

EA Visualizer Tutorial

Peter A.N. Bosman

Utrecht University



Department of Computer Science

June 1999

Preface

When the first working version¹ of the *EA Visualizer* was finished in 1998, it was tested by a group of users. Even though the system was accepted as being a powerful and useful one as well as being self contained, it was generally felt that the program would be more accessible to a greater audience when a tutorial with an operational setting would be available. Without such a tutorial, getting to know and understand the system is a task that requires some interest and curiosity as the help system must be explored in order to find explanations for system parts. Having a tutorial with a great amount of examples of a great variety would be easier to use in quickly getting a feel for the system as it directly allows the user to properly create an EA and run it without making quick and therefore mostly wrong decisions in defining an EA. So, even though the *EA Visualizer* is a self contained enough system to directly use and work with, this tutorial will greatly aid in quickly getting to know the basics of the system.

This tutorial is thus a guide to the *EA Visualizer* system. The reader is presented with a description of how to operate parts of the system without going into technical implementation details. After describing the system parts, a set of examples is given for both the single runs and the multiple runs way of running EAs. In this way, the reader can directly try out the system while reading this tutorial, giving a good feel of what it is capable of and of how things work. Next to such an operational description, other parts such as menu items are all described as well, so as to make this tutorial self-contained just as is the help system. This means that all parts of the *EA Visualizer* that the user can be confronted with in the user interface can be found here. At this point we should note that when items from the *EA Visualizer* are not clear while reading this tutorial, the system itself can be used directly to resolve this through the help system that is inbedded. There, a summary of all relevant components is given along with explanatory information for each of them. Next to an operational description of the *EA Visualizer* system, the *editor* version is described as well. Again first through explanation and second through examples, the user is guided in the use of the system.

Upon writing this tutorial, the current version of the *EA Visualizer* is 1.4. The general idea of the system has always been the same, so this tutorial is applicable also if your version of

¹Version 1.3

the system is more recent². Furthermore, the images in this tutorial are all taken from the same computer running the operating system WINDOWS 98, using the JDK 1.2 from SUN and screen resolutions of 1280×1024 and 1024×768 . This however does not withhold us to note that because of recent developments in JAVA technology, using the *EA Visualizer* is transparent in that on any system the interfaces are alike to such an extent that using any of them in a tutorial would be evenly explanatory. In some details you might find differences with your current working version, but the setup of the system is to such an extent the same that you will never find any harmful inconvenience when working with your own system while reading this tutorial. For any further information you can contact me by going to section 6.2 and address me with whatever remarks or comments you might have. In the mean time, I hope you will enjoy this scientific system and that it will aid and benefit you in your research as it does me every day.

Peter A.N. Bosman

June 1999

²This is always the case when starting the applet on the WWW or when downloading the latest version at: <http://www.cs.uu.nl/people/peterb/computer/ea/eavisualizer/EAVISUALIZER.html>

Contents

1	Operating the system	1
1.1	Starting up the system	1
1.2	The main GUIs	4
1.3	The help system	12
2	Views and the view system	15
2.1	Updating and redrawing	15
2.2	Internal views and external views	16
2.3	Graph drawer views	17
2.4	Saving views as images	23
2.4.1	Directly: using the EA Visualizer	25
2.4.2	Indirectly: using the OS	26
3	Single Runs	29
3.1	On single runs and multiple runs	29
3.2	Evolutionary algorithms in the EA Visualizer	31
3.3	The single run settings interface	36
3.4	Examples	40
3.4.1	Edge map recombination for the TSP	41
3.4.2	Bitcounting	50
3.4.3	Crowding, preselection, deterministic crowding and sharing schemes	54

3.4.4	Evolution strategies and elitist recombination/replacing	67
3.4.5	Advanced: MIMIC, FDA and others	73
4	Multiple Runs	93
4.1	Batch testing by expanding to multiple runs: how and why?	93
4.2	The multiple runs settings interfaces	94
4.3	Examples	102
4.3.1	Various recombination operators for the TSP	102
4.3.2	GA vs. ES on simple polynomes	117
4.3.3	Population sizing	129
4.3.4	Advanced: 1X-GA vs. UX-GA vs. MIMIC vs. FDA	137
5	Using the editor	151
5.1	Starting up the editor	151
5.2	The main GUI	152
5.3	About parameter components	157
5.4	Adding and removing classes	159
5.5	Editing and browsing classes	163
5.5.1	The class editor	163
5.5.2	The class browser	176
5.6	Examples	179
5.6.1	Adding the n-point crossover operator	179
5.6.2	The implementation of FDA	194
6	Miscellaneous	207
6.1	Installation	207
6.2	Contacting the author	210

Chapter 1

Operating the system

In this chapter, the basic operation of the system is described. The components of the main Graphical User Interface (GUI) are explained as they are what you will see on your screen most of the time. Other user interfaces that you will be confronted with while working with the *EA Visualizer* are described in other sections where they are most applicable and will be encountered while operating that part of the system. This means that the actual creation and running of EAs in an operational setting is described in sections 3 and 4.

Before starting the general description of the system, the program files must be installed and ready to be run. If you must still do this, you should first read section 6.1 and install the system so as to operationally follow the remainder of this tutorial.

1.1 Starting up the system

In section 6.1 a description is given of how to install the *EA Visualizer* and how to start the system using the JAVA runtime system. It is from that point on that we describe what happens on screen, as well as how and why. In this section we start at the very beginning, just after having executed the commandline to start the *EA Visualizer*.

The first thing that becomes visible on the screen is the *EA Visualizer* startup message window¹. In this frame the progress of the startup procedure can be monitored. As soon as the startup is completed, this frame disappears. In figure 1.1 the startup frame at the end of the initialization process is displayed, showing the intermediate steps that have all successfully been executed during startup.

¹As **Frame** is the JAVA equivalent of the window in WINDOWS, the word frame is freely used as a substitute for window in this tutorial.

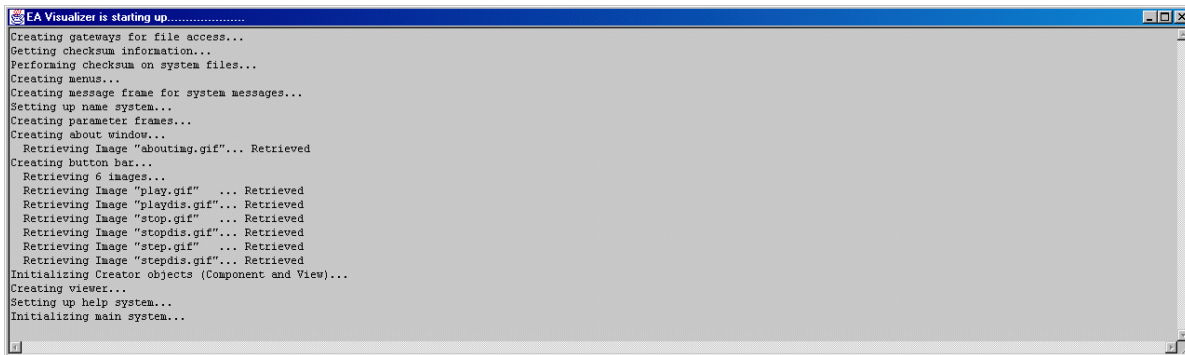


Figure 1.1: Startup frame at the end of system initialization.

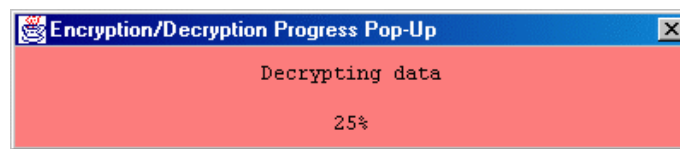


Figure 1.2: Encryption/Decryption progress popup window.

During the third step, a checksum is done on all system files. This checksum involves going over all files that should not be altered by the user and inspecting whether they are still valid. This is done based upon information that is retrieved in the second step. This information is stored encrypted and must first be decrypted in memory in order to retrieve the actual checksum information. To this end, a popup window appears that displays the progress in the decryption of the data, which is shown in figure 1.2.

When the checksum information is used to check the system files², it might turn out that some files are damaged. In this case, the system cannot start and the user is notified of this through a *Fatal Error* message in a big red window, containing the error message. Figure 1.3 shows such a window. In this case, a random system file was altered deliberately, which was detected by the *EA Visualizer*. After clicking the **Close** button at the bottom of this screen, the *EA Visualizer* will no longer start and the startup has failed, so the program will end right there.

If all is right, your system files are errorfree. The system will then dispose of the startup frame in figure 1.1 after execution of the all initialization steps and present the two main windows for the *EA Visualizer*, meaning it is ready to run. In the next section we describe these user interfaces.

²Some 250 files.

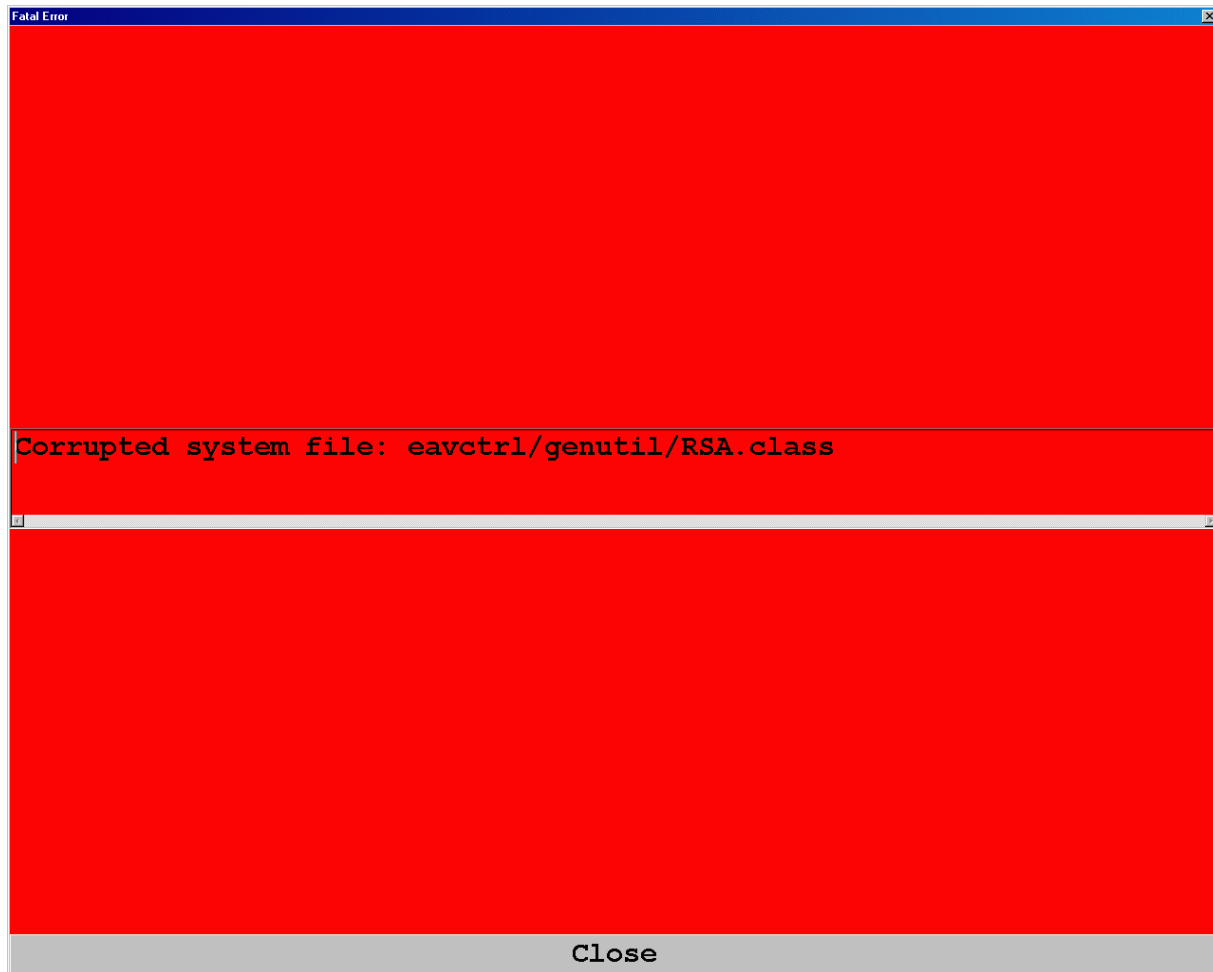


Figure 1.3: Fatal error reported by the system, making it unable to start.

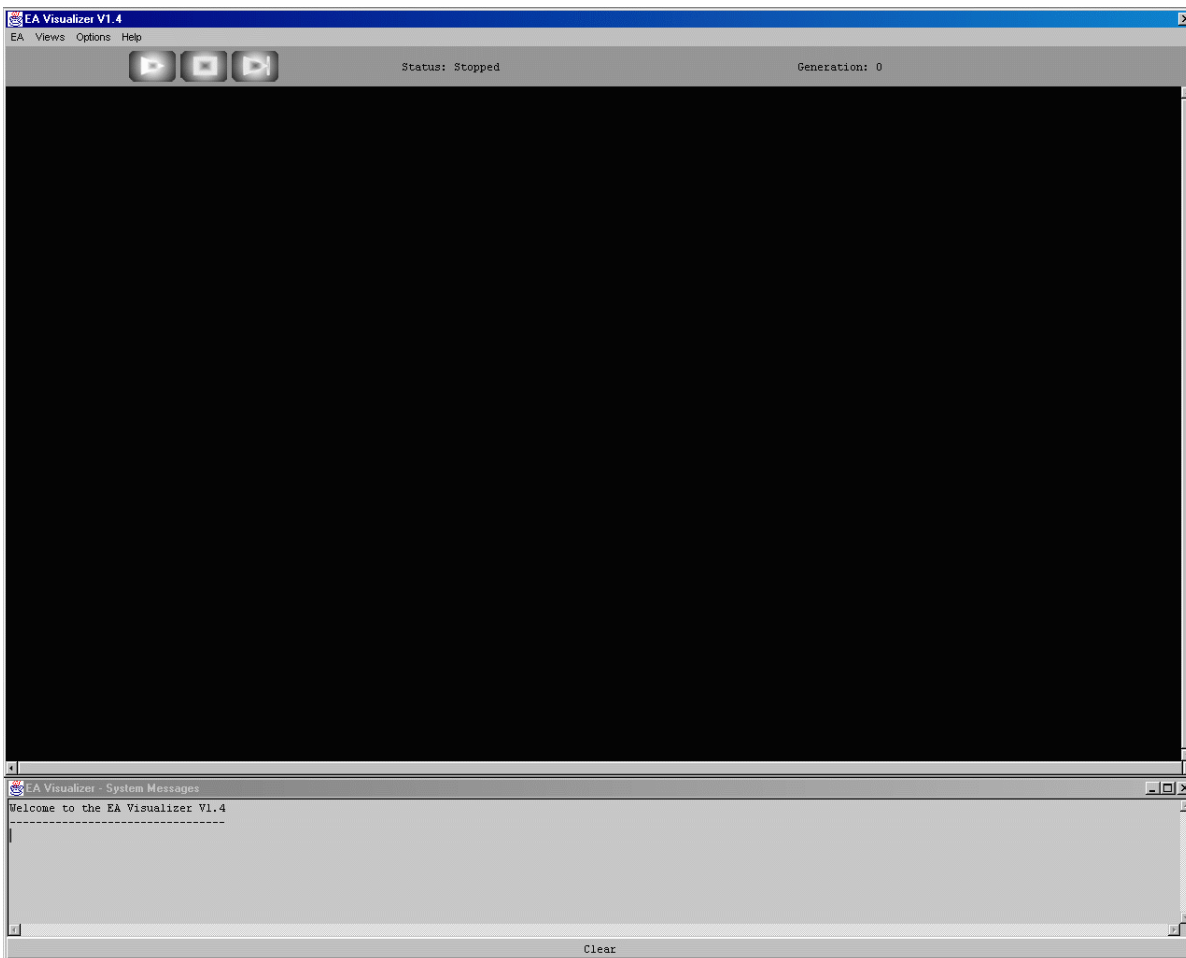


Figure 1.4: The main GUIs right after startup.

1.2 The main GUIs

Once the system has started, the two main GUIs are shown on the full desktop area as shown in figure 1.4. There are two frames, a larger one with a black background and a smaller one at the south with the same background as the startup frame. The upper frame is the main frame of the *EA Visualizer* and contains the viewing space where the visualization mainly takes place.

The lower frame is the system message frame where the *EA Visualizer* states system information when needed. When working with the system, messages are displayed here when for instance a new EA is created or when the EA is reset. Also, errors are reported here that warn you of incompatible settings that most likely result in undesired effects during runs. More urgent errors are directly displayed in a blocking popup message as you will notice. So the error messages that will appear in the system message frame are non fatal and could be regarded as warnings, but should mostly be taken very seriously.

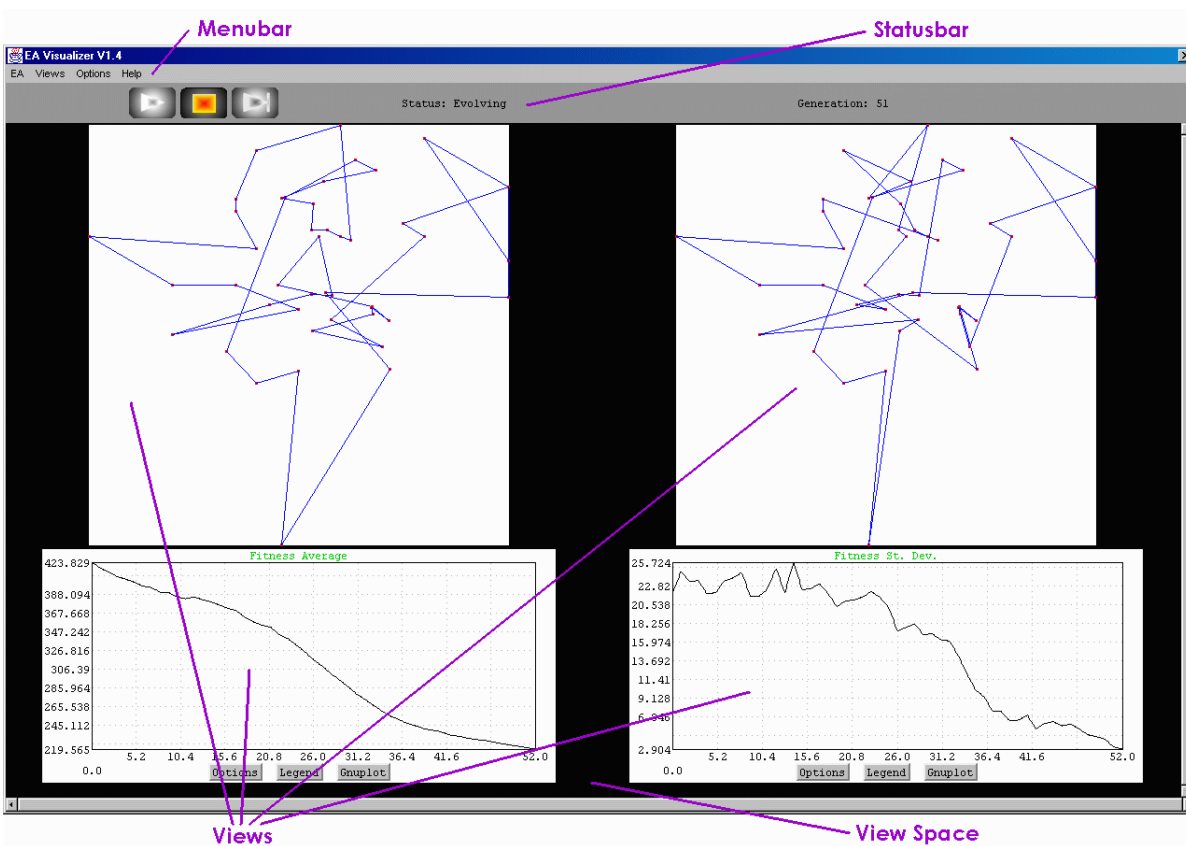


Figure 1.5: An overview of the most important frame in the system.

The upper and main frame with the black background is the frame in which all operations are done and as such is the main window of the *EA Visualizer*. It displays the results of the evolutionary algorithms through the use of views and facilitates the user both to start and stop the algorithm and open the frames to enter the settings for a new or a current evolutionary algorithm. A general overview is displayed in figure 1.5.

As shown in the overview, the frame consists mainly of three parts:

- The *menubar* allows you to manipulate both the evolutionary algorithm and the views for it that are currently active in the system. Also some general options can be set regarding the system behaviour. These options are described below.
- The *statusbar* is located just below the menu bar. On the left of it are three image buttons, much like the ones you'll find on your CD player. When a button is disabled, it will appear grey. Otherwise it will have a red-gold color. By pressing the *play* button (leftmost button) when it is enabled, the current evolutionary algorithm will start or continue to run. Pressing the *stop* button (button in the middle) will cause

the evolutionary algorithm to stop immediately after the current generation step has ended. In order to look closely at the progress the algorithm makes, the *step* button (rightmost button) can be used. Pressing this button will cause the *EA Visualizer* to perform exactly one generation step of the evolutionary algorithm.

On the right of the buttons, the status of the algorithm is displayed. This status can be one of the following: *Stopped*, *Evolving*, *Evolving (One Generation)*, *Terminated*. *Stopped* means that either the evolutionary algorithm is not running, but it hasn't terminated yet, so it can be started or continued, or there is no algorithm defined currently. *Evolving* implies that the algorithm is running until the termination condition is met or the user has pressed the *Stop* button. When the status indicates *Evolving (One Generation)*, the user has pressed the *Step* button. Finally, when the status indicates *Terminated*, the evolutionary algorithm has stopped because its termination condition has been met.

- The *View Space* is the black rectangle that is the body of the main frame. This space is used to place internal views on. The views can be made “active” by moving the mouse pointer over them. Once the mouse points at a certain view, that view will have a yellow bounding box around it. When pressing <CTRL> + F1 when pointing at a view, help will be displayed for that view. Any interactions that a view might have defined for it, can be utilized by using the mouse, no matter where the view is located in the view space. The internal views are layouted in the order that they were added in. This order can be changed through the menu option *Change Order Of Views* in the menu *Views* as described below.

We finish the description of the main GUIs in the *EA Visualizer* by going over the menubar and explaining all selectable menuoptions.

- *EA*. This is the main menu for the manipulation of evolutionary algorithms. New algorithms can be created and current ones edited or reset. This menu corresponds to the *File* menu found in most standard applications.
 - *New Multiple Runs EA*. Alternatively, the F2 key can be pressed on the keyboard to activate this menu option. This creates a new multiple runs EA that can be used for running a multiple of single run EAs a multiple of times. Different type of views can be added to average results over these multiple of runs. More on the difference between single run EAs and multiple run EAs can be found in section 3 where the single run EAs are introduced. Selecting this menu item opens a new interface in which the settings can be specified for a new multiple runs EA.
 - *Edit Current Multiple Runs EA (Reset If Applied)*. Alternatively, the F3 key can be pressed on the keyboard to activate this menu option. This menu option is only available when a multiple runs EA is currently installed in the system. It

opens the same interface as when a multiple runs EA would be created through the “create” menu option, but now it already contains the information on the current multiple runs EA that is installed. When the settings are applied, the *EA Visualizer* is reset. This is required because alterations within a multiple of runs do not allow for simple changes in the result that allow for direct continuation of the current process. This means that all views will be removed upon applying the changes and that the enumeration of the current multiple runs EA is reset to start over.

- *New Single Run EA*. Alternatively, the F4 key can be pressed on the keyboard to activate this menu option. This creates a new EA that can directly be used to be run and visualized. The actual creation and running of such EAs is described by operational examples in section 3. Selecting this menu item opens a new interface in which the settings can be specified for a new single run EA.
 - *Settings Current Single Run EA*. Alternatively, the F5 key can be pressed on the keyboard to activate this menu option. This menu option is only available when a single run EA is currently installed in the system. It opens the same interface as when a single run EA would be created through the “create” menu option, but now it already contains the information on the current single run EA that is installed. When the settings are applied, the changes in the EA are noted and reported in the system message frame at the bottom of the screen.
 - *Reset Single Run EA*. Alternatively, the F6 key can be pressed on the keyboard to activate this menu option. This menu option is only available if a single run EA is currently installed in the system. Selecting this menu option resets the entire algorithm by resetting each installed part.
 - *Reset Single Run Population*. Alternatively, the F7 key can be pressed on the keyboard to activate this menu option. This menu option is only available if a single run EA is currently installed in the system. Selecting this menu option resets the population by regenerating it. The other parts in the EA are left unharmed.
 - *Exit*. Alternatively, the F10 key can be pressed on the keyboard to activate this menu option. This opens a popup window in which a question is posed whether you really want to quit the system. Upon a confirmation, the system is terminated. Upon a negative response, the system resumes its tasks.
- *Views*. Through this menu, the views in the system can be manipulated. Views can be added and removed and their order can be interchanged so that they are laid out differently in the view space. Each individual view gets an entry in this menu so that its parameters can be altered. More on views can be found in section 2. The use of these menus is demonstrated in sections that demonstrate the system by example, being sections 3 and 4.

- *Add View*. Alternatively, the F8 key can be pressed on the keyboard to activate this menu option. This opens a new interface through which views can be added to the system once an EA (single or multiple) is installed and not running. More on views can be found in section 2.
- *Remove View*. Alternatively, the F9 key can be pressed on the keyboard to activate this menu option. This opens a new interface through which current views can be removed from the system once an EA (single or multiple) is installed and not running and views are currently active within the system.
- *Change Order Of Views*. This opens a new interface through which the order in which the currently active views are administered within the system can be altered. The way the views are added is directly used to layout the views in the view space. This regards of course only the internal views as the external views are separate windows. The internal views are layed out row by row, preserving the ordering of the views in the system. This means that as many views as possible are first placed on the first row. After this, the row is given the height of the highest view. Then the other rows are filled if there are views left. Because this might lead to an undesirable layouting of the views, the ordering of the views in which they are placed in the rows one by one can be altered through this menu option. More on internal and external views can be found in section 2.
- *View Names*. Below the *Change Order Of Views* option, the views that have been added and are currently active are placed. When one of these is selected, an interface is shown with the current settings of parameters for that view, which can then be altered.
- *Options*. In this menu, the way the *EA Visualizer* performs certain tasks can be altered. This mostly has to do with how the views are updated with information from the EAs.
 - *Single Run Views Updating*. Opens the interface as shown in figure 1.6. The *Update Interval* parameter is a number that specifies when to update the single run views with information from the EA on the current state of the evolutionary algorithm that is running in case of a single run EA. The number specifies the amount of generations that have to pass before sending such an update message to the views. Increasing the number implies that the views are less frequently processing information to visualize, but this also means that the visualized information is less extensive.

The *When To Update* parameter specifies this updating process more in general. When “Always” is selected, the views are updated every generation. If the selection is “By Generation Interval” (default setting), the updating is done every so many generations according to the number specified above. Finally when “Never” is selected, the views are never updated at all, except at termination.

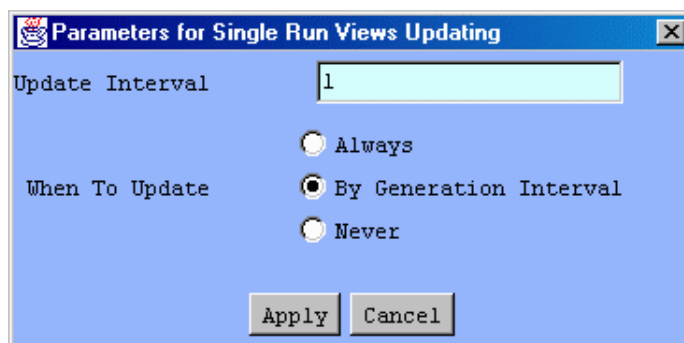


Figure 1.6: Entering parameters for the updating of single run views.

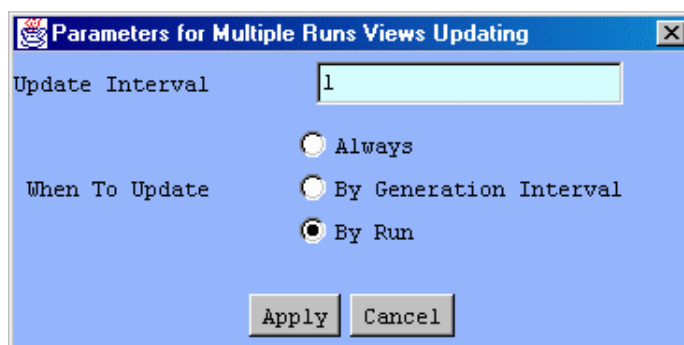


Figure 1.7: Entering parameters for the updating of multiple runs views.

- *Multiple Runs Views Updating*. Opens the interface as shown in figure 1.7. The *Update Interval* parameter is a number that specifies when to update the multiple runs views with information from the EA on the current state of the evolutionary algorithm that is running in case of a multiple runs EA. The number specifies the amount of generations that have to pass before sending such an update message to the views. Increasing the number implies that the views are less frequently processing information to visualize, but this also means that the visualized information is less extensive.

The *When To Update* parameter specifies this updating process more in general. When “Always” is selected, the views are updated every generation. If the selection is “By Generation Interval”, the updating is done every so many generations according to the number specified. When “By Run” (default setting) is selected, the views are updated when a run of an evolutionary algorithm terminates. As multiple runs views mostly only gather information at the termination of a run, this is the default setting.

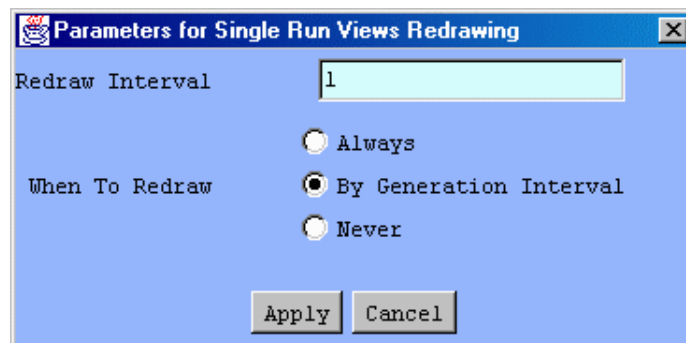


Figure 1.8: Entering parameters for the redrawing of single run views.

- *Single Run Views Redrawing.* Opens the interface as shown in figure 1.8. The *Redraw Interval* parameter is a number that specifies when to redraw the single run views in case of a single run EA. The number specifies the amount of generations that have to pass before requesting the system to redraw the views. The *When To Redraw* parameter specifies this redrawing process more in general. When “Always” is selected, the views are redrawn every generation. If the selection is “By Generation Interval” (default setting), the redrawing is done every so many generations according to the number specified above. Finally when “Never” is selected, the views are never redrawn at all, except at termination. Redrawing can make the run much slower as views can be graphically or statistically intensive. The information passed on to the views has *nothing* to do with the redrawing of the views, so no information is lost if the redrawing is set to occur less often than every generation. The information flow is determined by the view *updating* instead of the view *redrawing*. In other words, the final results are the same when the views are selected to never be redrawn as the only request to redraw will then occur at the very end (when the status equals *terminated*).
- *Multiple Runs Views Redrawing.* Opens the interface as shown in figure 1.9. The *Redraw Interval* parameter is a number that specifies when to redraw the multiple runs views in case of a multiple runs EA. The number specifies the amount of generations that have to pass before requesting the system to redraw the views.
 The *When To Redraw* parameter specifies this redrawing process more in general. When “Always” is selected, the views are redrawn every generation. If the selection is “By Generation Interval”, the redrawing is done every so many generations according to the number specified above. When “By Run” is selected, the views are updated when a run of the evolutionary algorithm terminates. When “After Final Run” is selected (default setting), the views are redrawn after the batch of runs for one and the same evolutionary algorithm. In other

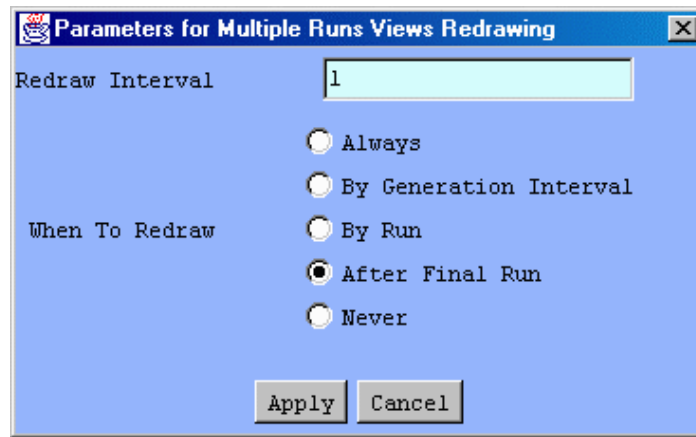


Figure 1.9: Entering parameters for the redrawing of multiple runs views.

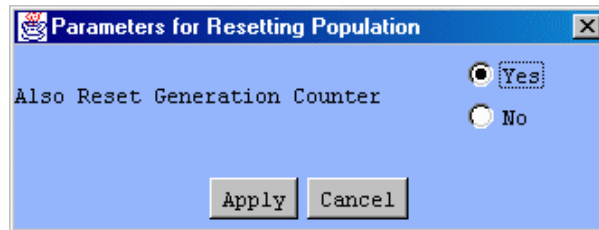


Figure 1.10: Entering parameters for the resetting of the population.

words, a multiple runs evolutionary algorithm is run a multiple of times for each of the settings required in order to average the results in some way. Redrawing the views “After Final Run” means that a redraw is requested every time for instance 20 runs have been run with the same settings. This setting is most commonly used as multiple runs views will have new information to display only at such a point since information can then be averaged in some way. Therefore, this is the default setting. Finally, when “Never” is selected, the views are only redrawn when the complete multiple runs evolutionary algorithm terminates.

- *Resetting Population.* Opens the interface as shown in figure 1.10. When “Yes” is selected, in using the *Reset Single Run Population* option from the *EA* menu, the generation counter (located on the status bar) is also reset, otherwise it is not.
- *Help.* In this menu, help information on the *EA Visualizer* can be found. The self contained help system can be opened and general information on the system can be found.



Figure 1.11: Information on the *EA Visualizer*.

- *Help Index*. Selecting this item opens the help system on the index page. The help system is described in section 1.3.
- *About EA Visualizer*. Selecting this item opens information on the *EA Visualizer* such as version, copyright and author as can be seen in figure 1.11. Pressing the *Close* button at the bottom of the interface dismisses the information and allows you to operate the system again as displaying the information blocks your input to the system.

1.3 The help system

The *EA Visualizer* is equipped with a fully self contained help system that can aid you at any point in time when using the system. Pressing the F1 key anywhere in the system will bring up the help system with related information on the location where you pressed the F1 key. This will always be information on the window in which you pressed the key. Alternatively, the help index can be brought up by selecting this in the help menu in the main GUI. Many items are not directly related to a window, so they cannot be found by

pressing F1 somewhere. The help index can also be displayed by pressing the *Index* button in the help interface. In the index all help topics are located and sorted alphabetically.

Just as in a WWW browser, the help information can be linked to other help information. Such a link is displayed in blue, using blue underlining. This is for instance the case for every entry in the help index. When you move the cursor over these links, it changes to a handcursor. Pressing the mouse button then opens that link and displays a new page with help information. Using the navigation buttons at the top of the interface, pages in your history of browsing the help pages can be brought up. The *Prev* button brings you to the previously visited page in your help browsing. Pressing the *Next* button brings you back to where you were before if you just pressed the *Prev* button. Pressing the *Close* button closes the help window and forgets your help browsing history. Basically, using the help system is based upon following the links and browsing through the index, which is really quite self explanatory. The index page of the help system is displayed in figure 1.12.



Figure 1.12: The index page of the help system.

Chapter 2

Views and the view system

In this chapter we shortly focus on the visualization part of the *EA Visualizer* without directly using it on evolutionary algorithms. In other words, we go over the viewing system that given the data coming from the EAs presents information to the user. In the next chapter the viewing system is implicitly demonstrated by working with EAs, just as is done in the chapter following it. Here we present the most important general aspects you should be aware of when working with the system.

2.1 Updating and redrawing

The views in the system visualize information coming from the EAs. This information is constructed each generation as the EA that runs goes through the evolution step. The information thus created is then collected by the *EA Visualizer* and shipped to the viewing system. The views are then updated and asked to redraw themselves with the new information. As such, the views can be seen as slaves that redraw only upon request and are provided with information.

Updating a view means that the information coming from the EA is presented to the view. The view will then in some way process the information provided. For instance, when regarding statistics, the information presented is processed to find for instance the average fitness of the population or the amount of certain substrings or schemata in the population. The key issue is that every view extracts the information required to visualize information. Note that this extraction is separate from the visualization.

The visualization is done when a view is drawn. This is thus done based upon the information extracted from the past updates. This internally stored data is used mostly to draw some image such as a graph. However, this redrawing is only done upon request by the system. Some visualizations might however take a lot of time, whereas you might only be

interested in the final visualization at the end of a run. As was noted in the first chapter when the menus in the system were explained, update intervals and redraw intervals can be specified so that redrawing can be postponed if it really takes a lot of time. Mostly, you will not want to do away with updating the views because the visualization you wish to do has to be done based upon the information from all past generations. So it is more likely that you will sometimes set a larger interval for redrawing than for updating. In any way, you should note that updating views is separate from redrawing views and that a lot of redrawing will slow down your EA.

This separation brings us to another note, which is automatic redrawing. Even though you might specify that the system only issues redraw request every so many generations, you might find that redrawing is done anyway in the mean time. This can happen when you are working on a multitasking system and have lowered the focus of the main GUI for the *EA Visualizer*. Once you request the focus for the main GUI of the system however, your operating system will issue a redraw to paint the contents of the window. The *EA Visualizer* then issues a redraw to the views, which results in displaying of course the latest information anyhow even if you selected the system to request redraws only every so many generations.

Also, views might redraw themselves as a result of interaction with the user (pressing the mouse button somewhere in the view for instance). This has nothing to do with redrawing called by the system while the EA is running, just as the automatic redrawing we just noted.

2.2 Internal views and external views

The viewing system is set up in a modular way just as are the EAs as we shall see shortly when discussing single run EAs. Furthermore views are capable of processing user input by using a mouse pointing device. Before defining what views look like, we must ask ourselves how a visualization comes into being. Mostly this is done in some graphical way as in drawing graphs. But the visualization could also be merely a list of numbers. We are now facing the problem that we want to offer the option to create views that graphically visualize information without much overhead in defining a new system in which a drawing of any type can be made, as well as the option to easily create text-based output that can be copied and pasted to save the information to a file. A solution exists in creating two types of views. This is what has been done in the *EA Visualizer*. On the one hand we have *internal* views that can directly use graphical methods for drawing visualizations and on the other hand we have *external* views that can be used as external frames without restrictions. For instance this will allow very easy presentation of text when using text components in a frame.

Next to identifying the type of views in the system, we have to remember that the internal views need to be layouted. This part is rather tricky because we have to describe in some way how to layout a collection of views neatly. Are we going to compute what is the best fit to some measure or are we going to apply a simple first fit heuristic? The latter approach may very well result in an undesirable layout. However, the latter approach also maintains the order in which the views were added and it is the most simple to implement. Furthermore if we allow the user to switch the order of the views in the management of the `EAViewer`, causing the layout to change as well, we need not worry about any not so good looking layouts. If some ordering is not satisfactory, the user can change it if desired. In order to employ a first fit heuristic however, we are to place views in rows from top to bottom, layouting them with equal space on all sides in one row. If a view does not fit on a row any longer, it is shipped to the next row. The row length is taken to be the maximum of the width of the widest view and the width of the canvas that the views are to be displayed upon. The layouting and the placing of the views is the easiest when all views in one row are thought of to have the same height and a total width that equals the maximum width. To this end a wrapper is used, which is a rectangle that is larger than an actual view but is invisible to the user. This wrapper makes it easier to do the layouting because we can at first determine what views will end up on what row, then determine the sizes of their wrappers so that the views have an equal amount of spacing between them on all sides so that they are centered as well. Finally all views are simply located in the center of their wrappers.

