


Figure 2.1: Creating a new schedule.

You will be presented the **Schedule Creation** dialog.

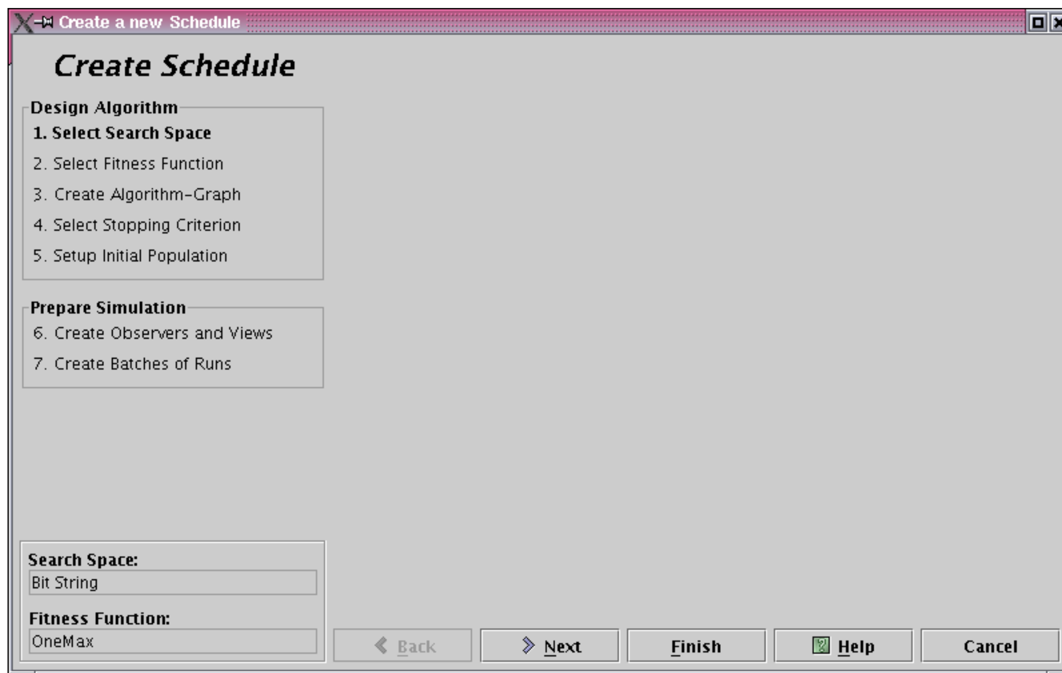


Figure 2.2: The Schedule Creation dialog.

The dialog consists of three parts. At the bottom of the window, you find the navigation buttons **Back** and **Next**. Use them to navigate back and forth through the wizard. Press **Help** to get context-sensitive help and hit **Finish** to end editing the schedule and run it. At any time you can press **Cancel** if you changed your mind and discard all changes you made to the schedule.

The progress in the wizard is displayed on the left side of the window. The step you are working on is highlighted. In the lower left corner you can see the chosen search space and fitness function.

The rest of the dialog will contain panels where you can construct your schedule step by step.

As you can see, some default selections are already done for you. In fact you can even press **Finish** right now to create a totally valid schedule but you will be warned that you won't get any feedback of the algorithm's progress. So let's take a quick tour through the **Schedule Creation** dialog.

2.3 General Notes

Before you start creating a schedule keep the following in mind:

- Whenever you can select a module from a list, there is most probably a **Configure** button near it. So whatever you choose, check the configuration of the module if it suits your needs. If there is a text field above the button that says "**Options**", this field will contain the current configuration of the selection in a human readable format. For more information about configuring modules, see Section 2.6: Configuration of Modules.
- Most modules you select from a list will show their description in a text area beside or below the list.
- The navigation buttons at the bottom are context sensitive. So, if **Finish** is enabled, your schedule is syntactically correct and you **may** run it. If, for example, the **Next** button is disabled, you forgot to select or configure something in the current step.
- Several modules of the schedule depend heavily on other modules. So, when you exchange a module for another one, modules selected and configured in following steps will most probably become invalid and might even not be available any more.

For a list of all available modules, refer to Appendix A: Modules Contained in FrEAK.

2.4 Designing the Algorithm

2.4.1 Step 1: Select a Search Space

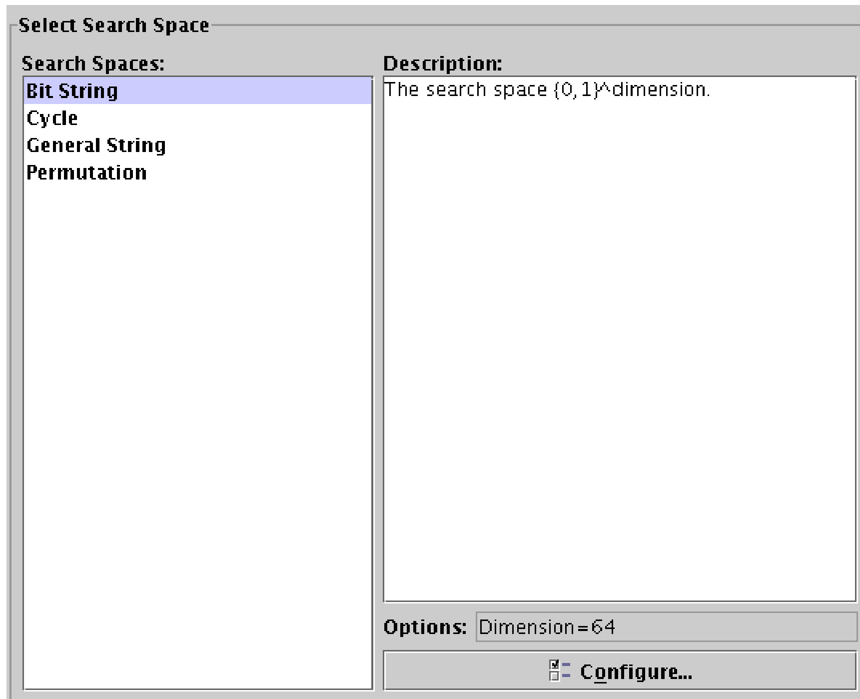


Figure 2.3: Selecting a search space.

First, you need to choose a *search space* your algorithm works on from the list (see Figure 2.3). Bit String is the default selection right now. If you want to change the search space, just click on another one and configure it using the **Configure** button.

For now, we are satisfied with Bit String with a dimension of 64. Press **Next** to continue.

2.4.2 Step 2: Select a Fitness Function

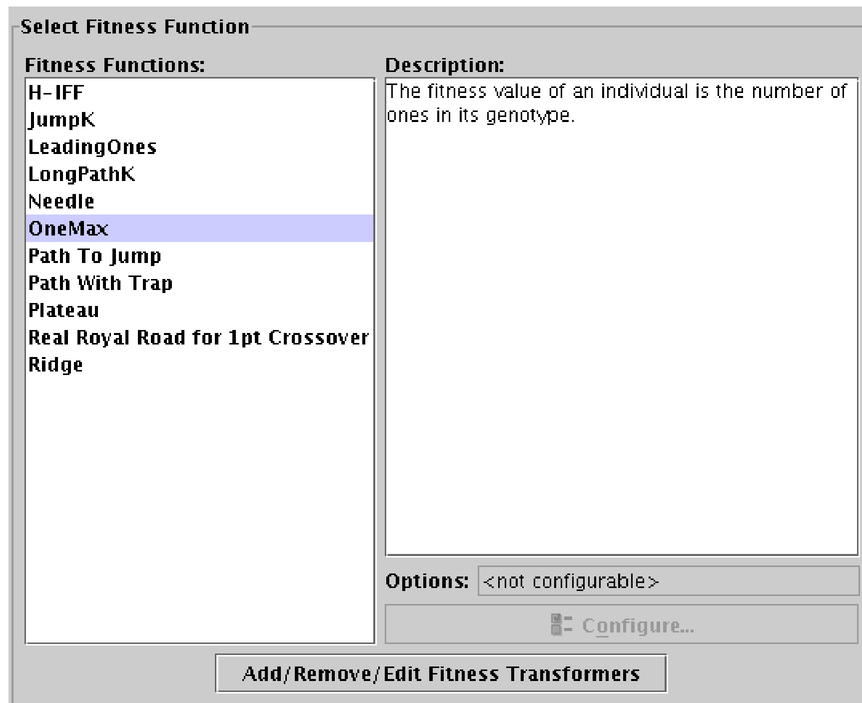


Figure 2.4: Selecting a fitness function.

In the second step you need to choose the *fitness function* your algorithm is supposed to optimize.

Just like the search space you can choose it by selecting one from the list and configure it by pressing the **Configure** button (see Figure 2.4).

The **Add/Remove/Edit Fitness Transformers** button is an advanced option to define *fitness transformers* that provide additional functionality to your fitness function. Fitness transformers are applied to your fitness function and transform the computed values.

If you want to know how to use fitness transformers, read the next paragraph, otherwise simply select RIDGE as fitness function and press **Next** to continue.

Defining Fitness Transformers

If you want to add fitness transformers to your schedule, click on **Add/Remove/Edit Fitness Transformers**. You will be presented the following dialog:

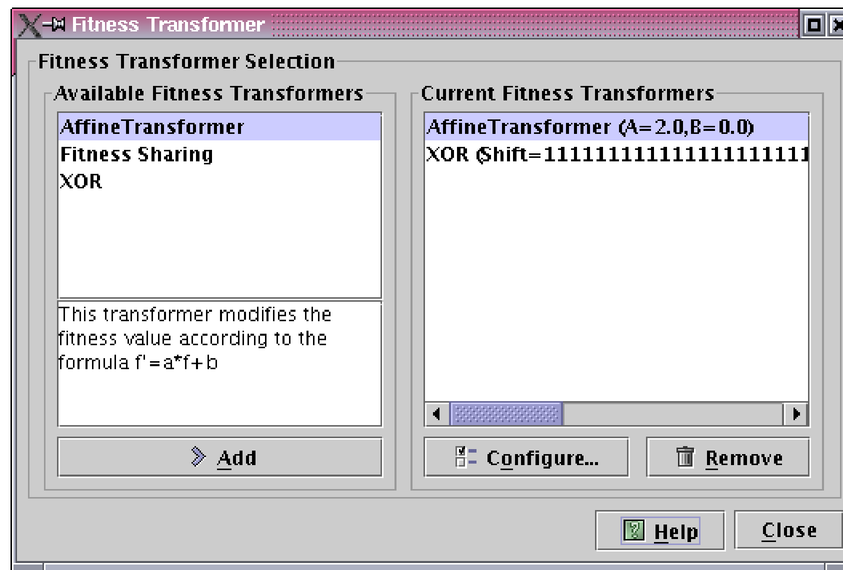


Figure 2.5: Defining fitness transformers

On the left side you can choose the fitness transformer you want to apply on your fitness function. Click on **Add** to use the selected transformer. The transformer will then appear in the list of your **Current Fitness Transformers** on the right. You can apply as many transformers as you want, they will be applied in the order specified by the list.