So basically, you can expect two type of views when working with the *EA Visualizer*, namely *internal* views that draw themselves directly on the *view space* of the main GUI and *external* views that are placed in frames exterior to the system. Added to that, the internal views are layouted as just mentioned, which can be influenced as noted in the first section.

2.3 Graph drawer views

One type of view is special in the *EA Visualizer*, which is the *Graph Drawer* view. Behind the scenes in the *EA Visualizer*, this is a general view that is capable of drawing graphs. An example of such a view is shown in figure 2.1. As this view is generally applicable, we discuss this view separate from the other views as we quickly go over its functionality.

Looking at figure 2.1, the main issue displayed is the graph itself. This graph is a twodimensional plot of two variables. Above the graph, a title is displayed. In the graph the graph entries are drawn, which can be multiple depending on the application using the graph drawer. The graph properties are mostly adapted during the updates that are performed by the system. This mostly means that the ranges along the axes change continuously. However, below the graph three buttons can be seen and the plotting of the graph can be influenced by pressing the leftmost of these that carries the name *Options*. By pressing

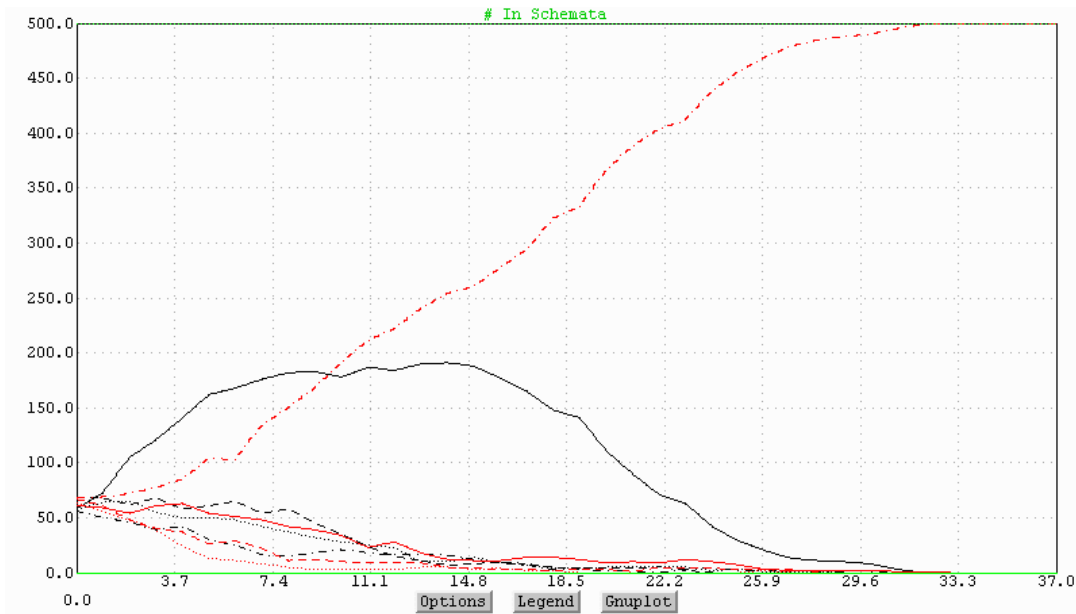


Figure 2.1: An example of an internal view using the graph drawer.

this button, an external window appears with the settings for the graph as can be seen in figure 2.2. After having set the options in the desired way, the changes can be applied, cancelled or tried by pressing the three buttons respectively at the bottom of the settings frame. Applying the settings means the settings are kept and the graph is drawn anew with the new settings. Canceling the settings means that all changes in the settingsinterface are discarded and that the view is redrawn with the same settings as when the options interface was opened. Finally, trying the settings means the options interface does not disappear and that the graph is only redrawn according to the current settings in the settingsinterface. In other words, cancelling after trying still resets the graph to the settings it used when the options frame was first opened. The options that can be set for any graph drawer are the following:

- *Title*. You can fill in any title you want for the graph in this field. This title is displayed centered over the graph (top).
- *Width*. The width of the graph in pixels (minimum value is 400).
- *Height*. The height of the graph in pixels (minimum value is 200).
- *X Range set*. Determines whether the *Graph Drawer* should itself determine the range along the *x* axis (horizontal) or that the values given by the user should be used. When this option is set to *Automatic*, the range is set so that all values that need to be displayed will precisely (eg. no larger than need be) fit in the graph.

- *X Range*. The range along the x axis (horizontal) that is to be used when *X Range set* is set to `Rigid`.
- *Y Range set*. Same as *X Range set* but now along the y axis (vertical).
- *Y Range*. Same as *X Range* but now along the y axis (vertical).
- *X Ticks type*. Sets how to compute the locations for the grid in the graph. `By step` means that for every amount as specified by the setting *X Ticks step* a grid line is placed. When `By amount` is selected, the *Graph Drawer* will attempt to place exactly that amount of grid lines in the graph (along the x axis). The grid lines are always placed at positions a positive or negative multiple of the tick step. This means it is not done based on the lower range bound of the graph. For instance when the step size is every 4 units and the lower value for the range is 10, the first gridline will appear at 12 as that is the smallest multiple of 4 greater than 10. Again to point out the difference, the first grid line is *not* placed at $10 + 4 = 14$.
- *X Ticks step*. The amount of units for every which a grid line should be placed when `By step` is selected for the *X Ticks type*.
- *X Tick amount*. The amount of gridlines that should be placed (independent of the range) when `By amount` is selected for the *X Ticks type*.
- *Y Ticks type*. Same as *X Ticks type* but now along the y axis (vertical).
- *Y Ticks step*. Same as *X Ticks step* but now along the y axis (vertical).
- *Y Tick amount*. Same as *X Tick amount* but now along the y axis (vertical).
- *Decimal precision*. The amount of decimals that should be used to display the numeric values along an axis. This means that rounding is done to the amount of decimals specified. *There are no recomputations done* for the given amount of decimal positions (for instance for the positions of the gridlines). The values are computed on beforehand and then rounded. Although an amount of 0 decimals can be specified, this only means that the value behind the dot will always be 0; one decimal is always a minimum in displaying the numeric data.
- *Sampling step size*. Sets how the amount of samples that are used to draw the graph entries must be determined. When `By amount` is selected, this value is taken directly from the *Samples* parameter entry. When `Clipping size` is selected however, the amount is taken to be the amount of pixels in the horizontal direction of the clipping area. The clipping area is the part of the graph that is the actual drawing (not the numeric data along the axis). This value is the minimum for a graph that in all cases doesn't cut off actual values.

- *Samples*. The amount of samples that should be taken to determine the actual graph. The more samples are used, the better the representation, but also the slower the drawing. For normal view sizes (500×250), 5000 samples is more than enough for a very smooth result.
- *Axis color*. The color for the axis and the numeric data along the axis. The color is to be specified in RGB format where the maximum value for any entry is 255 and the minimum is 0 (eg. 256 steps).
- *Grid color*. The color for the grid.
- *Background color*. The color for the background of the graph (default is white).
- *Title color*. The color for the title of the graph.

Next to these various settings that can be set for any graph drawer, there is a button for the *legend*. As the graph drawer can be used for a multiple of graph entries, a legend is required for the user to be able to distinguish between the different entries. By pressing the legend button, a separate frame is displayed that contains this legend information as can be seen in figure 2.3.

The legend frame consists of three parts. The top part contains the entries in the graph. These are listed top to bottom. On the left the type of line that is used to display the data is given (solid line, dotted line, etc.). On the right of that entry a summary is given of what the graph entry stands for. Whenever the supplier of data for the graph has set up more information that does not fit on one line, the button on the far right that says *Details* is enabled. When pressed, more specific information on that graph entry is displayed in the center part of the graph which is a textarea. Finally, the bottom part of the legend frame contains a button that allows the user to close the window.

These legends are mostly required when working with multiple runs as then mostly we are varying some parameters, implying different graph entries. For instance if we are comparing one-point crossover to two-point crossover and uniform crossover, we will typically have three lines in the graph specifying exactly the results using these types of recombination operators. In such a case, the legend is required to tell us which of the lines is which operator.

Finally, it is common practice to use the results from experiments and save them to disk. At a more professional level, the results are used in a book or a paper to be presented. In the general sense, saving pictures of views to disk is discussed in the next section, section 2.4. For views that use the graph drawer however, an additional option to what is described in section 2.4 is available.

This additional option lies within the third button at the bottom of any graph drawer, being the *gnuplot* button. By pressing this button, a new frame appears with GNU PLOT

Graph drawer options

Title: # In Schemata

Width: 800

Height: 450

x Range set: Automatic Rigid

x Range Lower: 0.0 Upper: 10.0

y Range set: Automatic Rigid

y Range Lower: 0.0 Upper: 10.0

x Ticks type: By step By amount

x Tick step: 5.0

x Tick amount: 10

y Ticks type: By step By amount

y Tick step: 1.0

y Tick amount: 10

Decimal precision: 3

Sampling step type: By amount Clipping size

Samples: 5000

Axis color Red: 0 Green: 0 Blue: 0

Grid color Red: 175 Green: 175 Blue: 175

Background color Red: 255 Green: 255 Blue: 255

Title color Red: 0 Green: 200 Blue: 0

Buttons: Apply, Cancel, Try Settings

Figure 2.2: Settings for the graph drawer.

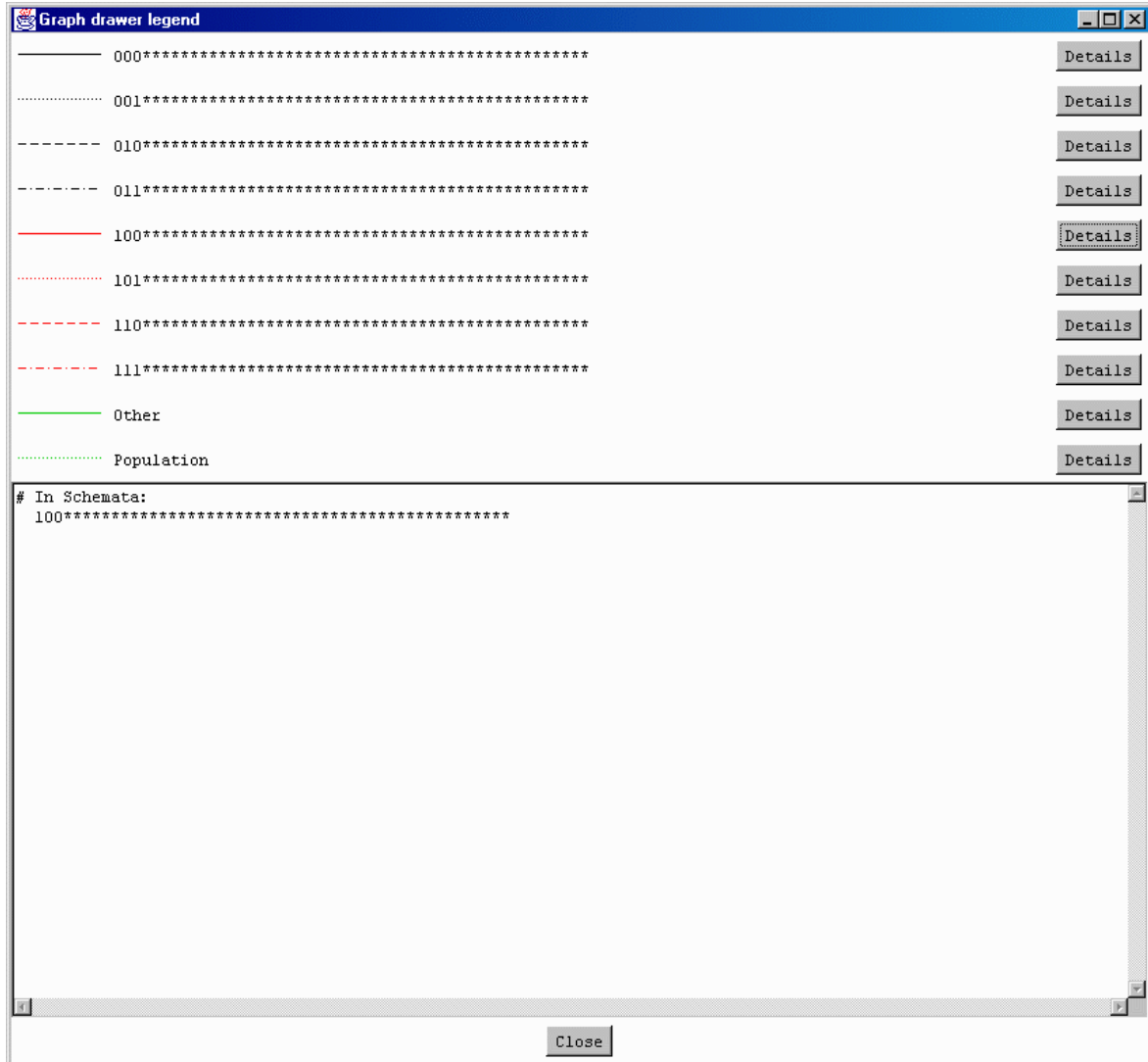


Figure 2.3: The legend for the graph drawer.

information. The program GNUPLOT is a command-driven interactive function plotting program that is freely available from the GNU community. This program is widely used by researchers around the world to create professional graphs. Using the *Gnuplot* button in any graph drawer view allows you to export the graph in the view to GNUPLOT. This exporting is done through the external frame that appears once the button is pressed as can be seen in figure 2.4.

The GNUPLOT data interface consists of two parts. The larger part of the frame is used at the top by a textarea that contains the actual GNUPLOT data. The data will be split over a multiple of files as the actual pointdata is saved in datafiles for GNUPLOT and one file contains the actual plotting commands. The files are displayed in the textarea at the top of a file. The filename is followed by a colon and is underlined with bars. In the lower part of the interface, a textfield shows again the names of the files to be saved and files that will be saved once the *Save* button is pressed.

Once saved, you will have files named `graphtype_datafileX`, where `graphtype` equals the name of the graph you have saved and `X` equals a number that denotes the how manyth datafile it is. Furthermore, there will be exactly one file named `graphtype_plotfile`. This file holds the commands for GNUPLOT. Running GNUPLOT on this plotfile by executing `gnuplot graphtype_plotfile`, a postscript file is generated on the standard output that is the GNUPLOT result of the graph you saved when using the graph drawer.

The export to GNUPLOT will maintain most of the settings you have entered for the graphs, such as ticks along the axis and axis boundaries, so the GNUPLOT representation is similar to the result in the *EA Visualizer*. However, using GNUPLOT, manipulating graphs is much easier, just as is most of all the combining of graphs. This requires only creating a new `plotfile` in which other commands are given to construct the desired result as the data is stored as numbers in datafiles. Otherwise, the graphical result of the *EA Visualizer* must be manipulated, which is a significantly more time-consuming task. For more information on GNUPLOT, you should refer to your local system administrator or the following URL of the GNUPLOT WWW site:

http://www.cs.dartmouth.edu/gnuplot_info.html

2.4 Saving views as images

As noted in the previous section when we discussed the GNUPLOT export option for graph drawing, the user will in the end want to save the results to disk and use them in a book or a publication of some kind. Therefore there should be a way to save the views as images in some graphics format. We have seen in this chapter that there are two types of views in the *EA Visualizer*, namely internal views and external views. The internal views are always graphical and it is intuitively clear that these graphics can be saved to disk just as

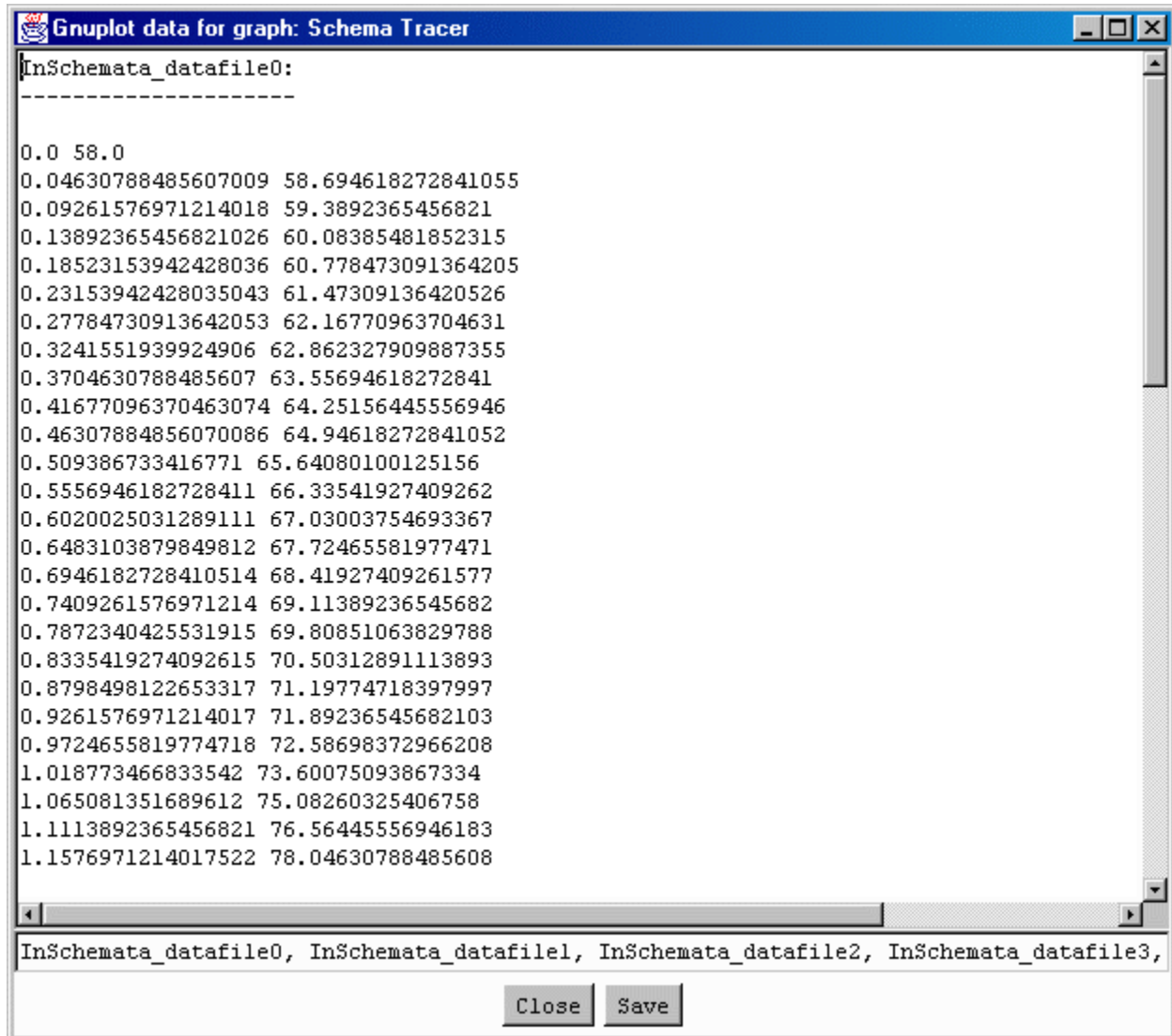


Figure 2.4: Exporting the graph drawer data to GNUPLOT.

easily as they can be presented on screen. The external views however can be of a great diversity and saving these as images to disk is not trivial to implement. Furthermore, the external views will mostly be used for presenting numerical data. As such, it is often not desired to save this numerical data as some graphical image. It is for this reason that external views cannot directly be saved by using the *EA Visualizer* and that the internal clipboard of the OS must be used to save the data. In the following we shall describe how to use the *EA Visualizer* to save internal views as graphical images and how to do this for any type of view using the operating system. In the latter case however we only describe WINDOWS and UNIX based systems.

2.4.1 Directly: using the EA Visualizer

Internal views are graphical representations of data. It is for this reason that it is intuitive that it is desirable that such views can be saved to disk as images so they can be used in reports. The *EA Visualizer* (not running as an applet) supports saving images in the portable PPM format. This format stems from the PBM family, which stands for *Portable BitMap*. The portable bitmap format is a lowest common denominator monochrome file format. It was originally designed to make it reasonable to mail bitmaps between different types of machines using the typical network mailers we have today. Now it serves as the common language of a large family of bitmap conversion filters. The PPM format stands for *Portable PixMap* and is an extension over the PBM to incorporate color. This format is widely supported in many operating systems.

Actually saving an internal view in this format requires you to place the cursor over the view you wish to save so that it becomes active in the *EA Visualizer*. This means a yellow rectangle will appear around the view. At this point, as noted before, you can press CTRL+F1 for direct help on the internal view. Alternatively, you can press the CTRL+S key combination on your keyboard to save the view. After pressing the key, a file dialog is presented in which a filename must be entered. Optionally, the extension `.ppm` can be entered. Without this extension, the *EA Visualizer* will automatically paste it behind the filename you enter. After having thus selected the filename under which to save the image data, the system will save the data. If you have selected an existing filename, either the file dialog will prompt you for acknowledgement for overwriting or the *EA Visualizer* will do so if the system finds out that after adding `.ppm` to the filename, that file already exists.

After saving the view as a PPM image, you can use an image processing program such as LViewPro under WINDOWS 95/98/NT or XV under X-WINDOWS to convert the image to for instance postscript if you want to use it in a L^AT_EX report. Under X-WINDOWS on UNIX, you might alternatively use the program `convert`, which is a very convenient program for converting between file formats.

2.4.2 Indirectly: using the OS

In the older versions of the *EA Visualizer*, directly saving internal views was not implemented for several reasons. Therefore, the OS had to be used to establish this. Here we describe how to do this in general and give examples for two different type of systems.

The general idea is that any OS has a so called *clipboard* which can be used to pin information on to exchange between programs. This clipboard is of a general kind and can hold any type of information. Using the *copy* and *paste* methods from your operating system, images can be *grabbed* from the *EA Visualizer* and be saved to disk using some graphics program for images. The general idea is to copy a graphic representation of the screen to the internal clipboard of your operating system and subsequently to paste this into some graphics program. You can then use this program to crop the image to the required selection and to save the data to disk. This of course also directly applies to external views just as much as it does to internal views. Next, we describe by example how to actually achieve this for two operating systems.

Windows 95/98/NT systems

In these systems, the internal clipboard can be used for an image. The desktop can be grabbed as an image and placed on the clipboard by pressing the `PrtScn` button on the keyboard. This captures the entire desktop (what you seen on your screen) as an image. More specifically, the `Shift` key can be held down while pressing the `PrtScn` button, which will grab as an image the currently active window only instead of the entire desktop. This means that when you activate the main GUI of the *EA Visualizer* by pressing anywhere in that window, by pressing `Shift+PrtScrn` you will place the entire main frame of the system as an image on the clipboard.

Having thus placed an image onto the clipboard, you can start some graphics program to paste the image into. On any WINDOWS system, the program PAINT is available, but it isn't very useful in working with the images. Alternatively, you are recommended to get some version of LVIEWPRO which is a much better program to view and edit pictures. You should start up any such picture editing program and use the paste operation of the program to paste the clipboard image to be the current image that is to be edited. By selecting the desired part of the view, you can then use the crop operation to crop to the part you wish to save and then save it in some graphics format using this graphical picture viewing or editing program.

Alternatively, you might want to save the numerical text data that is contained in some external view that has no direct means of saving data. To do this, you should select all of the text in the textarea or textfield you wish to save and then press `Ctrl+C` on the keyboard. Next, you should start up some text editing program (such as NOTEPAD) and press `Ctrl+V` there to paste the text data to the text editing program. You can then save this text to store the required data from the *EA Visualizer*.

Unix systems running X–Windows

Using X–WINDOWS, it is not possible to grab an image to the internal clipboard as is possible using WINDOWS variants. Instead, we assume that you have the standard picture viewer and editor XV installed on your system. You should start up this program when you wish to save graphical information from the *EA Visualizer* and display the controls by pressing the right mouse button in the graphics window of XV. In the control window of XV, you should then press the *Grab* button and select to hide the XV windows when grabbing. When then selecting to start grabbing, the XV windows disappear and you can grab entire windows by pressing the left mousebutton or grab rectangular selections by pressing the mousebutton in the middle. This will place the grabbed area in the graphics window of XV, which will then reappear. By using the controls of XV, you can now save the image in the desired format.

Saving the text data is similar as is the case when using WINDOWS variants. You should take care to select the part of text that you wish to save and then press **Ctrl+C**. By then opening an editor (such as NEDIT) and by then pressing **Ctrl+V**, the text is pasted into the text editor program and the text can be saved. Alternatively, if the use of the **Ctrl** keys does not work, you can still select the part you wish to save in some external view of the *EA Visualizer* and then start an editor program. However, you should now not press first **Ctrl+C** and then **Ctrl+V**, but press the mousebutton in the middle in the active editor window to copy and paste the text directly in the editor.

Chapter 3

Single Runs

To establish some form of a system description with respect to its actual usage, we could provide figures of the system and explain all components within it, describing all menu items and buttons in the system. Such is however already done in the help system. This means that when the user doesn't understand something, by simply pressing F1 the required information is displayed. There is something that is not contained in the help system as it is hard to provide a place for it unless we create a commercial product and incorporate a lot of external documentation in terms of short movies that explain how to use the system. What we are referring to is some means of how to use the system by example.

The system is provided with information on every part of it, but it does not have some means of showing the user how to typically run an algorithm and visualize the results. This addition is provided in this tutorial. We describe some examples of visualizations done and algorithms run with the *EA Visualizer*. Through a series of examples, we show the reader step by step how the algorithms are created and what actions are to be taken in order to get the system running. Before describing the settings in the *EA Visualizer*, we wish to note at this point that the examples shown in the following subsections are not to prove anything or to achieve interesting results with respect to evolutionary algorithms. They are merely meant to demonstrate the usage and the capabilities of the system.

3.1 On single runs and multiple runs

Evolutionary algorithms belong to the class of probabilistic algorithms because of their random aspect in applying operators with a specified chance, the creation of random genomes, etc. As such we can never rely on these algorithms in one single run to provide us with information from which we can then draw conclusions on how good or bad it is in any sense. Therefore, experiments with EAs are always (or at least *should* always be) performed a multiple of times. The results are then combined (for instance by averaging).

The whole idea of setting up the *EA Visualizer* has been to create a new way to look at evolutionary algorithms, namely through interactive visualizations. As multiple runs are mostly conveyed when the user is occupying him or herself with other activities because they take a lot of time, these interactive visualizations imply the main utilization of the *EA Visualizer* be through the usage of single runs.

Nevertheless, the position of these single runs and the interactive visualizations within the research should be reviewed here. Not only are indeed continuous and interactive visualizations a great tool to learn about and research evolutionary algorithms, they are also a way to quickly establish satisfaction regarding intuitions. In order to then generate a thorough investigation, a multiple of runs is still needed. Next to that, using the single run version on beforehand guards the user from wasting a lot of time in computing results over a multiple of runs on wrong settings. Still in the end, the user will want to be able to perform a multiple of runs with different settings in order to compare results. It is important to realize that a general system for evolutionary algorithms is not complete if in the end one is not capable of running an algorithm a multiple of times and of combining the results.

Furthermore, not only is it common practise (not only for *evolutionary* algorithms) to test an algorithm a multiple of times, but also to test it on problems with a different size or with different settings for the algorithm. As such, the availability of a multiple runs version that provides the option for running algorithms with different settings of all kinds is always a valuable and almost indispensable property.

Therefore, the *EA Visualizer* can be used in two important ways in order to investigate the behaviour of a certain evolutionary algorithm. These two approaches are known as the *Single Run EA* and the *Multiple Runs EA*.

The *Single Run EA* is the one time execution of a specific evolutionary algorithm with specific settings for its parameters. The execution can be observed but in addition the execution can be stopped at any time in between generations. At such a point the settings of the EA can be altered to inspect the influence of different strategies or parameters for a given problem in a direct sense. Further interaction is achieved through the views, but such is dependent on the functionality of a view.

The *Multiple Runs EA* facilitates in performing a thorough benchmark/performance test with multiple settings over multiple runs. It is widely accepted that because of the random aspect within each evolutionary algorithm, in order to make any conclusions about the performance, the results should be statistically combined over an amount of separate runs. The *Multiple Runs EA* in the *EA Visualizer* provides the possibility to enter different values for parameters (like selection size and population size), different instances for components (like selection strategy and recombination operator), possible links between them preventing a crossproduct in settings and the specification of the amount of times to run one specific evolutionary algorithm (specific parameter values and specific component instances).

In this chapter, we observe the direct way of running and visualizing EAs, which is thus called the *Single Run EA* in the system. It should be clear to the reader at this point what the difference is between the single run and the multiple runs and why these two versions are needed. In this chapter we shall also look at how Evolutionary Algorithms are represented in the system. Understanding the structure of EAs in the *EA Visualizer* strongly aids the user in understanding how EAs can be created and run. In this tutorial we do not go into why the representation (decomposition) of EAs is of the form we shall see, but will only present what it is like. For background information, the reader should refer to more technical papers on the *EA Visualizer* [5, 6].

3.2 Evolutionary algorithms in the EA Visualizer

Because of software engineering aspects, it was determined that the modeling of EAs should be *modular* [5]. This implies that the EAs are represented by components that work together in some general framework. We call these components a *decomposition* of EAs. By specifying this framework and the dataflow from one component to the other, we have defined a structure for EAs. An actual evolutionary algorithm is then no more than having an instance for each such component in the decomposition. For example, we have a *Recombinator* component and a possible instance is *Binary String – One Point Crossover*.

At the outset a common view on EAs is the basis of the framework in the *EA Visualizer*. This common view is depicted in figure 3.1. It holds nothing more than the running of an EA for one generation at a time until the termination condition is met. It is the *Evolve* step that truly determines the way EAs can be modeled in the system. This step is graphically presented in figure 3.2 and should be intuitively clear. All together, the following algorithm is constructed:

```

t = 0
initialize(P(t))
evaluate(P(t))
while not terminate(P(t)) do
    sel = select(P(t))
    mat = mate(sel)
    rec = for each mated collection m ∈ mat do recombine(m)
    mut = for each genome g in each recombined collection r ∈ rec do mutate(g)
    hyb = for each mutated genome h in each collection m ∈ mut do hybrid(h)
    rep = replace(hyb, P(t))
    P(t + 1) = select(rep)
    evaluate(P(t + 1))
    t = t + 1
od

```

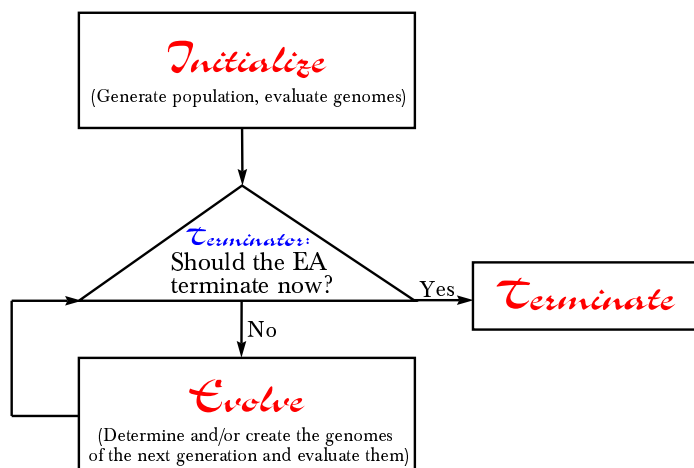


Figure 3.1: Coarse grained decomposition of evolutionary algorithms.

In this algorithm a number of components can be seen. Next to these components however, there are also components that are not directly visible from either figures or the above algorithm. In the following we shall give a description of all components present in the *EA Visualizer* that make up the decomposition and thus the general framework. When working through the examples in the remainder of this chapter, the reader is advised to look back at the decomposition to imagine how the instances selected in the example work together in the algorithm installed. Understand why parameters are set in what way and how instances work together to get a good feeling for modeling EAs in the *EA Visualizer*. First however, we shall now present the definition of the components in the decomposition of evolutionary algorithms in the *EA Visualizer*:

Name	Description
Population	A <i>Population</i> is a container for the genomes. When looked upon as a storage facility, the <i>Population</i> is nothing more than a collection of objects, but in the evolutionary algorithm it can come to hold more information than just such. For instance information on clusters or linkage between genomes can also be incorporated here, making the <i>Population</i> vastly more important than merely the holder of a collection of genomes. Like the <i>Fitness Function</i> , the <i>Population</i> serves as an environment. The difference is that the environment induced by the <i>Population</i> regards information about the structure and the linkage between genomes as opposed to information regarding the search space of the optimization problem.

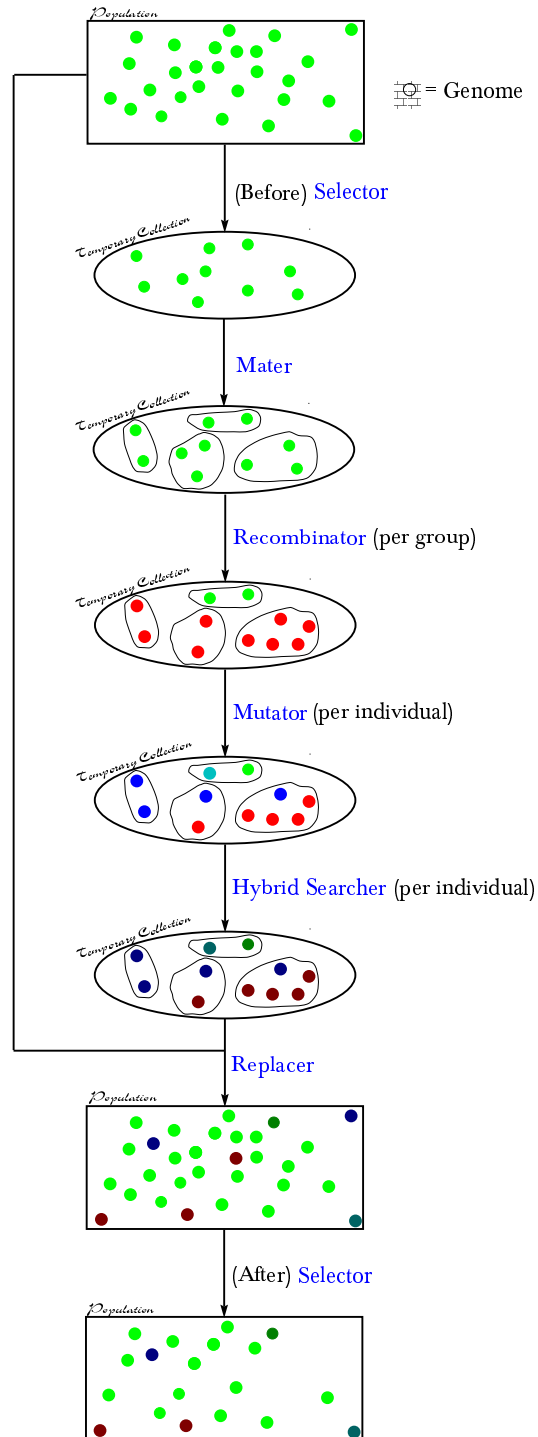


Figure 3.2: Fine grained decomposition of the generation step in evolutionary algorithms.

Name	Description
Fitness Function	<p>The <i>Fitness Function</i> rates the genomes and therefore the solutions according to their <i>fitness</i>. Very important alongside this definition is the fact that the <i>Fitness Function</i> of course also defines what a “better” fitness value is (maximization or minimization for instance). Observing this definition in terms of the algorithm, it is clear that the <i>Fitness Function</i> defines a mapping from the solution space to the real numbers. This points out that the <i>Fitness Function</i> plays the role of the environment in the optimization problem. It holds the information that is specific for the problem instance. Finally, the <i>Fitness Function</i> implicitly defines a <i>genotype-phenotype</i> mapping. To be more precise, it denotes the exact location in the problem space for each genome. It is clear that based on this location, the <i>Fitness Function</i> computes its fitness values. Hence the <i>Fitness Function</i> can map a genome onto its phenotypic equivalent.</p>
Genotype	<p>A <i>genome</i> represents a potential solution to a problem. It codes the problem specific information that describes a point in the search space. How the solution information is coded within a genome, is determined by the <i>Genotype</i>. Like in imperative programming languages, it designates the type of a value container where in this case the value container is a genome instead of a variable.</p>
Hybrid Searcher	<p>The <i>Hybrid Searcher</i> is an extension to the conventional evolutionary algorithm as it need not make use of evolutionary operators. It facilitates the optimization of individual genomes outside the evolution process. After both the <i>Recombinator</i> and the <i>Mutator</i> have been applied, a <i>Hybrid Searcher</i> is used to optimize every single offspring genome. The <i>Hybrid Searcher</i> has no further knowledge on the execution of the evolutionary algorithm in the larger setting. The system will provide it with the genome it needs to locally optimize when needed.</p>
Mater	<p>The <i>Mater</i> is an operator that puts together the parent genomes that were selected by the <i>before Selector</i> in groups. These groups need not be disjoint, but such is usually the case. The genomes that are placed together in a group will produce offspring together, for it is the groups that result from this operator that are transferred one by one to the <i>Recombinator</i>.</p>

Name	Description
Mutator	The <i>Mutator</i> implements the operation where the coincidental exploration of the search space takes place. The exploration is such because the mutation of a genome is mostly totally random. The <i>Mutator</i> has no further knowledge on the execution of the evolutionary algorithm in the larger setting. The system will provide it with the genome it needs to mutate when needed. These genomes are the offspring as resulting after the application of the <i>Recombinator</i> to the mated parents.
PRNG	In order to explore vast amounts of the search space (or at least to be able to do so), the usage of a certain random number generator can have a great effect. When using a random number generator that only generates so many different combinations, the exploration of any evolutionary algorithm is thereby inherently limited as well. In order to provide the user with a degree of freedom as to be certain of a well implemented <i>Pseudo Random Number Generator</i> , this functionality has been placed into a separate component.
Recombinator	The <i>Recombinator</i> implements the operation where the inheritance of genetic material as found in nature takes place. This operator takes an amount of parent genomes and by combining the information that is stored within their genetic structure in some sense, creates new offspring. In this way an exchange in solution information information takes place, causing a traversal through the search space of a certain kind to take place. The <i>Recombinator</i> has no further knowledge on the execution of the evolutionary algorithm in the larger setting. The system will provide it with the parents it needs to recombine when needed. These groups of parents were compiled prior to the application of the <i>Recombinator</i> through usage of the <i>Mater</i> .
Replacer	The <i>Replacer</i> determines (before any optional <i>after selection</i> operator) which genomes will be in the population in the next generation. The replacement strategy implements a way to place the offspring back into the population and thereby possibly replacing already present genomes. The <i>Replacer</i> is provided with the current population as well as the offspring and relation between these offspring genomes and the parents that created them. Based on that information, a selection is made as to see what genomes survive. As such, this operator is part of the selection process.

Name	Description
Selector	The <i>Selector</i> is an operator that is capable of selecting genomes from a population. This selection process needs not to select any genome at most once. Individual genomes can be selected multiple times. It is not said that a <i>Selector</i> has to select the better genomes, but usually this will be the case. In the evolutionary algorithm framework used, selection is applied twice during the evolution of one generation. First, it is applied as the <i>before Selector</i> , selecting the genomes that will act as parents during the current generation. Who is to survive is not yet pointed out, as the <i>Replacer</i> specifies which genomes will eventually be placed in the population. Second, the selection operation is applied directly after replacement has taken place. Here, as the <i>after Selector</i> , it <i>does</i> define exactly who is to survive, for only the genomes that are selected this time are placed within the resulting population.
Similarity	In order to be able to compare genomes and use this information when defining evolutionary operators, the <i>Similarity</i> component is part of an evolutionary algorithm in the general framework. A <i>Similarity</i> component takes two genomes and determines to what extent they are equivalent. This information can for instance be used when mating genomes to create offspring or when replacing genomes in the current population to find the best or worst match.
Terminator	Eventually we want the evolutionary algorithm to terminate. For this task, a separate component has been created, being the <i>Terminator</i> . The component is provided with all the relevant information from the evolution process during the last generation. The <i>Terminator</i> then determines based on this information whether or not the algorithm should terminate.