An example: Imagine you have added the fitness transformers as in Figure 2.5. The XOR transformer will become the fitness function of your schedule. If any part of the algorithm now wants to evaluate the fitness value of an individual it will pass the individual to this transformer. The XOR transformer then does its processing by applying an xor-bitmask on the individual thus creating a new individual. This new individual is passed to the next transformer, in this case the Affine Transformer. This transformer first fetches the fitness value of the next transformer, in this case from the real fitness function itself. Then it applies its transformation (being a multiplication with 2 in this case) and returns the resulting value as fitness value. This fitness value is the final fitness value your algorithm works with.

In this example you cannot only learn the function of fitness transformers in general, you can also see that there are fitness transformers (e.g. the XOR transformer) that do their transformation on incoming individuals rather than the fitness values they get from the underlying fitness function. Keep that in mind when you choose your transformers.

Of course, you can configure each transformer after adding it to the list of **Current Fitness Transformers** by clicking on the **Configure** button.

If you are satisfied with the fitness transformer setup, click on **Close** to get back to the schedule editor dialog. You will notice a small table containing information about your selected fitness transformers below the List of available fitness functions (see Figure 2.6).

Fitness Transformer

Transformer	Configuration
XOR	Shift=1111111111111111111111111...
AffineTransformer	A=2.0,B=0.0

Figure 2.6: Setup of fitness transformers.

But let's get back to editing the schedule...

2.4.3 Step 3: Create an Algorithm Graph

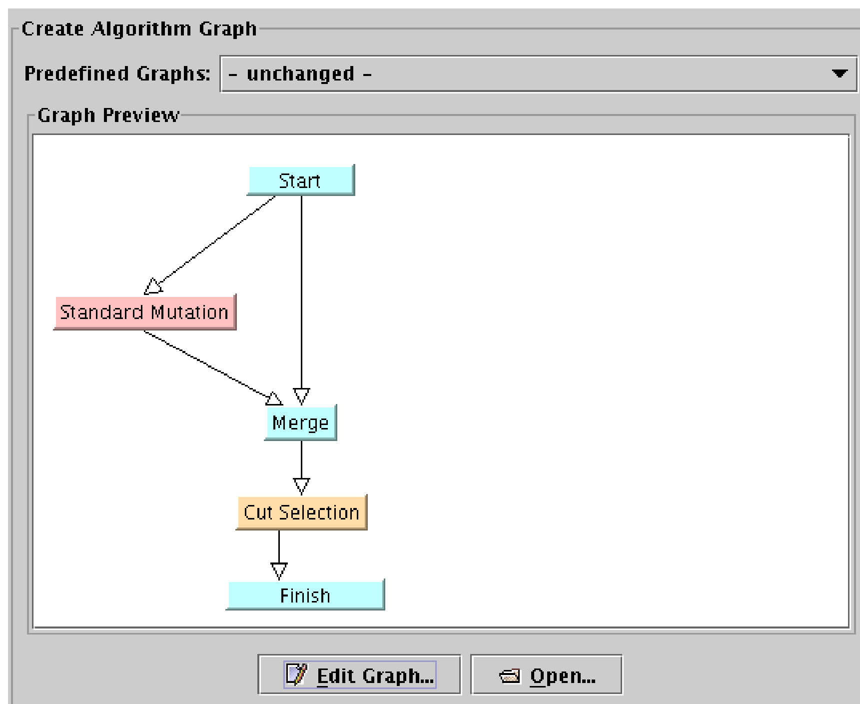


Figure 2.7: Creating an algorithm graph.

In this step you specify the *algorithm graph* which is the core of your algorithm. For further explanations on algorithm graphs, read Section 1.2: Evolutionary Algorithms or Chapter 4: Designing Algorithms.

In the center of the window, you see a preview of the current algorithm graph. You can either choose a predefined graph from the drop-down list on top, or create your own graph by clicking the button **Edit Graph**.

For now we use the default graph. To see how to create custom graphs, see Chapter 4: Designing Algorithms.

Press **Next**.

2.4.4 Step 4: Select a Stopping Criterion

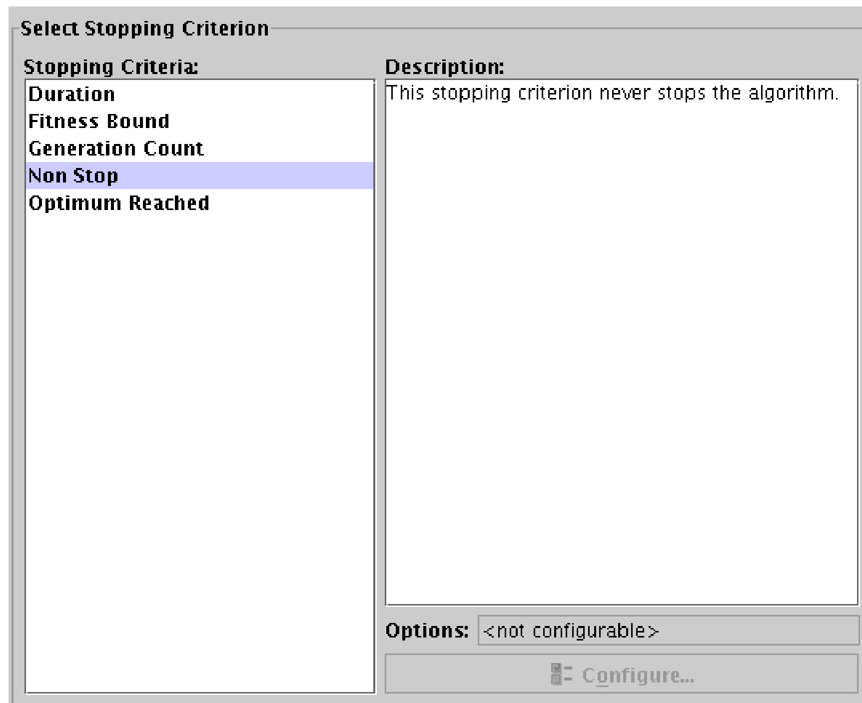


Figure 2.8: Selecting a stopping criterion.

Stopping criteria tell the algorithm to stop the current run if a specified condition is fulfilled. Select a stopping criterion from the list of available stopping criteria (Figure 2.8). Again, after selecting one you can configure it by pressing the **Configure** button.

Actually the Non Stop criterion is selected which lets your algorithm run forever (more precisely: until you stop it manually). Select Optimum Reached and press **Next**.

Note:
The Optimum Reached stopping criterion is a good example for a module that is not available with every fitness function. Fitness functions that don't know their optimal value will not allow this stopping criterion so that it won't be displayed in the list of available stopping criteria.

2.4.5 Step 5: Select Population Model and Initialization

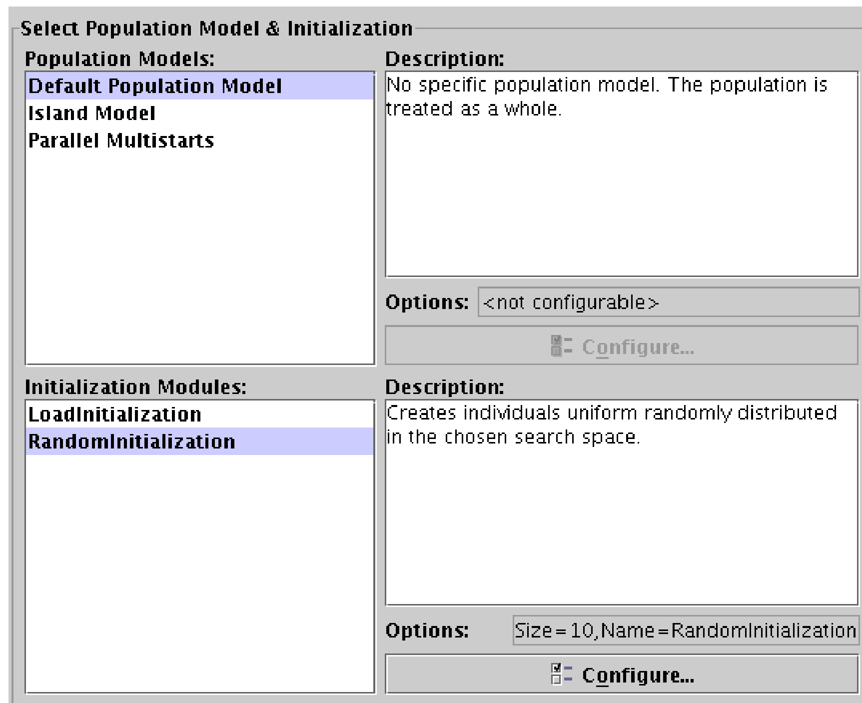


Figure 2.9: Selecting population model and initialization.

This step consists of two parts that both deal with the population of the algorithm and how this population is initialized. In the upper panel you select your *population model*. The population model is an optional choice and allows you to maintain subdivisions inside the population. The DEFAULT POPULATION MODEL is appropriate if you do not want to use subdivisions at all.

In the lower panel you select the *initialization module* which creates the initial population. You can lookup how these modules work in Appendix A: Modules Contained in FrEAK.

For now, leave the population model and the initialization module as it is to create a randomly chosen population of size 10.

With the completion of this step, you also completed the algorithm itself. What's missing is the collection and display of the data the algorithm produces. This is configured in the next step.

2.5 Preparing the Simulation

2.5.1 Step 6: Add Observers and Views

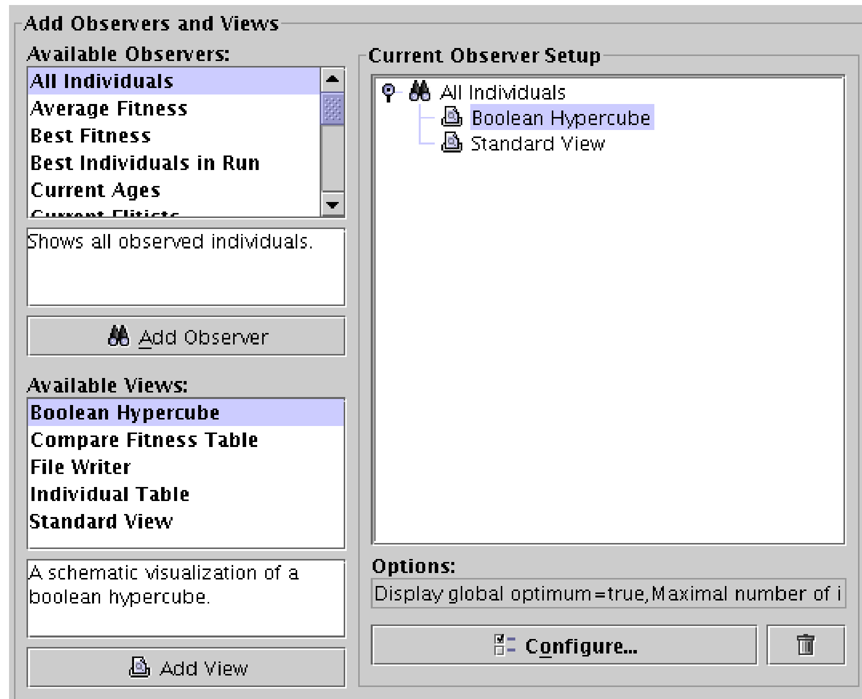


Figure 2.10: Adding observers and views.

Now that your algorithm is complete, you surely want to see what it does and how it performs. FrEAK uses two types of modules to accomplish your needs, *observers* and *views*. Observers collect and compute data that is produced by the algorithm and then displayed by one or several views on the screen or written to a file. There is (virtually) no limit of the number of observers and views you can use to process data.

The **Current Observer Setup** shows your current setup of selected observers and views in a tree view where observers are containers for views and views are leaves in the tree.

At first, the **Current Observer Setup** is empty, so you will most probably want to add an observer now. Select All Individuals from the list of available observers and press the **Add Observer** button or double click on the list entry to add the observer to your current setup. This Observer just forwards the individuals to its views so they can display them in whatever way they want.

Since observers without any views do not make sense, you should add some views now, too. Select Boolean hypercube from the list of available views and press **Add View** or double click on the list entry. The Boolean hypercube displays Bit String individuals in a graphical visualization. Also add the view Individual Table to the observer. This view displays all individuals in a table.

Note:
The list of available views depends on your selection in the **Current Observer Setup**. A view has to match the observer it is assigned to so the list contains all views that match the currently selected observer (if you selected an observer) or the owning observer of the view (if you selected a view).

You can delete observers and views by selecting them and pressing the trash can button.

You should now have the setup as shown in Figure 2.10.

2.5.2 Step 7: Creating Batches of Runs

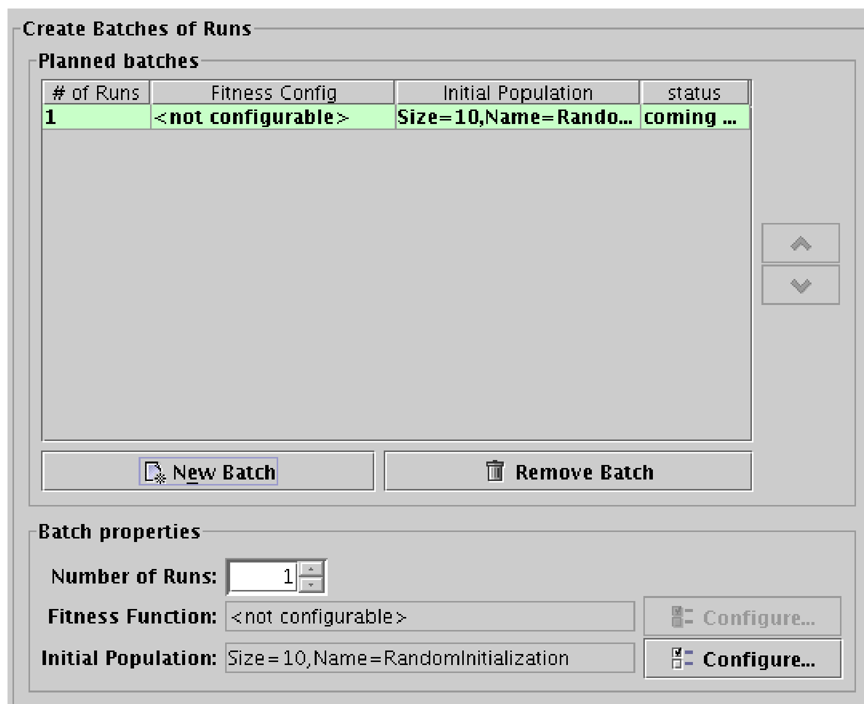


Figure 2.11: Creating batches of runs.

If you have read Section 2.1.2: What is a Batch?, you should know by now what a batch is. Here, you can setup your batches. Figure 2.11 shows the default batch that is created automatically and cannot be removed. The default batch is always the first and contains the configuration of your fitness function and initialization module as entered in the previous steps. As you can see, the table-row has a light green background and the last cell of the row (in the status-column) says "*coming up*". Rows can have 3 different colors which reflect the progress of the corresponding batch:

green background / status: coming up The batch has not yet started. You can modify any setting of this batch.

yellow background / status: running The batch is currently running. At least one run has already started. You can modify nothing but the number of runs this batch has. You cannot decrease the number of runs below the number of already completed runs (plus the actual running one) though.

red background / status: finished This batch has been finished completely. All runs have completed. Nothing can be changed here.

You can add batches and remove them with the **New Batch** and **Remove Batch** buttons and you can change their order with the arrow-buttons to the right. To modify the settings of a batch, first select it from the table by clicking on the corresponding row (the row will become bold). As soon as you select a batch, the **Batch Properties** panel at the bottom will show the configuration of it. You may then change the number of runs, the fitness function configuration and the configuration of the initial population. Changes are applied directly to the batch.

Try adding a few batches with different configurations and press **Finish**.

Well done. Your schedule is now complete. Now you might want to simulate (run) your schedule to see what it really does. This is covered in Chapter 3: Controlling Runs.

2.6 Configuration of Modules

2.6.1 Configuration of Properties

Whenever a module offers the option to be configured, the **Property Dialog** shown in Figure 2.12 (with varying contents of course) pops up:

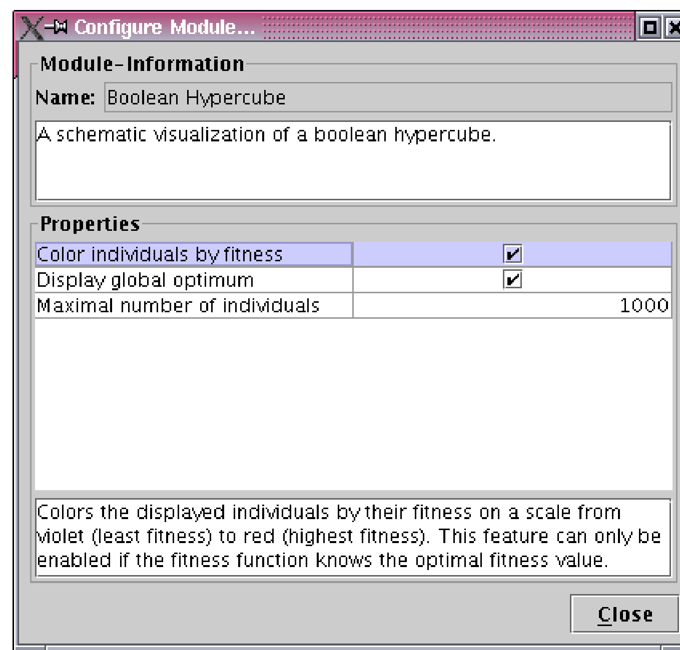


Figure 2.12: Property dialog of view BOOLEAN HYPERCUBE.

It displays some information about the module you are configuring, in particular the name and the description of the module as well as the properties the module allows to be configured.

Figure 2.12 shows the property dialog for a BOOLEAN HYPERCUBE. As you might have guessed, you can change the values displayed in the dialog to your needs. Numeric values can be edited, checkboxes can be set or unset, and so on.

2.6.2 Configuration of Event Sources

Some modules offer the possibility to be registered as event listener for certain events. An example for such a module is an observer that computes data from a specified set of individuals. This observer has to be notified when new individuals are available by an event. In fact, most observers work that way. So where does this event come from? Which other module is the *event source* for this event?

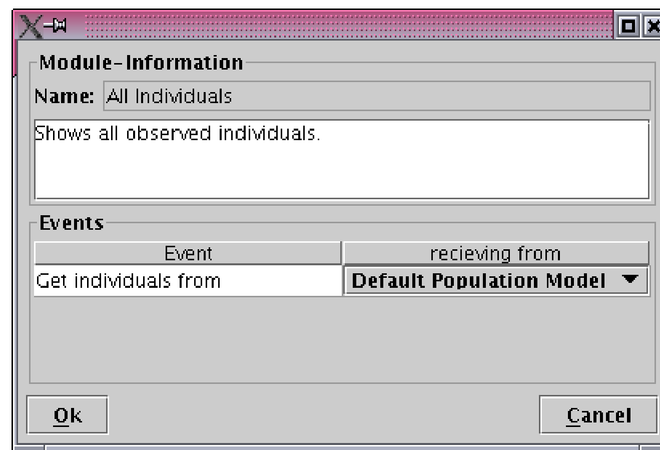


Figure 2.13: Property dialog of observer ALL INDIVIDUALS.

Modules that can be registered as event listeners have a **Property Dialog** as shown in Figure 2.13. (Here, the **Property Dialog** of the ALL INDIVIDUALS observer is shown.) You will notice a table of events each displayed with a name and an event source where the module receives this event from. The event source may or may not yet have been set to a valid source (in fact most, if not all, modules will be assigned to meaningful default event sources), but you can change the event source at any time by selecting an item from the drop down box of the event.

Some events allow only one event source (mainly the SCHEDULE itself), where other events even allow *outports* of operators of the graph to be selected as event source. See Figure 2.14 for an example.