3.3 The single run settings interface

By selecting **New Single Run EA** or by pressing **F4** on the keyboard, an interface is displayed in which the settings can be entered for a single run evolutionary algorithm. This means that for each component of the general framework for evolutionary algorithms as stated above, the user must select an instance. The selected and available instances are placed in the lists at the center of the interface. An overview of this interface is given in figure 3.3.



Figure 3.3: The GUI for single run EA settings.

Whenever a selected instance has parameters, the *Parameters* button on the right side of the list belonging to that selected instance is enabled. By pressing this button, an interface is displayed with the *Parameter Components* in which the parameters for the selected instance are to be filled in.

At the bottom of the interface, the recombination and mutation chances are to be specified. If your screen resolution is not high enough to host all of the components, the lists are divided over two pages and the recombination and mutation chances are located at the bottom of the second page.

Once for all components an instance has been selected and all the parameters are filled in, the *Create EA* or *Apply* button can be pressed to dismiss the interface. The name of the button is dependent on whether you are editing the settings for a *new* or for the *current* evolutionary algorithm respectively. Pressing the *Cancel* button discards any changes and dismisses the interface as well.

Four components are so called *Dependency Imposing* components. These components are used by other components in the process of creating the next generation of genomes. Because of this, the selection of the other instances for components can be dependent on what selection has been made for these *Dependency Imposing* components. These *Dependency Imposing* components are the *Genotype*, the *Fitness Function*, the *Similarity* and the *Population*. When selecting an instance for a *Dependency Imposing* component, the interface will filter all the other lists for the components so as to see which instances are allowed to be selected now that this instance for this *Dependency Imposing* component has been selected. For instance, when selecting the *Genotype* to be **Binary String**, the other lists of instances for components will only list the instances that can be used for that genotype. This will mean amongst other things that the *ES Recombinator* will disappear as will the *ES Mutator*.

Next to entering the settings by hand, settings can be loaded and saved (if the system is not being run as an applet). Loading and saving can be established by pressing the *Load Settings* and *Save Settings* buttons respectively at the bottom of the interface. By pressing the the *Load Settings* button, an interface appears as is depicted in figure 3.4. This interface shows a large textarea where the summaries for the settingsfiles are shown once a file is selected in the list of selectable files at the bottom of the interface. Once the desired settingsfile is located, the settings can be actually loaded by pressing the *Load* button. This will dismiss the interface and alter the settings interface according to the settings in the settingsfile that was selected. The interface can also be dismissed without loading settings by pressing the *Cancel* button.

Saving settings in the *EA Visualizer* is done through the interface that becomes visible after pressing the *Save Settings* button in the settings interface for the single run EA. This interface for saving settings is depicted in figure 3.5. The interface shows two textareas. In the above textarea the summary for the settingsfile to be saved can be edited. The lower textarea shows settingsfiles that currently exist in the system. The settingsfiles that were

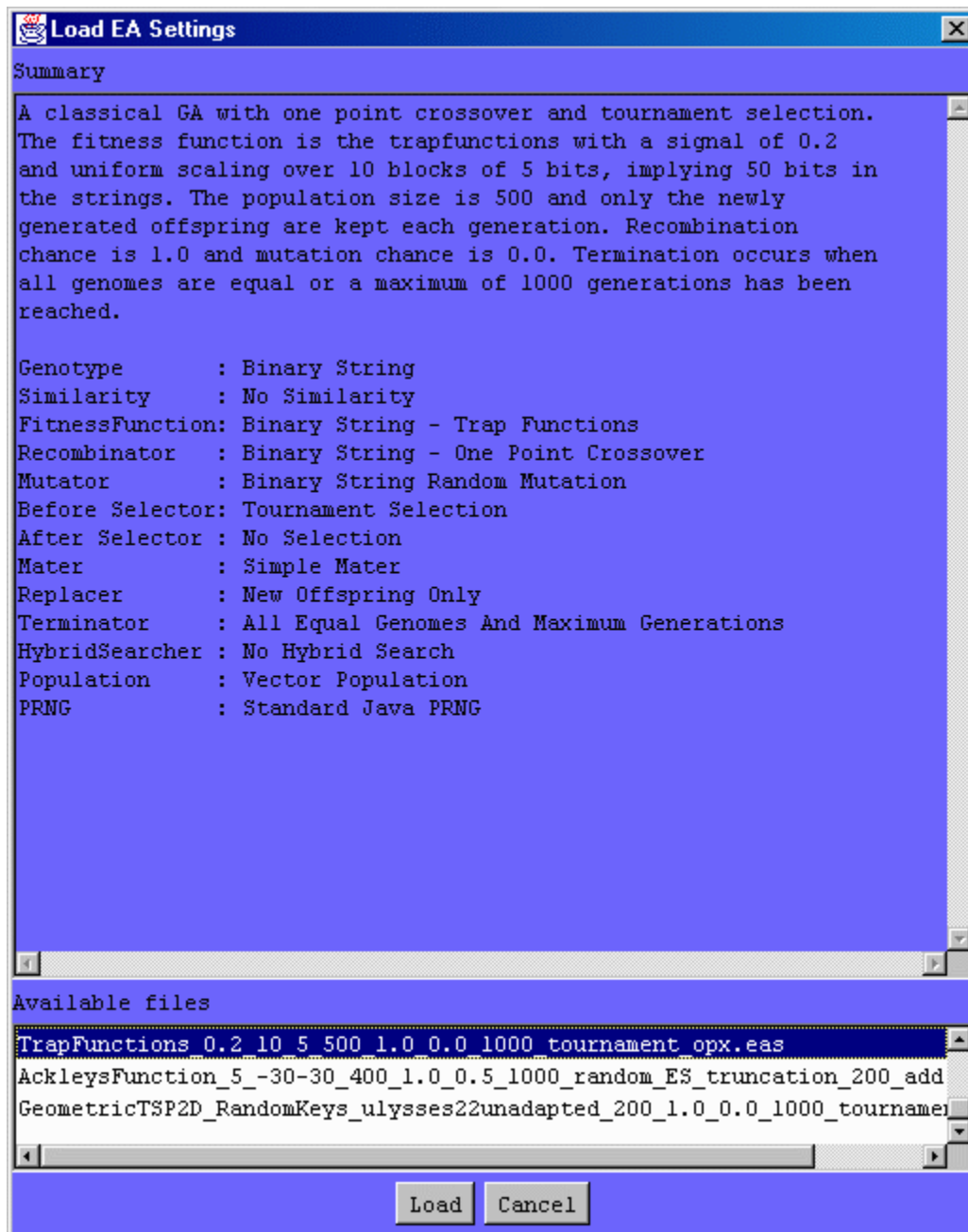


Figure 3.4: Loading single run EA settings.

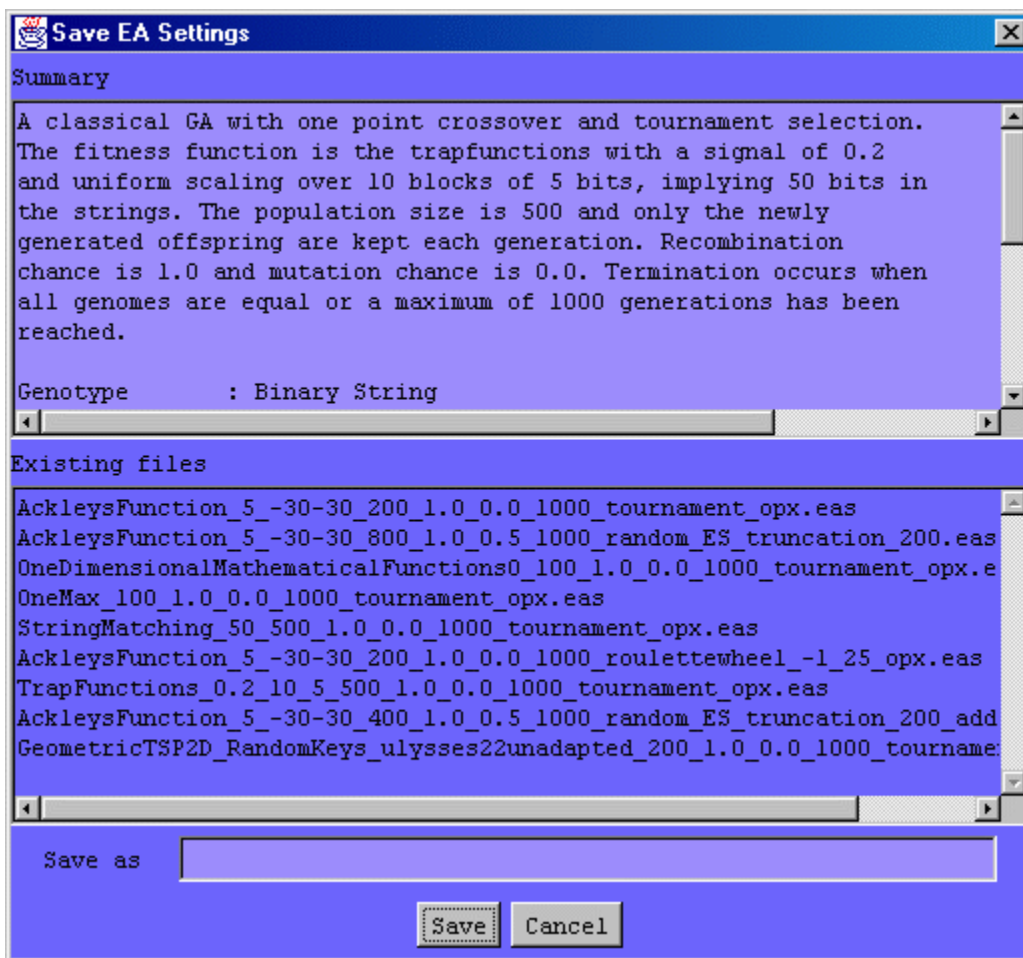


Figure 3.5: Saving single run EA settings.

shipped in the original configuration cannot be overwritten. At the bottom of the interface the filename for the settingsfile can be entered. Once the desired information is entered, the settings can be actually saved by pressing the *Save* button. This dismisses the interface, gathers the information from the settings interface and actually saves the information to disk. The interface can also be dismissed without saving settings by pressing the *Cancel* button.

3.4 Examples

In this section we present various examples. The examples are chosen so that there is a variety in applications, showing the diversity of the *EA Visualizer* system. The examples range from simple to more difficult, starting off with something simple. The examples

presented are sometimes described in less detail than in others, but nevertheless the reader is guided in the use of the *EA Visualizer*, especially when the examples get more involved.

3.4.1 Edge map recombination for the TSP

In a first example you are taken through the *EA Visualizer* step by step and are shown through what steps an EA can be created, run and visualized. In the next section, more examples will be given over a variety of applications. However, in this section the process of establishing such an EA visualization is described in full detail, whereas the subsequent examples are presented more briefly.

The standard package of the *EA Visualizer* contains a basic working set for the 2-dimensional symmetric geometric traveling salesman problem. This problem states that we are to find the shortest path between a given set of cities in the 2-dimensional plane in such a way that every city is visited exactly once and the city where we start is also the city where we end. The distances between the cities are the Eulerian distances. It follows that the resulting tour is a Hamilton cycle. The 2-dimensional geometric version of the TSP (Traveling Salesman Problem) was chosen because easy visualizations that are both interesting and fun to see can be created. In this section we see how to use the *EA Visualizer* to try to find an approximation to the problem using evolutionary algorithms.

At the outset we choose to use one of the better recombination operators for the numbered list representation of the solutions to the TSP. This recombination operator is the *edge map* recombination strategy. This is one of the better “blind” operators that stem from the first generation of operators for the TSP. Blind operators do not utilize any domain specific information from the problem at all. Only the information from the parents is used to generate new tours. The edge map recombination operator has turned out to be one of the best of this kind. Mathias and Whitley [11] have shown that the edge crossover can even be improved further, which resulted in an edge-2 and an edge-3 crossover operator. The edge map recombinator uses a so called *edge map*. This edge map is a table in which each city is stored. Behind each city a list is placed which holds the cities that are neighbours to this city in the parent tours. These lists therefore all have a length no larger than four cities. The recombination process is then performed as follows:

1. Select first city from one of both parents to be the current city.
2. Remove the current city from the edge map lists.
3. If the current city has any remaining edges, go to step 4, otherwise go to step 5.
4. Choose the new current city from the edge map list of the current city as the one with with shortest edge map list.
5. If there are any cities left, select the city with the shortest edge map list to be the current city and go to step 2.

It follows that the operator according to the above definition creates 1 offspring genome for every two parents. This means we have to select twice as many parents so as to get enough offspring back to replace the current population and keep the population size equal.

We are now ready to enter the settings of the evolutionary algorithm. By pressing F4 in the main window or by selecting **NEW SINGLE RUN EA** from the EA menu, we open the settings window and enter the settings as is depicted in figure 3.6. At the top of the window we select to use the **TSP Numbered List** as the genotype. The similarity component is not interesting as we are not concerned with niching or any other means of similarity usage. We therefore select the only thing we are allowed to select: **No Similarity**. The fitness function is one of the most interesting parts of the settings. We select the **Geometric TSP Numbered List** and notice that the *Parameters* button on the right of it is enabled. By pressing that button we can specify the parameters for the component which in this case is of course the locations of the cities. As this is an unusual type of parameter component, we show this situation in figure 3.7. At the top we can enter the coordinates of the cities. We can also directly enter the cities by clicking them in the white rectangular area. We wish to use the TSPBIB library set `eil101` that contains 101 cities. After reformatting the data in this set to be the right input to the *EA Visualizer*, we enter the locations for the cities, resulting in the drawing of the city locations as can be seen in figure 3.7. By pressing *Apply* we select to go with the entered settings.

We continue to enter the settings for the evolutionary algorithm by selecting that we wish to use edge map recombination as the recombination operator and no mutation whatsoever as the mutation operator. The outset of the evolutionary algorithm determines how we should select our remaining components. We choose to select some classical configuration by installing a selector on beforehand and using only the offspring genomes as the genomes of the next generation. This leads to selecting tournament selection as the before selector. We enter to select 400 genomes from the population with a tournament size of 2. Note that the tournament selection strategy is found by scrolling down the list of available selectors. The after selector is set to no selection and the mater to the simple mater that is to create mating groups of size 2 so as to achieve the classical configuration as requested. Continuing on this note we select to have the **New Offspring Only** as the replacement strategy which is once again located in the lower part of the list. This replacement strategy only takes the offspring genomes as the individuals of the population of the next generation. Furthermore we employ no hybrid searcher and select to use the standard population implementation, namely the **Vector Population**. We specify to have 200 genomes in the population which is correct with respect to the selection of 400 genomes by the before selector because the recombinator creates one offspring genome for every two parents implying that the 400 selected genomes are recombined into 200 offspring. Finally we select to use the standard prng that is provided within any JAVA distribution. We finish entering the settings for the evolutionary algorithm by pressing the *Create EA* button, accepting the chance of 1 for recombination and 0 for mutation (of which the latter is not of interest because we have a non contributing mutation operator).

The screenshot shows a dialog box titled "New Single Run EA (modified)" with a close button (X) in the top right corner. The dialog is organized into several sections, each with a label on the left and a list of options in the center, with a "Parameters" button to the right of each list.

Section	Options	Action
Genotype	Random Keys Genotype TSP Numbered List	Parameters
Similarity	No Similarity	Parameters
Fitness Function	Geometric TSP 2D	Parameters
Recombinator	TSP Numbered List - Distance Preserving Crossover TSP Numbered List - Edge Map Recombination TSP Numbered List - Order Crossover	Parameters
Mutator	No Mutation	Parameters
Before Selector	Roulettewheel Selection Tournament Selection Truncation Selection	Parameters
After Selector	No Selection (Select All) Random Selection Roulettewheel Selection	Parameters
Mater	Random Mater Simple Mater	Parameters
Replacer	New Offspring Only Preselection	Parameters
Terminator	All Equal Fitness Values And Maximum Generations All Equal Genomes All Equal Genomes And Maximum Generations	Parameters
Hybrid Searcher	2 Opt Heuristic For The TSP No Hybrid Search	Parameters
Population	Vector Population	Parameters
PRNG	Standard Java PRNG	Parameters

At the bottom of the dialog, there are two input fields: "Recombination Chance" with the value "1.0" and "Mutation Chance" with the value "0.0". Below these fields are four buttons: "Create EA", "Load Settings", "Save Settings", and "Cancel".

Figure 3.6: Entering the settings for a single run.

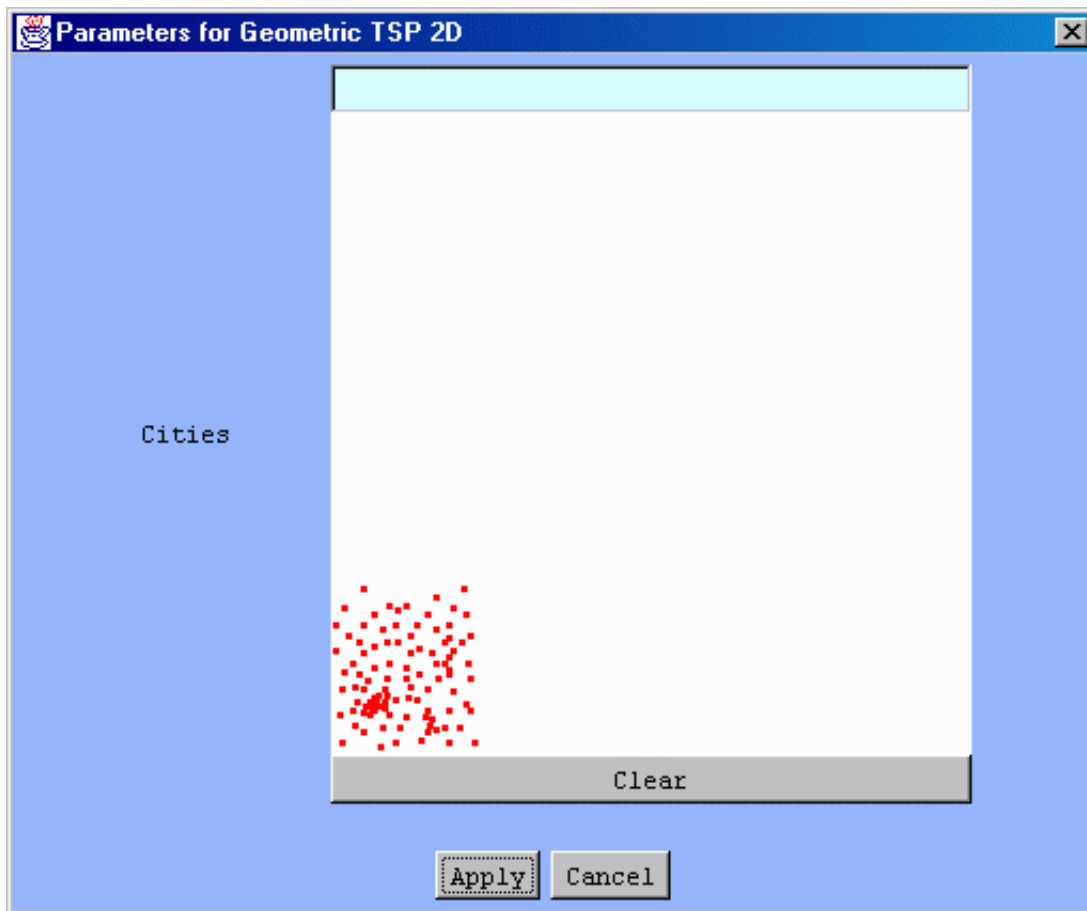


Figure 3.7: A special kind of `ParameterComponent`, entering parameters for the 2-dimensional geometric TSP.

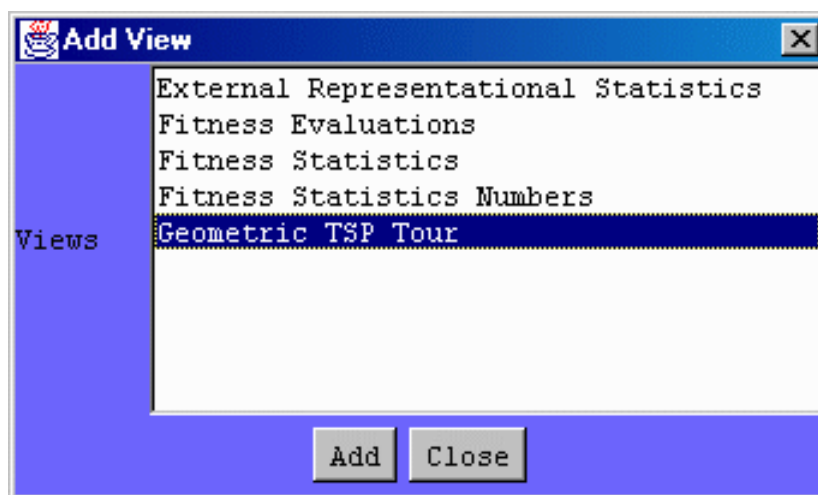


Figure 3.8: Adding views to visualize information.

Once the evolutionary algorithm has been created, we press the F8 button or select **Add View** from the **Views** menu to add views in order to visualize information about the evolutionary algorithm. The window that appears resulting from this action is shown in figure 3.8. We select to add four views. First of all we wish to see directly some of the solutions. The most interesting solutions are the best and the worst genome. As we are optimizing a 2-dimensional geometric TSP, we add two **Geometric TSP Tour** views that show the best and the worst tour in the population. Below these views we specify to have two statistics views to have some statistical feedback on the evolution process over a multiple of runs. These views are both **Fitness Statistics** views that display the fitness average of the population and the fitness standard deviation. These numbers are displayed as a function of the generation counter. After adding all these views, we press the *Close* button in the **Add View** window and we are ready to run the evolutionary algorithm.

Running or stopping the evolutionary algorithm is done by using the three buttons underneath the menubar in the main window of the system. The buttons have been chosen similar to those of a CD or cassette player and are therefore intuitively easy to use. The leftmost of the buttons is the “play” button and can be used to start or continue the algorithm. The button in the middle is the “stop” button and can be used to stop the algorithm at some point. Finally the rightmost button can be used to do *one* generational step and is called the “step” button. All buttons are grey when they are disabled. They turn to a yellow-red combination color when they are enabled. By pressing the “play” button, the evolutionary algorithm starts and we can observe the best and the worst tours that it contains each generation as well as the fitness statistics we chose to view. In figure 3.9 the situation is displayed after 235 generations.

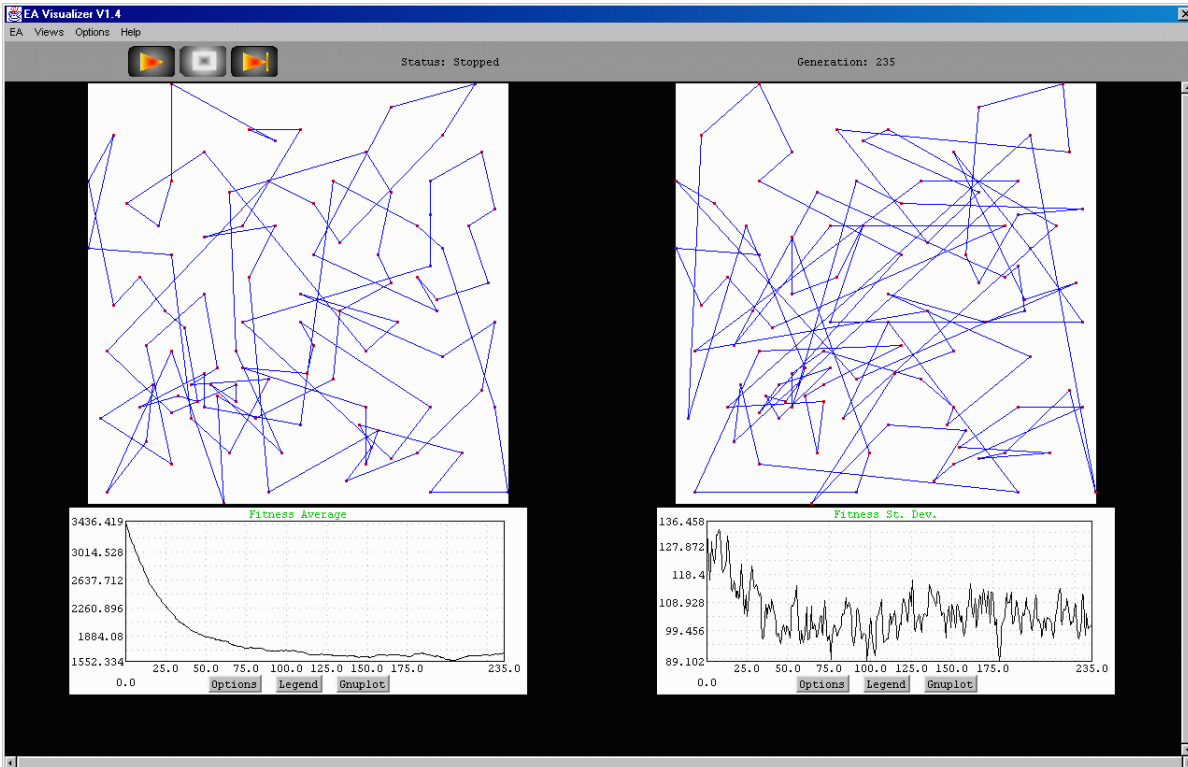


Figure 3.9: The evolutionary algorithm in progress.

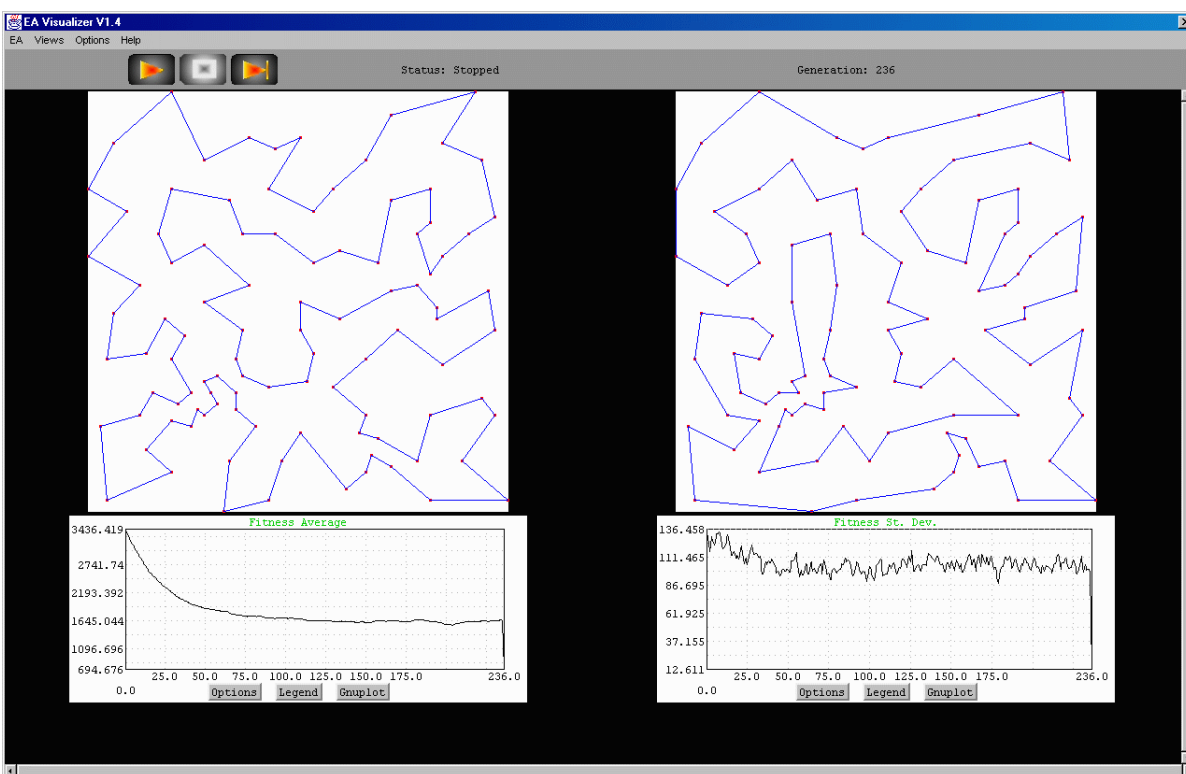


Figure 3.10: A single step of the evolutionary algorithm with a 2-Opt heuristic.

After these 235 generations we observe that the fitness average of the population is not decreasing fast enough anymore to our satisfaction and the tours are far from something interesting yet. The standard deviation is still rather large, but seen the fitness average this tells us only we have some rather bad and even worse tours. So after 235 generations we edit the settings of the current evolutionary algorithm by pressing F5 or by selecting **Settings Current Single Run EA** from the EA menu. We select to use the 2-Opt hybrid searcher for the TSP and use the *step* button to do one step in the algorithm. The results are shown in figure 3.10. This demonstrates the influence the user can have directly on the evolutionary algorithm and the conclusions that can be drawn when applying some new strategy somewhere right in the middle of a run.

Because we have no interest in this section to investigate any properties of the evolutionary algorithm or the selected recombination operator specifically, we do not draw any conclusions about any results here. As we are demonstrating the system itself and we want to provide an example of typical usage, we would for instance at this point like to just let the original algorithm run again with this intermediate result. We establish this by editing the settings of the current evolutionary algorithm again and by removing the hybrid searcher. Then by pressing the “play” button, we continue evolution. After 603 generations, the results have turned worse again and the nice approximations that we saw after one step

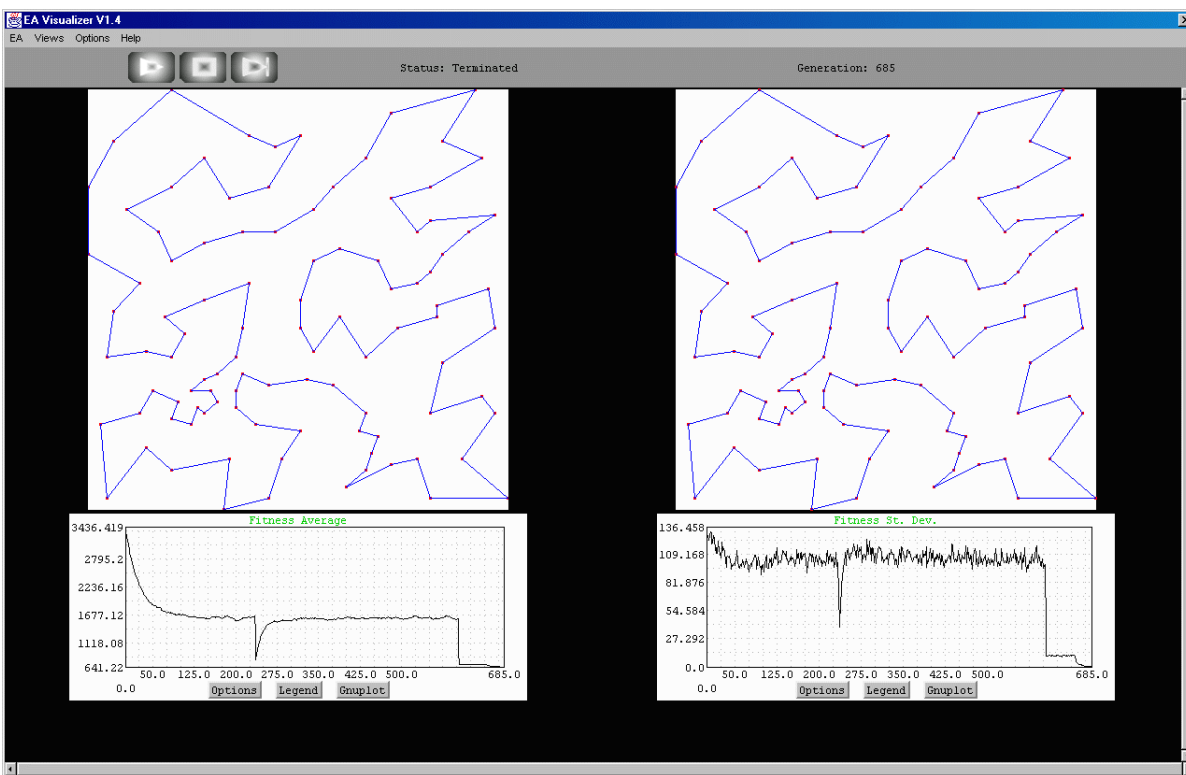


Figure 3.11: The resulting visualizations upon termination.

with the hybrid searcher have somewhat vanished. The algorithms was therefore altered to again employ the hybrid searcher from generation 603 and on. The standard deviation and fitness average immediately dropped, but the selective pressure was still not great enough to converge to a single solution. To this end, elitist replacing (see section 3.4.4) was employed as replacement strategy after 655 generations. After 30 more generations, the algorithm finally terminates with a tour that is slightly better than the one first found when first using the 2-Opt searcher. This situation is depicted in figure 3.11. Note how all the buttons are now disabled and the status is marked as **Terminated**.

Finally, we use the *EA Visualizer* to magnify the result by enlarging the view. This is done by selecting the view in the **Views** menu and altering its dimensions. Once again we note that we are not interested in drawing any conclusions about the results for any reason, but we do wish to finish this example by providing a nice view of the results. The tour that was finally found is depicted in figure 3.12 and is a rather acceptable result at first glance. This concludes our stereotypic example of using the system with a single run version. The number of options is far greater than demonstrated here and can be made as great as required by implementing some interesting usage of the interactivity offered by the *EA Visualizer*. We merely set out to provide some feeling for how to use the system.

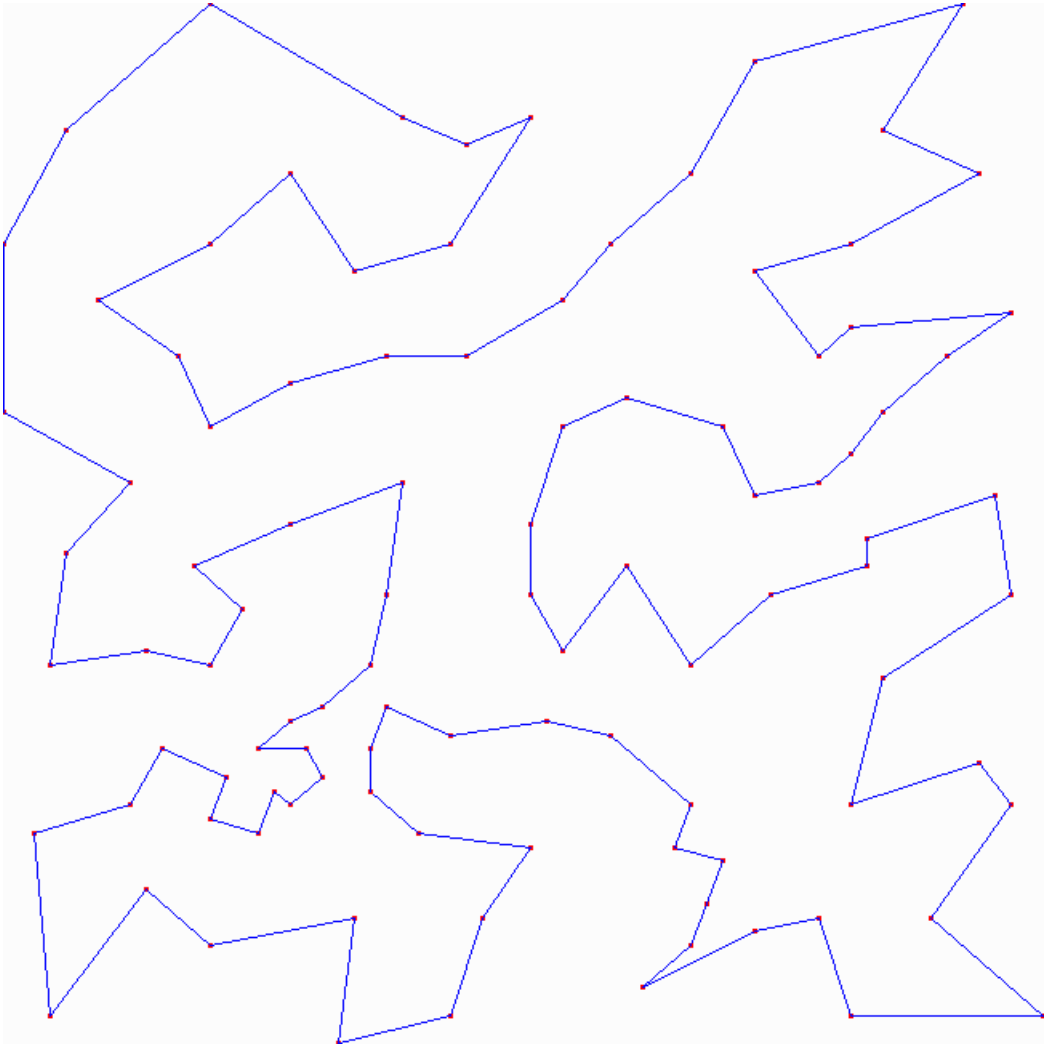


Figure 3.12: The resulting tour, displayed magnified by the system.

3.4.2 Bitcounting

One of the most simple optimization problems for genetic algorithms is the *bitcounting* or *one-max* problem. In the *EA Visualizer*, a fitness function is defined that implements a fitness landscape according to this fitness function. The `Bit Counting` fitness function is a simple fitness function that counts in the 1 symbols or the 0 symbols in the binary string. This fitness function is therefore defined specifically for the `Binary String Genotype`. There are two ways in which bits can be counted in this fitness function, but we only regard one of these two options, namely *One-Max*. This problem is the classic bitcounting in which the fitness value of a binary string is equal to the amount of bits set to 1.

We wish to run a simple GA with one-point crossover on this problem and observe the trajectory and the form of the binary strings in the population. At the same time we would like to trace some schemata to see what type of binary strings occur more frequently in the population.

First of all, we need to specify how to set up the *EA Visualizer*. In the sections containing the examples of creating and running single run EAs, including this one, we provide the settings in a table. After having read the extensive example description in section 3.4.1, the reader should be able to configure the *EA Visualizer* using only the table information and as such, doing so could be seen as an exercise. The following table shows the settings to be entered for a new single run EA:

Component	Instance	Parameters
<i>Genotype</i>	Binary String	String length 100
<i>Similarity</i>	No Similarity	—
<i>Fitness Function</i>	Binary String – Bitcounting	Counting type One-max
<i>Recombinator</i>	Binary String – One Point Crossover	Offspring Arity 2
<i>Mutator</i>	No Mutation	—
<i>Before Selector</i>	Tournament Selection	Selection size 100 Tournament size 2
<i>After Selector</i>	No Selection (Select All)	—
<i>Mater</i>	Simple Mater	Grouping size 2
<i>Replacer</i>	New Offspring Only	Report popsize warnings Yes
<i>Terminator</i>	All Equal Genomes	—
<i>Hybrid Searcher</i>	No Hybrid Search	—
<i>Population</i>	Vector Population	Population size 100
<i>PRNG</i>	Standard Java PRNG	Seed <i>Any</i> Use Random Seed Instead Yes

The value *Any* for the seed parameter of the *PRNG* component means you may pick any value. Here, you may alternatively set the second parameter of the instance to *Yes* for a

random number to be picked by the system to use as a seed. Having entered these values in the settingsinterface, we press the *Create EA* button, after which the system will notify us in the message frame that the creation of the new EA is completed. We are now supposed to add views for visualization, after which we desire to run the EA and witness the results.

We select *Add View* from the *Views* menu or alternatively press F8 in the main GUI and select to add the population dots view which will give us an overview of the population contents. As our population holds 100 members of string length 100, we select the width of the view to be equal to the height of the view so that the bits are represented as squares rather than rectangles and go with the initial settings for a view of 200×200 pixels. Pressing the *Apply* button immediately displays the population dots view, which should give you the view of a random population, meaning a random mix of blue and red dots in a square view. As we noted in the beginning of this example, we wish to observe some schemata to see what happens in the population using this method. To this end, we select to add a *Schema Tracer* view. First we must now decide which schemata to trace. Because all bits in all positions are equal in contribution to the search problem, we may for instance only regard the first so many bits to trace some schemata. We decide we want to see all schemata with fixed bits for the first three positions and *don't care* symbols for the remaining positions. To this end, we enter the following string for the *Schemata to trace* parameter:

```
fff*****
```

Furthermore, we select to view the number of individuals in the population that are matched by a schema by selecting *# In Schemata* from the *Statistic* parameter options. We do the same for two more schema tracer views, but now select to view the fitness average and fitness standard deviation. Having added these views, we stop the adding of views by pressing the *Close* button in the interface for adding views and behold the viewing space of the *EA Visualizer* with four views as can be seen in figure 3.13.

At this point, we press the *Play* button in the buttonbar and observe the evolution process and the visualization thereof in the *EA Visualizer*. Quickly already, we can observe in the population dots view that the view is turning red. This means that the amount of ‘1’ symbols in the population is increasing. The optimum is of course a string with only such symbols, so we indeed hope that the entire population dots view will end up red. The final result in our run can be seen in figure 3.14. If all is right, the *# In Schemata* graph will show that with the exception of the lines for the two additional schemata *Other* and *Population* (check the legend by pressing the *Legend* button, see figure 3.15), all lines start at approximately the same point on the vertical axis. This is of course because in a random population, the amount of members in each schema is expected to be $\frac{1}{2^3} \cdot \text{popsize} = 12\frac{1}{2}$.

It should be clear that given the result of the search, the only graph entry that increases to a maximum of 100 individuals in the *# In Schemata* graph is the schema entry with

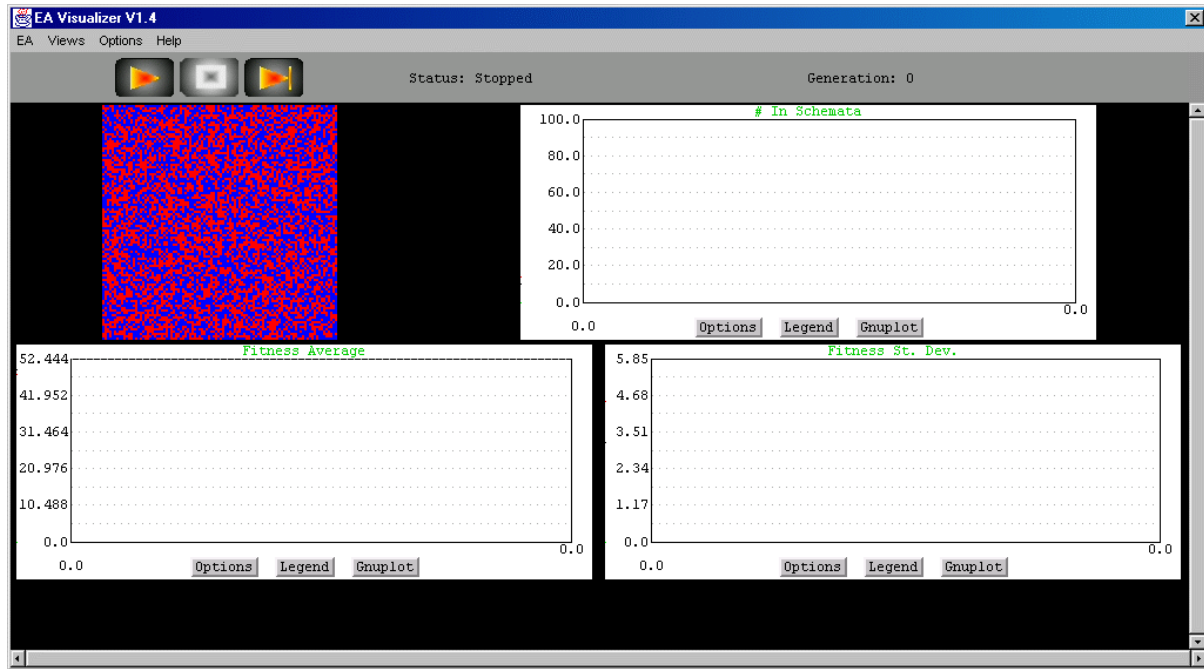


Figure 3.13: The *EA Visualizer* upon initialization with the selected views for bitcounting.

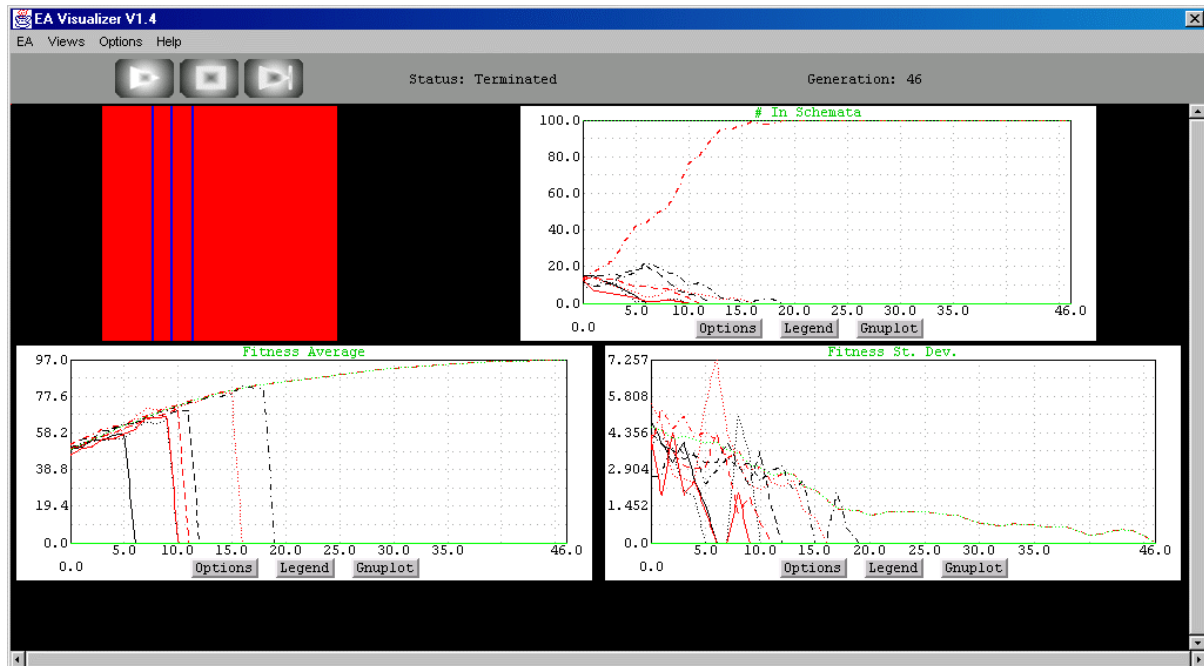


Figure 3.14: The *EA Visualizer* upon termination with the selected views for bitcounting.

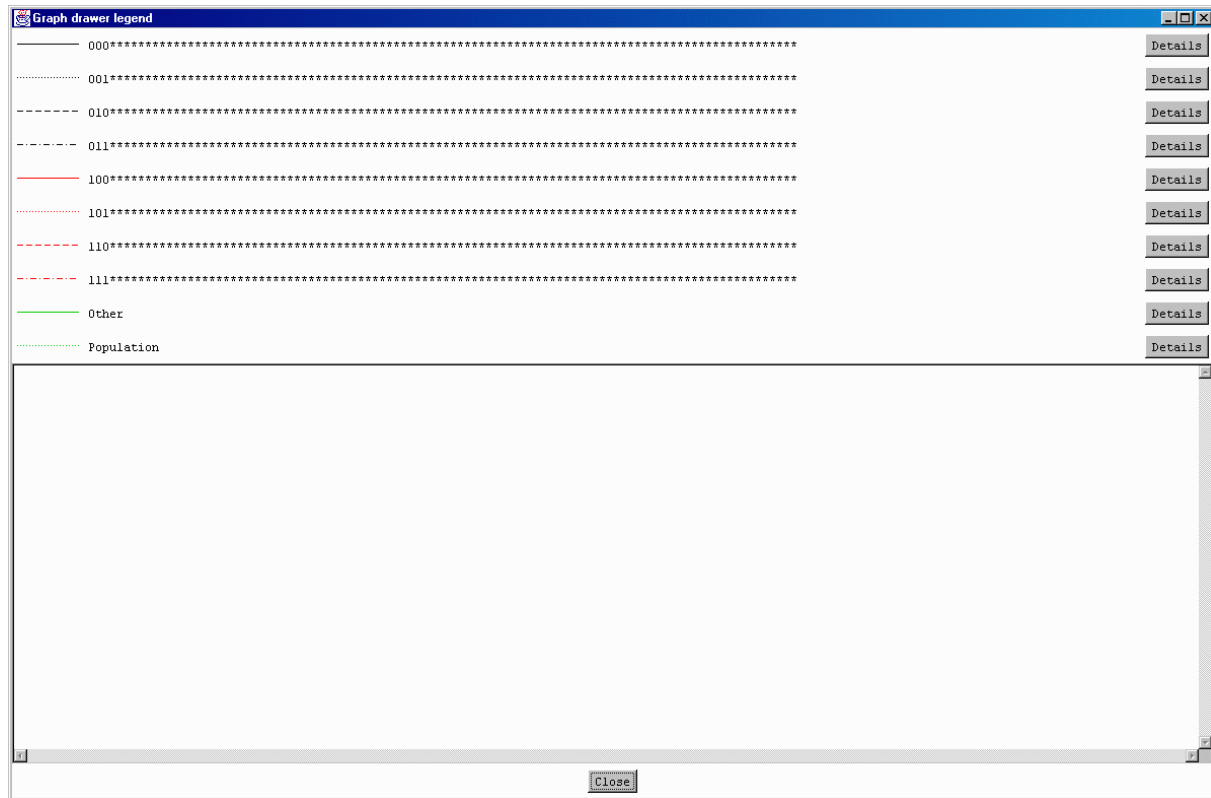


Figure 3.15: The legend for the graph entries for the selected views for bitcounting.

the leading 111 substring. All other individuals will over time decrease to 0. When this happens, the average fitness graph entry will immediately fall down to 0 as well, because the average fitness can no longer be computed and is therefore set to 0 at such a point in time. This means that in the average fitness graph, you will see the one schema after the other “drop out”. In the standard deviation graph you will see that the schema entries will gradually go to a 0 entry, with the 111 substring leading schema holding out the longest. All other information on this simple problem found in the graphs should be clear to the user at this point. We finish this simple example by showing the use of the *Gnuplot* option for the graph drawers. We press the *Gnuplot* button at the bottom of the # In Schemata graph to get the interface for saving the graph drawer data in GNPLOT format. The data that presents the graph entries and the commands for GNPLOT are shown in the interface as well as the files that are to be saved. By pressing the *Save* button, a file dialog appears with the title *Pick/enter any file in desired directory*. This tells us that we should pick any file in the directory in which we wish to save the GNPLOT files. After doing so, we select to save the data by pressing the *Save* button in the file dialog. This will create the files in the selected directory. After running GNPLOT on the plotfile that has resulted by executing `gnuplot InSchemata_plotfile`, a postscript image is generated, which is presented in this tutorial as figure 3.16. The graph entry legends were abbreviated so as to avoid clutter in the image. To this end, the plotfile was adapted by altering the titles for the graph entries. This shows again the power of using an external interactive and powerful plotting program such as GNPLOT, as the data being in textform is easily manipulated and then processed to form an image.

3.4.3 Crowding, preselection, deterministic crowding and sharing schemes

Crowding and preselection are two notions in the world of genetic algorithms that have been around for some time. Both are meant to preserve the diversity in the population. It has been shown however that in practice stochastic errors in the replacement of population members work to create a significant amount of genetic drift, causing unsuccessfulness at preserving population diversity as noted by Mahfoud [10]. At this point we are not going to take a closer look at reasons or experimental explanations to confirm these results in a scientifically justified way. What we will do however is briefly go over the crowding and the preselection scheme as well as the scheme that was introduced by Mahfoud [10] to overcome problems introduced by crowding and preselection. Subsequently we shall look at the most popular way in GA history to achieve multimodal function optimization, which is *sharing*.

Crowding

The crowding strategy makes use of a crowding factor CF which is the number of genomes from the current population that are grouped together. This group is randomly chosen

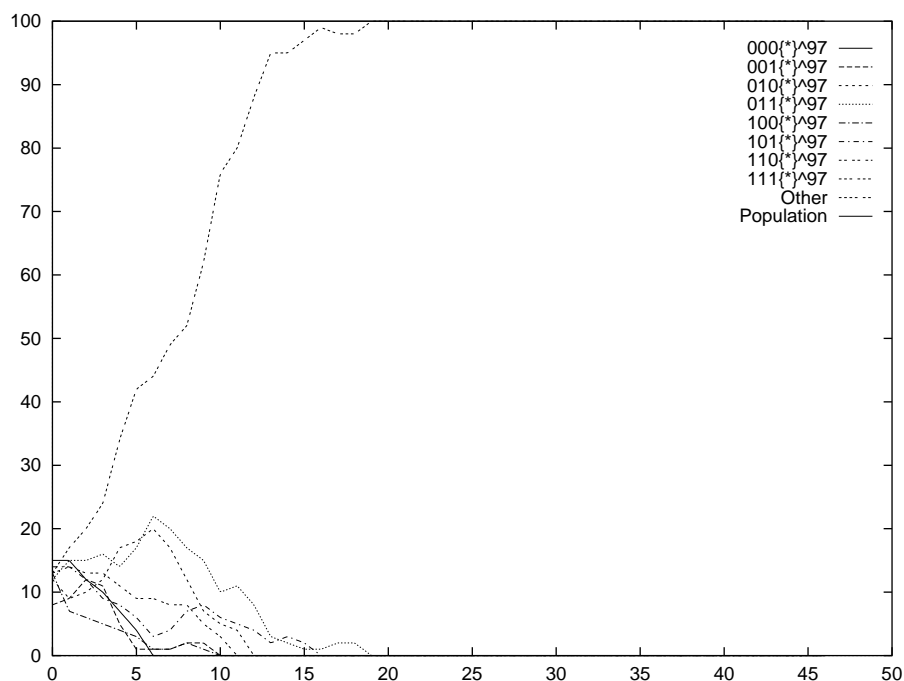


Figure 3.16: The resulting graph from GNUPLOT for # In Schemata.

from the entire population. The genome from the next population to add then finds from this group the most similar genome and replaces that one in the population. The implementation in the *EA Visualizer* disregards the newly added genomes in the subsequent replacement of all the new genomes of the new generation. In order to find the most similar genome, the *Similarity* component is used. The true crowding strategy also employs a *generation gap* G which is a percentage that specifies the portion of the population that is to be selected and subsequently replaced in the population through the crowding factor replacement.

Preselection

The preselection replacement strategy works directly on the offspring and the parents that created the offspring through recombination. To place an offspring into the population, the worst parent is found. If the offspring is better than that parent, it replaces that parent.

Deterministic Crowding

The deterministic crowding replacement strategy was presented as an improvement over *crowding* and *preselection*. For each offspring, the nearest parent according to some distance measure is found and if it's better, it replaces that parent in the population. In order to find the most similar genome, the *Similarity* component is used.

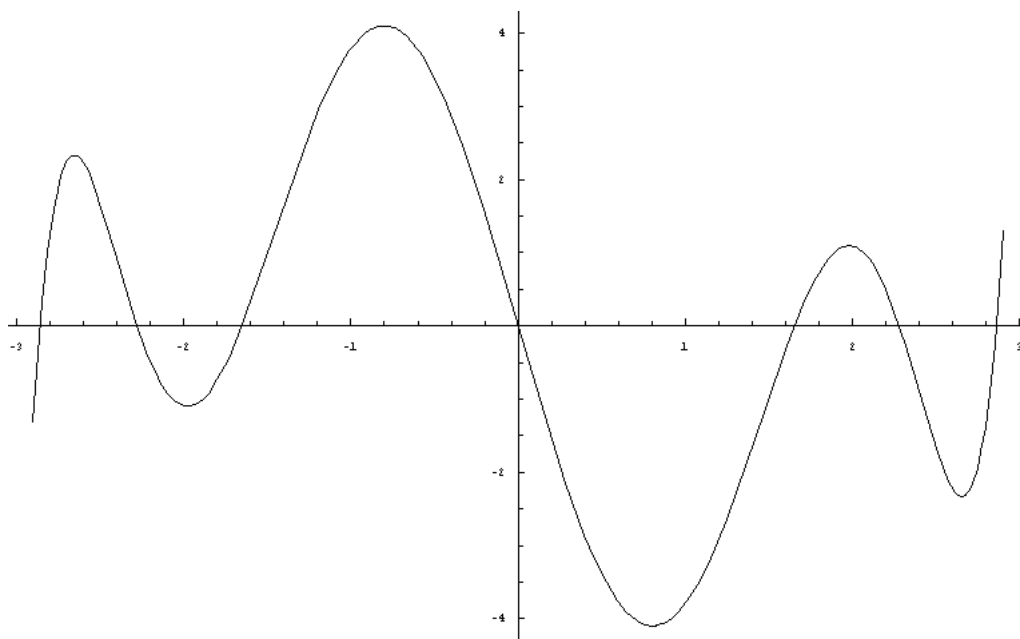


Figure 3.17: The polynome $\frac{4}{1000}x^9 + \frac{5}{1000}x^7 - \frac{4}{5}x^5 + 5x^3 - 8x$ with $x \in [-2.9, 2.9]$.

According to Mahfoud [10], his deterministic crowding mechanism would be an improvement over normal crowding and preselection and he experimentally verified this through showing results using the three strategies on multimodal function optimization. This form of optimization concerns the finding of more peaks than just the optimal one. Peaks that stand out in general are required to be found. The most extreme type of example in this field is maximizing the $\sin(x)$ function with $x \in [0, 8\pi]$ for instance. The maximum value is found for $x = \pi(2k + \frac{1}{2})$ with $k \in \{0, 1, 2, 3\}$ and so there isn't a single solution for x . An example of multimodal function optimization for functions that do not have maxima all at the same value is the maximization of the polynome $\frac{4}{1000}x^9 + \frac{5}{1000}x^7 - \frac{4}{5}x^5 + 5x^3 - 8x$ with $x \in [-2.9, 2.9]$. This function is depicted in figure 3.17. The maximum value for the polynome is obtained for $x \approx -0.80145$ and equals approximately 4.1. In the light of multimodal function optimization however it is not the goal to find only this optimum, but also the other peaks which are local optima in terms of optimization. As crowding, preselection and deterministic crowding are said to allow for preservation of population diversity, this should result in the finding of all peaks. We shall now turn to installing the three different algorithms in the *EA Visualizer* and running them to observe the results.

We start out with the crowding scheme. From the menubar in the *EA Visualizer* we select to create a new single run EA. As we are to tackle the problem using genetic algorithms, we select to use the **Binary String** as the *Genotype* and set the string length to 25 bits. The *Similarity* component is of great importance now as it defines during the run to what extent two binary strings are similar. We start out with the simplest of similarity measures, being

the **Bitwise Difference**. This difference measure is equal to the amount of bits that are matched, scaled to $[0, 1]$. In other words, the *Hamming* distance is expressed through this *Similarity* component. The fitness function is of course set to be the polynome optimizer with its parameters set according to the specifications above. The *Recombinator* is set to be the classical one-point crossover with two offspring. As we do not wish to have mutation in the algorithm, we specify to use **No Mutation** as the *Mutator*. The effect of this is the same as selecting any other mutator and keeping the mutation chance at 0. The *Before Selector* is taken to be tournament selection with a selection size of 100 and a tournament size of 2. The *After Selector* is not to be used so we set this to be **No Selection (Select All)**. The *Mater* is to mate the selected genomes in groups of two so that they can be sent in exactly these groups of two to the *Recombinator*. Therefore we install the **Simple Mater** as the *Mater* and set the grouping size to 2. The simple mater does nothing more than go over the population from top to bottom and mate the genomes in groups of the specified size. As such this approach is the fastest. Only slightly slower (negligable speed reduction) is the random mater that goes over the population in a random order until all genomes have been mated. We can however suffice with the simple mater because the tournament selection mechanism shuffles the population to go into a new tournament round. This indeed means that when no selection is used as the *Before Selector*, the random mater must be used so as to make sure that no restrictions are introduced in what genomes can be mated. This is all explained in the help system as well which can be reached by pressing F1 on the keyboard at any time. Continuing on the settings, the *Replacer* is set to be the **Crowding** replacer with a crowding factor of three¹. If the crowding mechanism indeed succeeds in maintaining the population diversity, it is no use to select to have a *Terminator* that terminates when all genomes are equal because that will then never happen, so we select to have the **Maximum Generations** instance for the *Terminator* component with 100 as the maximum of generations. We employ no hybrid searcher and set the population to be the simple and standard **Vector Population** with a population size of 100 genomes. The prng is set to be the standard JAVA prng with a seed of 902480582260, which is the time in milliseconds at which the parameter component in which the seed can be specified was made. It doesn't matter what number it says here. The recombination chance is set to be 1.0 and the mutation chance to be 0.0. When we presented the first single run example in section 3.4.1, we already showed the interface in which to enter the settings (the interface coded in the **EASettingsInterface** class) in figure 3.6. The following table with the instances to select for the components and the parameters to set for these instances should therefore suffice for the reader to try out the example him or herself.

¹Note that in the classical sense of crowding, we have a generation gap of 100%.

Component	Instance	Parameters
<i>Genotype</i>	Binary String	String length 25
<i>Similarity</i>	Binary String – Bitwise Difference	—
<i>Fitness Function</i>	Binary String – Polynome Optimizer in [a, b]	Function $0.004x^9 + 0.005x^7 - 0.8x^5 + 5x^3 - 8x$ Lower limit (a) -2.9 Upper limit (b) 2.9 Optimization Maximize
<i>Recombinator</i>	Binary String – One Point Crossover	Offspring Arity 2
<i>Mutator</i>	No Mutation	—
<i>Before Selector</i>	Tournament Selection	Selection size 100 Tournament size 2
<i>After Selector</i>	No Selection (Select All)	—
<i>Mater</i>	Simple Mater	Grouping size 2
<i>Replacer</i>	Crowding	Crowding Factor 3
<i>Terminator</i>	Maximum Generations	Maximum generations 100
<i>Hybrid Searcher</i>	No Hybrid Search	—
<i>Population</i>	Vector Population	Population size 100
<i>PRNG</i>	Standard Java PRNG	Seed 902480582260 Use Random Seed Instead No

After the *Apply* button has been pressed, we are ready to add views to the system to observe the events as they evolve. By pressing F8 we are presented with the views that we can add. In the light of what we set out to achieve, namely multimodal function optimization, it is no use to trace the best and worst genome or the fitness average. It is more interesting to inspect things like cluster analysis and replacement errors. These have not (yet) been implemented in the *EA Visualizer* however, but if the reader finds it interesting, it is a good exercise to implement exactly this in a view. For now, we select to add the **Locations On Polynome** view to actually see in a phenotypic sense where the genomes are going over the generations. The maximum value for the graph is set to 5 and the minimum to -5. The result is a graph in which a blue line is the polynome and the red crosses are the locations of the genomes². Furthermore to view what happens with the genetic material in the population, we add a **Binary String Population Dots** view that will show every bit in every genome in the population from top to bottom. As we have a string length of 25 and 100 genomes in the population, the width/height ratio should be $\frac{1}{4}$ to have every bit to be a square instead of a non-square rectangle. We therefore set the width of the view to be 100 pixels and the height to be 400 pixels. The result is a view in which every red dot is a 1 symbol in the population and every blue dot a 0 symbol. The genomes themselves are the rows in the block. At this point we are ready to start the algorithm to see what happens and so we “press play” in the menubar. After only a few

²Note that the polynome is completely point symmetric in (0, 0).

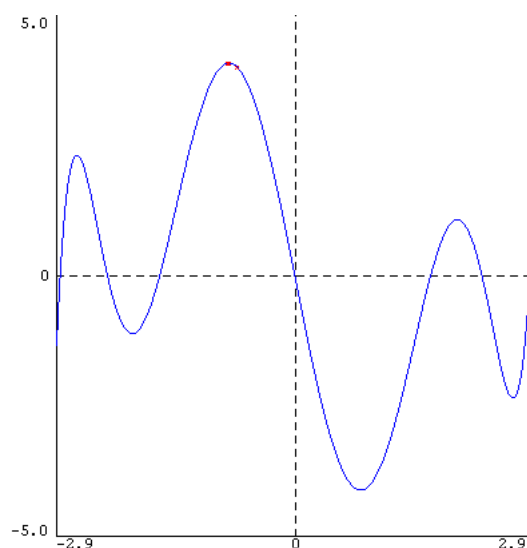


Figure 3.18: Polynome locations after 100 generations using crowding and genotypic similarity.

generations, we indeed see that the population does not directly converge to the maximum value and that the suboptima are also kept populated by genomes. However this is only shortlived and after some time only the largest peak is still populated, however not all genomes are at the same position as crowding still keeps them apart. The population dots view shows us indeed that in some sense the diversity is kept as the least significant bits (the rightmost) are still quite random. This poses exactly the problem with the approach because the observed diversity is in genotypic space, while we wish to keep the diversity in phenotypic space of course. The resulting polynome and the locations of the genomes on it are depicted in figure 3.18 and the resulting population contents are shown in figure 3.19.

Even though we have at this point deduced that it will be much better to have a similarity measure in phenotypic space, as this is only an example we will hold that thought for the moment and first now try preselection with the same similarity measure. To do this, we press F5 or select **Settings Current Single Run EA** from the EA menu. We alter the *Replacer* to be **Preselection** instead of **Crowding** and note how the *Parameters* button is immediately disabled because there are no parameters to be set for preselection. It is at this point that we must realize that the preselection operator in the *EA Visualizer* only accepts a single offspring genome to set back amongst its parents. As such we need to alter the parameters of the one-point crossover operator installed and set the amount of offspring it generates to one. After installing the new *Replacer* and by altering the parameters for the *Recombinator*, we press the **Apply** button and note in the **System Messages** window how the system reports that a new *Replacer* has been installed and that the views have been checked to still be valid for the altered contents of the evolutionary algorithm. The EA

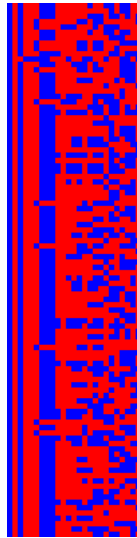


Figure 3.19: Population contents after 100 generations using crowding and genotypic similarity.

remains however in the `Terminated` state and we must first reset the algorithm before we can continue. At this point we point out that there are two ways of resetting the algorithm, namely through `F6` and `F7` which represent the resetting of the entire algorithm and only the population respectively. Resetting the entire algorithm means that the system notifies each instance that is currently installed that it has to reset itself. This must be used when instances are installed that have some sort of memory which must be reset when a new independent run is to be started. In our case it suffices to only reset the population which is done by pressing `F7`. Note how the generation counter is also reset when we do so. This is done because in the `Options` menu the `Resetting Population` option is set to also reset the generation counter upon resetting of the population through the pressing of `F7`. We now have a completely reset EA but now with preselection instead of crowding and we once again start the algorithm by clicking the *play* button. The results are shown in figures 3.20 and 3.21. The results are even worse than for crowding because the population has converged almost to a single point. Indeed if the other peaks could be populated as well in a similar fashion, the preselection operator would work better here because the genomes converge more to a single point in that optimum whereas crowding kept more diversity. Note that the remark that preselection works better is of course restricted to this problem. Also, during the run with preselection, the diversity witnessed again was in the genotypic sense, providing us with yet more assurance that a phenotypic distance measure is essential in this type of problem. On a practical note, we mention at this point that the views presented here are directly captured from the *EA Visualizer*. This has been done by using the copy and paste methods of WINDOWS 98.

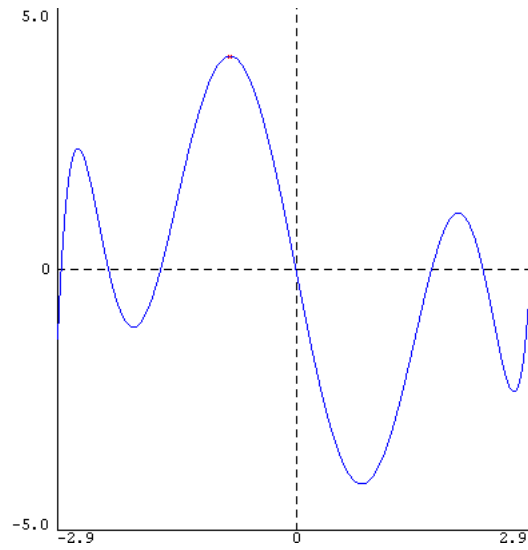


Figure 3.20: Polynome locations after 100 generations using preselection and genotypic similarity.

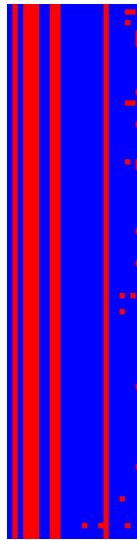


Figure 3.21: Population contents after 100 generations using preselection and genotypic similarity.

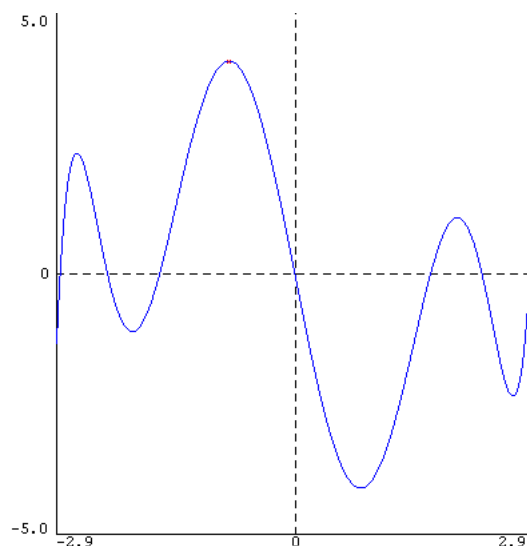


Figure 3.22: Polynome locations after 100 generations using deterministic crowding and genotypic similarity.

Concluding the triple of crowding, preselection and deterministic crowding approaches, we wish to test deterministic crowding. In order to do this, we select to edit the settings for the current single run EA and set the *Replacer* to be the **Deterministic Crowding** replacer. As this operator can handle again two offspring, we set the offspring arity for the one-point crossover operator back to two. After pressing the **Apply** button and resetting the population, we are ready to press again the *play* button to start the algorithm. Behaviour similar to that of preselection is observed and rather quick convergence to the highest peak takes place. The deterministic crowding replacer shows again results that point out that the genotypic matching is all wrong for problems such as these. We shall not go into details further at this point since this is only an example. The results upon termination after 100 generations are depicted in figures 3.22 and 3.23.

Even though at this point we have clearly demonstrated how easy it is to incorporate mechanisms such as crowding, preselection and deterministic crowding, the feeling is rather unsatisfactory as the results with respect to the problem of multimodal function optimization are very poor. We noted before however that a phenotypic similarity measure is required for this problem and so in figure 3.24 we present all the results for the three mechanisms anew, but now using the **Binary String - Range Difference** similarity measure that uses the locations of the genomes with respect to the x range, meaning that two genomes are said to be more equal when their x coordinates are closer together. Indeed the results are much better now as can be seen in figure 3.24.

To illustrate further how less standard items are easily incorporated in the *EA Visualizer*, we will shortly describe and operationally introduce the most popular approach to mul-

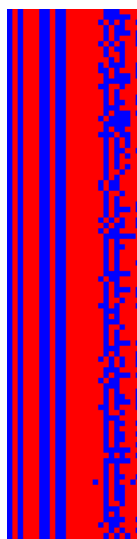


Figure 3.23: Population contents after 100 generations using deterministic crowding and genotypic similarity.

timodal function optimization in genetic algorithms, which is called *sharing*, and should be able to achieve better results than seen so far. The sharing scheme that we will test is an extended sharing scheme that uses cluster analysis methods especially for multimodal function optimization. The general idea is to divide the population into clusters that are adaptively formed and to degrade the fitness of the members in a cluster by an amount that is dependent on the amount of genomes in the cluster. This way the overpopulated clusters will have lower fitness values for their members and subsequently we will more likely get a good spread over the suboptima as their fitness is relatively better because the amount of individuals in a cluster at that point is less. To implement this, we need a new kind of expansion for the *EA Visualizer*. We need to create a new *Population* that acts as an environment with information on the whereabouts of the genomes. The result is the **Sharing Clustered Population** which we shall introduce next.

The **Sharing Clustered Population** is an implementation based on a sharing scheme as proposed by X. Yin and N. Gernay [14]. The additional information that this population holds over the standard population (the **Vector Population** in the *EA Visualizer*), is the clusters over which the population is divided. Each genome is assigned to a cluster. To find these clusters, a distance measure is needed, which is established through the usage of the *Similarity* component. The clusters are determined with the *Adaptive MacQueen's KMEAN Algorithm*. The implementation of the clustering algorithm together with the sharing scheme is as follows (taken from the article by Yin and Gernay [14]):

After the reproduction and the crossover processes in each generation, the clustering algorithm is applied to cluster all the individuals in the population into different niches. The number of individuals in each niche are determined. Then,

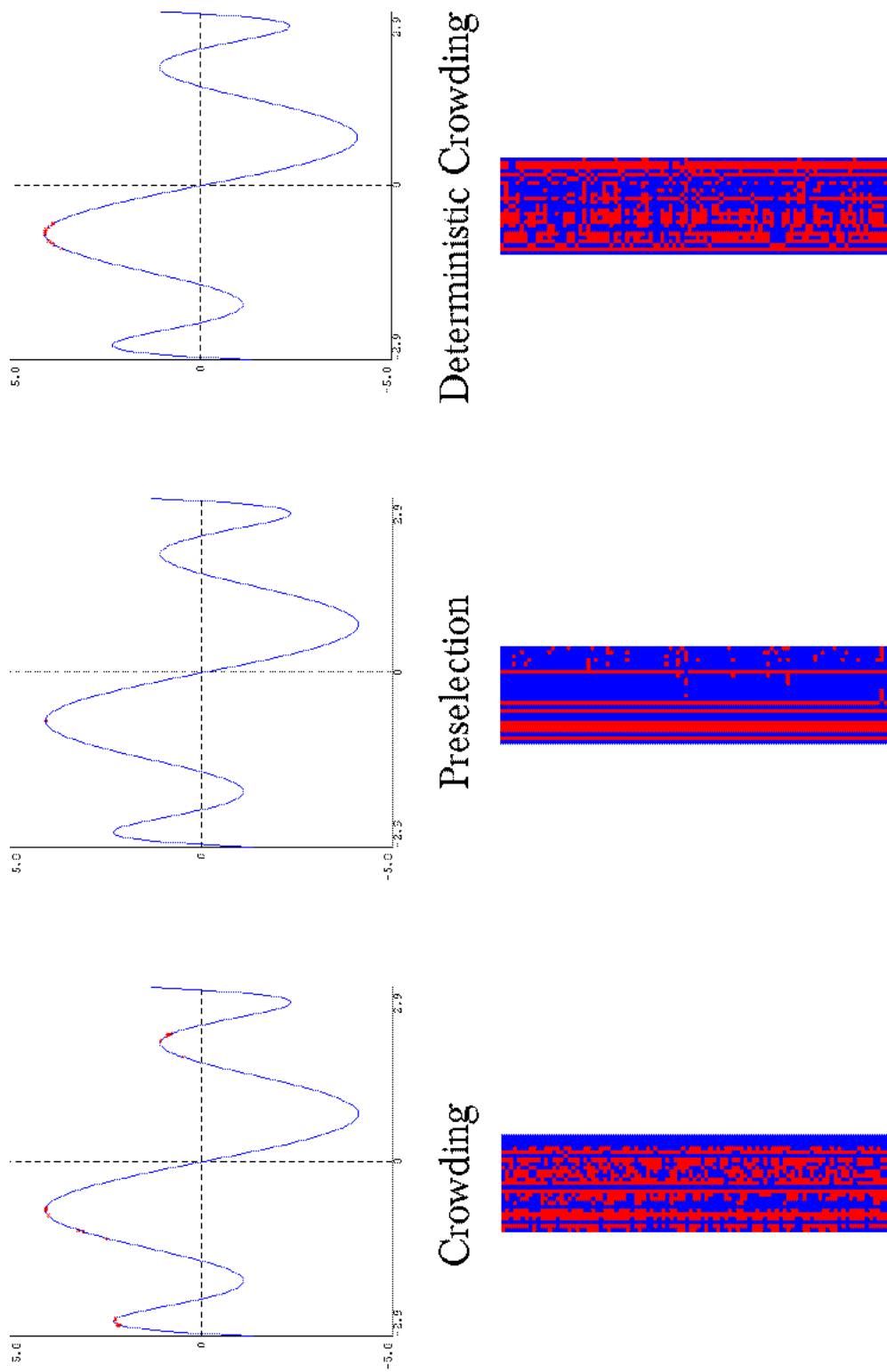


Figure 3.24: Results after 100 generations for all replacers using phenotypic similarity.

for individual i , its potential fitness f_i is divided by the approximated number of individuals in the niche to which it belongs m'_i :

$$f'_i = \frac{f_i}{m'_i} \quad \text{with } m'_i = n_c - n_c * (d_{ic}/2 * d_{\max})^\alpha \text{ for each } x_i \text{ in cluster } c$$

In the above, n_c is the amount of genomes in the cluster, d_{ic} is the distance between individual i and its niche's centroid, d_{\max} is the maximum distance between clusters (parameter for the Adaptive MacQueen's KMEAN Algorithm), and α is a constant. The KMEAN algorithm is an algorithm that finds clusters amongst data entries based upon a distance measure between those data entries. In our case the data entries are the genomes in the population and the distance measure is the *Similarity* component. We set out to test the clustering sharing method on the same problem using as many similar settings as possible. We initiate the example using the same views as before and the following settings for the single run EA (the settings for the population were taken from the paper by Yin and Gernay [14], except for the maximal centroid distance, which was increased specifically for this problem):

Component	Instance	Parameters
<i>Genotype</i>	Binary String	String length 25
<i>Similarity</i>	Binary String – Bitwise Difference	—
<i>Fitness Function</i>	Binary String – Polynome Optimizer in $[a, b]$	Function $0.004x^9 + 0.005x^7 - 0.8x^5 + 5x^3 - 8x$ Lower limit (a) -2.9 Upper limit (b) 2.9 Optimization Maximize
<i>Recombinator</i>	Binary String – One Point Crossover	Offspring Arity 2
<i>Mutator</i>	No Mutation	—
<i>Before Selector</i>	Tournament Selection	Selection size 100 Tournament size 2
<i>After Selector</i>	No Selection (Select All)	—
<i>Mater</i>	Simple Mater	Grouping size 2
<i>Replacer</i>	New Offspring Only	Report popsize warnings Yes
<i>Terminator</i>	Maximum Generations	Maximum generations 100
<i>Hybrid Searcher</i>	No Hybrid Search	—
<i>Population</i>	Sharing Using Adaptive Clustering	Population size 100 Initial Number Of Clusters 10 Minimal Centroid Distance 0.05 Maximal Centroid Distance 0.5 Alpha 1
<i>PRNG</i>	Standard Java PRNG	Seed 902480582260 Use Random Seed Instead No

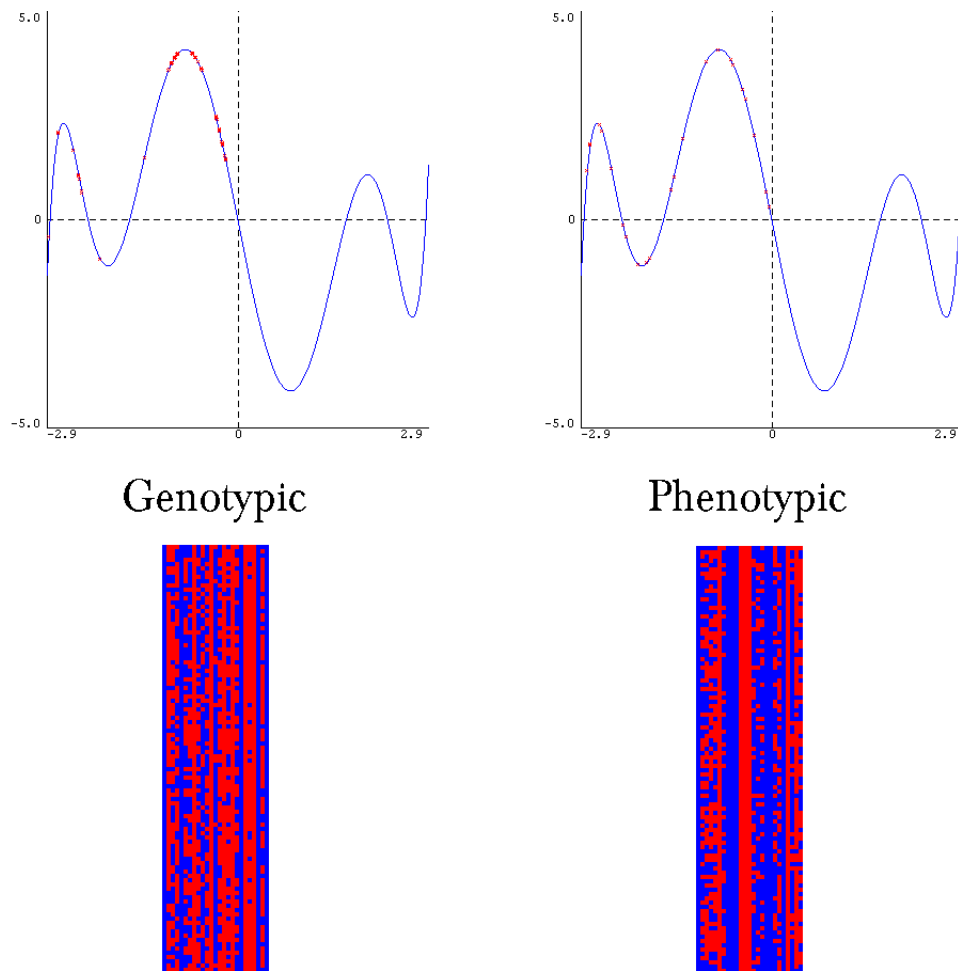


Figure 3.25: Results after 100 generations for the clustering sharing scheme using genotypic and phenotypic similarity.