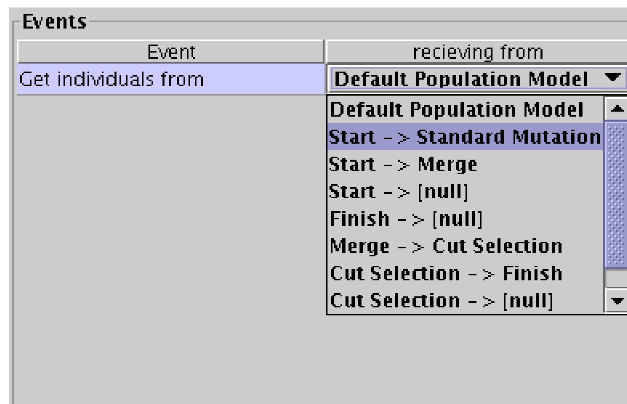


Figure 2.14: Possible event sources for the observer All Individuals.

For more information about operators and outports, see Section 4.1: Operator Graphs: Concepts.

Of course there are modules that have both properties and event sources to be configured. An example for these modules are the **Parameter Controllers** described in Section 4.3: Parameter Controllers.

If you are done with configuring the module, click on **OK** to return to the **Schedule Creation** dialog.

Chapter 3

Controlling Runs of Evolutionary Algorithms

3.1 Getting Started

When starting FrEAK, you see an empty window as shown in Figure 3.1.

Figure 3.1: The window after starting FrEAK.

Note that almost all menu items are disabled at the beginning. The first step to take is to create a new schedule by choosing **New** from the **File** menu. The creation of new schedules is described in Chapter 2: Creating a Schedule. Alternatively, you may open an existing schedule by choosing **Open** from the **File** menu.

The **Help** menu provides further information. You can open this document there and take a look at the **About** dialog.

When a schedule has been created or opened, you see a control panel on the left side and a desktop with the selected views on the right side. At the top of the control panel, the current search space and fitness function is displayed. Below, you see the current run and generation.

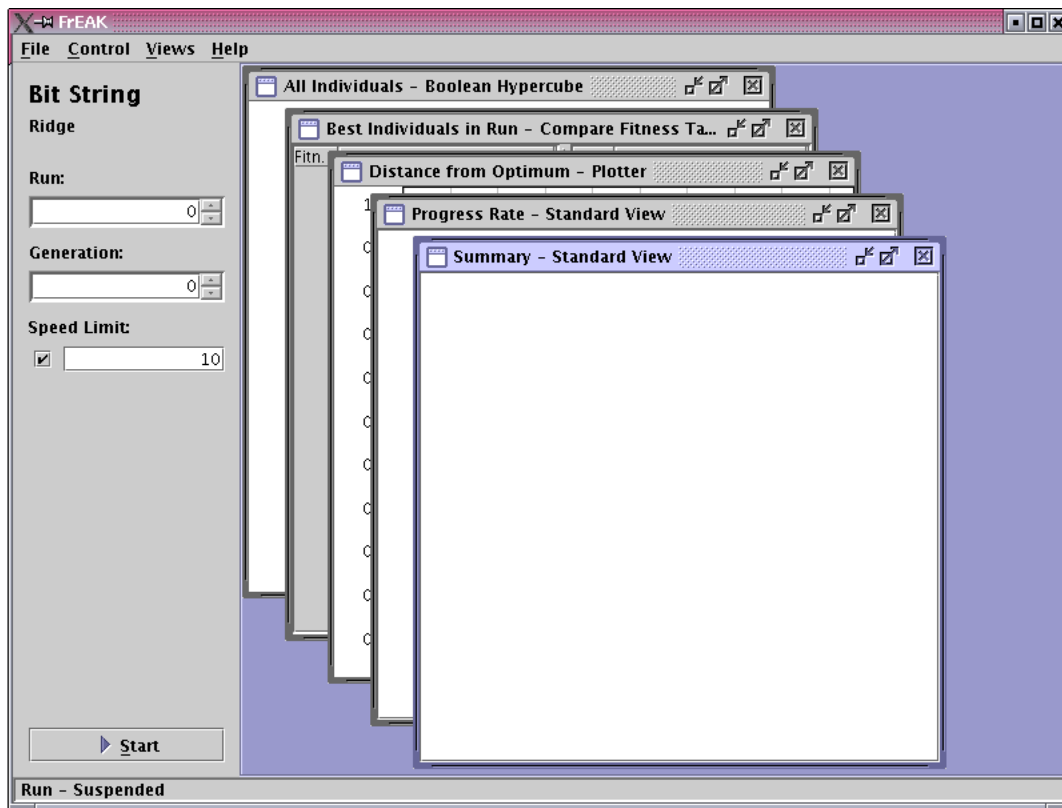


Figure 3.2: The FrEAK window after creating a schedule.

Before starting the simulation you can choose if you want to set a speed limit for the simulation. The speed limit represents the maximal number of generations per seconds and can be enabled and disabled using the corresponding check box.

With a click on the **Start** button you start the first run.

The current run can always paused by using the **Suspend** button. Some operations are only available while the run is suspended, such as editing the current schedule, modifying the speed limit, current run and current generation, or configuring views.

Press the **Start** button again and the simulation is continued.

3.2 Watching Replays

FrEAK distinguishes between *running mode* and *replay mode*. In replay mode, you can watch past runs and past generations in the current run. Switch to suspended mode by hitting the **Suspend** button and enter the desired run and generation number in the control panel. FrEAK automatically switches to replay mode and looks for the specified generation in the specified run. This may eventually take some time because parts of the run have to be simulated in the background to reach the desired generation.

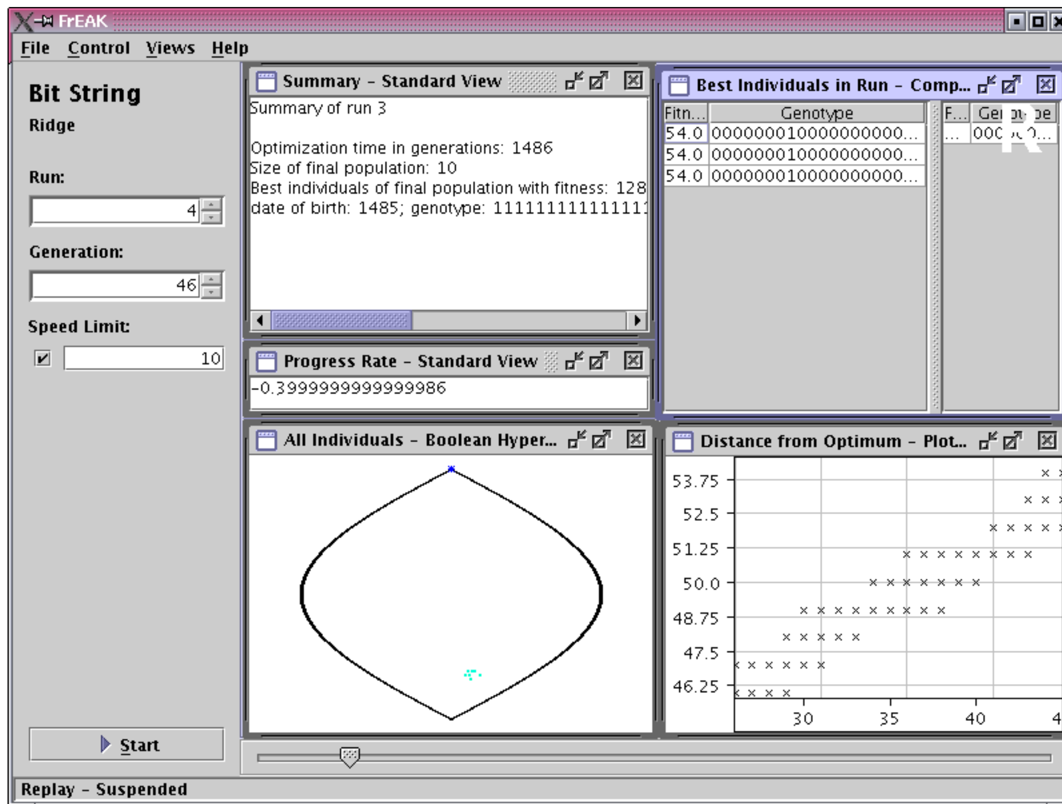


Figure 3.3: The FrEAK window during replay mode.

While in replay mode, a slider below the desktop is displayed indicating the current generation measured according to all generations that have already been simulated. Use the slider to quickly jump between generations. Furthermore, a white blinking “R” is shown in the background on the desktop to show that you are in replay mode.

Suspending is handled just like in running mode. You may pause the replay via the **Suspend** button anytime. Hitting **Start** again, FrEAK continues playing the replay.

When replay is finished and the latest simulated generation is reached, FrEAK automatically turns back to running mode and continues simulating new generations.

3.3 Editing the Current Schedule

You can modify the setup of the current schedule to a certain degree depending on the progress of it. This is done via the same dialog that was used to create a new schedule. To open this dialog, select **Edit...** from the **Control** menu. The handling of this dialog is explained in detail in Chapter 2: Creating a Schedule with the difference that several options are not available and you start with Step 7: Creating Batches of Runs if your schedule was already started. To be exact, the following things *can* be done with such a schedule: (every other setting can only be watched but not modified)

- You can select and configure another stopping criterion in Step 4: Select a Stopping Criterion.
- You can change the number of runs of upcoming (to any number) and running (not below the number of already finished/started batches) batches in Step 7: Creating Batches of Runs.
- You can change the configuration of the fitness function of upcoming batches (batches that have not yet been started) in Step 7: Creating Batches of Runs.
- You can change the configuration of the initial population of upcoming batches in Step 7: Creating Batches of Runs.

Note:
If you edit a schedule that was not yet started, the dialog works *exactly* the same as described in Chapter 2. If it was already started, several components (listboxes, buttons) will be disabled.

Please refer to Chapter 2: Creating a Schedule on how to work with the schedule editor.

3.4 Menu Items

The menu provides several functions which are described in detail here. Note that there are shortcuts for the most important functions.

3.4.1 File Menu

This menu is used for schedule management and to exit FrEAK.

New Create a new schedule with the Schedule Creation wizard. For details see Chapter 2: Creating a Schedule.

Open Open an existing schedule.

Save Save the current schedule.

Save as Save the current schedule requesting a file name.

Exit Exit FrEAK.

3.4.2 Control Menu

Contains several operations from the control panel.