Note that we have again started out using a genotypic similarity measure. We have seen above in the examples for crowding, preselection and deterministic crowding that such a similarity measure leads to unsatisfactory results as the similarity measure does not allow for clusters to form in a space that we want the clusters to form, namely the phenotypic space which for this problem is the range of the x axis. Now using this explicit clustering mechanism it is even more obvious that a phenotypic similarity measure is required, because that is the only logical space to base the cluster creation on. The results of both the genotypic and the phenotypic similarity measure are depicted in figure 3.25 and from the results for the genotypic similarity only it is already obvious that the search for clusters is more powerful in the sharing scheme as can be observed from both the resulting population contents as well as the locations on the polynome.

Finally, the article by Yin and Germary closes with the introduction of a mating restriction. This mating restriction implies that only genomes that come from the same cluster are allowed to be mated together and thus to produce offspring. This encourages the forming of tight clusters even further. Unfortunately however, as can be seen from the results in figure 3.26, the restriction is just a little too tight for the settings that we entered and the amount of clusters is reduced to exactly one, being far from the desired result. To remedy the too fast selection, we can introduce a mild random mutation. Doing this in itself shows the diversity and ease in use of the *EA Visualizer* because after the 100 generations that resulted in the undesired effect, we simply select the **Binary String Random Mutation** as the mutator and set the mutation chance to 0.01. Furthermore we alter the termination condition to have a maximum of 1000 generations just to be able to continue evolving with the current result. After these changes, we start the algorithm again and witness that after another 100 generations, the diversity is already a lot greater, even with the slight mutation rate. These results are shown in figure 3.26.

3.4.4 Evolution strategies and elitist recombination/replacing

In this section, we briefly go over an example that concerns the evolutionary algorithm known to be well suited for real function optimization, being *Evolution Strategies*. The evolution strategies are reviewed in an article by Bäck, Schütz and Khuri [1] and presented more extensively at a later time by Bäck and Schwefel [2]. The main difference with the genetic algorithm is the representation which in the case of evolution strategies (ES) is a vector $(x_1, \dots, x_n, \sigma_1, \dots, \sigma_{n'}, \alpha_1, \dots, \alpha_{n''})$, consisting of n object variables, standard deviations for individual mutations and optionally additional angles for mutations in typical directions (correlated mutations). The mentioned standard deviations for individual mutations are commonly referred to as the *meta*-parameters. The coding of the problem is done through the x_i parameters. The σ_i are positive and algorithmically enforced to stay larger than a specified ε . The α_i are angles between $-\pi$ and $+\pi$ and are enforced to stay within these bounds also by circularly mapping them into the feasible range whenever it is left by mutation. The implementation in the *EA Visualizer* follows to a large extent the implementation by Bäck and Schwefel [2].

There are three possible combinations for n' and n'' to choose from for applying the mutation operator:

- $n' = 1, n'' = 0$. Standard mutations with one single standard deviation controlling mutation of all components of x .
- $n' = n, n'' = 0$. Standard mutations with individual step sizes, $\sigma_1, \dots, \sigma_n$, controlling mutation of the corresponding object variables x_i individually.
- $n' = n, n'' = \frac{n(n-1)}{2}$. Correlated mutations with a complete covariance matrix for each individual.

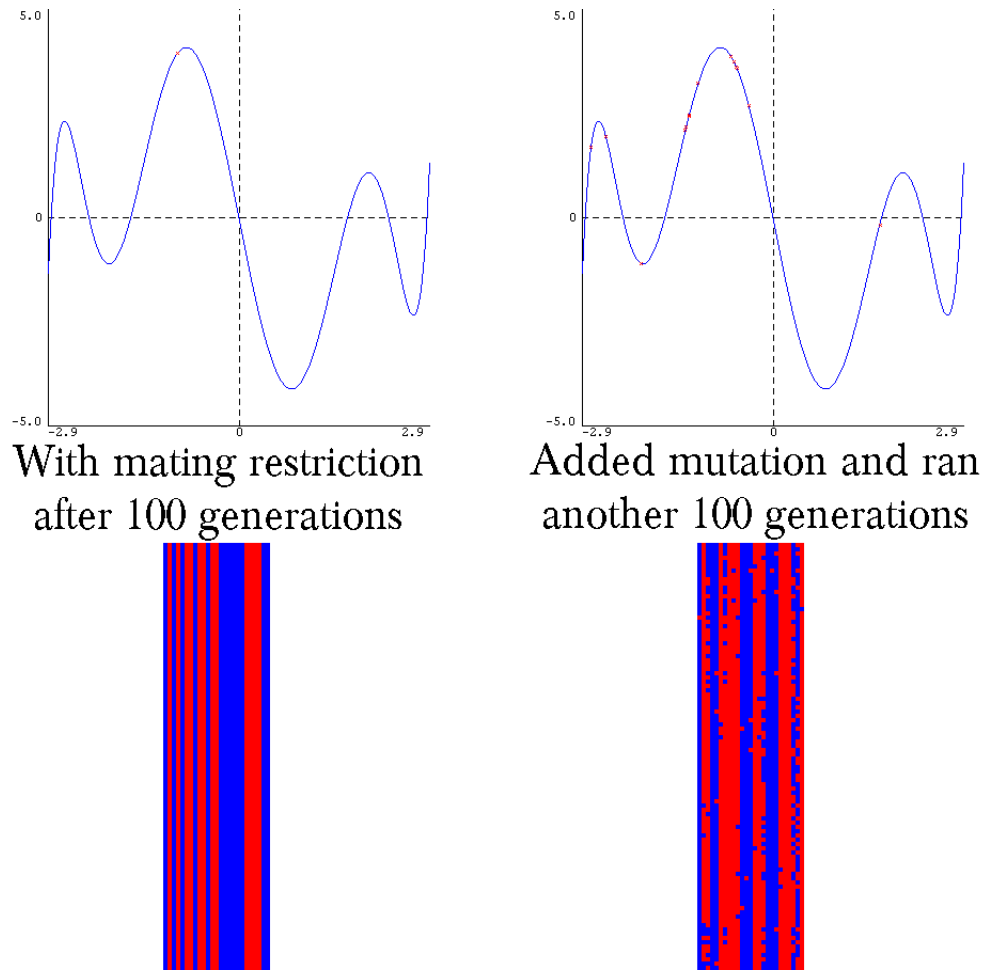


Figure 3.26: Results after adding a mating restriction and subsequently mutation.

A string length of 0 in the *EA Visualizer* implies that the setting of the length parameter will be skipped so that other parts in the EA can adapt the length of the string without problems. In other words, a length of 0 implies non-fixed length for the ES string.

Mutation is performed by first mutating the meta parameters according to $\sigma'_i = \sigma_i e^{(z_i + z_0)}$. Here z_i is a normally distributed variable $N(0, \tau')$ and z_0 is a normally distributed number $N(0, \tau)$.

Second, the angle parameters are mutated according to $\alpha'_i = \alpha_i + z_i$. Here z_i is again a normally distributed number, but now with $N(0, \beta^2)$. Empirically, $\beta = 0.0873$ (about 5 degrees) has shown to yield good results.

Finally, the actual variables are mutated according to $x'_i = x_i + cor_i(\sigma, \alpha)$. Here the vector *cor* can be computed as $cor = Tz$ where z is a vector of normally distributed numbers with $N(0, \sigma_i^2)$ and T is the product over all matrices $T_{pq}(aj)$ with $j = \frac{(2n' - p)(p + 1)}{2 - 2n' + q}$. The rotation matrices T_{pq} are unit matrices except that $t_{pp} = t_{qq} = \cos(aj)$ and $t_{pq} = -t_{qp} = -\sin(aj)$.

If there is only a single meta parameter s_1 and no angle variables α , the actual variables are mutated simply like this: $x'_i = x_i + \sigma'_1 N(0, 1)$. If we have multiple meta parameters σ_i and no angle variables, the mutation of the actual variables is like this: $x'_i = x_i + \sigma_i N(0, 1)$. In the case we have multiple meta parameters and multiple angle parameters, the mutation is performed using the matrices as described above. By experimental results, the following settings have proven successful: $\tau = \frac{1}{\sqrt{2n}}$ and $\tau' = \frac{1}{\sqrt{2\sqrt{n}}}$. The mutation operator in the *EA Visualizer* that was defined according to the above specification (**ES Mutation** instance) does *always* perform mutation, regardless of the mutation probability set. If no mutation is required, the operator that was specifically designed for this should be used (**No Mutation** instance). In this example we shall use the variant for mutation where $n' = n, n'' = 0$.

General recombinators have been specified in the past of which the classical are *intermediate* and *discrete* crossover. Discrete crossover acts as uniform crossover and determines per entry in the string from which of the two parents the entry must be inherited. Intermediate crossover takes the average value for the entries in the chromosomes. Both crossover operators can directly be used to support more than two parents. Research has shown that the best results are achieved when discrete recombination is used for the object variables and intermediate recombination for the meta parameters.

Next to the specialized mutation and recombination mechanisms, the selection operator is also of a special form in evolution strategies. For the sake of showing the diversity of the *EA Visualizer*, we shall restrict ourselves to the $(\mu + \lambda)$ selection mechanism. This mechanism states that there are μ parents and λ offspring. The μ genomes selected to survive are in the $(\mu + \lambda)$ selection mechanism chosen from the both the parents and the offspring (hence the + in the name). How this selection is done is in the strictest of senses unspecified, but normally this is achieved by simple truncation selection that selects the μ best genomes.

Incorporating evolution strategies in the *EA Visualizer* means that a new *Genotype* needs to be created, along with a new *Mutator* and a *Recombinator*. This is exactly what has been done alongside the expansion of the polynome optimization component to also allow for evolution strategies genomes. The requirements to thus install a classic ES in the *EA Visualizer* that optimizes the same polynome as we used in section 3.4.3 are now clear and the following table shows how to actually configure the *EA Visualizer*:

Component	Instance	Parameters
<i>Genotype</i>	Evolution Strategies	Length 1 Min. init range values -2.9 Max. init range values 2.9 Min. init range meta-parameters 0 Max. init range meta-parameters 1 Combination type sd = n, alpha = 0
<i>Similarity</i>	No Similarity	—
<i>Fitness Function</i>	Evolution Strategies – Polynome Optimizer in $[a, b]$	Function $0.004x^9 + 0.005x^7 - 0.8x^5 + 5x^3 - 8x$ Lower limit (a) -2.9 Upper limit (b) 2.9 Optimization Maximize
<i>Recombinator</i>	ES Crossover	Crossover type for values Discrete Crossover type for meta-parameters Intermediate Crossover type for angle-parameters Intermediate
<i>Mutator</i>	ES Mutation	General Tau 0.7071067811865 Individual tau 0.7071067811865 Minimal St. Dev. 0.0 Beta <i>Any</i>
<i>Before Selector</i>	No Selection (Select All)	—
<i>After Selector</i>	Truncation Selection	Selection size 100 Percentage 50 Which To Use Selection size
<i>Mater</i>	Random Mater	Grouping size 2
<i>Replacer</i>	Add Offspring To Population	—
<i>Terminator</i>	All Equal Genomes	—
<i>Hybrid Searcher</i>	No Hybrid Search	—
<i>Population</i>	Vector Population	Population size 100
<i>PRNG</i>	Standard Java PRNG	Seed <i>Any</i> Use Random Seed Instead Yes

The recombination chance and mutation chance are both set to 1.0. After creating the EA by pressing the **Create EA** button, views can once again be added to the system. As we are looking for the optimum value, we are interested in the best and worst value so far in the system. Therefore, we add a **Best & Worst Evolution Strategies Genome** to the views. Next, we add two **Fitness Statistics** views that show us the average and the variance in fitness values over the population. Lastly, a **locations on polynome** view is added to actually see where the individuals in the population are located on the polynome. Using all standard sizes however, we have now created a clumsy ordering of the views. With the 1024x768 pixels as the screen resolution that is used to write this example alongside, the two statistics views are situated next to each other and just fit in the width of the window, separating the best and worst view from the locations on the polynome view. Putting these latter two views next to each other would save height and perhaps save us from having to use the scrollbar as is now required because the total height of the view is larger than the available viewing space height. To remedy this, we select from the **Views** menu the option **Change Order Of Views** and move the polynome locations view all the way to the top. When we apply the view order switching, the system relayouts the views and indeed we save a lot of height, but we can yet not see everything without scrolling vertically. To this end we change the dimensions of the polynome locations view by selecting it from the **Views** menu. We set the **Width** to be 400 pixels because we seem to have some width left in the first row of the views and for the sake of the example would like to use that width. Furthermore we reduce the height to 250 pixels. After applying the results, finally everything fits on screen and we are ready to start the algorithm by pressing the *play* button.

After fifteen generations the optimum has already been found to a very high precision as can be seen in figure 3.27 and can be compared with the maximum value for the polynome that was stated in the previous example section (section 3.4.3). However, the algorithm can proceed beyond the fifteen generations because not all genomes are totally equal yet. This is of course a problem in evolution strategies because the mechanism takes very long to terminate as the offspring are always mutated and it takes a long time to get a population with 100 members without finding any better ones at all under the circumstances of the precision offered by the ES genotype. After 300 generations we decide to stop the algorithm ourselves.

In order to speed up the termination of the algorithm that uses the ES genotype, we could introduce the *elitist* recombination strategy [13]. Elitist recombination is an augmented recombination operator that creates offspring and then uses elitist behaviour to see which of the parents and offspring will survive into the next generation. This truncation selection kind of behaviour is thus done in a local fashion for each pair of parents and offspring. What the elitist recombination operator is thus actually doing is performing replacement. In the *EA Visualizer* to this end a special *Replacer* component has been created. Indeed, the elitist recombination specification is a composite one and is therefore not a neatly one. The elitist behaviour described is a replacement strategy as it selects which of the parents

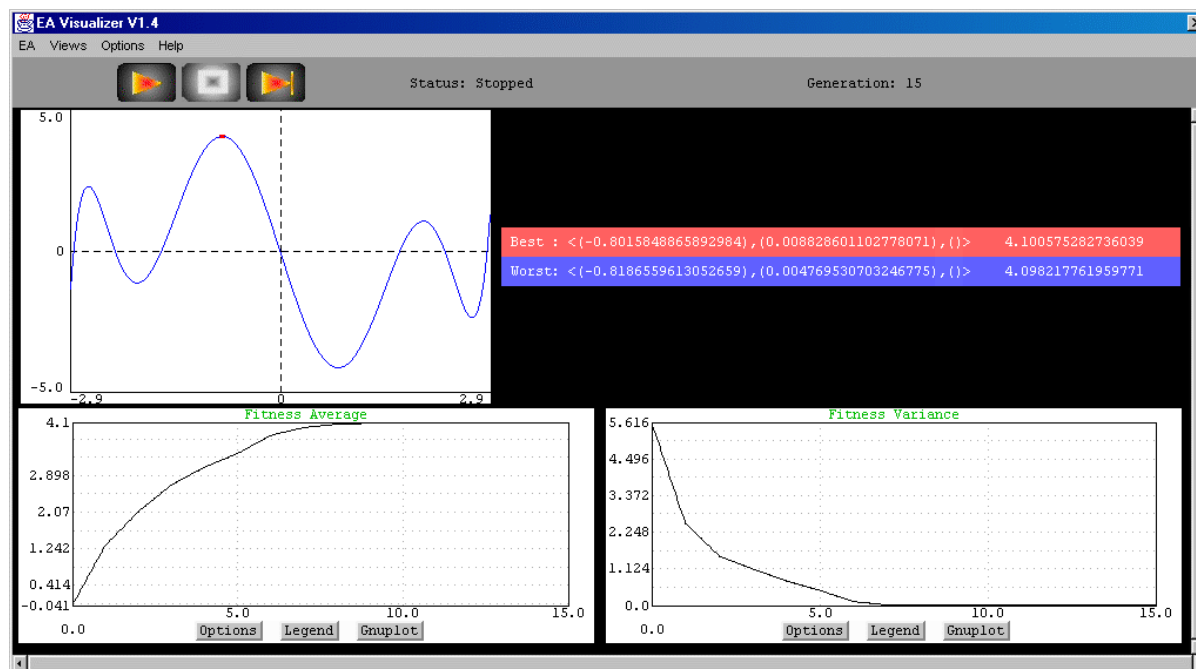


Figure 3.27: The results of running the ES after 15 generations.

and offspring that were involved in a single *Recombinator* application are to be placed in the population to go into the next generation. Therefore the name *elitist recombination* is somewhat inappropriate. A better naming would be *elitist replacing* and indeed in the *EA Visualizer* a *Replacer* component instance is created called *Elitist Replacing* that is the implementation of the elitist behaviour as stated.

Incorporating the *Elitist Replacing* replacement operator at the expense of the currently installed operator (*Add Offspring To Population*) will generally make an evolutionary algorithm terminate much faster as offspring are directly not selected if a recombination and subsequent mutation has resulted in a lesser offspring than the parent. So instead of global behaviour, the elitist replacement strategy is much more strict and has local behaviour that speeds up convergence, but also very easily causes premature convergence because of quick loss of diversity in the population. We do not address the theory of that further at this point as we only serve to provide the reader with the example. The resulting algorithm is thus a composite algorithm that uses both the evolution strategies approach but without the truncation selection globally on both the parents and the offspring afterwards but with the elitist replacing that is actually truncation selection locally on both the parents and the offspring. The only parameter for the *Elitist Replacing* instance of the *Replacer* component is the *Selection Size* which specifies how many genomes should be selected from both the parents and the offspring that are concerned with a single application of the *Recombinator* operator. To set the correct value for this

parameter, at this point it should be noted that the **ES Crossover Recombinator** takes n parents and creates only a single offspring because of the possible intermediate crossover which obviously allows only for a single different offspring to be generated. Concluding, in order to keep the population at a size of 100 every generation, the local elitist replacing strategy should select 2 out of the 3 genomes involved in the recombination process because the *Mater* creates 50 groups of 2 genomes from the 100 population members, so from each of these groups 2 genomes should again be selected. Setting these parameters however seems to make the altered ES converge even slower and easily go over 300 generations again. To really speed up the convergence process and to see how easy it is to alter settings and incorporate new strategies and make new combinations, we select to have tournament selection as the *Before Selector* that selects 100 genomes instead of the selection mechanism that simply selects all genomes. Now we have a twofold selection scheme and the pressure is very high, leading to much quicker convergence. Indeed, after 48 generations the so constructed new EA terminates with a near optimal solution as can be seen in figure 3.28. The best and worst view for the ES shows the best and worst string upon termination which are of course the same because of the termination condition that signals termination upon equality of all genomes. The strings are indeed of length 2 as we require for this one-dimensional problem only a single object parameter x_1 and thus we have also a single meta parameter σ_1 , which are exactly the real number entries in the string that can be seen in the best and worst view (meaning that the first number in the string found there is the value for x in optimizing the polynome $p(x)$). Behind the strings the fitness value (which equals the polynome response value $p(x)$) is stated.

3.4.5 Advanced: MIMIC, FDA and others

Of late, new approaches to GA like problem optimization have come into being that can be placed in the history of linkage learning [5, 7]. Linkage learning is a term that has come forward out of the classical GA history and is therefore deemed to be tackled by a GA that allows for finding relations to aid its selection and recombination of material. However the notion of linkage learning can be taken in a much more general fashion, namely as the cohesion of the bits in the coding string with respect to the fitness landscape (search space). The important thing is thus finding structure amongst the bits and exploiting the found structure. In terms of GAs the idea was to find the building blocks and respect the building block borders in the exchange of material between strings. However, this is not the only way in which the linkage information can be processed. When the linkage information is made more explicit in terms of statistics, one might be able to say things such as bit i is set to 1 with chance 0.9 when bit j is set to 0 implying that a certain linkage has been found between bit j and bit i . This is exactly the new perspective that has been employed by De Bonet, Isbell and Viola in the creation of MIMIC [4] and in an extended version by Baluja and Davies [3]. An overview of related material in this field can be found in an article by Bosman and Thierens [7] that regards the linkage information processing in

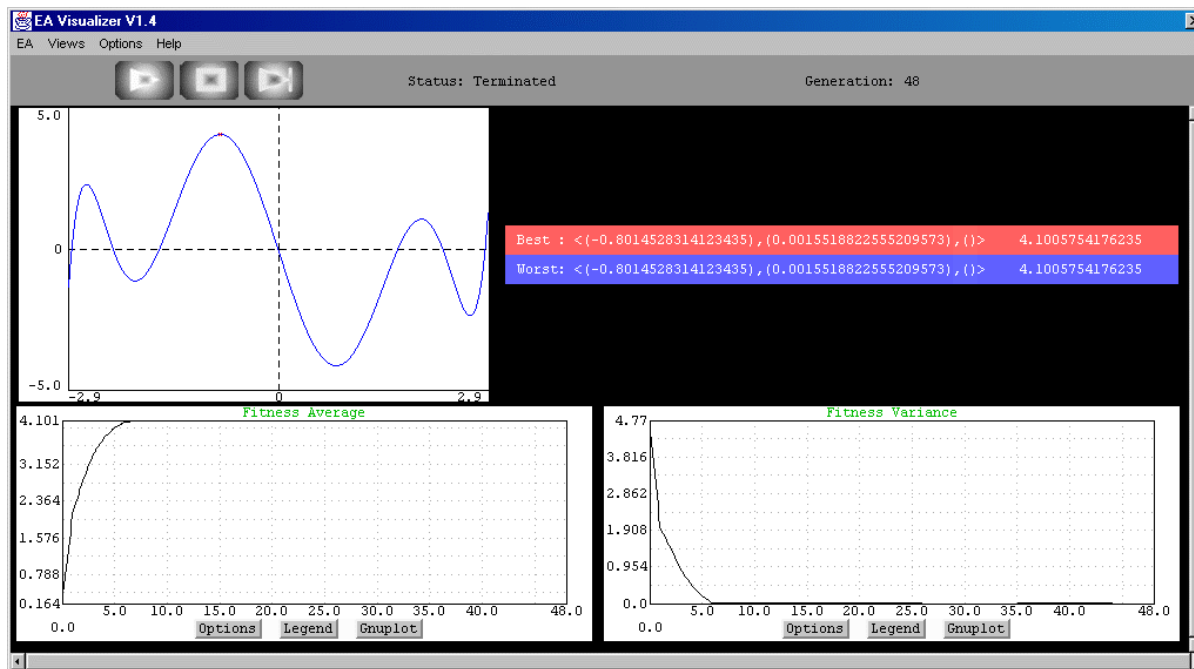


Figure 3.28: The results of running the adapted ES with tournament selection and elitist replacing.

such algorithms in a general fashion. Next to MIMIC and the improvement, at this point we wish to point out that there is also a general framework for such algorithms exhibiting a large expressional power, which goes by the name of FDA [12].

The exact contents of this approach initiated in 1996, extended in 1997 and formalized in 1998, are first set out in a detailed manner. The new approach is interesting and proves that we should not fall into a closed world assumption. Incorporating explicit statistics is actually something that the other GAs of late were already making use of. These algorithms proved to be good so it is natural to question ourselves how much better the new approach is in which the concept of explicit statistics is fundamental and the more random GA approach is disposed of. Furthermore, it is very interesting to note that *no* restrictions are posed a priori with respect to the order of building blocks in the problems that can be solved. If only for these reasons it is already interesting to investigate this approach more closely. In this tutorial, we refrain however from investigating such issues and refer the reader to other material to this end [5, 7, 12]. Here, as an advanced issue in using the *EA Visualizer* we first describe these different types of GA like approaches to give the reader a feel for the algorithms, after which we show how to install these algorithms in the *EA Visualizer*.

In theory

First we formally present the three frameworks referred to above. We shall first discuss the framework called MIMIC. The second framework is based upon MIMIC and extends it by using a less restricted way for the generation of probability distributions. We present the theory for both frameworks by giving background information as well as the actual algorithms. Finally, we briefly present a generalization of the two former methods, namely FDA.

MIMIC

The framework known as MIMIC [4] attempts to find optima by estimating probability densities. MIMIC stands for *Mutual Information Maximizing Input Clustering* which expresses the general idea of the framework: trying to find clusters in the coding structure by maximizing the linkage information between pairs of bits. Before going into details that specify how this is achieved within MIMIC, we should think about what this means. To this end we picture binary strings like in genetic algorithms. Finding linkage information between pairs of bits would then mean that we find for every bit some other bit that it is linked to according to some measure. This measure should be the implementation of something like the following expression:

When bit a is linked to bit b , this means that there is a meaningful relation specifying that when bit b is set to 1, bit a is set to 1 with a certain chance.

In more mathematical terms, this would mean that there is a conditional chance $p(a = 1|b = 1)$ that “makes sense” when we link bit a to bit b . Now since we work with *pairs* of bits, for every bit there is exactly one other bit that it will be related to when we wish to create a full probability distribution. To keep the distribution connected, we do not wish to generate subcycles and we will get some form of a tree in which each bit has at most one parent bit in the tree. In the light of what we just noted above, the child bit is then bit a and the parent bit is bit b . Note that indeed linking bits is in this way a *one-way* relation. If the relation was not *one-way* we could get direct subcycles and we would not get a connected distribution. Both MIMIC and the framework in the next paragraph work with such restricted models, which is why we introduce this notion. As we shall see, MIMIC further restricts the form of probability distribution to a specialized tree: the chain. The more recent framework in the next paragraph removes this restriction. We shall now move to more specific and mathematical terms to make precise the MIMIC framework. In the following, parts and fragments from the introductory paper for MIMIC [4] are repeated.

The joint probability distribution over a set of random variables $X = X_i$ is

$$p(X) = p(X_1|X_2 \dots X_n)p(X_2|X_3 \dots X_n) \dots p(X_{n-1}|X_n)p(X_n)$$

From a practical and engineering point of view this reads from right to left something like:

We can set bit n to 1 with a certain chance $p(X_n)$. Once we do such, we can set bit $n-1$ to 1 with a certain chance because we have been given the value for bit n and we know the conditional chance $p(X_{n-1}|X_n)$ which tells us with what chance to set bit $n-1$ given the fact that bit n is valued either 0 or 1. The remainder of the expression to the left is analogous.

Of course what the expression says is that the chance at X having some value, or to be more precise the chance at every X_i having some value a_i , is equal to the chance that X_n has value a_n multiplied by the chance that X_{n-1} has value a_{n-1} given the fact that X_n has value a_n and so on. Why this is true is explained in the following simple general example with $n = 3$:

$$\begin{aligned}
 p(X) &= p(X_1 = a_1 | X_2 = a_2 \wedge X_3 = a_3) p(X_2 = a_2 | X_3 = a_3) p(X_3 = a_3) \\
 &= p(X_1 = a_1 | X_2 = a_2 \wedge X_3 = a_3) \frac{p(X_2 = a_2 \wedge X_3 = a_3)}{p(X_3 = a_3)} p(X_3 = a_3) \\
 &= p(X_1 = a_1 | X_2 = a_2 \wedge X_3 = a_3) p(X_2 = a_2 \wedge X_3 = a_3) \\
 &= \frac{p(X_1 = a_1 \wedge X_2 = a_2 \wedge X_3 = a_3)}{p(X_2 = a_2 \wedge X_3 = a_3)} p(X_2 = a_2 \wedge X_3 = a_3) \\
 &= p(X_1 = a_1 \wedge X_2 = a_2 \wedge X_3 = a_3)
 \end{aligned}$$

The last expression in the above derivation is trivial and something we all agree on being the expression for stating the chance that each random variable has a certain value.

So what we have now is a different expression for the same thing. The only difference is that we now use *conditional* probabilities. These conditional probabilities however are exactly what we described above in the intuitive description of finding linkage between pairs of bits. The only thing is that we were merely to use *pairs* of bits. This means that we cannot have subexpressions such as $p(X_1 | X_2 \dots X_n)$ as each bit was only allowed to be linked to one other bit³. This implies that we can only create distributions of the following kind:

$$\hat{p}_\pi(X) = p(X_{i_1} | X_{i_2}) p(X_{i_2} | X_{i_3}) \dots p(X_{i_{n-1}} | X_{i_n}) p(X_{i_n})$$

In this expression π stands for a permutation of the numbers between 1 and n , $\pi = i_1 i_2 \dots i_n$. The distribution $\hat{p}_\pi(X)$ uses π as an ordering for the pairwise conditional probabilities. The goal is then to find the permutation π that maximizes the agreement between $\hat{p}_\pi(X)$ and the true distribution $p(X)$. This agreement is measured in the paper on MIMIC [4] by using the Kullback–Liebler divergence, which is a nonnegative distance measure which is zero when two distributions are identical. When written out, it becomes clear that the optimal π is the one that minimizes the following expression:

$$J_\pi(X) = h(X_{i_1} | X_{i_2}) + h(X_{i_2} | X_{i_3}) + \dots + h(X_{i_{n-1}} | X_{i_n}) + h(X_{i_n})$$

The fact that we have already chosen to use a permutation of $1 \dots n$ to approximate the true joint distribution, leads to the fact that the resulting tree of dependencies will be a chain as

³So actually we are only regarding second order statistical information.

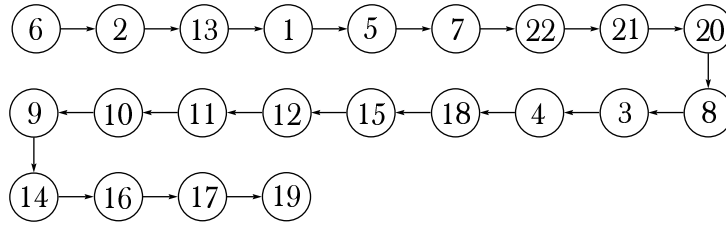


Figure 3.29: An example of the chain of dependencies generated by MIMIC.

can be seen in figure 3.29. To find the optimal permutation π , all $n!$ permutations could be searched. When n becomes even but medium large however, the amount of permutations to search is already intractably high. So in the interests of computational efficiency, the creators of MIMIC employ a straightforward greedy algorithm to pick a permutation:

1. $i_n = \arg \min_j \hat{h}(X_j)$.
2. $i_k = \arg \min_j \hat{h}(X_j | X_{i_{k+1}})$, where $j \neq i_{k+1} \dots i_n$ and $k = n - 1, n - 2, \dots, 2, 1$.

In this algorithm, $\hat{h}()$ is the empirical entropy. Given a discrete random variable X over an alphabet α , the entropy $h(X)$ is defined as follows:

$$h(X) = - \sum_{a \in \alpha} p(X = a) \log p(X = a)$$

The algorithm also uses the notion of conditional entropy $h(Y|X)$ where in the algorithm $\hat{h}(Y|X)$ is used to denote once again *empirical* entropy instead of true entropy. To work with the algorithm, we must provide the definition of conditional entropy first when we are additionally given a discrete random variable Y over an alphabet α' :

$$h(Y|X) = h(X, Y) - h(X) = - \sum_{a \in \alpha} \sum_{b \in \alpha'} p(X = a \wedge Y = b) \log p(X = a \wedge Y = b) - h(X)$$

The only factor that is now unclear is the term *empirical*. This means no more however than the fact that the information on which the chances $p(x)$ are based, is computed from the information at hand. In the case of the MIMIC framework, this means that the required chances are computed from the individuals in the population. For instance, in the first line of the algorithm, the index of the variable with the lowest entropy is determined. We have seen that to compute the entropy $h(X)$ however, we need chances $p(X = a)$ for each $a \in \alpha$. We will be working with binary strings, so our alphabet α will be none other than $\{0, 1\}$. This means that we need to have $p(X_j = 0)$ and $p(X_j = 1)$, where the X_j are the random variables that represent the bit at the j -th position in the binary string. So we

determine the empirical chance $\hat{p}(X_j = 1)$ by regarding every individual in the population, looking at the bit at position j in each such individual and counting the amount of 1 symbols we encounter. If we have counted m_1 ones at position j over the population which holds m individuals, we have $\hat{p}(X_j = 1) = \frac{m_1}{m}$ and of course $\hat{p}(X_j = 0) = 1 - \hat{p}(X_j = 1)$, since our alphabet α has size 2. The computation is analogous for finding the values for $\hat{p}(X = a \wedge Y = b)$.

To compute all these empirical probabilities, we require for each string a running time of $O(n^2)$ where n is taken to be the length of the binary strings in the population. Over the entire population this implies a running time of $O(mn^2)$, which is the running time for the greedy algorithm presented above for finding the permutation.

What has been established so far is the creation of a probability distribution. A search component is still required, as we have only seen what to do given a number of samples to update the distribution but not the population. Given that distribution however, we can generate new samples in a straightforward way as is described in the paper on MIMIC [4]:

1. Choose a value for X_{i_n} based on its empirical probability $\hat{p}(X_{i_n})$.
2. For $k = n - 1, n - 2, \dots, 2, 1$ choose element X_{i_k} based on the empirical conditional probability $\hat{p}(X_{i_k} | X_{i_{k+1}})$.

There is nothing new here anymore since we have already shown how to compute the empirical probabilities that are required. The only thing that isn't directly clear from the algorithm is something that has been clarified above already when we gave an engineering view on the joint probability distribution over a set of random variables $X = \{X_i\}$. The first step of the algorithm is clear, but the second step leaves it to the user to understand that once the value for X_{i_k} has been chosen to be a_{i_k} , the value for $X_{i_{k-1}}$ is equal to $a \in \alpha$ with chance $\hat{p}(X_{i_{k-1}} = a | X_{i_k} = a_{i_k})$ which is what we explained in words earlier. It means that to create a new sample, the string should be built up dynamically, from the right to the left with respect to the permutation π , incorporating the decisions made along the way.

The full MIMIC framework can now be specified:

1. Generate a random population of m candidates chosen uniformly from the input space. Extract the medium fitness and denote it θ_0 .
2. Update the parameters of the density estimator of $p^{\theta_i}(x)$.
3. Generate more samples from the distribution $p^{\theta_i}(x)$.
4. Set θ_{i+1} equal to the Nth percentile of the data. Retain only the points less than θ_{i+1} (in case of minimization).
5. If the values of $C(x)$ have ceased to improve, stop. Otherwise go to step 2.

Steps 2 and 3 in the algorithmic framework above are the algorithms we saw earlier in this paragraph. Concluding, we can state that MIMIC attempts optimization by subtracting second order statistics from the better part of a population, using these statistics to pairwise relate bits and then to generate more samples from the established probabilistic relations, hoping that these samples will be better than the ones seen previously as the evidence for the linkage that was used to generate the new samples was subtracted directly from the better genomes in the population.

Baluja and Davies optimal dependency trees

In 1997 Baluja and Davies came forward with an extension [3] to the MIMIC framework. The idea at the outset is the same as with MIMIC. The approach is to try to learn the structure of the search space through the usage of a density estimator based on second order statistics. At first glance the only restriction dropped with respect to MIMIC is the form of the graph that is allowed for the linkage relations. Whereas MIMIC allowed only one chain of relations to exist to express the linkage between pairs of bits, the framework of Baluja and Davies (which has not been given a name) allows for probability distributions where each random variable is linked to exactly one other variable. The restriction is thus dropped that each bit also is allowed to *be linked to* by exactly one other variable. This means that the structure that can be used to estimate the true joint probability distribution of the set of random variables as introduced in the former paragraph, is now a tree in which each node is thus allowed exactly one parent but multiple children. Any bit that is now a node in the tree is linked to its parent. Just as with MIMIC there is one bit that is not linked, which is in this case of course the root of the tree. How the structure of the tree is actually determined we shall see later on in this paragraph.

Even though the new structure for the probability distribution is the most obvious difference between MIMIC and the framework by Baluja and Davies, there are more nuances that just might make the approach fundamentally different after all. Just as with MIMIC, the framework by Baluja and Davies generates samples at each generation and just as with MIMIC the better few are chosen. However, this selection is now done from the generated samples only, whereas MIMIC retained the best samples selected from both the generated samples and the best samples seen so far. In other words, MIMIC has an elitist component that the framework by Baluja and Davies does not contain. This might just make the approach to be fundamentally different.

We shall now move again to more specific and mathematical terms to make precise the framework. In the following, parts and fragments from the introductory paper for the framework [3] are repeated.

Even though MIMIC is the one that “remembers” the better individuals from the past, the framework by Baluja and Davies does not completely “forget” them. In the new framework an estimator array is maintained that holds for every (i, j, a, b) with $i \in \{1..n\}$, $j \in \{1..n\}$, $a \in \alpha$, $b \in \alpha$ a value that is an estimate of how many recently generated “good”

bit strings have had bit i set to a and bit j set to b . To stress the influence of recently generated strings, the array is multiplied by a decay factor (a real value between 0 and 1) every generation. Just as with MIMIC, unconditional and conditional probabilities are determined empirically every generation. These values are however now computed directly from the estimator array. The array itself is updated every generation by looping over all strings in the population and by incrementing the array at (i, j, a, b) when bit i has value a and bit j has value b . Based upon the mutual information measure $I(X_i, X_j)$ between all pairs of variables X_i and X_j the tree is then generated. The mutual information measure is somewhat similar to the entropy measure used in MIMIC:

$$I(X_i, X_j) = \sum_{a,b \in \alpha} p(X_i = a \wedge X_j = b) \log \frac{p(X_i = a \wedge X_j = b)}{p(X_i = a)p(X_j = b)}$$

The mutual information measure is a measure for how closely two parameters are linked. The higher the mutual information, the greater the linkage. So in any algorithm that searches for the tree of dependencies we will want to find pairs that maximize the mutual information measure⁴.