Start Start or continue the current run.

Suspend Suspend the current run. Several operations are enabled only if the current run is suspended.

Skip Run Cancel the current run and initialize the following one. This menu item is only available if the current run is suspended.

Edit Edit the current schedule. For details see Section 3.3: Editing the Current Schedule.

3.4.3 Views Menu

This menu is used to manage the current views. The upper part contains general window management functions for the views. Below, all active views are listed and can be activated by choosing the corresponding menu item. Last, the selected view can be configured.

Tile Tile the views on the desktop. Minimized and closed views are not considered.

Cascade Reorder the views on the desktop in form of a cascade. Minimized and closed views are not considered.

Minimize All Minimize all visible views.

Restore All Restore all closed and minimized views.

Close All Close all opened views. You can restore all view windows with the menu item **Restore All** or simply click on the menu item corresponding to the view you want to restore.

Configure Selected View If the active view is configurable and the current run is suspended, you can configure the view in the configuration dialog by choosing this menu item. For help on the configuration dialog see Section 2.6: Configuration of Modules.

3.4.4 Help Menu

The **Help** menu offers you a global help.

Help

Tutorial

About

Chapter 4

Designing Algorithms with FrEAK

As mentioned in Chapter 1.2: Evolutionary Algorithms, FrEAK visualizes the control flow of evolutionary algorithms as operator graphs.

This chapter explains the main concepts of FrEAK's operator graphs (Section 4.1: Operator Graphs: Concepts) and describes how to work with the graphical editor for these operator graphs (Sections 4.2: The Graph Editor and 4.3: Parameter Controllers).

4.1 Operator Graphs: Concepts

This section explains the main concepts of FrEAK's operator graphs. If you are impatient, you can skip this section and return to it if you are confused by anything in the following sections of this chapter.

An operator graph visualizes how a given generation becomes the next generation. Operator graphs consist of *nodes* and *edges*. Each node defines an operator that works on some of the individuals of the population. Nodes have *ports* where edges can connect to (see Figure 4.1). Only one edge can be connected to a port. Edges define the flow of individuals.

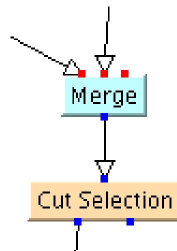


Figure 4.1: Operators with Ports.

The edges are directed, that means each edge has a source and a target port. Each port can only work either as a source or as a target port; those working as source port are called *inports*, those working as target port are called *outports*.

Most kinds of operators have a fixed number of inports and outports. But some kinds of operators, like Start, Finish, or Merge, can have any number of incoming or outgoing connections. In FrEAK, these operators are said to have *floating inports* or *floating outports*, respectively.

For instructions how to work with nodes and edges in operator graphs, please refer to Section 4.2.1, Adding and Removing Nodes and Edges.

Another important concept in FrEAK is that of *properties*. Many operators can be configured to some extent, a mutation operator for example may let you specify the mutation probability. Each attribute of an operator that is configurable is called a property. For instructions how to change properties of operators, please refer to Section 4.2.2, Changing Properties of Modules.

4.2 The Graph Editor

The graph editor is divided into three areas (see Figure 4.2).

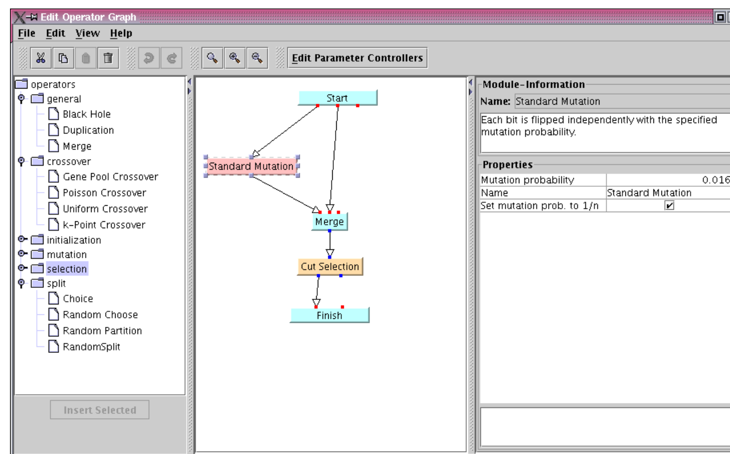


Figure 4.2: The Graph Editor Dialog.

The editor itself is found in the middle. To the left, there is a tree of operators that can be inserted into the graph, and to the right, information about the currently selected module is displayed. You can also edit properties of some of the operators here, see Section 4.2.2.

The menu and the tool bar provide standard editor commands like creating a new, empty operator graph, opening and saving operator graphs as well as cut/copy/paste and zooming the graph display.

Note:
It is only necessary to save the graph if you plan to reuse it later when creating other schedules.

In addition to these commands, you can open the Parameter Controller Setup with the appropriate tool bar button or by selecting **Edit Parameter Controllers** from the **Edit** menu. Section 4.3: Parameter Controllers describes how to work with parameter controllers.

When you have finished creating the operator graph, close the editor dialog by using the file menu's **Close** item. If your operator graph contains syntax errors, you will see a warning message (see Figure 4.3).

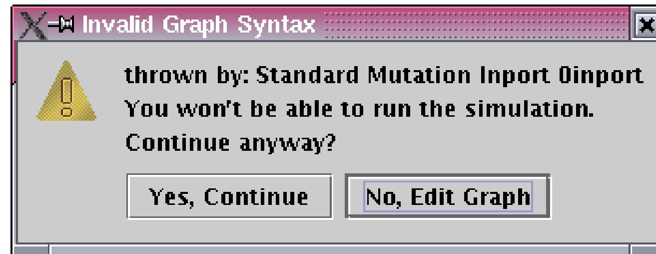


Figure 4.3: Error Message: Graph Has Syntax Errors.

Selecting **Yes, Continue** and trying to run the resulting schedule will most likely produce errors, so you should select **No, Edit Graph** and try to correct the error. Make sure there are no operators that have unconnected, non-floating inports and that the graph contains no cycles.

If the graph doesn't have errors or you have selected **Yes, Continue** on the warning message dialog, the editor dialog is closed and you return to the schedule creation wizard.

4.2.1 Adding and Removing Nodes and Edges

To add an operator to the graph, simply select it from the operator tree. You can insert the operator by double clicking or by using the **Insert Selected** button.

To connect two operators, select a free outport of the source node. Hold and drag the mouse to a free inport of the target node and release the mouse to establish the connection.

Note:
The mouse cursor changes to a hand when it is over a port that can be selected in the given situation. Please keep in mind that operator graphs must not contain cycles. You are not allowed to draw edges that would result in a cycle.

As explained in Section 4.1: Operator Graphs: Concepts, some operators have a floating number of inports or outports. The graph editor automatically inserts a new, unconnected port whenever a floating port is connected, so you don't have to take care about that. The graph editor also removes the disconnected port when an edge is removed. In other words, operators with floating ports always have an unconnected port.

Note:
While "normal" ports are drawn in blue, floating ports are drawn in red so you can recognize them easily.

To remove an element of the graph, select it with the mouse and press DEL or use the appropriate item from the tool bar or the menu. You can also click and drag the mouse to select multiple elements.

Note:
You cannot delete the Start or Finish operators.

4.2.2 Changing Properties of Modules

To the right of the graph editor itself is the property inspector of the operator that is currently selected in the operator graph (see Figure 4.4).

Figure 4.4: The Property Inspector

This inspector displays some information of the operator, such as its name, its description, and the description of its ports.

Underneath the description of the ports, you can see and edit the current values of the operator's properties. To edit a numeric or textual property, just doubleclick into the appropriate cell in the table and type the new value. Confirm your changes by pressing ENTER. Values that can only be either *true* or *false* can be set by checking or unchecking the checkbox.

At the bottom of the property inspector, the description of the selected property is displayed.

4.3 Using Parameter Controllers

The concept of parameter controllers has been added to FrEAK to account for adaption. A parameter controller contains some attributes which in this context are called *parameters*. A parameter controller can listen for some events, and during the handling of an event it can modify its parameters.

What is important is that the changes in the parameters are reflected in properties of operators in the graph. To be more precise, an operator and one of its properties can be assigned to a parameter. When the value of a parameter changes, the associated property is set to the same value.

4.3.1 Adding Parameter Controllers

You can open the parameter controller setup dialog from the graph editor dialog with the appropriate tool bar button or by selecting **Edit Parameter Controllers** from the **Edit** menu. In the left of this dialog, you can see the interface for adding and removing parameter controllers (see Figure 4.5).

You can add parameter controllers from the available list to the active list (by double clicking on them or by selecting and clicking the button between the two lists), and also select parameter controllers from the latter list and remove them by clicking the button labeled **Remove**.

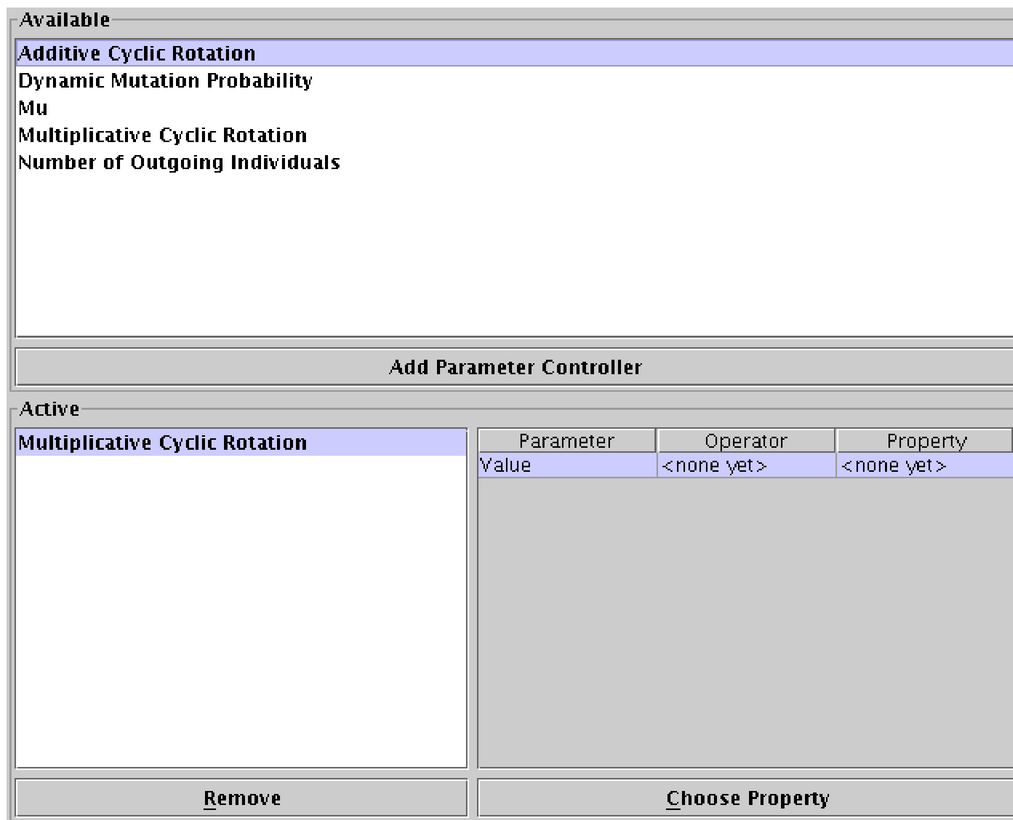


Figure 4.5: Adding parameter controllers.

Once you have added a parameter controller, you can configure it. This means mainly setting the sources for the events it can listen to and assigning operators with their properties to the parameters. Furthermore, the functionality of the parameter controller may be parametrized; in this case, you can modify some properties.

4.3.2 Assigning Event Sources and Configuring Parameter Controllers

If you click on a parameter controller in the active list, its properties and events will be displayed in the module information panel at the right of the parameter controller setup dialog (See Figure 4.6). In the subpanel labeled **Events** you find a table with one line for each event the parameter controller can handle, containing the description or name of the handled event and a drop down box which can be used for setting the event source for the event. Section 2.6.2: Configuration of Event Sources describes in detail how to specify event sources.

Above, you find the subpanel labeled **Properties** in which another table containing the properties of the parameter controller is located. Here you can set these properties and thus modify the parameter controller's behaviour.

Note:
The **Properties** subpanel is only available if the selected parameter controller is configurable.

Figure 4.6: The information panel for the selected parameter controller.

4.3.3 Assigning properties to parameters

If a parameter controller is selected in the active list, a table containing its parameters is displayed at the right of the parameter controller main panel (see Figure 4.7). To assign a property to a parameter, you must select the parameter in the table and click the button **Choose Property**.

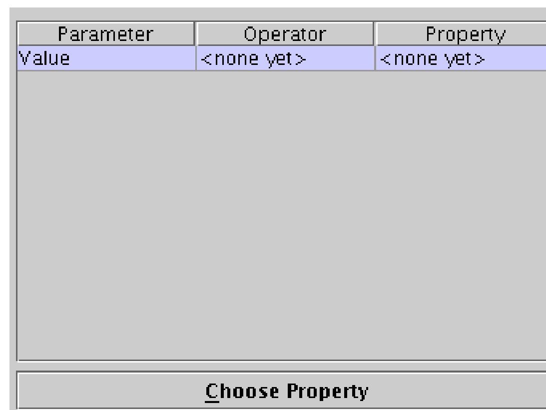


Figure 4.7: Setting up properties.

After doing this, you will see a window like the one in Figure 4.8. There is a list of all operators in the graph which contain properties compatible with the parameter you have selected. You can open the list of properties for each operator by clicking on the symbol left to a gate's name. After this, select the property you want and click **OK** to make the selection take effect.

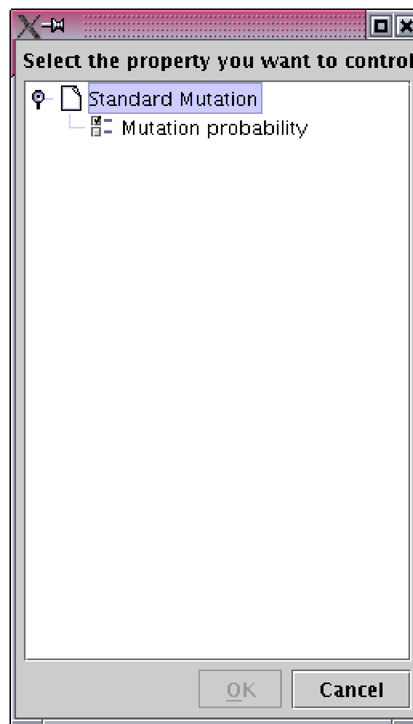


Figure 4.8: Selecting properties to control.

Appendix A

Modules Contained in FrEAK

A.1 Search Spaces

Bit String BIT STRING represents the well–known search space $\{0,1\}^n$. This search space is implemented separately from GENERAL STRING in order to obtain greater performance on this important search space.

Cycle CYCLE represents the search space consisting of all permutations in the symmetric group S_n which form a cycle (e.g. a TSP-tour). That are all permutations of the elements $1, ..., n$ which form a cycle.

General String GENERAL STRING represents the search space $\{0, ..., k-1\}^n$ with $0 < k < 256$.

Permutation PERMUTATION represents the search space consisting of all permutations in the symmetric group S_n. That are all permutations of the elements $1, ..., n$.

A.2 Fitness Functions

A.2.1 Fitness Functions on Bit Strings

OneMax This fitness function simply counts the number of set bits within the genotype of the individual. Obviously, it ranges from 0 to n.

So, the fitness value of an individual with genotype $x = (x_1, \ldots, x_n)$ is

$$\text{OneMax}(x) := \sum_{i=1}^{n} x_i.$$

LeadingOnes The fitness function LeadingOnes calculates the size of the block of set bits which begins at the leftmost bit. Therefore, the fitness value ranges from 0 to n.

The fitness value of an individual with genotype $x = (x_1, \ldots, x_n)$ is

$$\text{LeadingOnes}(x) := \sum_{i=1}^{n} \prod_{j=1}^{i} x_j.$$

Needle The fitness function Needle is easy to define, but hard to optimize. The fitness value of each individual is zero except for one individual, the so called *needle* with the genotype $x = (1, \ldots, 1)$. So, an evolutionary algorithm gets no information about the location of the optimum, it searches for the needle in the haystack.

The fitness value of an individual with genotype $x = (x_1, \ldots, x_n)$ is

$$\text{Needle}(x) := \prod_{i=1}^{n} x_i.$$

LongPathK The definition of the fitness values belonging to this fitness function consists of two parts: The values of search points belonging to a path P and the values of search points not belonging to the path P. Let $k \in \{1, \ldots, n\}, k \mid n-1$.

The path P is defined recursively by $P_k^1 := (0, 1)$ and

$$P_k^n := \left(0^k v_1, 0^k v_2, \ldots, 0^k v_l, 0^{k-1} 1 v_l, \ldots, 01^{k-1} v_l, 1^k v_l, 1^k v_{k-1} \ldots, 1_1^v\right)$$

where $P_k^{n-k} = (v_1, \ldots, v_l)$.

For genotypes on the path, the fitness value is $n^2 + i$ where i is the position on the path. For the other individuals it is $n^2 - \left(n \cdot \sum_{i=1}^{k} x_i\right) - \sum_{k+1}^{n} x_i$.

So, the fitness value of an individual with genotype $x = (x_1, \ldots, x_n)$ is

$$\text{LongPath}_k(x) := \begin{cases} n^2 + i & \text{if } x = v_i, \\ n^2 - n \cdot \sum_{i=1}^{k} x_i - \sum_{i=k+1}^{n} x_i & \text{otherwise.} \end{cases}$$

H–IFF This fitness function only works on search spaces with dimension $n = 2^k$. Within the individual's genotype, a bit string of length n, all blocks with length 2^i for each $i \in \{1, \ldots, k\}$ are created. If all bits within a block are equal, the length of the block is added to the fitness value.

So the fitness value of an individual with genotype $x = (x_1, \ldots, x_n)$ is

$$\text{H–IFF}(x) := \sum_{h=0}^{k} \left(2^h \cdot \left(\sum_{i=0}^{2^{k-h}-1} \left(\prod_{j=1}^{2^h} x_{2^i \cdot i+j}\right) + \left(\prod_{j=1}^{2^h} (1 - x_{2^i \cdot i+j})\right)\right)\right).$$

JumpK This fitness function is similar to OneMax, but it contains a gap of low fitness values giving hints to move away from the optimum. The gap is specified by a parameter $k \in \{1, \dots, n\}$ and contains all genotypes with a OneMax value within $\{n-k+1, \dots, n-1\}$.

For individuals outside the gap, the fitness value is OneMax$+k$, thus rising with the OneMax value. But for individuals inside the gap, the fitness value is $n -$ OneMax, thus falling with the OneMax value.

So the fitness value of an individual with genotype $x = (x_1, \dots, x_n)$ is

$$\text{Jump}_k := \begin{cases} n - \sum_{i=1}^{n} x_i & \text{if } n - k < \sum_{i=1}^{n} x_i < n, \\ k + \sum_{i=1}^{n} x_i & \text{otherwise.} \end{cases}$$

PathToJump The fitness of an individual with genotype $x = (x_1, \dots, x_n)$ is defined as

$$\text{PathToJump}(x) := \begin{cases} n + i & \text{if } x = 1^i 0^{n-i}, \\ 3n & \text{if } x \in I, \\ n - \sum_{i=1}^{n} x_i & \text{otherwise,} \end{cases}$$

where I is the island of optimal search points containing all genotypes x fulfilling the following conditions: the OneMax value is in $(\frac{3}{4}n, \frac{7}{8}n)$ and the distance of x to every point on the path $1^i 0^{n-i}$ is at least $\frac{n}{16}$. This fitness function shows that the dynamic (1+1) EA can outperform each static (1+1) EA drastically.

PathWithTrap The fitness of an individual with genotype $x = (x_1, \dots, x_n)$ is defined as

$$\text{PathWithTrap}(x) := \begin{cases} 3n & \text{if } x = 1^n, \\ n + i & \text{if } x = 1^i 0^{n-i}, \\ 2n & \text{if } x \in T, \\ n - \sum_{i=1}^{n} x_i & \text{otherwise,} \end{cases}$$

where T is the trap of only locally optimal search points. The trap contains all genotypes x fulfilling the following conditions: the OneMax value is in $(\frac{n}{4}, \frac{3}{4}n)$ and a point on the path $1^i 0^{n-i}$ exists with a distance to x in the interval $(\frac{n}{12}, \frac{n}{16})$. Furthermore, the distance of x to every point on the path has to be at least $\frac{n}{24}$.