The final part missing in the description of this new framework is the creation of the tree. The authors have used a method for finding the optimal model within the restrictions posed for the probability distributions that was presented by Chow and Liu [8] in 1968. We shall not repeat that algorithm here, as it is part of the framework and is therefore part of the algorithm found at the end of this paragraph. The results are that the tree constructed minimizes the Kullback–Liebler divergence posed in the paragraph on MIMIC that we sought out to minimize in the first place. An example of the tree structure that can now be found for the pairwise linkage between bits can be seen in figure 3.30.

The full framework by Baluja and Davies can now be specified (the following algorithm is a quote from the original paper [3] that introduced the framework):

INITIALIZATION:

For all bits i and j and all binary assignments to a and b , initialize $\mathbf{A}[X_i = a, X_j = b]$ to \mathbf{C}_{init} .

MAIN LOOP: Repeat until Termination Condition is met.

1. Generate a dependency tree

- Set the root to an arbitrary bit X_{root} .
- For all other bits X_i , set $\text{bestMatchingBitInTree}[X_i]$ to X_{root} .
- While not all bits have been added to the tree:
 - Out of all the bits not yet in the tree, pick the bit X_{add} with the maximum mutual information $I(X_{\text{add}}, \text{bestMatchingBitInTree}[X_{\text{add}}])$, using \mathbf{A} to estimate the relevant probability distributions.

⁴This is contrary to the approach in MIMIC where the goal was to *minimize* the entropy. Note that the idea is the same, only the measures differ.

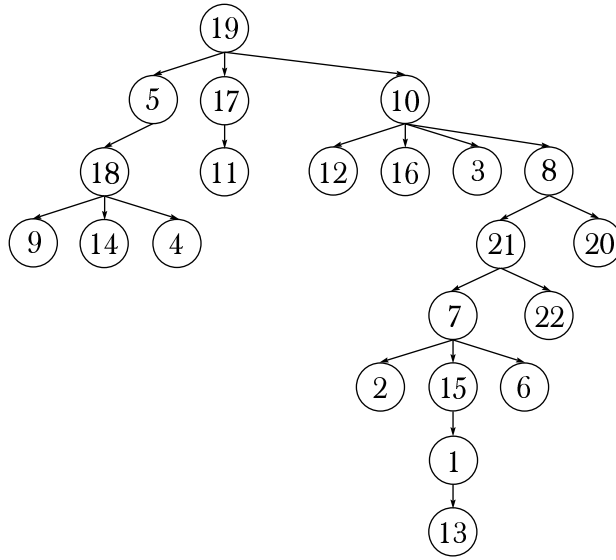


Figure 3.30: An example of the tree of dependencies generated by the framework by Baluja and Davies.

- Add X_{add} to the tree, with $\text{bestMatchingBitInTree}[X_{add}]$ as its parent.
 - For each bit X_{out} still not in the tree, compare $I(X_{out}, \text{bestMatchingBitInTree}[X_{out}])$ with $I(X_{out}, X_{add})$. If $I(X_{out}, X_{add})$ is greater, set $\text{bestMatchingBitInTree}[X_{out}] = X_{add}$.
2. Generate \mathbf{K} bit-strings based on the joint probability encoded by the dependency tree generated in the previous step. Evaluate these bit-strings.
 3. Multiply all of the entries in \mathbf{A} by a decay factor α between 0 and 1.
 4. Choose the best \mathbf{M} of the \mathbf{K} bit-strings generated in step 2. For each bit-string \mathbf{S} of these \mathbf{M} , add 1.0 to every $\mathbf{A}[X_i = a, X_j = b]$ such that \mathbf{S} has $X_i = a$ and $X_j = b$.

Concluding, we can state that the framework by Baluja and Davies attempts optimization in a way very similar to MIMIC. In fact, we could just as well call this second framework MIMIC as the difference is not all that much except for a greater expressional power in the same (limited) space of second order statistics. The fact that the two approaches are quite alike already becomes clear from the fact that the general introduction in the paragraph on MIMIC already really poses the natural result of the tree for the probability distribution. The tree could just as well have been taken as the chain, which would have caused MIMIC to be even closer to the framework by Baluja and Davies than it is now.

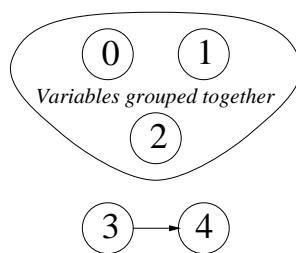


Figure 3.31: Example of factorized structure used by FDA: $p(X_0, X_1, X_2)p(X_4|X_3)p(X_3)$

FDA

Finally, we briefly present the generalizing framework called FDA [12]. In this algorithm a complete factorization of the distribution is taken as input and used to build the network structure:

$$p(X) = \left(\prod_i p\left(\prod_{X \in b_i} X \mid \prod_{X \in c_i} X \right) \right) p\left(\prod_{X \in b_0} X \right)$$

Where $c_i = s_i \cap d_{i-1}$, $b_i = s_i \setminus d_{i-1}$, $d_i = \bigcup_{j=1}^i s_j$ and the s_i make up the factorization as input sets with $\bigcup s_i = \{X_0, X_1, \dots, X_n\}$. Using an actual factorization for the problem at hand, FDA can very efficiently perform optimization. In FDA, the estimation of distributions is separated into two parts, namely *model selection* and *model fitting*. Selecting the right model regards finding the right network with the right topology. Fitting the model regards the fitting of the data at hand into the model and thus finding the values for the parameters in the network. This separation is used commonly because of the difficulty of model selection. In FDA, the model selection is thus left to the user as the sets s_i that determine the network have to be specified. Model fitting however is then done entirely and very efficiently by FDA. The model selection part is very difficult as the amount of possible networks is very large and grows exponentially with the amount of variables. It is the implications of model selection we are investigating here. In other words, if we don't have a good description of the required network, we want to know what the implications are of using some other structure for the network instead. The results on FDA tell us that on the one side given a right distribution structure, the problem can be solved efficiently. However, it also tells us that in that case algorithms such as MIMIC and the extension as described above might be too restrictive because of the type of relations allowed in the network. This is obvious since FDA can use joint probabilities directly over multiple variables such as $p(X_0, X_1, X_2)p(X_4|X_3)p(X_3)$ instead of something like $p(X_0|X_1)p(X_1|X_2)p(X_2)p(X_4|X_3)p(X_3)$ as can be seen in figure 3.31. The FDA algorithm can be specified in such a way that the former algorithms can be placed in a larger framework along with the framework for FDA. This allows the example given here within this tutorial to be more accessible. We now specify the full FDA algorithm:

1. Generate a random population of N individuals.
2. Select τN individuals with $0 < \tau < 1$.

3. Update the distribution with selected individuals.
4. Generate a new population of N individuals using the new distribution.
5. Incorporate the best individual from the previous generation (elitism).
6. If termination condition is not met, go to step 2.

The steps in which the distribution is updated and a new population is generated using the new distribution is done by using a strategy as found in MIMIC or the framework by Baluja and Davies. Using FDA implies that the distribution of variables is used to sample the probabilities using the individuals. Once the probabilities for the fixed distribution are thus established, the new samples can be generated according to this distribution. Whereas the other frameworks above adapt their internal structure for the variables in the probability density structure, FDA thus uses a fixed structure which is only updated in the sense of the probabilities themselves.

In practice

In the above we have in detail described both theoretically and to some extent operationally to clarify the methodology, three evolutionary frameworks that attempt to perform optimization through linkage between bits. The theory brings a novel approach and it differs in some ways from evolutionary algorithms seen so far. For instance, classical genetic algorithms contain recombination and mutation operators that work on subsets of two individuals from the population and single individuals respectively. Furthermore with the whole structure that admits selectors, replacers and terminators, the classical genetic algorithm fits perfectly in the structure of the *EA Visualizer*. One could even say that the *EA Visualizer* was to a large extent finetuned to work perfectly for such algorithms. How then do we incorporate a different framework such as MIMIC? Certainly it is an evolutionary framework as it uses old samples, learns from these samples and generates new samples that are again evaluated, but there is not an obvious operator such as recombination that directly swaps information between parents. However, we claimed the *EA Visualizer* to be a general framework and part of the proof is of course given in putting it to the test by implementing such alternative evolutionary frameworks as MIMIC. We shall first explain how MIMIC fits into the structure of the *EA Visualizer*. It is clear from the similarities between MIMIC and the approach by Baluja and Davies that the latter will most definitely fit as well if we can make MIMIC fit into the *EA Visualizer*. We shall therefore concentrate on MIMIC and then briefly show how the other frameworks can be constructed in the *EA Visualizer* as well.

MIMIC

To see how MIMIC can be incorporated in the *EA Visualizer* we should first reconsider

the notion of *Recombinator* the way it was introduced when we set up the *EA Visualizer*. We should not be focused on wanting to have some direct mechanism that swaps alleles between two parents for instance. This is rather tempting of course since such is the approach of many evolutionary algorithms, especially the classic ones. In general however, as we stated before, the *Recombinator* is an operator that simply receives parent genomes from the system and is required to generate offspring genomes. As such it is obvious what part of MIMIC this applies to. In MIMIC, all genomes are treated at the same time as information is extracted from the entire population. Therefore we require the *Recombinator* (which we will also call MIMIC in the *EA Visualizer*) to receive one group of parents per generation where this group is nothing less than the entire contents of the population. To ensure this, we require a new *Mater* that will mate all the parents into a single group.

We have now ensured that every generation all the genomes from the population are passed on to the MIMIC *Recombinator*. It should be clear that this new operator implements most of the algorithmic contents of the MIMIC framework. The MIMIC *Recombinator* will generate the distribution (the linkage chain, see figure 3.29), generate the samples and return these samples as the offspring. The generation of the distribution and of the samples are exactly the two algorithms we saw in the paragraph on MIMIC in section 3.4.5. This however does not complete our incorporation of MIMIC in the *EA Visualizer*. The MIMIC framework also has its own way of retaining genomes. It generates more samples and then retains the Nth percentile as is written in the paper on MIMIC [4]. This is however something we can incorporate easily as we made the *EA Visualizer* capable of handling extensive selection mechanisms. We simply take as the *Before Selector* the selector that selects all genomes (which is really hardly any selection at all) so that all parents are preserved when shipping the contents to the *Mater*. The resulting offspring from the *Recombinator* are then put through to the *Mutator* which we set to be the mutator that doesn't mutate any genomes at all. The thus resulting genomes are replaced into the population through the *Replacer* and it should be clear by now that we need to use the replacer that adds the offspring to the population so that the *After Selector* that is set to truncate the population at the Nth percentile will select the better genomes from both the old parents as well as the newly generated offspring.

The requirements for the settings of the other components is trivial, except for that of the *Terminator*. The termination condition is something that in the description of the MIMIC framework has not been made completely clear. Termination is supposed to occur “when values of $C(x)$ cease to improve”. This could mean for instance that the average fitness does no longer improve, but it could also mean that the best fitness does not seem to improve. However, for as long as the genetic material within the population is not equal for every genome, the MIMIC framework might yet induce a new probability distribution that generates even after some time of no improving fitness values, better offspring. When we look more precise, what we are saying is that as long as the probability vector does not seem to have stabilized, we could yet find better values. Still yet, even when the probability vector *has* stabilized, we could still generate offspring we haven't seen before,

unless it has stabilized with chances only set to 0 or 1. Under the assumption that the latter will happen, the *Terminator* that terminates when all genomes are equal can indeed be employed. To save ourselves from the situations in which the framework has problems with coming up with only equal genomes (for instance the probability for the last bit in the distribution is not 1 or 0 and all others are, resulting in two genomes that dominate the population), a maximum of generations is set.

The following table contains the settings that need to be employed to install the MIMIC framework in the *EA Visualizer*:

Component	Instance
<i>Genotype</i>	Binary String
<i>Similarity</i>	Any
<i>Fitness Function</i>	Any
<i>Recombinator</i>	MIMIC
<i>Mutator</i>	No Mutation
<i>Before Selector</i>	No Selection (Select All)
<i>After Selector</i>	Truncation Selection
<i>Mater</i>	Mate All Genomes In One Group
<i>Replacer</i>	Add Offspring To Population
<i>Terminator</i>	All Equal Genomes And Maximum Generations
<i>Hybrid Searcher</i>	Any
<i>Population</i>	Vector Population
<i>PRNG</i>	Any

Baluja and Davies optimal dependency trees and FDA

It should now be clear how the framework by Baluja and Davies as well as FDA can be implemented and installed in the *EA Visualizer*. In the following we shall briefly describe how this can actually be established.

Just as with MIMIC, new recombinators are generated that do most of the mathematics that were introduced in the papers [3, 12] that introduced the framework. For Baluja and Davies, the new recombinator prepares the estimate array by multiplying it by α , updates the estimate array from the samples by counting the different combinations for pairs of bits, generates the dependency tree from the estimate array and returns the newly generated samples as offspring. The selection mechanism for the framework is slightly different as the parents are not retained. This means no more than that we use as the *Replacer* the replacer that places only the offspring in the population. Just as was the case with MIMIC we decide to simply select to install the terminator that signals termination when all genomes in the population are identical or a maximum of generations is reached. The remainder of the settings remain unaltered with respect to MIMIC (Note that all that had to be added to the system is the implementation of the *Recombinator* for the new

framework, all other parts were already present in the system and could simply be selected to be part of the evolutionary algorithm when running the *EA Visualizer*):

Component	Instance
<i>Genotype</i>	Binary String
<i>Similarity</i>	Any
<i>Fitness Function</i>	Any
<i>Recombinator</i>	Baluja And Davies Optimal Dependency Trees
<i>Mutator</i>	No Mutation
<i>Before Selector</i>	No Selection (Select All)
<i>After Selector</i>	Truncation Selection
<i>Mater</i>	Mate All Genomes In One Group
<i>Replacer</i>	New Offspring Only
<i>Terminator</i>	All Equal Genomes And Maximum Generations
<i>Hybrid Searcher</i>	Any
<i>Population</i>	Vector Population
<i>PRNG</i>	Any

We conclude this subsection by presenting the settings for FDA. Once again we note that by using these settings with the different recombination operators (MIMIC, Baluja and Davies and FDA), we can perhaps in a better way compare these algorithms as we have one general selection scheme. Concluding we have the following table of settings in the *EA Visualizer*:

Component	Instance
<i>Genotype</i>	Binary String
<i>Similarity</i>	Any
<i>Fitness Function</i>	Any
<i>Recombinator</i>	FDA
<i>Mutator</i>	No Mutation
<i>Before Selector</i>	Truncation Selection
<i>After Selector</i>	No Selection (Select all)
<i>Mater</i>	Mate All Genomes In One Group
<i>Replacer</i>	Best Sample Elitist Replacing
<i>Terminator</i>	All Equal Genomes And Maximum Generations
<i>Hybrid Searcher</i>	Any
<i>Population</i>	Vector Population
<i>PRNG</i>	Any

Running the algorithms

In this section we finally get to running the algorithms. We shall however refrain from going into this with great detail as such would imply a comparative study of the different algorithms. Such is much better achievable when using multiple runs EAs, which is described in the next chapter. We show there how to directly compare these algorithms. Here we shall only run each algorithm once in the general truncation selection style as presented in the last table describing the algorithm settings in the *EA Visualizer* for FDA.

Firstly, we set out to run MIMIC on the *trap functions* fitness function. Its name is in plural as the function has a fully deceptive characteristic (so it's meant to *trap* the genetic algorithm) for certain values for the parameters of the function. For other values of the parameters, the function can be easy. Since there are thus many flavors to this function, it's really a collection of functions, which is the reason for the plural aspect in the name of the function.

The function works on binary strings that are split up in building blocks of equal size. Each of these building blocks contributes a fitness values itself, separate from the other values. For every block the fitness can be computed in the same way (when uniformly scaled). When we set n to be the length of a building block, m the amount of building blocks in one string (thus the length of one binary string is mn), $o(x)$ the amount of ones in a given string x , d the so-called signal (a real value between 0 and 1), the fitness evaluation of a single building block x is the following:

$$f(x) = \begin{cases} -\frac{1-d}{n-1}o(x) + 1 - d & \text{if } o(x) < n \\ 1 & \text{if } o(x) = n \end{cases}$$

It follows that the maximum fitness value is equal to m , which occurs when all bits are set to '1' and thus all m building blocks have a fitness contribution of 1. The fitness function for a single building block is thus dependent on the amount of '1' symbols in the input. The fitness evaluation function for a single building block is depicted in figure 3.32.

Running MIMIC in the truncation selection framework presented above means that we create a new single run EA with the instances for the components as noted in the last table presented above, with the exception of the *Recombinator*, which should of course be set to *MIMIC*. Furthermore, the fitness function is to be set to the **Binary String - Trap Functions** instance. We set out to run the framework on strings of 50 bits in length and set the trap functions parameters to 10 blocks of length 5 with a signal of 0.2. We want to set $\tau = 0.5$ for no apparent reason, only just to try some value. This means we tell the before selector instance to select 50 percent and to use the percentage as selection ratio. We employ a population of 200 members. Finally, we must specify the parameter values for the parameters for MIMIC. We set out to generate 200 new samples as this will be the new population and thus its size must equal the current population size. For scientific research, the distribution MIMIC uses can be fixed, but we shall refrain from that here, so we leave that parameter unspecified. Having thus created a new algorithm, we set out

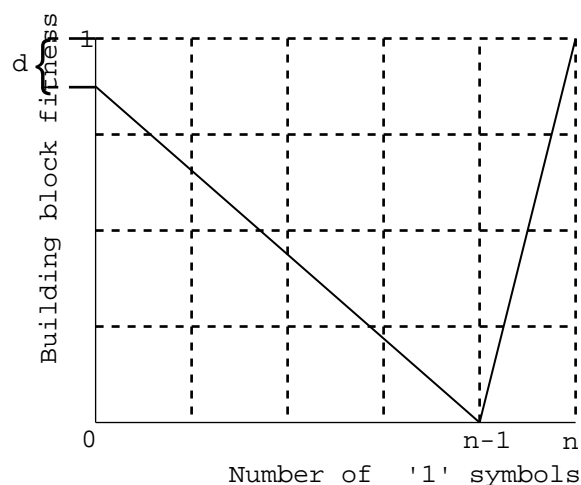


Figure 3.32: The fitness contribution function for a single building block.

to run it using the *EA Visualizer* by pressing the *Create EA* button and subsequently add some views. The views we wish to add are a population dots view with a width of 50 pixels and a height of 200 pixels to keep the bits square, a best and worst binary string view, a schema tracer that plots # In Schemata for the schema with 5 leading 1 symbols and 45 trailing don't care symbols and a trap functions building blocks correct view that tells us the percentile of building blocks that have been found correct (all 1 symbols) over all building blocks in the population. Subsequently, we run the algorithm by pressing the *play* button. The typical results for a single run with MIMIC are displayed in figure 3.33. As can be seen when experimenting, MIMIC is quickly led to the complete suboptimum, which is a string of only 0 symbols.

Next, we wish to test the framework by Baluja and Davies. To be more precise, we want to test how different the topology of the network is and what implications this has on the optimization power of this different framework. To this end, we should note that the framework by Baluja and Davies differs in that it uses an estimator array. The reader can check however that when the initial value c_{init} and the decay factor α are set to 0, the estimator array is disabled, meaning the algorithm only uses statistics from the *current* population just as in MIMIC. So, just by pressing F5 or by selecting to edit the current single runs EA from the EA menu in the *EA Visualizer* and only selecting a new *Recombinator*, namely **Baluja and Davies Optimal Dependency Trees**, we can directly test the adapted second approach once we set the decay factor and the initial value both to 0. Doing so, the typical results for a single run with this adapted second approach are displayed in figure 3.34. As can be seen when experimenting here, this adapted second framework is also quickly led to the complete suboptimum.

Finally, we wish to test the completely factorized approach found in FDA. The expectations are of course that this last approach will yield very good results as we can specify the perfect distribution on beforehand. First we must find out what this distribution is. The best distribution is of course one where the individual building blocks are acknowledged as being individual and completely separate blocks and will be processed as such. Furthermore, the bits in one such building block should be processed as a whole and not in some chain structure. This implies that the best probability density structure for the problem is:

$$p(X) = \prod_{i=0}^9 p(X_{5i}, X_{5i+1}, \dots, X_{5i+4})$$

Once again, we can just by press F5 or select to edit the current single runs EA from the EA menu in the *EA Visualizer* after running the tests with the adapted framework by Baluja and Davies. After doing so, we only have to select a new *Recombinator*, namely FDA. By specifying the distribution to be the above, which can be done by specifying the sets s_i to be $s_i = \{5i, 5i + 1, \dots, 5i + 4\}$ for $i \in \{0, 1, \dots, 9\}$. These sets are completely disjoint. In the *EA Visualizer*, these settings can be entered as the distribution for FDA by typing as one string:

```
(0 1 2 3 4)(5 6 7 8 9)(10 11 12 13 14)(15 16 17 18 19)(20 21 22 23 24)(25 26 27 28 29)
(30 31 32 33 34)(35 36 37 38 39)(40 41 42 43 44)(45 46 47 48 49)
```

Having done so, we can test FDA with this distribution once we apply the settings and reset the current EA by pressing F6 or by selecting to do so from the EA menu. Running the algorithm, the typical results for a single run with FDA are displayed in figure 3.34. As can be seen when experimenting, this time we have an algorithm that is *not* led to the complete suboptimum, but finds the optimum instead virtually every time the algorithm is run. The typical results for a single run with FDA is displayed in figure 3.35.

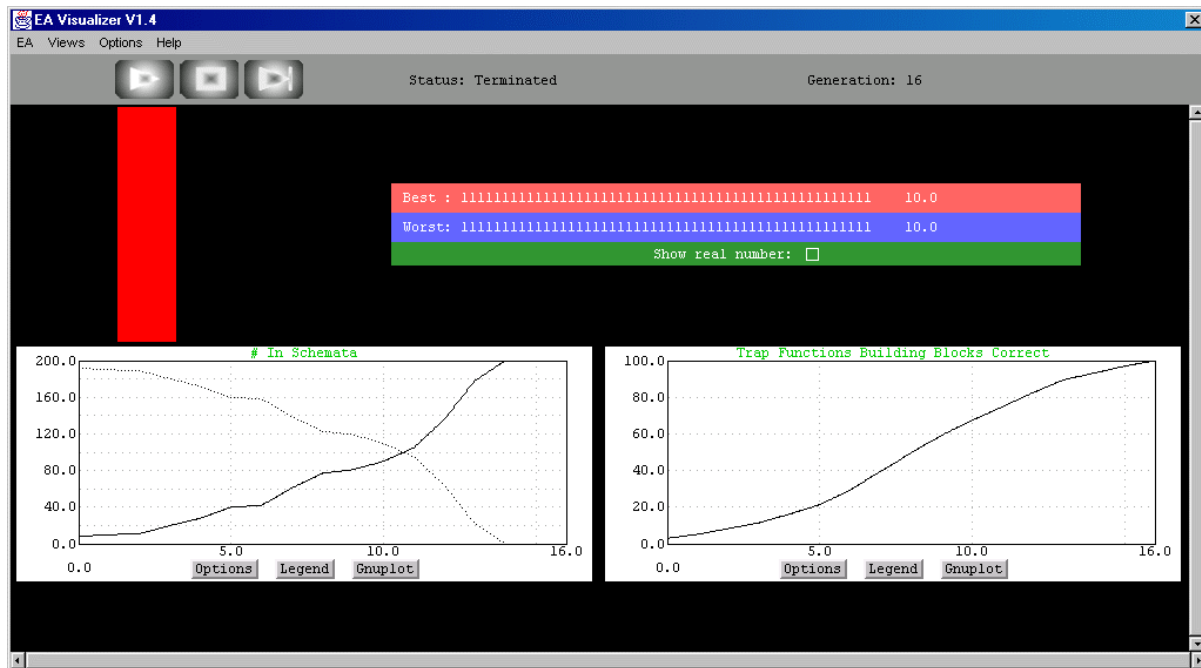


Figure 3.35: The typical results of running FDA a single time on the trapfunctions.

Chapter 4

Multiple Runs

Just as was the case for single run evolutionary algorithms, we describe through a system description and moreover through examples how to use multiple runs EAs in the *EA Visualizer*. First, we describe why multiple runs are required and what their goal in use are. Second, we set out to give a description of the interfaces you will encounter when creating a new multiple runs EA. Third and finally, we give various examples that will guide you in a practical way through the system and show you how to use it. On beforehand we point out again however we do not run these algorithms to prove anything, but only to demonstrate the usage and capabilities of the *EA Visualizer*.

4.1 Batch testing by expanding to multiple runs: how and why?

Before describing how to use multiple runs EAs in the *EA Visualizer* system, it is convenient to first note why multiple runs in extension to the single runs are required and how they can be useful. In section 3.1 we already pointed out why in general multiple runs are required. Running multiple runs that run a single EA a multiple of times and then visualize averaged results is required to be able to make scientific remarks and for instance be confident about the stability of or theories for an evolutionary algorithm.

The system must therefore be capable of running EAs a multiple of times. In the general setting of the *EA Visualizer* however, facilitating this neatly is complex. At first glance, this might not seem to be the case because making a self coded GA run iterated is no more difficult than adding another loop to the code. This is however required for each type of test anew whereas a general model will prevent this. Furthermore, constructing a general approach helps in finding solutions to whatever loops required when coding them directly. Also, coding it in general also prevents the user from having to once again and after all

make facilities for visualization of the results after a multiple of runs. Again, when using the *EA Visualizer*, this is not required. However, this does mean we have to allow for such *general* multiple runs facilitations to exist in the system.

The simplest approach for the general case is to allow the user to create an amount of single run algorithms that are consecutively run a specified amount of times. This is far from desirable as it happens all too quickly that we wish to test all population sizes between 2 and 200 with a step of 4, which would require the definition of 49 EAs. Furthermore, we have then not even varied other settings as well.

A much better approach is to allow for the varying of settings. The parameter values must then be variable so that the influence of a parameter can be inspected. In order to test different strategies, we must be able to test different instances for a single component. We must however not forget that we have *dependency imposing* components. This implies that we cannot test a multiple of instances for those components in a single specification. Otherwise we might get inconsistencies within the dependencies that are imposed. Also, we might want to enumerate sets of parameters or instances simultaneously (for instance amount of genomes to select and population size). This means that we wish to test only simultaneously iterated combinations of values for two parameters instead of individually iterated combinations of values. Thus we need to be able to avoid crossproducts between settings.

To be able to install a multiple of instances for the dependency imposing components, we can allow for a *multiple of multiple runs* to be specified. Each of these multiple runs can be run a specified amount of times for each combination of parameter values and instances. In each such multiple run, the instances for the dependency imposing components are fixed, but their parameters are variable. The instances for the other components can be selected with respect to the dependencies now imposed. For each instance, a multiple of parameter values can be entered. For one such multiple runs the links can be specified, indicating what sets of parameters or instances must be enumerated simultaneously. This is nontrivial, as not all combinations are allowed to be linked as can be seen in figure 4.2. The structure for instances and parameters are shown in figure 4.1. It is obvious that the complexity of multiple runs is far beyond that of single runs.

4.2 The multiple runs settings interfaces

By selecting **New Multiple Runs EA** or by pressing F2 on the keyboard, an interface is displayed in which the settings can be entered for a multiple runs evolutionary algorithm. To be more precise, this interface will allow the user to specify a multiple of multiple runs evolutionary algorithms. As noted before in section 4.1, this is required because of the dependency imposing components that might present problems when making combinations. In practice, it hardly ever occurs that you will want to have a multiple of multiple runs EAs.

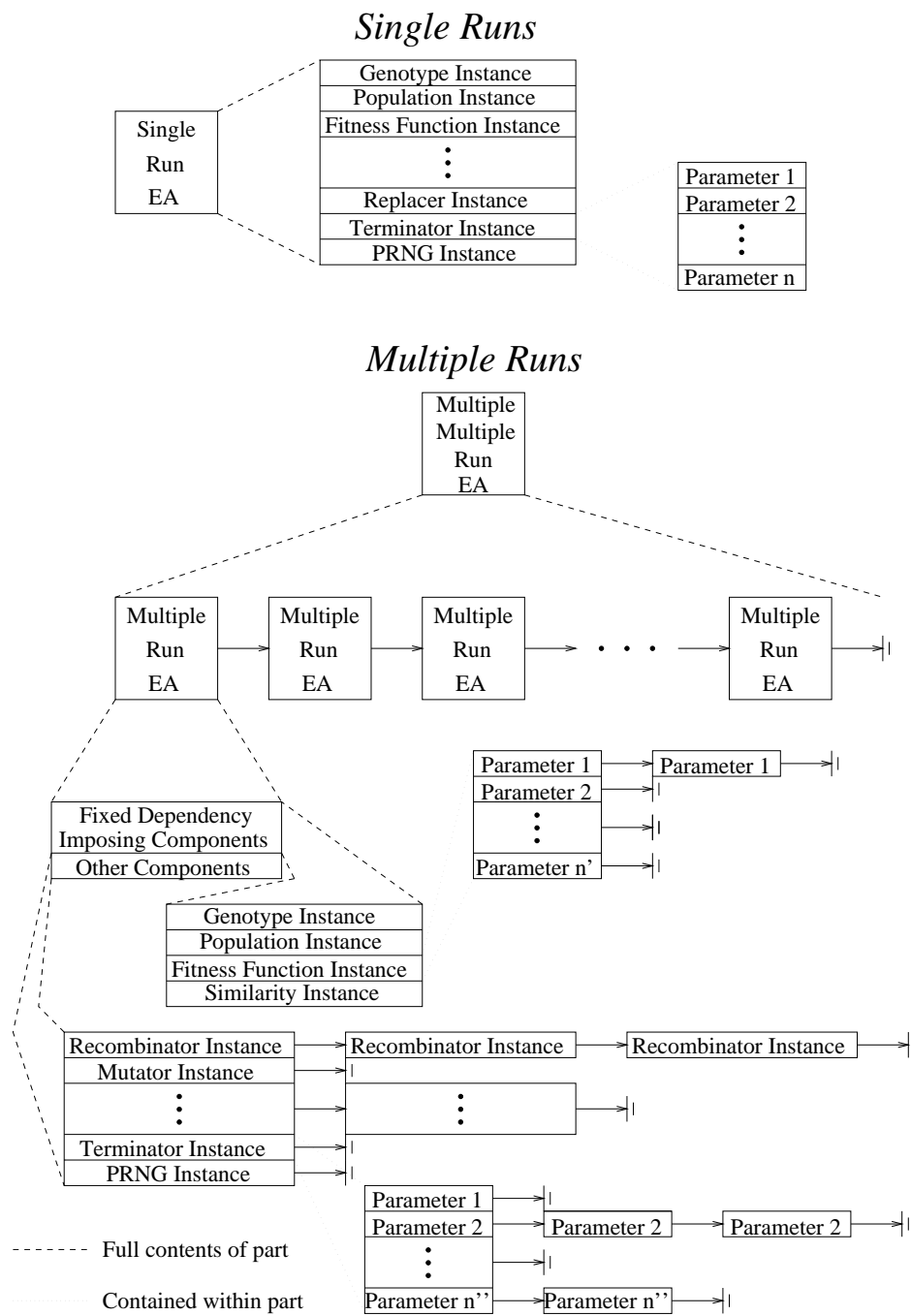
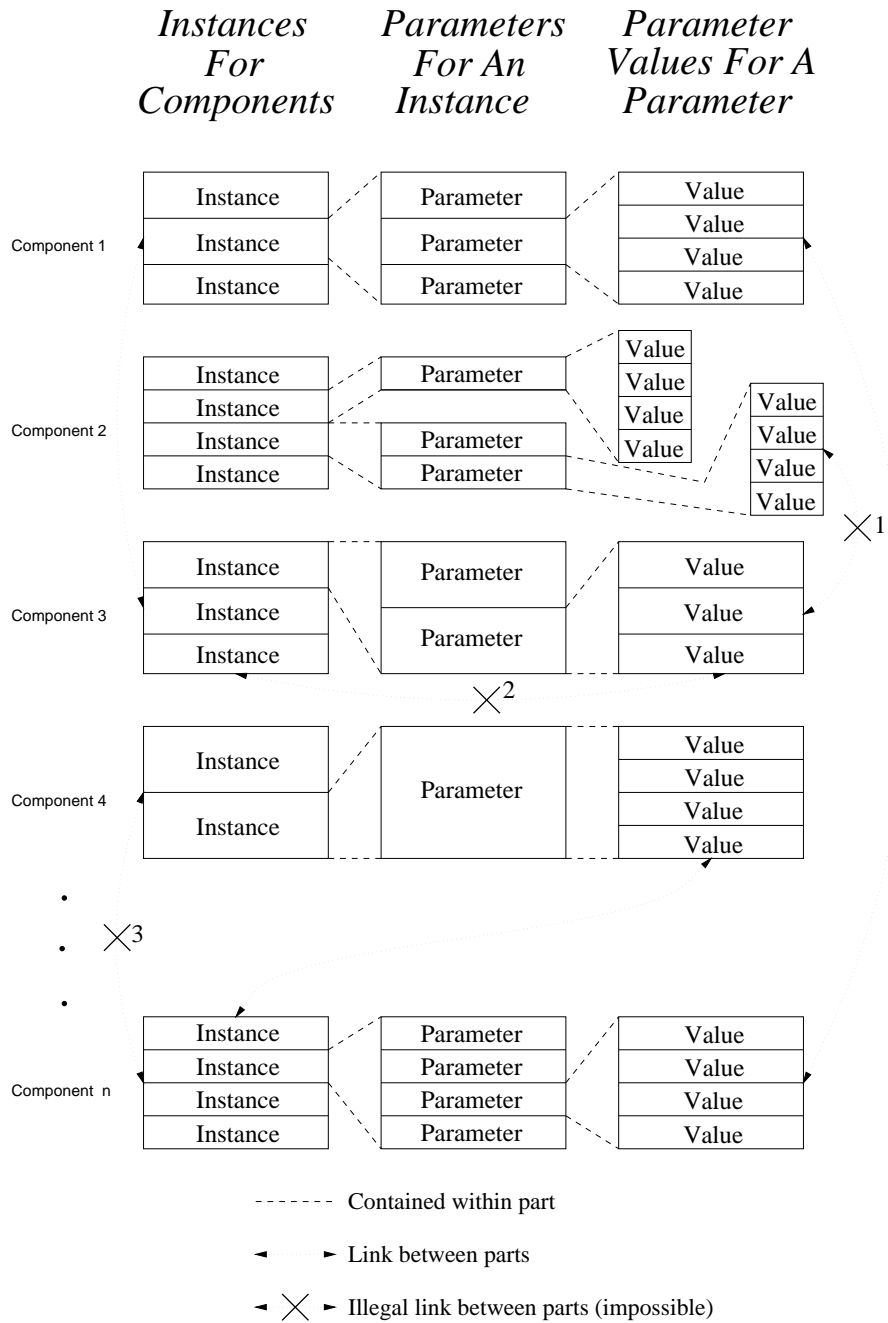


Figure 4.1: Structure of single and multiple runs with example instances and parameters.



Illegal links:

- 1: Wrong arity, not the same amount of values to enumerate
- 2: Parameters values for one instance with all instances of the same component
- 3: Wrong arity, not the same amount of instances to enumerate

Figure 4.2: The structures to be enumerated and an example of links between them.

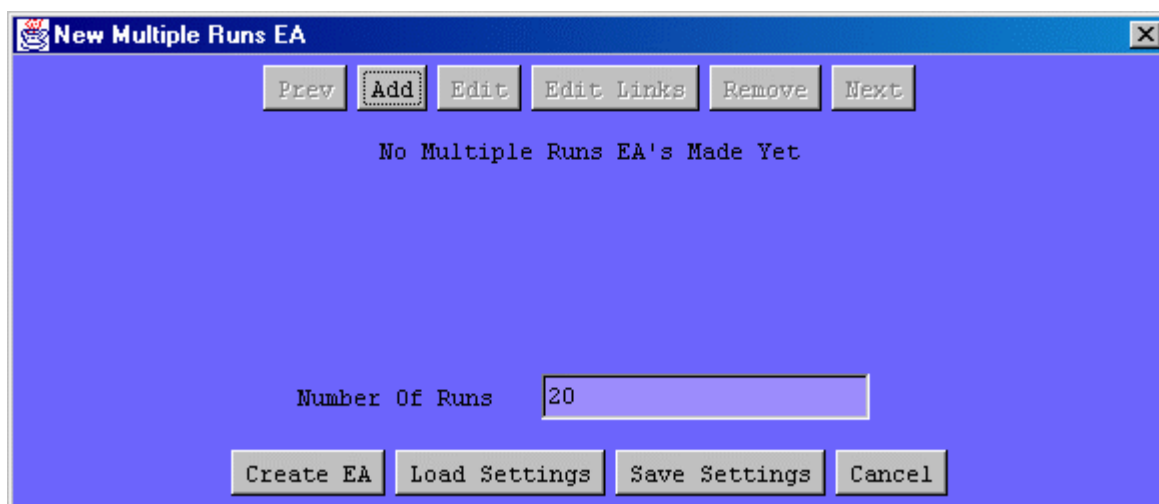


Figure 4.3: The GUI for multiple multiple runs EA settings.

Mostly, one multiple runs EA with one set of dependency imposing components will suffice, but this might not always be the case. In any way, in the first interface, which is depicted in figure 4.3, the settings are made for multiple multiple runs evolutionary algorithms. One multiple runs evolutionary algorithm consists of possibly multiple selections for instances of components that are part of an evolutionary algorithm and possibly multiple parameters for these instances. For each of these multiple runs evolutionary algorithms however, the *Dependency Imposing* components are fixed at the outset. In order to vary the instances of these *Dependency Imposing* components as well, a full multiple runs evolutionary algorithm is therefore actually a multiple of multiple runs evolutionary algorithms.

The interface displays information on all the multiple runs evolutionary algorithms that have been selected. The multiple of multiple runs evolutionary algorithms are stored in a list. If any multiple runs evolutionary algorithm is currently available, the instances of the *Dependency Imposing* components are written out in the center of the interface. In the following, we give a description of the buttons that are located at the top in this interface.

- *Prev.* Goes to the previous multiple runs evolutionary algorithm in the list.
- *Add.* Adds a new multiple runs evolutionary algorithm by first opening a multiple runs EA dependency imposing components interface, followed by a multiple runs EA settings interface.
- *Edit.* Edits the current multiple runs evolutionary algorithm by opening a multiple runs EA settings interface.
- *Edit Links.* Edits the links between the selected instances and the parameters for the current multiple runs EA by opening a multiple runs EA links interface.

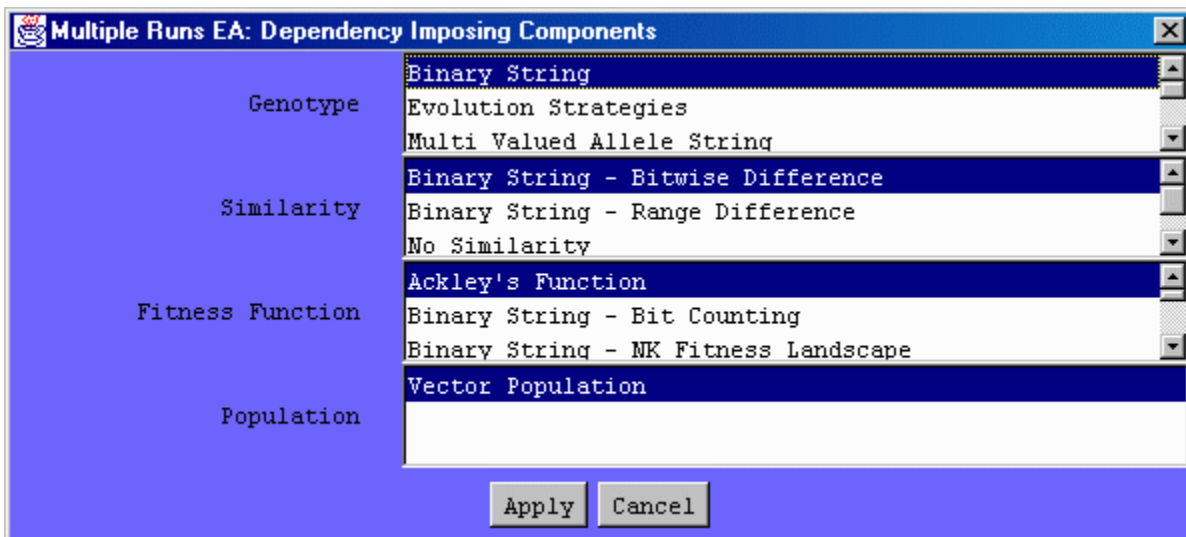


Figure 4.4: The GUI for multiple runs EA dependency imposing components.

- *Remove*. Removes the current multiple runs evolutionary algorithm from the list.
- *Next*. Goes to the next multiple runs evolutionary algorithm in the list.

At the bottom of the interface, a textfield is located as well as two more buttons. In the textfield, the amount of times the current multiple runs evolutionary algorithm is to be run with one specific combination of settings is stated. Once all multiple runs evolutionary algorithms have been filled in, the *Create EA* or *Apply* button can be pressed to dismiss the interface. The name of the button is dependent on whether the user is editing the settings for a *new* or for the *current* evolutionary algorithm respectively. Pressing the *Cancel* button discards any changes and dismisses the interface as well.

Selecting to add a new multiple runs EA first takes you to a multiple runs EA dependency imposing components interface as depicted in figure 4.4. In this interface, the instances for the *Dependency Imposing* components that are to be used in one multiple run evolutionary algorithm are selected. This means that you are to select an instance for each component. These instances are placed in the lists at the center of the interface. The parameters for the instances are not edited here, this is done in the multiple runs EA settings interface.

Having selected which *Dependency Imposing* components will be used for the multiple runs EA, the *Apply* button can be pressed, which dismisses the interface and opens a multiple runs EA settings interface as can be seen in figure 4.5. In this interface, all settings except for the instances for the *Dependency Imposing* components are made for one multiple runs evolutionary algorithm. This means for each component of the general framework for an EA other than the *Dependency imposing* components, at least one instance has to be selected. These instances are placed in the lists at the center of the interface.

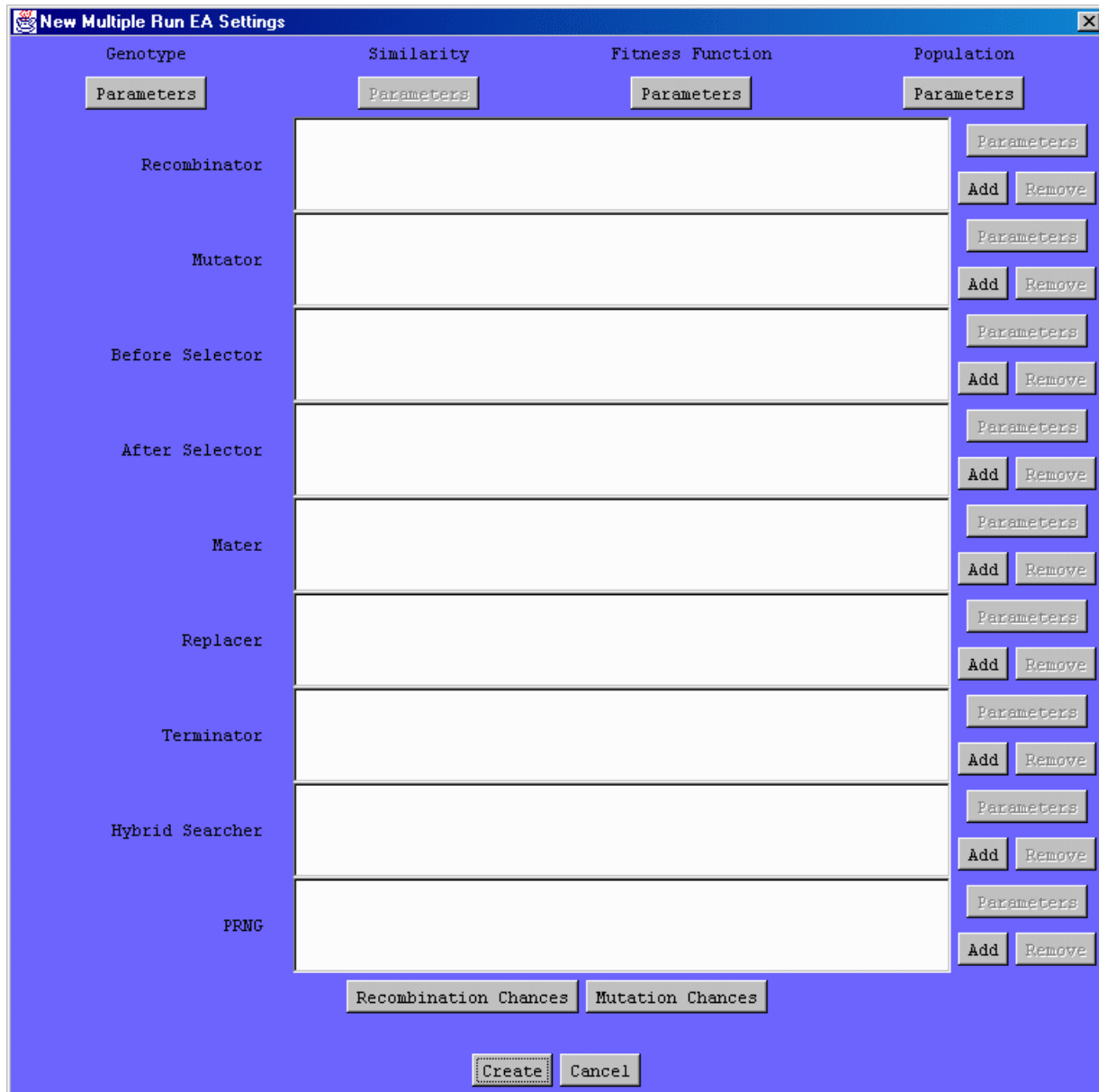


Figure 4.5: The GUI for multiple runs EA settings.

Whenever a selected instance has parameters, the *Parameters* button on the right side of the list belonging to that selected instance is enabled. By pressing this button, an interface is displayed with the parameter components in which to fill in the parameter values for the selected instance (possibly a multiple of values).

As this interface facilitates the selection of the instances for the components for a multiple runs evolutionary algorithm, a multiple of instances can be selected. Whenever the *Add* button besides a component list is pressed, an interface is displayed with the selectable instances for the component. An instance for a component can be removed by pressing the *Remove* button besides a component list.

The parameters for the *Dependency Imposing* components can be specified by pressing the *Parameters* buttons at the top of the interface. Furthermore, to specify the recombination and mutation chances, the buttons at the bottom of the interface can be pressed. This will trigger an interface to be displayed in which the information can be entered. If your screen resolution is not high enough to host all of the components, the lists are divided over two pages and the recombination and mutation chances buttons are located at the bottom of the second page.

Once for all components at least one instance has been added and all the parameters are filled in, the *Create* or *Apply* button can be pressed to dismiss the interface. The name of the button is dependent on whether the user is editing the settings for a *new* or for the *current* evolutionary algorithm respectively. Pressing the *Cancel* button discards any changes and dismisses the interface as well.

Lastly, links can be specified for one multiple runs EA when required. The parameters for these links can be entered when the *Links* button is pressed in the multiple multiple runs EA settings interface as depicted in figure 4.3. This will trigger a multiple runs EA links interface to appear as depicted in figure 4.6. When creating a new multiple runs evolutionary algorithm or editing a currently active one, this interface presents the user with the option to specify links between the different settings so that they not be subjected to a cartesian product in settings when enumerating them. The links that may be created can be very diverse. The links that can be specified can only concern settings that have the same arity (amount of instances). A list of links of possibly a different arity can be defined.

No links can be defined within one component from a higher to a lower level. This means that the instances of a component cannot be linked to any values for a parameter of one of those instances.

The interface contains four lists. The *Components* list contains the components except for the *Dependency Imposing* components that make up the evolutionary process. By selecting one of them, the textarea on its right side shows the instances for that component that have been added for this multiple runs evolutionary algorithm. By adding this component to a link by pressing the *Add Link* button below this list, the enumeration of the instances

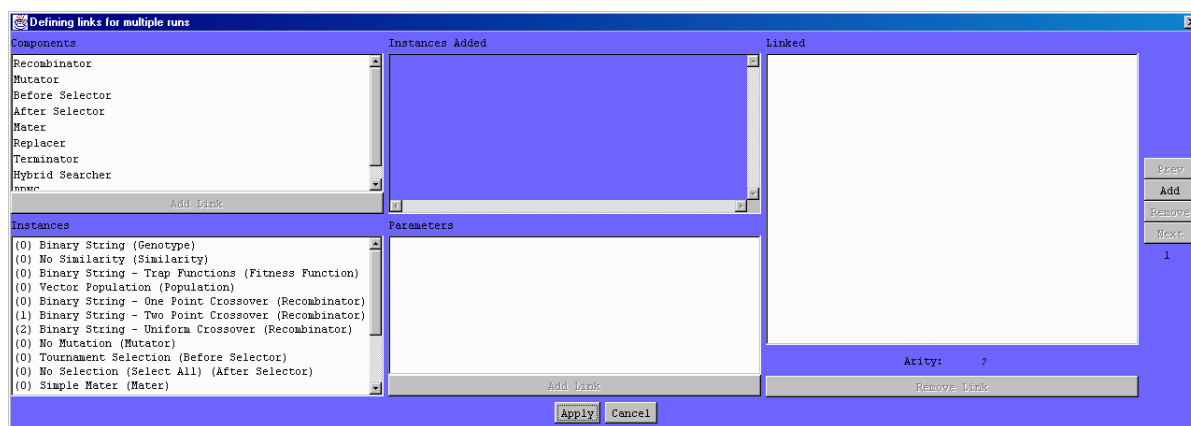


Figure 4.6: The GUI for multiple runs EA links.

for the added component is done together with the enumeration of the other items that are already in the link.

The *Instances* list contains all of the instances that have been selected for all of the components that make up the evolutionary process. By selecting one of them, the list on its right side will contain the parameters that belong to that instance (if the amount of values set for them matches the arity of the current link). By selecting one of these parameters from that list (the *Parameters*) list and by then pressing the *Add Link* button under the *Parameters* list, the values that have been specified for this parameter of the particular instance for a component will be enumerated together with the enumeration of the other items that are already in the link.

Multiple links of different arities can be defined. The contents of the links are displayed in the list on the right of the interface, the *Linked* list. Below that list, the arity of the link is displayed. By pressing the *Remove Link* button below the list, the linked item that is currently selected in the list is discarded of and the information is put back in the other lists. As the user is allowed to create multiple links of different arities, a mechanism should be supplied to handle all the links within that list. This is taken care of by the buttons located on the right side of the *Linked* list. By pressing the *Prev* button, the previous link in the list is displayed, by pressing the *Next* button, the next link in the list is displayed. In order to add a new link, the *Add* button can be pressed and in order to remove the full contents of a link as well as the link itself, the *Remove* button can be pressed. Below these buttons, a number is shown indicating the how manyth link is currently being edited.

By pressing the *Apply* button, the links are accepted and the interface is dismissed. By pressing the *Cancel* button, any changes are discarded and also the interface is dismissed. An example of a link you might want to specify is the size of the *Population* together with the selection size of the *Before Selector*. This means that instead of selecting 10 out of 10, 10 out of 20, 10 out of 30, 20 out of 10, 20 out of 20, 20 out of 30, 30 out of 10, 30 out

of 20 and 30 out of 30 genomes in *nine* different evolutionary algorithms, 10 out of 10, 20 out of 20 and 30 out of 30 genomes are selected in only *three* evolutionary algorithms if the sizes for both parameters for the instances are 10, 20 and 30.

4.3 Examples

In this section we present various examples just as we did for the single runs EA. The examples are once again chosen so that there is a variety in applications, showing the diversity of the *EA Visualizer* system. The examples range from simple to more difficult, starting off with something simple. The examples presented in this section are sometimes described in less detail than in others, but nevertheless the reader is guided in the use of the *EA Visualizer*, especially when the examples get more involved.

4.3.1 Various recombination operators for the TSP

In a first example you are once again taken through the *EA Visualizer* step by step and are shown through what steps a multiple runs EA can be created, run and visualized just as was done in section 3.4.1. In the next section, more examples will be given over a variety of applications. However, in this section the process of establishing such an EA visualization is described in full detail, whereas the subsequent examples are presented more briefly. We once again visit the two dimensional TSP, but this time we inspect various recombination operators for it at the same time in such a way that we can compare them for a given problem.

The main idea is to take one problem instance and to run a GA with different types of TSP recombination strategies. We also have a 2-opt hybrid operator we could investigate, but in this comparative example we only wish to investigate the different recombination operators. The *EA Visualizer* is as a standard equipped with the following recombination operators for the TSP:

Operator	# Parents	# Offspring
Cycle crossover	2	2
Distance preserving crossover	2	2
Edge map recombination	2	1
Order crossover	2	2
Partially mapped crossover	2	2

Note that the different recombination operators do not all share the same input–output relations with respect to amount of genomes. All operators namely produce two offspring

from two parents with the exception of the edge map recombination operator, which produces only a single offspring genome. Therefore, once we start comparing the recombination operators, we should first think about how to set up a selection framework that gives the operators equal opportunity. Using the same amount of parents gives a problem that the edge map recombination operator will only provide half the amount of offspring, meaning we should then add the offspring to the population to apply selection afterwards so that in all cases the population will contain the same amount of genomes each generational step. However, the selection pressure is then higher for the edge map recombination operator, which is not fair in comparing the recombination operators only. Thus the amount of parents needs to be twice as great, but by doubling the population size we again give the edge map recombination operator an unfair advantage. Concluding, we must set the before selector to select twice as many genomes when edge map recombination is installed. In such a case, the population size remains the same for all operators and only in the case of edge map recombination more parents are selected but this is only done so as to find more offspring. Moreover, taking the selection size equal to the population size (with the exception of when the edge map recombination operator is installed), keeps the population size at all times the same when we only take the new offspring as the new population and gives no bias with respect to the selection strategy either. This is thus our approach in this multiple runs example.

Firstly, we request to create a new multiple runs EA by pressing the F2 key or by selecting to create a new multiple runs EA from the EA menu. Firstly, we select to average the results over 20 runs by entering this value at the bottom of the interface that is showing (as already shown in figure 4.3). Secondly, we then select to *Add* a new multiple runs EA by pressing that button. Next, we then select to set the *Genotype* to the TSP Numbered List, the *Similarity* to No Similarity, the *Fitness Function* to Geometric TSP Numbered List and the *Population* to Vector Population as can be seen in figure 4.7.

After pressing *Apply*, we are to enter the parameters and the instances for the remainder of the components in the multiple run settings interface we saw before in figure 4.5. Firstly, we enter the parameters for the fitness function by entering the data of the `ulysses22` dataset from the TSPLIB library¹. These coordinates are the following:

```
(38.24, 20.42)(39.57, 26.15)(40.56, 25.32)(36.26, 23.12)(33.48, 10.54)(37.56, 12.19)
(38.42, 13.11)(37.52, 20.44)(41.23, 9.10)(41.17, 13.05)(36.08, -5.21)(38.47, 15.13)
(38.15, 15.35)(37.51, 15.17)(35.49, 14.32)(39.36, 19.56)(38.09, 24.36)(36.09, 23.00)
(40.44, 13.57)(40.33, 14.15)(40.37, 14.23)(37.57, 22.56)
```

Having entered these in the textfield above the canvas that defines the cities and by pressing enter, the cities are displayed on the canvas as the cities that will be used in the TSP as

¹The coordinates that should actually be used for comparing benchmarks with the TSPLIB dataset should be converted through polar coordinates, but we shall refrain from that here.

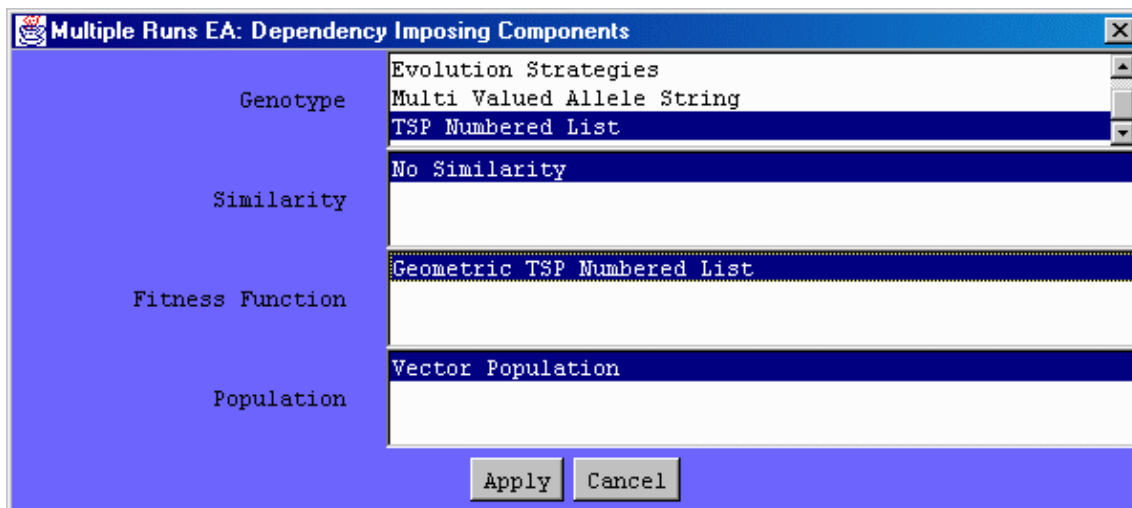


Figure 4.7: Selecting instances for the dependency imposing components.

can be seen in figure 4.8. Secondly, we specify the population parameters by selecting the population size to be anything from 50 to 1000 with steps of 50 as depicted in figure 4.9.

Next, we select what instances to install for the *Recombinator*, which is the basis of our example. To be more precise, we thus install all recombinators we mentioned above. Specifically, we install the edge map recombinator last because we are to install different parameters for other components as well according to this specific recombinator instance. Therefore, for convenience, we install this recombinator last. None of these recombinators has parameters, so the selection of instances is enough for the *Recombinator* component. As the *Mutator* we take the *No Mutation* operator that doesn't alter any tour it receives.

Going to the *Before Selector*, we want to use tournament selection as is done mostly in GAs, but as noted above we must take care in selecting the amount of genomes. We noted that for the edge map recombination operator, we require to select twice as many parents. This means that we must have just as many tournament selection instances as we have recombinators so that the recombinators can be linked to the *Before Selectors* so that each recombinator has its own *Before Selector* where the last *Before Selector* is of course configured to select twice as many parents. In other words, we first add four tournament selection operators as a *Before Selector* and configure their parameters to have a selection size of 50 to 1000 with a stepsize of 50 and a tournament size of 2 at all times as is depicted in figure 4.10. Next, we add a fifth tournament selection operator for which we specify to select twice as many genomes, meaning 100 to 2000 with a stepsize of 100 as can be seen in figure 4.11.

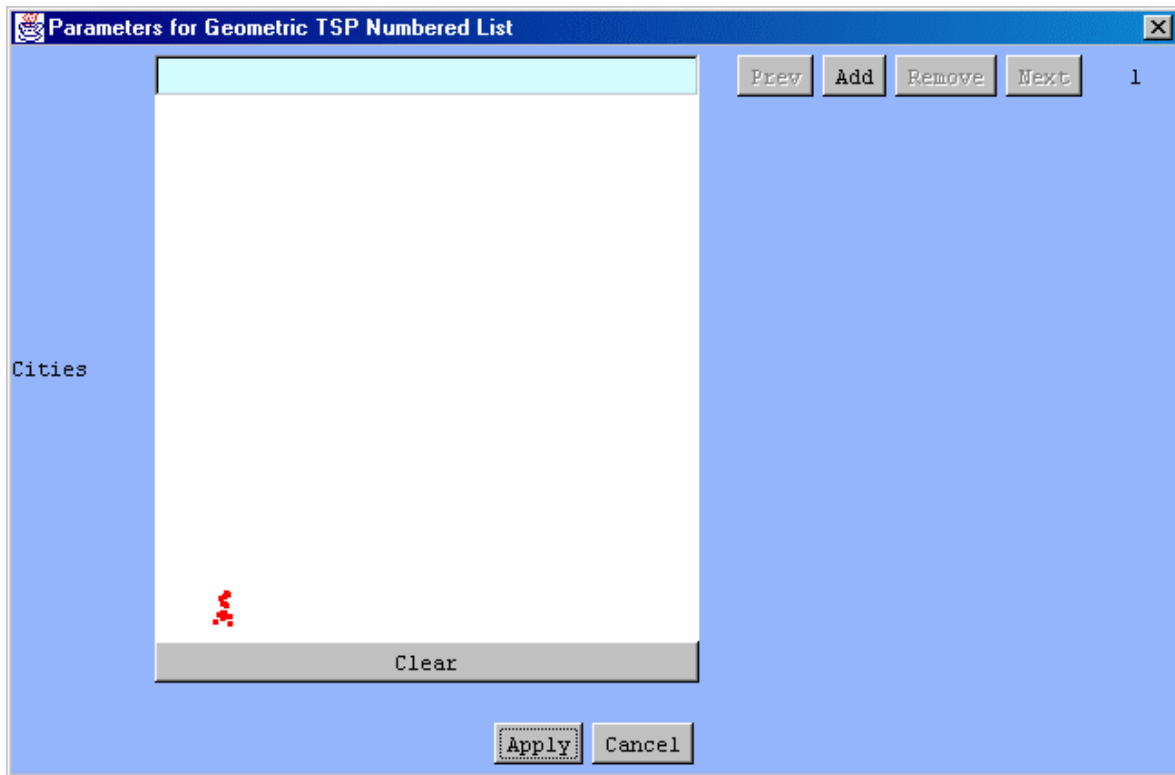


Figure 4.8: Entering the cities that define the TSP problem instance.

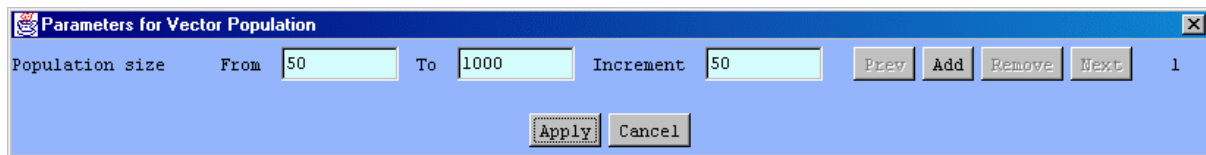


Figure 4.9: Entering the sizes for the population.

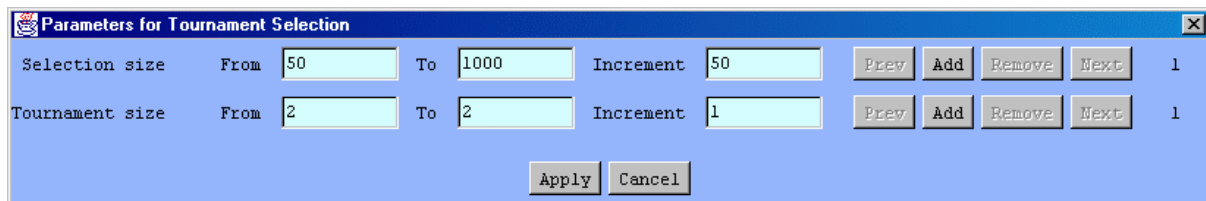


Figure 4.10: Entering the parameters for tournament selection for the first four recombination operators.

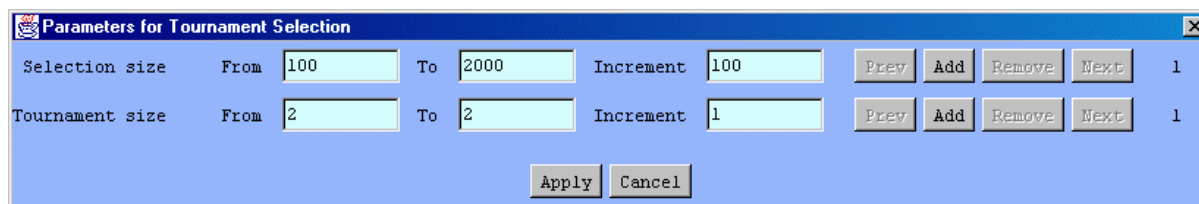


Figure 4.11: Entering the parameters for tournament selection for the edge map recombination operator.

As in a standard GA, we employ no selection afterwards and thus select to have **No Selection (Select All)** as the only instance for the after *Selector*. As the *Mater*, we can suffice in taking a **Simple Mater** at all times that mates the population from top to bottom in groups of two. As the tournament selection operator always shuffles the population, we require no **Random Mater**². The parameters for this *Mater* are that the grouping size must be set to two, which will of course give sets of two parents for the recombination operators. As the *Replacer* we install once again to achieve in some sense a classical GA the **New Offspring Only Replacer**. This will cause the newly generated offspring to make up the population of the next generation. For the operators *Edge Map Recombination*, *Distance Preserving Crossover* and *Order Crossover* however we wish to employ **Elitist Replacing** as well as the *Replacer*. Installing this replacer in the cases of those operators is done because of the way the operators maintain diversity after some generations. This will cause the population to not completely converge, resulting in termination only after so many generations without any better results. Employing both tournament selection on beforehand and elitist replacing afterwards is a very strict and disruptive selection mechanism, but for the sake of this example, we shall not allow ourselves to be bothered by this. We set the only required *Terminator* to be the **All Equal Genomes And Maximum Generations** instance with 1000 generations as a parameter for the maximum amount of generations. As said before, we employ **No Hybrid Search** as the *Hybrid Searcher*. Finally, we take the **Standard Java PRNG** as the random number generator and set to use a random seed chosen by the system. To finish the settings for the multiple runs EA, we set the recombination chances to always be 1 and the mutation chances to always be 0. Of course, we could choose any other chance for the mutation probability, since the installed *Mutator* does not have any effect on the genomes. In any way, we are finished with selecting the desired instances and their parameter values. The resulting interface is depicted in figure 4.12.

Having entered the settings for the multiple runs example, we must define the links that constrain the way in which the combination of parameters and instances just set are enumerated. To specify these links, we press the *Edit Links* button in the multiple multiple

²It actually is no difference in the running time for the *Mater*, so you might use a **Random Mater** instead at all times.

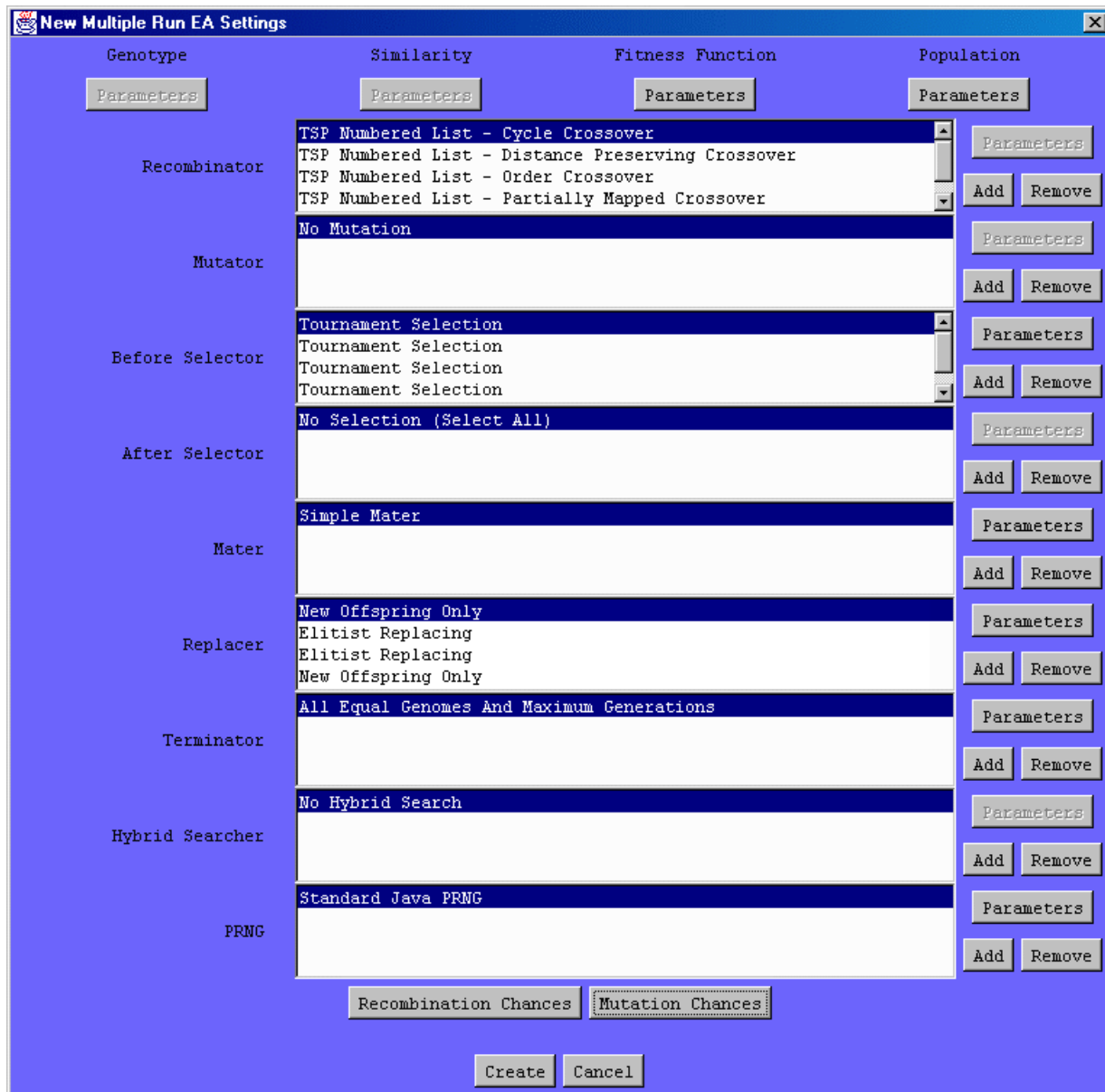


Figure 4.12: The resulting GUI for the multiple runs settings.

runs interface. First of all, we must make sure that the instances for the recombination operators are linked to the instances of the *Before Selector* as well as the instances of the replacing mechanism. This is because as we stated before the edge map recombination operator requires to select more genomes on beforehand and for some recombination operators we wish to use the elitist replacing operator and for others not. This implies that for each recombination operator we have specified a *Before Selector* and replacement operator, resulting in five recombination operators as well as *Before Selectors* and replacement operators. Now however we must tell the system that these instances need to be enumerated simultaneously because we want recombination operator i to be installed with *Before Selector* i and replacement operator i instead of respectively instances i , j and k with $i \neq j \neq k$. Specifying this is done by linking the instances for the mentioned components. This can be done by selecting first the *Recombinator* component from the topleft list in the interface for editing links. By doing so, the textarea on the right of this list displays the selected instances for the recombination operator. At this point, the button underneath the topleft list should be pressed to actually add the instances to the link. After doing so, the rightmost list that holds the link information shows that the arity of the link is now equal to the amount of instances (which is 5) that were selected for the recombination operator. Furthermore, the rightmost list now contains an entry that specifies that the instances of the *Recombinator* are now part of the link. Finally, the list of components at the topleft is now filtered to only still contain the *Before Selector* and the *Replacer*. This is because for all the other components we didn't select to have five instances, so we can't link them to something that has an arity of five of course. Furthermore, the instances for the *Recombinator* were just added, so this is also no longer included in the topleft list. Subsequently, the remaining entries in the topleft list should be clicked and added to the link by pressing the button underneath that leftmost list just as was just done for the *Recombinator* component. This results in the linking of the instances for the three components as was desired. The resulting interface after this first step in defining the links is shown in figure 4.13.

Next to linking the instances of the components, we need to link something else as well. We have selected to have an increasing population size. Along with a certain population size however, we require a certain selection size to keep the amount of genomes in the population equal each generation. This means that the increasing amount of genomes that are to be selected as we defined earlier, must match (or be twice as much in case of the edge map recombination operator) the amount of genomes in the population. This means that the parameter values for the population must be linked to the parameter values for the *Before Selector*. We have however selected to install five instances for the *Before Selector* because the instance that is to be selected when the edge map recombination operator is to be installed, must select twice as many genomes as the population size has. This means that the population size values must be linked to *each* range of parameter values of the instances for the *Before Selector*. In order to do this, we must first define a new link, which can be done by pressing the button *Add* in the list of buttons on the right of the rightmost list. Next, the *Vector Population* should be selected from the bottomleft list. By doing

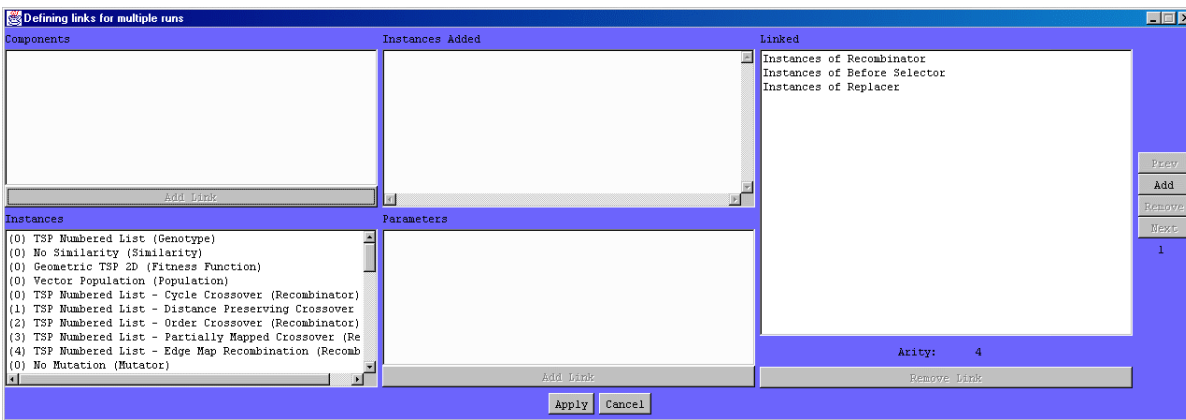


Figure 4.13: The resulting interface after defining the first link for the instances.

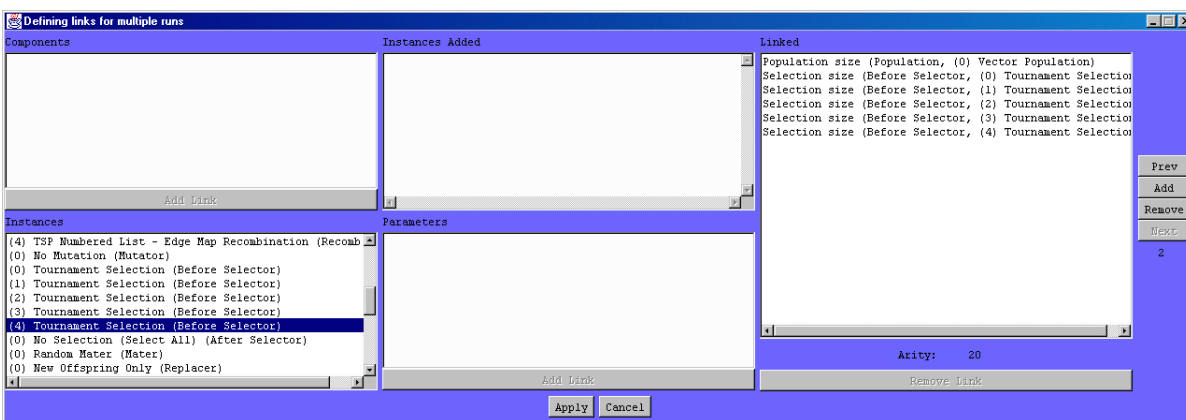


Figure 4.14: The resulting interface after defining the second link for the instances.

so, the list on the right displays the parameters for the selected population instance. By selecting the population size and by subsequently pressing the button beneath the list that holds the parameters, the values for the population size are added to the new link. Now we are required to add the selection sizes for the selection operators. This can be done by subsequently selecting the different instances for the *Before Selector* in the bottomleft list, selecting the selection size parameter in the list on its right and pressing the button underneath that list to add the parameter values to the link. The link on the right will show the added items to the link as they are added. After having added the selection size for each of the five instances for the *Before Selector*, we have finished specifying the links. The resulting interface after the second and final step is shown in figure 4.14.

After having specified the multiple runs EA through all of the interfaces just described, the system is ready to run. We must however first of course add views in order to see anything as the system runs the evolutionary algorithms. After this is done, the system is ready to

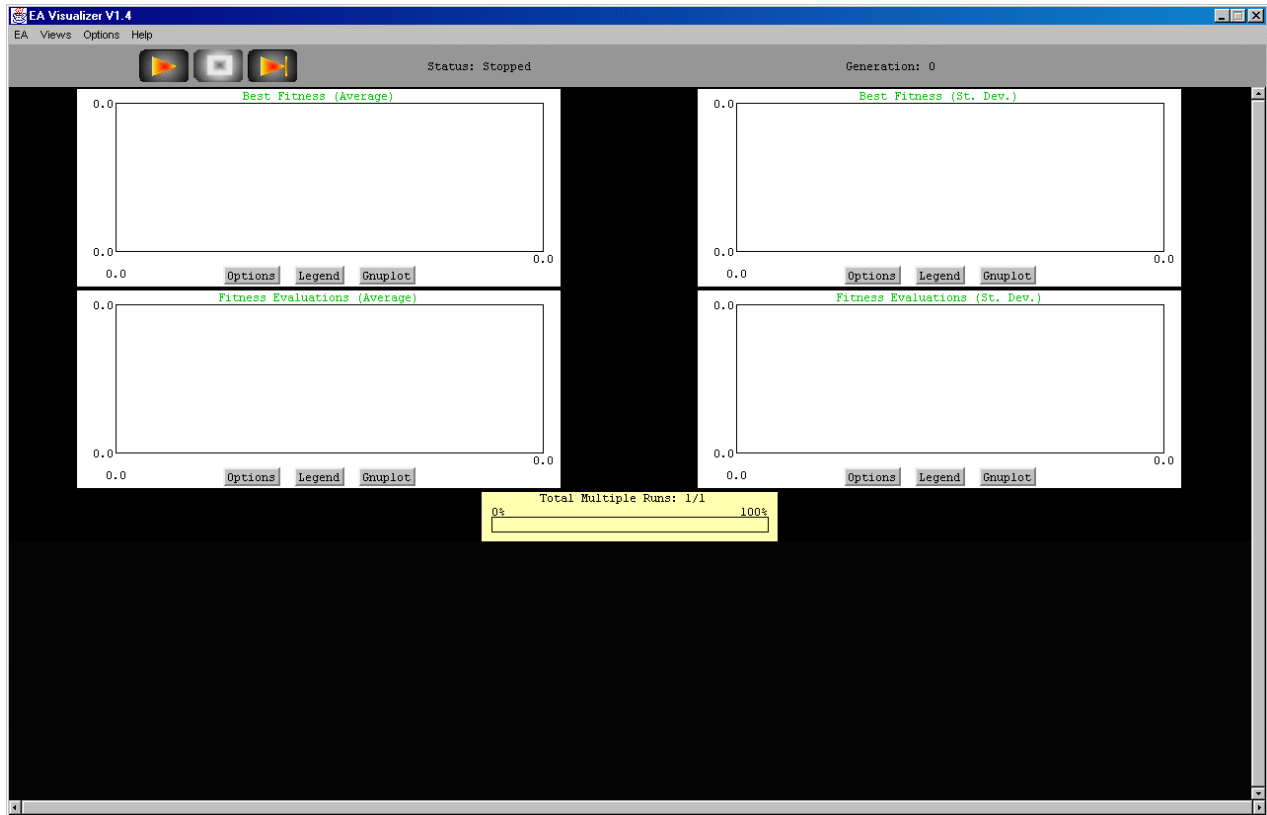


Figure 4.15: The *EA Visualizer*, ready to run after setting up the multiple runs EA.

run and show you the results you requested. We shall add some general statistics views and set their titles. Having done so, the system will look like the image in figure 4.15.