This fitness function shows that the dynamic (1+1) EA can get trapped and be outperformed by some (1+1) EAs.

Plateau The individuals with genotypes $1^i 0^{n-1}, 0 \leq i < n$ form a plateau of equal fitness values, they all get fitness values $n + 1$. The optimum is 1^n, it gets the fitness value $2n$. All other individuals get fitness $n-$OneMax, thus giving hint to reach the plateau near 0^n.

So, the fitness value of an individual with genotype $x = (x_1, \ldots, x_n)$ is

$$\text{Plateau}(x) := \begin{cases} n+1 & \text{if } x = 1^i 0^{n-i}, i \in \{1, \ldots, n-1\}, \\ 2n & \text{if } x = 1^n, \\ n - \sum_{i=1}^{n} x_i & \text{otherwise.} \end{cases}$$

Ridge This fitness function is very similar to PLATEAU, but individuals on the plateau (genotypes of form $1^i 0^{n-i}$) get fitness value $n + i$.

So, the fitness value of an individual with genotype $x = (x_1, \ldots, x_n)$ is

$$\text{Ridge}(x) := \begin{cases} n+i & \text{if } x = 1^i 0^{n-i}, i \in \{1, \ldots, n\}, \\ n - \sum_{i=1}^{n} x_i & \text{otherwise.} \end{cases}$$

Real Royal Road for 1pt Crossover This fitness function was created to demonstrate (provably) that crossover can be essential. Like in JUMPK, there is a gap which has to be crossed.

So, the fitness value of an individual with genotype $x = (x_1, \ldots, x_n)$ is

$$\text{Real Royal Road for 1pt Crossover}(x) := \begin{cases} 2n^2 & \text{if } x = 1^n, \\ n \cdot \sum_{i=1}^{n} x_i + b(x) & \text{if } \sum_{i=1}^{n} x_i \le n - \frac{n}{3}, \\ 0 & \text{otherwise,} \end{cases}$$

where $b(x)$ is the length of the longest coherent block consisting only of ones.

A.2.2 Fitness Functions on General Strings

Championship Problem The championship problem deals with the soccer-league. There is one club, which is your favorite club, and at any within the current season you ask yourself: "Is it possible for my club to win the league?". At this point of time, each club has a certain amount of points and the games which aren´t played yet are also fixed. Also the distribution of points per game is known. With this information one can try to solve it, but in general this problem is NP-hard.

In this fitness function the games are represented by three values: 0,1,2, which have the meaning of the three possible results in a game: lose, win, and draw.

To configure the fitness function one can specify the number of clubs playing in the league, the points for all clubs in the league's current situation and the games which aren´t played yet. The rule of point distribution can be specified, too.

The fitness function will interpret an individual as a list of game results and calculate the points according to it. By convention, it is sufficient not to have less points than the leading club.

A.2.3 Fitness Functions on Permutations

Sorting The problem of sorting a sequence can be considered as an optimization problem where the goal is to maximize the *sortedness* of the sequence. Precisely, a permutation of the given sequence is searched which sorts this sequence. W. l. o. g., the sorted sequence is the identical permutation $(1, 2, ..., n)$ with n the dimension of the search space.

There are five different measures of sortedness implemented in FrEAK. Some of these measures lead to minimization problems. To gain maximization problems, the fitness value in these cases is computed by the measure multiplied with -1.

- INV(π) measures the number of pairs in the correct order.
- HAM(π) measures the number of elements at the correct position.
- RUN(π) measures the number of maximal sorted blocks, called *runs*.
- REM(π) measures the length of the longest sorted subsequence.
- EXC(π) equals the minimal number of exchanges to sort the sequence.

A.2.4 Fitness Functions on Cycles

Travelling Salesperson Problem There is a number of locations named $1, 2, \ldots, n$ with n the dimension of the search space, and a person who wants to visit all of them and return to the starting position with minimal cost. There is a cost matrix, which specifies the cost to travel between two locations.

So, a permutation in cycle representation is searched, which has minimal cost.

At this point of time, the cost matrix can only be created randomly by the program.

Euclidean Travelling Salesperson Problem The same problem like TSP, but instead of a cost matrix, the euclidean distance between points in $\mathbb{R}^2$ will be computed for the cost of the tour.

A.2.5 Fitness Transformers

Affine Transformer Performs an affine transformation on the fitness values: if the fitness function returns the fitness value $f(x)$, the transformer returns $a \cdot f(x) + b$.

Note that a transformer will not be able to localize the optimal individual (for use in the analysis only) if the parameter a is not positive, even if the original fitness function can supply this information.

Fitness Sharing To enhance diversity, fitness sharing derates the individual fitness by dividing it by the individual niche count. The niche count is the sum of shared function (sh) values, calculated by distances between the individual and all individuals of the population.

$$F'(i) := \frac{F(i)}{\sum_{j=1}^{\mu} sh(d(i,j))}$$

If the distance between two individuals is greater or equal to the parameter σ_{share}, they do not affect each other. The parameter α determines the shape of the sharing function (sh).

$$\mathrm{sh}(d) := \begin{cases} 1 - (d/\sigma_{share})^{\alpha} & \text{if } d < \sigma_{share}, \\ 0 & \text{otherwise.} \end{cases}$$

XOR Transformer This transformer only works on Bit String individuals. Sometimes, you may wish to move the optimum of a fitness function, for example to check if your algorithm handles 1s like 0s in the bit string. Then you can wrap it with this transformer to modify the fitness function.

Every time a fitness value is requested for an individual its genotype is xor-ed with a (configurable) bit string. So, the resulting fitness function is a different one.

A.3 Operators

A.3.1 General Operators

Black Hole A dummy operator used to express a dead end for individuals.

Duplication Duplicates the incoming individuals for all outports.

Merge All incoming individuals are merged together. Note that the result is a not a set but rather a multi-set, so if an individual is contained in multiple input lists, there will be duplicates in the resulting set of individuals.

A.3.2 Crossover

Bit String

Gene Pool Crossover Takes a whole population of size μ as input and creates a specified number of children. The number k of ones at position i in the population (gene pool) is counted and the i-th bit in each child is set to one with probability k/μ.

k-Point Crossover Chooses k crossing points at random. The offspring will be generated by substrings of both individuals. The first substring of bits will be taken from the first parent until the first crossing point. The next substring will be taken from the other parent until the next crossing point. This procedure is repeated until the whole genotype is processed.

Poisson Crossover Each position becomes a crossing point with a specified probability. So, the distance between two crossing points is negative binomially distributed (asymptotically this equals a poisson distribution).

Uniform Crossover Each bit is taken with probability 1/2 from the first parent and with probability 1/2 from the second parent.

General strings

Gene Pool Crossover Takes a whole population of size μ as input and creates a specified number of children. For each character, the number of occurrences k_{char} at position i in the population (gene pool) is counted and the i-th position in each child is set to the character with probability k_{char}/μ.

k-Point Crossover Chooses k crossing points at random. The offspring will generated by substrings of both individuals. The first substring of bits will be taken from the first parent until the first crossing point. The next substring will be taken from the other parent until the next crossing point. This procedure is repeated until the whole genotype is processed.

Poisson Crossover Each position becomes a crossing point with a specified probability. So, the distance between two crossing points is negative binomially distributed (asymptotically this equals a poisson distribution).

Uniform Crossover Each bit is taken with probability 1/2 from the first parent and with probability 1/2 from the second parent.

Random Respectful Crossover The positions which are identical in both parents are taken over, the characters at the other positions are chosen at random.

Cycle

Inver-Over This is a special operator for the TSP. This operator mutates every individual with connections between cities from other randomly selected individuals. Each individual competes with its offspring.

Maximal Preservative Crossover Chooses two crossing points at random and takes over the subtour in the middle part of the second individual. Then the operator tries to use edges in the parents to complete the tour. Edges in the first individual have a greater priority than those in the second individual. If both parents don't have permitted edges starting at the last city visited, then the first following city in the first individual which hasn't been visited yet becomes the next city on the tour.

Common

Note:
The following crossover operators work on permutations and on cycles. On both search spaces they operate syntactically identical, which means that the semantic depends on the chosen search space. To understand the following descriptions of the operators you need to know that the genotypes of permutations and cycles are stored both in arrays. But these arrays are interpreted as function tables in connection with permutations and they are interpreted as cyclic notations of one cycle in connection with cycles.
Consider the genotype 841593627. In connection with permutations it represents the permutation $1 \to 8$, $2 \to 4$, $3 \to 1$, $4 \to 5$, $5 \to 9$, $6 \to 3$, $7 \to 6$, $8 \to 2$, $9 \to 7$. In connection with cycles it represents the permutation $8 \to 4$, $4 \to 1$, $1 \to 5$, $5 \to 9$, $9 \to 3$, $3 \to 6$, $6 \to 2$, $2 \to 7$, $7 \to 8$.

Order Crossover This operator works on permutations and on cycles.
Order crossover is based on 2-point crossover operations. At first, two crossing points are chosen at random. These crossing points divide each individual into 3 parts.
For example
12|3456|789 and
84|1593|627.
The information in the middle part of the first individual is taken over, so we get the permutation
??|3456|???
The remaining values $(1, 2, 7, 8, 9)$ are inserted (starting in the right part) in the order $(2, 7, 8, 1, 9)$ in which they appear in the second individual staring in the right part.
So, the created child is:
193456278.

Order Crossover 2 This operator works on permutations and on cycles.
First, k is chosen uniformly at random from $\{1, ..., n-1\}$ (for permutations / cycles out of S_n). Then, k randomly chosen positions are marked. We take over these k positions from the second individual in the order in which they are found in the first individual.
Example:
ind1 = 123456789
ind2 = 841593627
$k = 4$, marked are the positions $2, 4, 6, 8$.
We take from the second individual $4, 5, 3, 2$.

The order of these elements in the first individual is $2, 3, 4, 5$. That's why the child is of the form
?2?3?4?5?
Now we add the remaining positions from ind2:
821394657.

Partially Mapped Crossover This operator works on permutations and on cycles. Partially Mapped (or Matched) Crossover (PMX) is a well known crossover operator for cycles.

A.3.3 Initialization Operators

Random Initialization Creates the specified number of individuals uniformly at random distributed within the search space.

A.3.4 Mutation

Bit String

k-Bit Mutation Flips exactly k randomly chosen bits per bit string.

Standard Mutation This operator represents the default mutation on $\{0, 1\}^n$. It flips every bit independently from the others with a specified mutation probability. The default value for the mutation probability is $1/n$, so in mean only one bit flips per bit string while big mutations still remain possible.

General String

k-Char Mutation Alters exactly k randomly chosen positions. Altering a position means that the character is replaced by a different randomly chosen character.

Standard Mutation Alters each position independently from the others with a specified mutation probability. Altering a position means that the character is replaced by a different randomly chosen character.

Cycle

k-Opt This operator cuts the permutation into k segments and reorders them randomly. Each segment can be placed in a forward or reverse order.

Common

Exchange This operator works on permutations and cycles and performs a number of exchange operations which is poisson distributed with the specified parameter λ (Lambda). With every exchange operation, two elements at randomly chosen positions are exchanged.

Jump This operator works on permutations and cycles and performs a number of jump operations which is poisson distributed with the specified parameter λ (Lambda). With every jump operation, a randomly chosen element at position i jumps to a randomly chosen position j while the other elements between position i and position j are shifted in the appropriate direction.

Jump & Exchange This operator combines jump and exchange operators. Again, the operator executes a number of local operations which is poisson distributed with the specified parameter λ (Lambda). When executing a local operation, either a jump operation or an exchange operation is performed. The probability that exchange will be used as local operation can be specified.

A.3.5 Selection

Cut Selection Selects the individuals with highest fitness values deterministically. You can adjust the number of selected individuals by configuring the corresponding parameter.

If some individuals have the same fitness value, the individuals are preferred, which are created within the current generation. So you get children instead of their parents.

Fitness Proportional Selection A specified number of individuals are selected randomly according to their fitness values. With the normal configuration, the probability for in individual to be selected is proportional to its fitness value. With the option *inverse* enabled, individuals with lower fitness are preferred:

$$Prob(x \text{ selected}) = \begin{cases} \frac{f(x)}{\sum\limits_{x \in P} f(x)} & \text{normal selection,} \\ \frac{\max\limits_{x \in P}\{f(x)\} - f(x) + 1}{\sum\limits_{x \in P} f(x)} & \text{inverse selection.} \end{cases}$$

You can also specify the option to get only unique individuals, so no individual is returned twice or more.

Tournament Selection This operator performs tournaments with sets of individuals and selects the winners. First, the individuals are chosen randomly from the population and then the individual with the highest fitness is determined. This individual is put into the resulting population and the procedure is repeated until enough individuals are chosen.

The size of the tournaments and the number of selected individuals can be specified.

Uniform Selection Selects individuals uniform randomly from the population.

A.3.6 Split

Random Choice This operator chooses one of its outports uniformly at random and sends all incoming individuals there.

Random Partition This operator distributes the incoming individuals equally over the outports. So each outport receives the same number of individuals. Of course this operator is only applicable if the number of incoming individuals is a multiple of the number of outports.

Random Split The individuals are split into different lists. The number of lists to create is configurable. The individuals are distributed according to the uniform distribution, which means the probability for each list and individual is equal.

A.4 Population Models

Default Population Model The default model that does not use subdivisions but simply treats the population as a whole.

Island Model The population is partitioned into a number of subpopulations, so–called *islands*. Then each island has a number of generations of isolated evolution, called *epoch*. After an epoch a number of individuals is exchanged throughout the islands.

Parallel Multistarts Subdivides the population into subpopulations of size 1. That way, multiple parallel runs of the algorithm with one individual are simulated.

A.5 Parameter Controllers

Additive Cyclic Rotation This controller rotates a parameter between an upper and lower bound. Each generation a fixed (but configurable) value is added to the current value of the parameter. If it exceeds the upper bound (drops below the lower bound) it is set to the lower (upper) bound.

Dynamic Mutation Probability This parameter controller can be used to simulate a dynamic (1+1) EA. It controls one real–valued property which in most cases will be the mutation probability of a mutation operator. First, this property is set to $\frac{1}{n}$ (where n denotes the dimension of the search space). Then, it is doubled each time a new generation is created. If the property reaches a value larger than $\frac{1}{2}$ it is set back to $\frac{1}{n}$.

Mu This controller sets the controlled property to the number of individuals put into the operator graph. This is very convenient if you´re going to implement a $(\mu + \lambda)$EA because the number of individuals put into the graph has to be the number of individuals coming out of the graph. So the number of selected individuals may be set to μ with this controller.

Multiplicative Cyclic Rotation This controller rotates a parameter between an upper and lower bound. Each generation the current value is multiplied with a fixed (but configurable) value. If the current value exceeds the upper bound (drops below the lower bound) it is set to the lower (upper) bound.

Number of Outgoing Individuals Sets the controlled parameter to the number of individuals going through the observed outport.

A.6 Stopping Criteria

Duration This stopping criterion measures the time passed since the start of the current run. If it exceeds the configured period, the run is stopped.

Generation Count This stopping criterion simply counts the generations up to a previously set number. If this number of generations is reached, it stops the current run.

Non Stop This isn´t a stopping criterion in its original sense. It tells the program to run non stop, so the run has to be stopped manually.

Fitness Bound Sometimes the optimal fitness value of a fitness function isn't known or the user is just interested in approximating an optimal search point. In these cases he can use the stopping criterion FITNESS BOUND. A fitness bound can be specified and if it is reached the algorithm is stopped. Furthermore, it can be configured how often the criterion is checked and if either the algorithm is stopped as soon as one search point has reached the fitness bound or not until the whole population has reached the bound.

Optimum Reached This stopping criterion stops the current run if either a individual or the whole generation reaches the optimum. This choice is configurable. The period with which this checks are performed can also be specified.

A.7 Observers

All Individuals This observer simply collects all observed individuals.

Average Fitness Computes the individuals' average fitness value.

Best Fitness Computes the best fitness value within the individuals.

Best Individuals in Run Computes the best individual seen in the current run. Please note that to work properly, the fitness of an individual must be deterministic and independent of its environment (e.g. the surrounding population or the number of the current generation). The fitness function must return the same fitness value for an individual with every evaluation.

Current Ages Computes the individuals' ages in generations.

Current Elitists Computes the elitists, the best individuals within the observed individuals.

Current Fitness Computes the individuals' current fitness values.

Distance from Optimum This observer requires a fitness function providing a metric to measure the distance between genotypes and a unique global optimum. Then, the distances of the individuals to the global optimum are computed.

Diversity The observer requires a search space with a metric. It computes the diversity within the individuals, which is defined here as the average distance between pairs of individuals.

Fitness Variance Computes the variance within the individuals' fitness values. The variance is defined as sum of the squared differences between the individual´s fitness value and the average fitness value.

Number of Individuals This observer simply computes the number of observed individuals.

Progress Rate A metric on the search space and a fitness function providing a unique global optimum is required to use this observer. The progress rate measures the decrease in the average distance to the global optimum. It is defined as the difference between the average distance to the optimum within the current population and last generation's population.

Quality Gain Computes the gain in average fitness. The quality gain is defined as the difference between the average fitness within the current population and last generation's population.

Run Time Computes the run time of all observed runs, that is the number of generations until the selected stopping criterion stops the run.

Selection Intensity Computes the selection intensity, which is defined here as the quota of individuals born in the current generation.

Summary Draws a summary of all runs.

A.8 Views

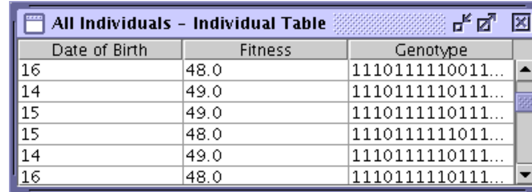
Boolean Hypercube A schematic visualization of a boolean hypercube. The individuals are drawn as small circles. The y–position of such a circle is determined by the number of ones in the genotype, the x–position is determined by the position of the ones in the genotype: the leftmost position of all individuals with i ones belongs to the genotype 1^i0^{n-i}, the rightmost position belongs to the genotype $0^{n-i}1^i$.

Figure A.1: Boolean Hypercube View

File Writer Writes the displayed textual data into a file. You may choose a file in the configuration dialog; if no file is specified, a new file named like `data0000.log` is generated automatically with a unique number.

Figure A.2: File Writer View

Individual Table This view shows the individuals in tabular form. Each row represents one individual and the columns show the date of birth (in generations), genotype and current fitness value of the corresponding individual.

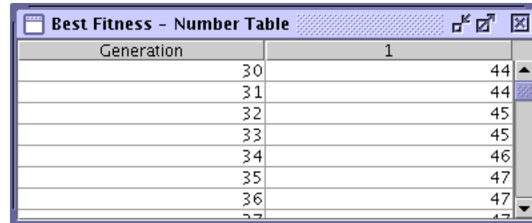


All Individuals - Individual Table

Date of Birth	Fitness	Genotype
16	48.0	1110111110011...
14	49.0	1110111110111...
15	49.0	1110111110111...
15	48.0	1110111111011...
14	49.0	1110111110111...
16	48.0	1110111110111...

Figure A.3: Individual Table View

Number Table This view displays the given numbers for the last generations. The number of generations that are to be displayed can be specified in the configuration dialog or set to the value 0 to allow unlimited storage.



Best Fitness - Number Table

Generation	1
30	44
31	44
32	45
33	45
34	46
35	47
36	47

Figure A.4: Number Table View

Plotter A Plotter displaying the received numbers over generations.

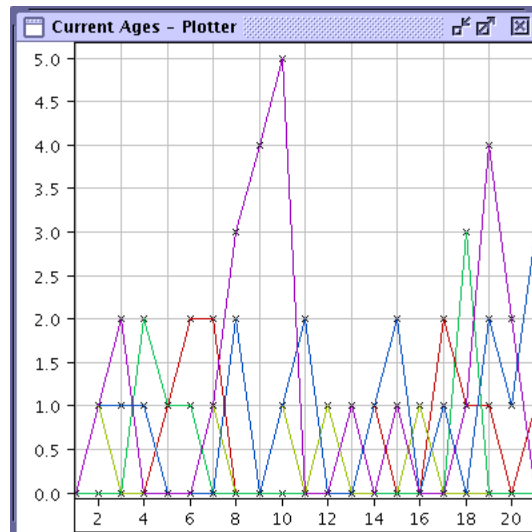


Figure A.5: Plotter View

Standard View Creates a textual output from the given data.

Figure A.6: Standard View