What we wish to view is some statistics concerning the best individual upon termination. As all genomes are taken to be equal upon termination since this is the exact termination condition, the best genome fitness is equal to the average genome fitness. The statistics we wish to view are the average, standard deviation and the best value over all of these best values that result from the individual evolutionary algorithms. Next to these fitness statistics that tell us something about the search performance in terms of finding quality solutions, we want to have some means of knowing how long it took the algorithms. To this end, we also want to know how many times the fitness function was called upon to evaluate a genome. To this end, we require two views, both for which we display two instances on screen. The views that we require are the **Multiple Run Fitness Statistics** and the **Multiple Run Fitness Evaluations**. In both cases, we will require to specify a so called *environment* that will identify the different evolutionary algorithms when making a legend. We shall first present the idea of the environment, after which we will present the two views we shall use. Finally, we shall turn back to our example by noting what instances we shall actually use to display the results and by presenting the results.

Differentiating between different settings in a multiple runs EA

When using a multiple runs view to retrieve information for different combinations in both instances of components and parameters for these instances of evolutionary algorithms, the values that result from the process have to be grouped in some way. This is done through the usage of an *environment*. With the help of an environment, it can be specified when two runs in a multiple runs EA should be seen as belonging to the same evolutionary algorithm (e.g. classes of evolutionary algorithms).

This is required because the *EA Visualizer* as we noted before runs every combination that the user has specified through multiple runs settings and links runs EAs a multiple of times. It is clear that for each of such runs, the results belong to the same evolutionary algorithm. However, when for instance making a graph that shows results with at the base (x) axis the population size, a change in population size does not mean that we have a different evolutionary algorithm. The EA is the same, but the population size has only changed. This means that the population size parameter should be neglected when determining whether two evolutionary algorithms are taken to be the same.

This *Parameter Component* consists out of three lists named *Components*, *Parameters* and *Neglected*. Without putting anything into the environment, everything that might be different in the settings (be they instances of components (e.g. one-point or two-point crossover) or parameter settings (e.g. select 50 or 100 genomes)), causes a new entry to appear in the environment. When for instance we wish not to distinguish between the different instances of the recombination operator, we can add the *Recombinator* component to the *Neglected* list by pressing the *Add* button under the *Components* list. When we decide not to differentiate between selection sizes, we add the parameter *Selection Size* of the selection mechanism from the *Parameters* list to the list of neglected items. This last example will cause an evolutionary algorithm which is used a multiple amount of times with a different selection size to be seen as one evolutionary algorithm, which is convenient when we choose the selection size to change at the same rate as the population size. The image in figure 4.16 shows an example in which the appearance of this parameter component is shown.

Multiple Run Fitness Statistics

The *Multiple Runs Statistics View* can be used to view a variety of statistical information regarding the fitness values of the genomes as a function of the population size. This view is created for a multiple runs evolutionary algorithm. The information that can be generated therefore has a wider variety and a potential higher complexity than the single runs version. When adding the view, the user is prompted for the type of statistical information that is to be displayed. This setting is divided over two parts. The first part (*Statistic*) concerns what values to collect after each termination of a run. The second part (*Over All Runs*) concerns what to do with each collection of values after all runs for one combination have been finished. The following options are available for the *Statistic* part:

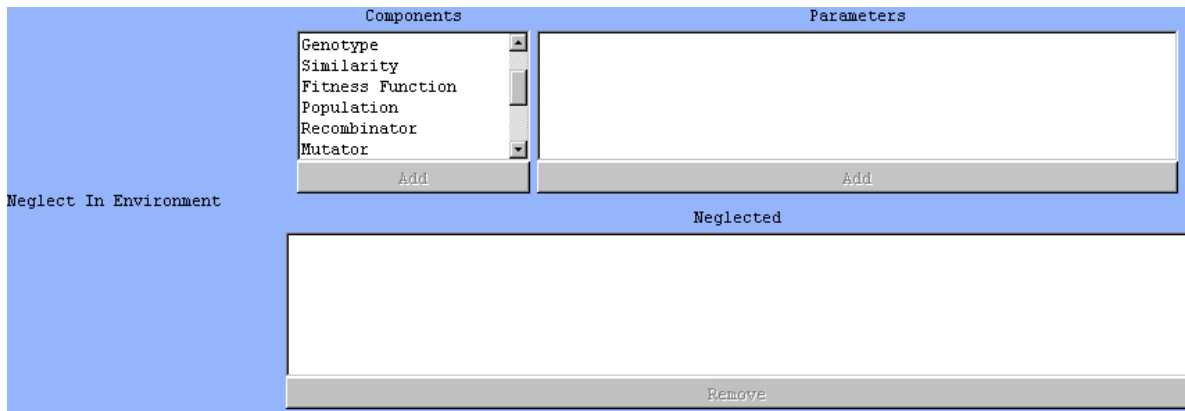


Figure 4.16: An *environment* to distinguish between evolutionary algorithms.

- *Fitness Average*. The average fitness value over all genomes in the population.
- *Fitness St. Dev.*. The standard deviation of the fitness values over all genomes in the population.
- *Best Fitness*. The fitness value of the genome with the best (with respect to the *Fitness Function*) fitness value over all genomes in the population.
- *Worst Fitness*. The fitness value of the genome with the worst (with respect to the *Fitness Function*) fitness value over all genomes in the population.

The following options are available for the *Over All Runs* part:

- *Averaged*. The average value over all values at the end of the runs.
- *St. Dev.*. The standard deviation value over all values at the end of the runs.
- *Best*. The best (with respect to the *Fitness Function*) value of over all values at the end of the runs.
- *Worst*. The worst (with respect to the *Fitness Function*) value of over all values at the end of the runs.

As this view is a multiple runs view that displays information for different combinations in both instances of components and parameters for these instances and places this information as a function of the population size, the values that result from the process have to be grouped in some way. This is done through the usage of the *environment* we just mentioned. This is the last parameter that has to be filled in when using a *Multiple Runs Statistics View*. The statistical information that is retrieved is based upon the states of the

EA at the termination of a run. When the amount of runs is reached that the EA is supposed to be run with the same parameters, the resulting statistics are displayed. Finally, this view is an internal view that displays the values in a graph. As such the information will be displayed in the *view space* of the *EA Visualizer*.

Multiple Run Fitness Evaluations

The *Multiple Runs Fitness Evaluations View* can be used to view a variety of statistical information regarding the amount of fitness evaluations as a function of the population size. This view is created for a multiple runs evolutionary algorithm. The information that can be generated therefore has also in this case a wider variety and a potential higher complexity than the single runs version. When adding the view, the user is here also prompted for the type of statistical information that is to be displayed. The statistic to select determines what to do with the collection of fitness evaluations numbers after all runs for one combination have been finished. The following options are available:

- *Average*. The average function evaluations numbers.
- *St. Dev.*. The standard deviation of the function evaluations numbers.
- *Most*. The most function evaluations required over the total of runs.
- *Least*. The least function evaluations required over the total of runs.

The results again have to be grouped as this is also a multiple runs view. Again this is done by using the environment we presented above. This is the last parameter that has to be filled in. The statistical information that is retrieved is based upon the state of the EA at the termination of a run, just as is the case for the statistics view. When the amount of runs is reached that the EA is supposed to be run with the same parameters, the resulting statistics are displayed.

Running the tests and gathering the results

Before getting the system in the state as shown in figure 4.15, we must add the views that we just mentioned. The first two views are the statistics graphs we just mentioned and the next two are the fitness evaluations graphs. To add the views, we first select (after selecting to add views to the system) to add two MULTIPLE RUN STATISTICS VIEWS. For both views, the *Statistic* is set to *Best Fitness* in accordance with what we wanted to see. The *Over All Runs* parameter is set for the two views respectively to *Averaged* and *St. Dev.*. The environment is the most tricky part, but by now it should be quite clear that what we have to neglect in the environment are the population size and the selection sizes for the tournament selection mechanisms that are installed as the *Before Selector* instances. This is because the population size is what the views will use along the *x* axis, so it is clear

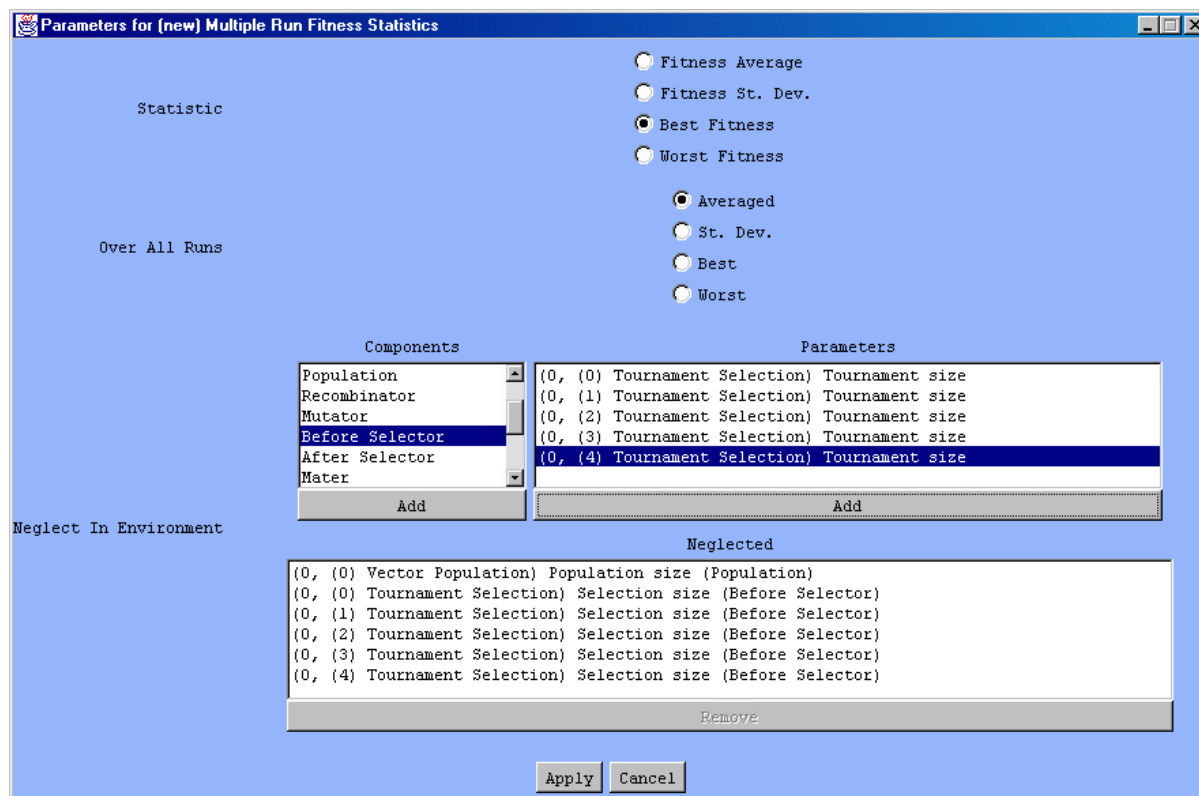


Figure 4.17: The parameters for the first Multiple Run Statistics View.

that this parameter should be neglected. Next to that, the selection sizes are linked to the population size, so they will always be altered when the population size is altered, meaning they must also not indicate that two EAs are different. Concluding, the parameters for the first view we add are shown in figure 4.17.

After adding in a similar manner (with all the same parameters for the environment parameter component) the other statistics views and the fitness evaluations views, we alter the titles for the views by pressing the *Options* button for every graph view and by altering the title in the interface that pops up. Having altered all the titles, we obtain the system that is ready to run the multiple runs as we saw depicted before in figure 4.15. The final results after pressing the *play* button in the main interface when the system is ready to run (figure 4.15) are graphs that are displayed in the view space of the *EA Visualizer*. Each graph can however be exported to GNU PLOT data as we saw and explained earlier in section 2.3. We use this professional graph plotting program to present the final results of this example. How to use the datafiles that result from exporting to GNU PLOT we also already saw in section 2.3 (the next section shows in more detail how to work with the GNU PLOT data). As we altered the titles of the graphs on beforehand as shown in figure 4.15, the resulting GNU PLOT files have different filenames and do not overwrite each other. The resulting GNU PLOT graphs are shown in figures 4.18 through to 4.21.

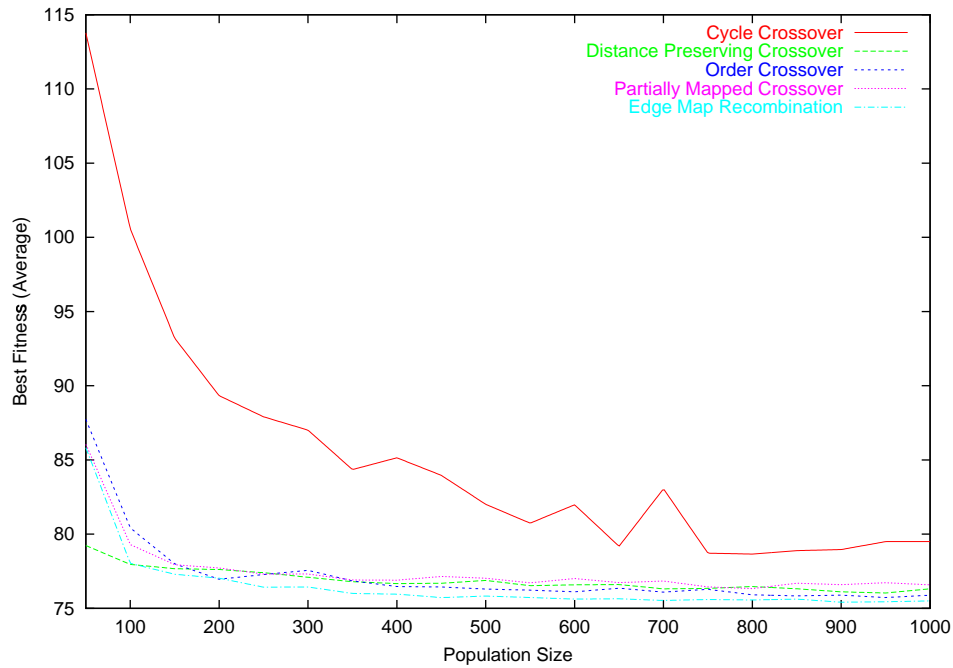


Figure 4.18: The average best fitness value over 20 runs.

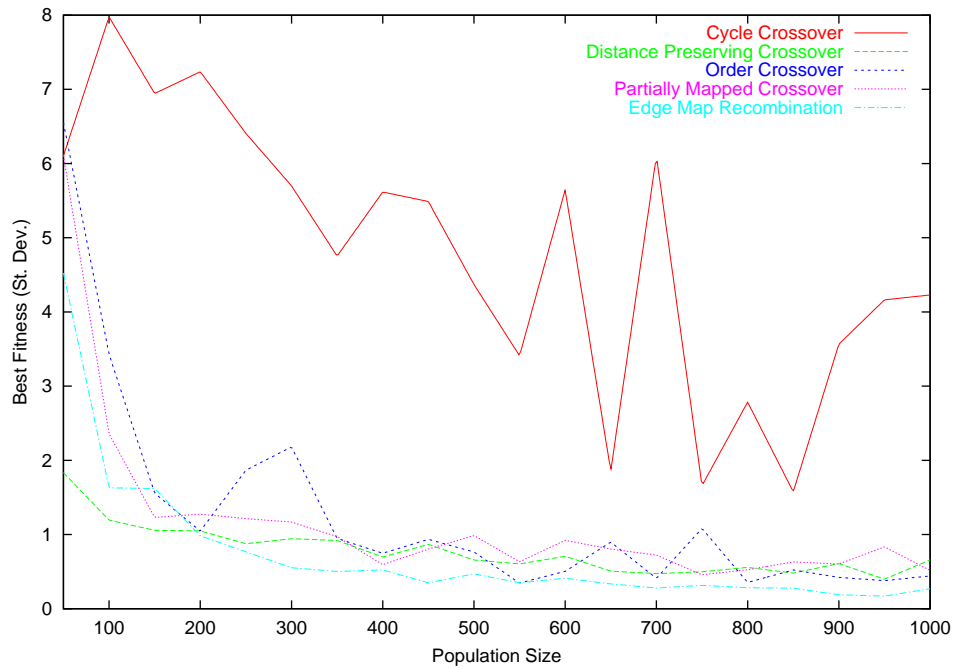


Figure 4.19: The standard deviation over the best fitness values over 20 runs.

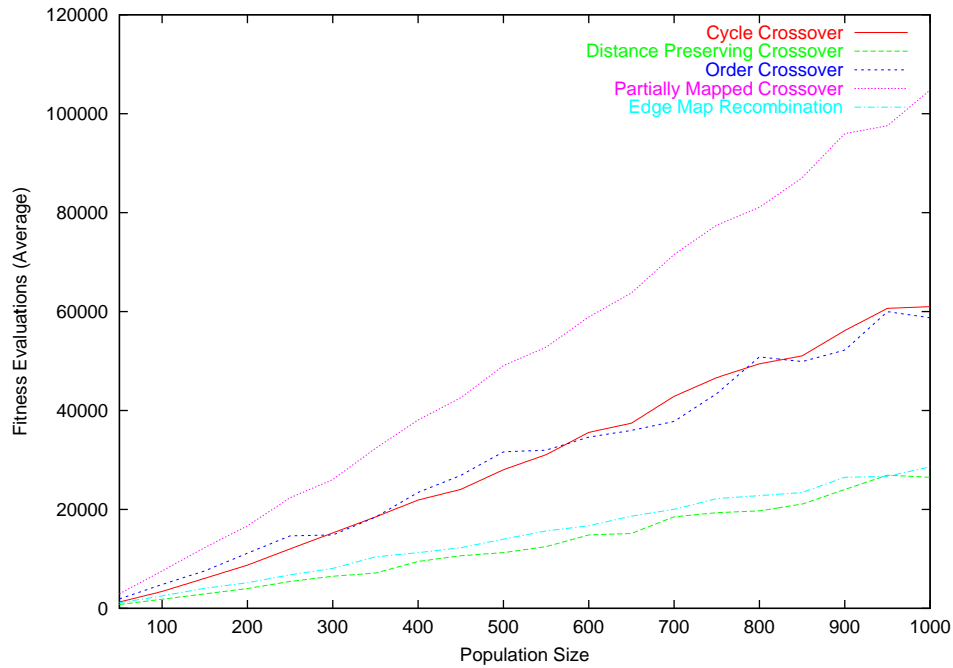


Figure 4.20: The average fitness evaluations over 20 runs.

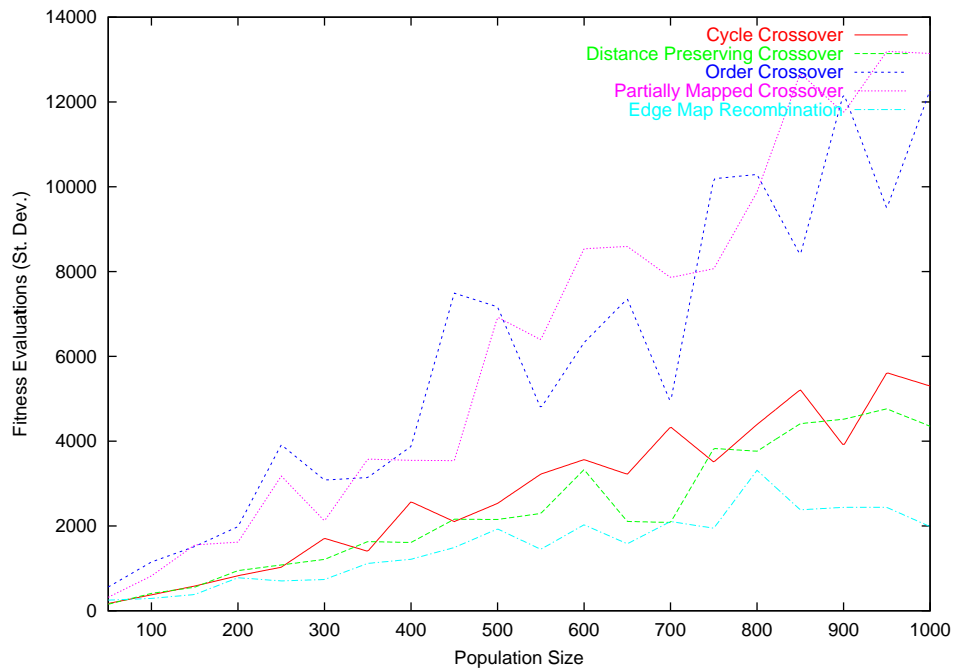


Figure 4.21: The standard deviation over the fitness evaluations over 20 runs.

4.3.2 GA vs. ES on simple polynomes

As a first multiple runs example, we wish to do some tests on simple polynomes to take a look at the results between a GA and an ES (Evolution Strategies, see section 3.4.4). The idea of this example is to vary the optimization problem by taking four different polynomes and by observing how over the board a quite standard GA performs in comparison to an ES. Once again, we note that all of this is merely stated here as an *example* that serves to show the versatility of the program and not the use of either genetic algorithms or evolution strategies in any way.

A key issue is to understand how we wish to compare the methods. First, we must establish the precision that the GA can handle through the amount of bits to use to code the real numbers along the x axis. In this example we shall take a look at precision and to this end we shall take 50 bits for the GA, so the optimal solution for the polynome problem can be approximated well enough by discretization of the search space.

Another key issue is that we must realize that the evolution strategy is most likely not going to terminate very rapidly because of the mutation that is *always* applied so that therefore (even though the mutation size changes over the generations) the equality termination condition will not be met for some time. A plausible decision on behalf of this reasoning is then to use a termination condition that terminates when the standard deviation amongst the fitness values lies below some ε . What we shall however do in this example is more in the light of the precision mentioned earlier and we shall allow a maximum of generations for the evolutionary algorithms. We shall allow the algorithms to run for 25 generations, keeping the amount of fitness evaluations equal for both the GA and the ES by choosing appropriate population sizes. We shall increase the population size and see what type of GA or ES performs better on what problem. In this example we wish to perform a real large amount of tests, which normally would not be advised because of the clutter in the resulting graphs. We shall test four different polynome problems, ranging from more easy to more difficult and we shall use one–point, two–point and uniform crossover for the GA, whereas we shall use the three different methods for mutation for the ES that were presented in section 3.4.4. We shall now first present the polynome problems we shall use as test functions, followed by the setup of the genetic algorithm and the evolution strategy that we shall test.

Polynome problems

There are two things we can specify when defining a polynome problem in the *EA Visualizer*. First of all, we may define the polynome that we wish to inspect and second of all, we may define the type of optimization for the polynome problem, namely whether the objective is to maximize or minimize the function value or to find the root of the function³. As we noted, we will test four different optimization problems with polynomes. The poly-

³The actual objective function is then to minimize the absolute value of the function.

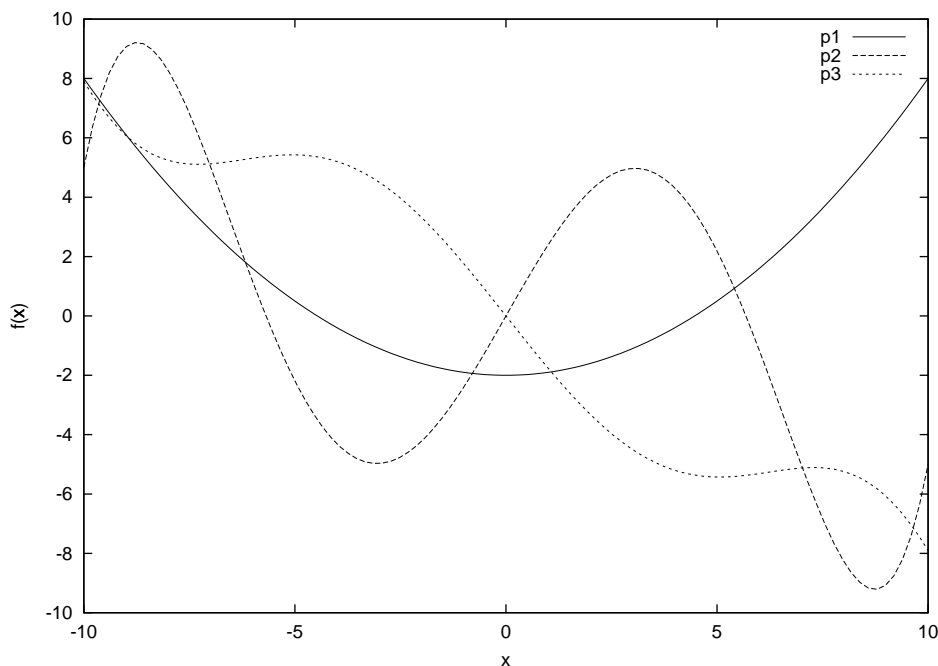


Figure 4.22: The polynomes used for the optimization problems.

nomes that were selected are quite arbitrarily chosen and are all regarded at the interval $[-10, 10]$.

The first two problems are defined for the function $p_1(x) = 0.1x^2 - 2$ that is displayed in figure 4.22. This function can be used to find $\sqrt{20}$ numerically by using the root finding optimization measure. We shall use this as our first polynome problem. The best fitness value is therefore equal to 0. We can also use this problem to minimize this simple function to find as optimum the value of -2 .

The next function we use is the slightly more interesting function $p_2(x) = 0.0007x^5 - 0.1x^3 + 2.5x$. We want to maximize this function. The optimum can be found for $x \approx -8.73842734$ with $p_2(x) \approx 9.21383715$. Finally, we want to minimize the function $p_3(x) = 0.000000004x^9 + 0.00000001x^7 - 0.0003x^5 + 0.036x^3 - 1.8x$. The minimum for this function is found for $x = 10$ with $p_3(x) = 7.9$. All these functions $p_1(x)$, $p_2(x)$ and $p_3(x)$ can be seen in figure 4.22.

Genetic Algorithm setup

The setup for the GA is a rather canonical one. The string length as we specified earlier is taken to be 50 bits. As just mentioned, we shall vary the recombination operator. The operators used will be the classical one-point, two-point and uniform crossover operators. The mutation operator shall be left out of the experiments. We leave experimenting with

this to the reader as an exercise. Mating the genomes will be done by using the *Simple Mater* with a grouping size of two, so that the crossover operators all receive two parents to recombine. The use of the selected mating mechanism is allowed because of the fact that we shall use tournament selection to perform selection on beforehand to select the parent genomes. We shall employ a standard tournament size of two for this selection operator. The remaining issue we need to specify is the replacement strategy. As we wish to use a rather standard genetic algorithm, the instance we use for the *Replacer* component is the *New Offspring Only* instance. This gives us a generational GA because all of the population is renewed by the offspring genomes that result from crossover. In a single run EA in the *EA Visualizer*, specifying this would come down to the following settings:

Component	Instance	Parameters
<i>Genotype</i>	Binary String	String length 100
<i>Similarity</i>	No Similarity	—
<i>Mutator</i>	No Mutation	—
<i>Before Selector</i>	Tournament Selection	Selection size n Tournament size 2
<i>After Selector</i>	No Selection (Select All)	—
<i>Mater</i>	Simple Mater	Grouping size 2
<i>Replacer</i>	New Offspring Only	Report popsize warnings Yes
<i>Terminator</i>	Maximum Generations	Maximum Generations 25
<i>Hybrid Searcher</i>	No Hybrid Search	—
<i>Population</i>	Vector Population	Population size n
<i>PRNG</i>	Standard Java PRNG	Seed <i>Any</i> Use Random Seed Instead Yes

From the above table, the settings for the *Fitness Function* have been removed because of the fact that the parameters for the polynome optimization problem will be varied to model a multiple of polynome problems. In addition, because of the fact that the *Recombination* component will have multiple instances (multiple crossover operators), we have removed this component from the table above as well.

Evolution Strategy setup

For the evolution strategy, we also wish to install a rather standard type. From the two types of selection mechanisms that are commonly used, we choose to use $(\mu + \lambda)$ selection which is different from the selection scheme used in the GA we specified. The $(\mu + \lambda)$ selection scheme is of an elitist form because the final population is selected by choosing μ individuals from both the μ parents as well as the λ offspring. As we mentioned earlier, we wish to keep the amount of fitness evaluations equal for the GA and the ES. To this end, we note that in a population of size n , the GA generates n offspring each generation which all have to be evaluated. This means that we may allow n offspring also in the ES. As the recombination operator for evolution strategies takes an arbitrary amount of parent

genomes and generates only a single offspring, we must select a multiple of n genomes as parents and group these together. Just as was the case for the GA, we shall group the genomes in groups of two. This means that we need a before selector to select at random $2n$ parent genomes that will be mated by the *Random Mater* in n groups of size 2. Note that the *Simple Mater* cannot be used now, because the population will not be shuffled by the random selection mechanism. The recombination operator that will be applied to each of the n groups will give n offspring genomes. These offspring genomes have to be added to the population. This is done by using the *Add Offspring To Population* instance for the *Replacer* component. This will then result in a population that consists of both the genomes from the previous generation and the offspring genomes. From this, as is the case in the $(\mu + \lambda)$ selection mechanism, n genomes have to be selected. This selection requires the evaluation of the genomes in the population. However, only the newly generated n offspring genomes in this population have to be evaluated, so we have indeed the same amount of fitness evaluations as in the case of the genetic algorithm. To select the n genomes, the truncation selection mechanism is used. The settings for a single run EA in the *EA Visualizer* are the following (the settings for the *Fitness Function* are once again omitted and the evolution strategies single actual value is displayed as initially distributed in the range $[a, b]$, where these values are of course dependent on the (polynome) problem at hand.):

Component	Instance	Parameters
<i>Genotype</i>	Evolution Strategies	Length 1
		Min. init range values a
		Max. init range values b
		Min. init range meta-parameters 0
		Max. init range meta-parameters 1
		Combination type <i>Varied</i>
<i>Similarity</i>	No Similarity	—
<i>Recombinator</i>	ES Crossover	Crossover type for values Discrete
		Crossover type for meta-parameters Intermediate
		Crossover type for angle-parameters Discrete
<i>Mutator</i>	ES Mutation	General Tau 0.7071067811865
		Individual tau 0.7071067811865
		Minimal St. Dev. 0.0
		Beta <i>Any</i>
<i>Before Selector</i>	Random Selection	Selection size $2n$
<i>After Selector</i>	Truncation Selection	Selection size n
		Percentage 50
		Which To Use Selection size
<i>Mater</i>	Random Mater	Grouping size 2

Component	Instance	Parameters
<i>Replacer</i>	Add Offspring To Population	—
<i>Terminator</i>	Maximum Generations	Maximum Generations 25
<i>Hybrid Searcher</i>	No Hybrid Search	—
<i>Population</i>	Vector Population	Population size n
<i>PRNG</i>	Standard Java PRNG	Seed <i>Any</i> Use Random Seed Instead Yes

Running the tests and gathering the results

Before we can actually run the tests, we must of course enter the settings for the multiple runs EA in the *EA Visualizer*. We shall quickly go over how this can be done, after which we show the resulting system that displays many cluttered graphs that contain far too much information. Finally we show that by using `GNUPLOT` we can easily alter the resulting numbers to create graphs that *are* transparent and give a good overview of the results.

We must first realize that we have two distinct instances for the *Genotype* component in the system. As the *Genotype* component is a dependency imposing component, this means that we cannot directly install a multiple of instances for it as we can for the *Recombinator* for instance as we saw in the first example in section 4.3.1. This means that we require to install two multiple multiple runs EAs, where the first is for the GA and the second is for the ES so that the first uses the `Binary String` and the second uses the `Evolution Strategies` as an instance for the *Genotype* component.

We start by selecting `New Multiple Runs EA` from the EA menu in the main system. We first concentrate on the multiple GA implementations and then in a completely separate fashion concentrate on the multiple ES implementations. First, we press the *Add* button to add a new multiple runs EA and select the instances for the four instances for the dependency imposing components to be `Binary String`, `No Similarity`, `Binary String - Polynome Optimizer in [a,b]` and `Vector Population`. Having done so, we are required to enter the remainder of the settings after pressing the *Apply* button. The parameters for the *Genotype* are of course that we desire a string length of 50 bits. The fitness function parameters are slightly more complicated, but the lower and upper limits are set to -10 and 10 respectively for all problems. The `Function` parameter is set to the four functions we proposed earlier, namely $p_1(x)$ twice, followed by $p_2(x)$ and $p_3(x)$. This can be done by entering a polynome and by subsequently pressing the *Next* button behind the polynome component. For all of these functions we need to specify what optimization type we wish to use, so we enter four values for this parameter as well, namely *Root*, *Minimize*, *Maximize* and again *Minimize* respectively for the four polynome problems. After defining the parameters, we shall of course require to link these two parameters as we want them to be enumerated simultaneously. This we shall come to shortly. The parameters for the population size are set to range from 2 to 20 with an increase of 2. We do not expect

these problems to be very difficult, so we will not require very large population sizes. As we already discussed when defining the setup for the GA, we install three recombination operators and for all other components the instances we specified. We set the selection size for the *Before Selector* to be the same as the population size as required. The resulting interface should look like the one in figure 4.23.

Having defined the instances and their parameters for the multiple runs GA, we define the links after pressing the *Edit Links* button. We require to define two links, namely the one linking the population size to the selection size as we also saw in section 4.3.1 but also the one that links the definition of the polynomes to the optimization type for the polynome problems so that we have exactly 4 optimization problems instead of $4 \cdot 4 = 16$. Knowing what is to be linked, we leave the actual specification of these links to the user as an exercise. Note that two *separate* links are required.

Next, we have to define the multiple runs ES. This is done by pressing the *Add* button in the **New Multiple Runs EA** interface a second time. First we are then to specify again the instances for the dependency imposing components. In this second case, these are of course *Evolution Strategies*, *No Similarity*, *Evolution Strategies - Polynome Optimizer in [a,b]* and *Vector Population* respectively. Having done so, we are again required to specify the parameters and instances for the other components. For the genotype, we specify the length (amount of coding variables) to be 1 as the polynome problems are only one-dimensional. The minimum and maximum init ranges are set to -10 and 10 respectively. The init range for the meta parameters is set to $[0, 1]$. Finally, our three ES variants are constructed by specifying to use all three combinations of value-parameters, meta-parameters and angle-parameters that are available. The arity of this particular parameter of the *Genotype* is thus three. The parameters for the *Fitness Function* are set in a completely analogous fashion as was the case for the GA, just as will be the link that regards the *Fitness Function*. The population size is also defined exactly the same as in the case of the GA settings we just entered. The remaining instances for the components are again conform to the ES setup we described above. The only thing we must note is that the *Before Selector* which is set to select at random selects 4 to 40 genomes with a stepsize of 4, which is double the amount of the population size. Why this is required we already noted earlier, but at this point we note how to actually specify the correct values. The values for the *After Selector* are set to range from 2 to 20 with a stepsize of 2, just as is the case for the population size. Note that this requires us to link three parameters (population size, selection size for both *Before* and *After Selector*) with respect to the population size, which is one more than was the case for the GA. The links that go with the multiple runs ES are again twofold and consists of a population size link and a link for the polynome problems. We again leave the actual specification of this to the user. The resulting settings interface should in any way be equal to what is shown in figure 4.24.

Lastly, we note that we run all experiments 20 times by entering this value in the **Number Of Runs** field in the **New Multiple Runs EA** interface for both the multiple runs GA as well the multiple runs ES. The results we wish to see are of the same type as in the case of

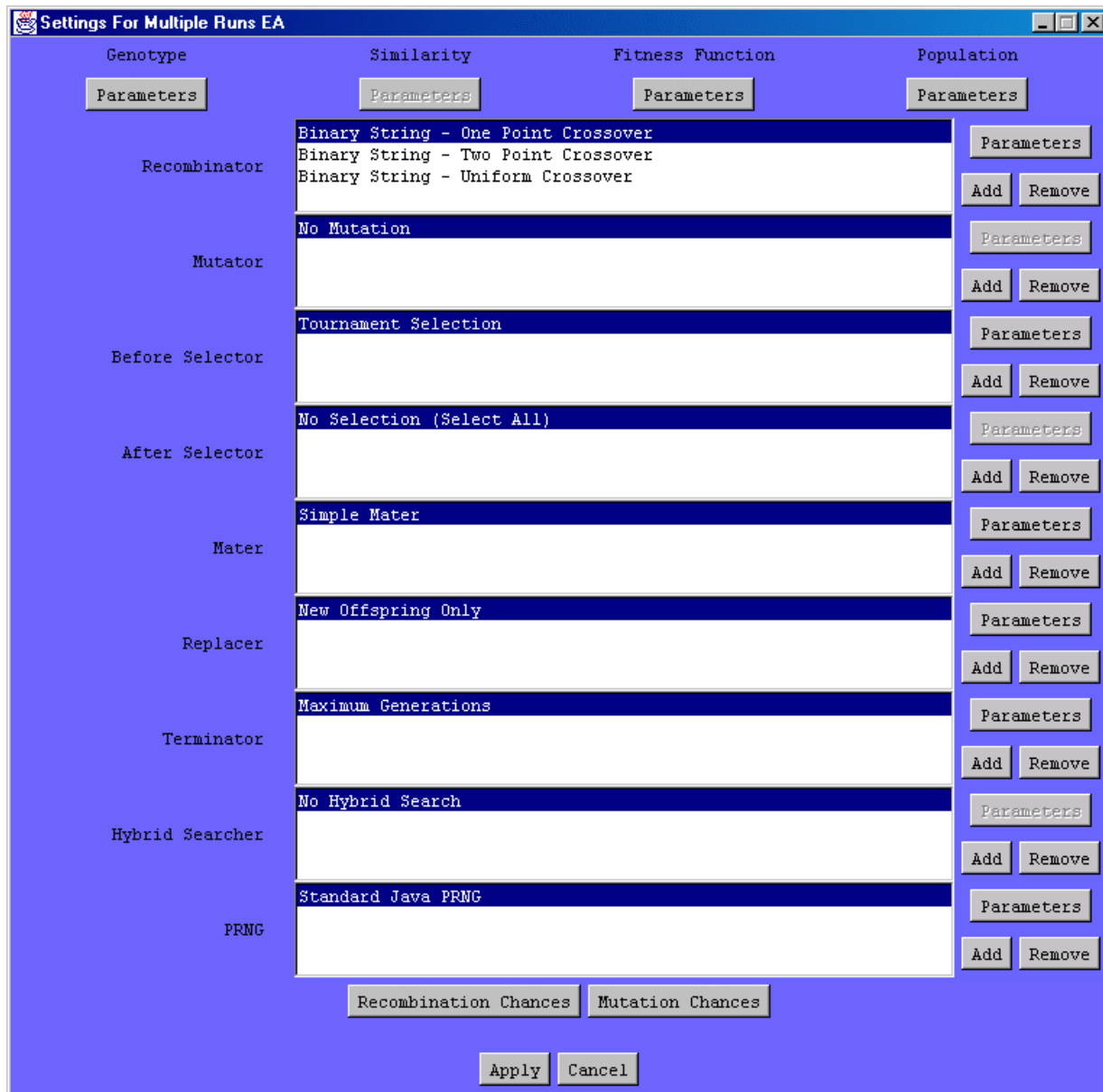


Figure 4.23: The settings for the multiple runs GA for the polynome problems.

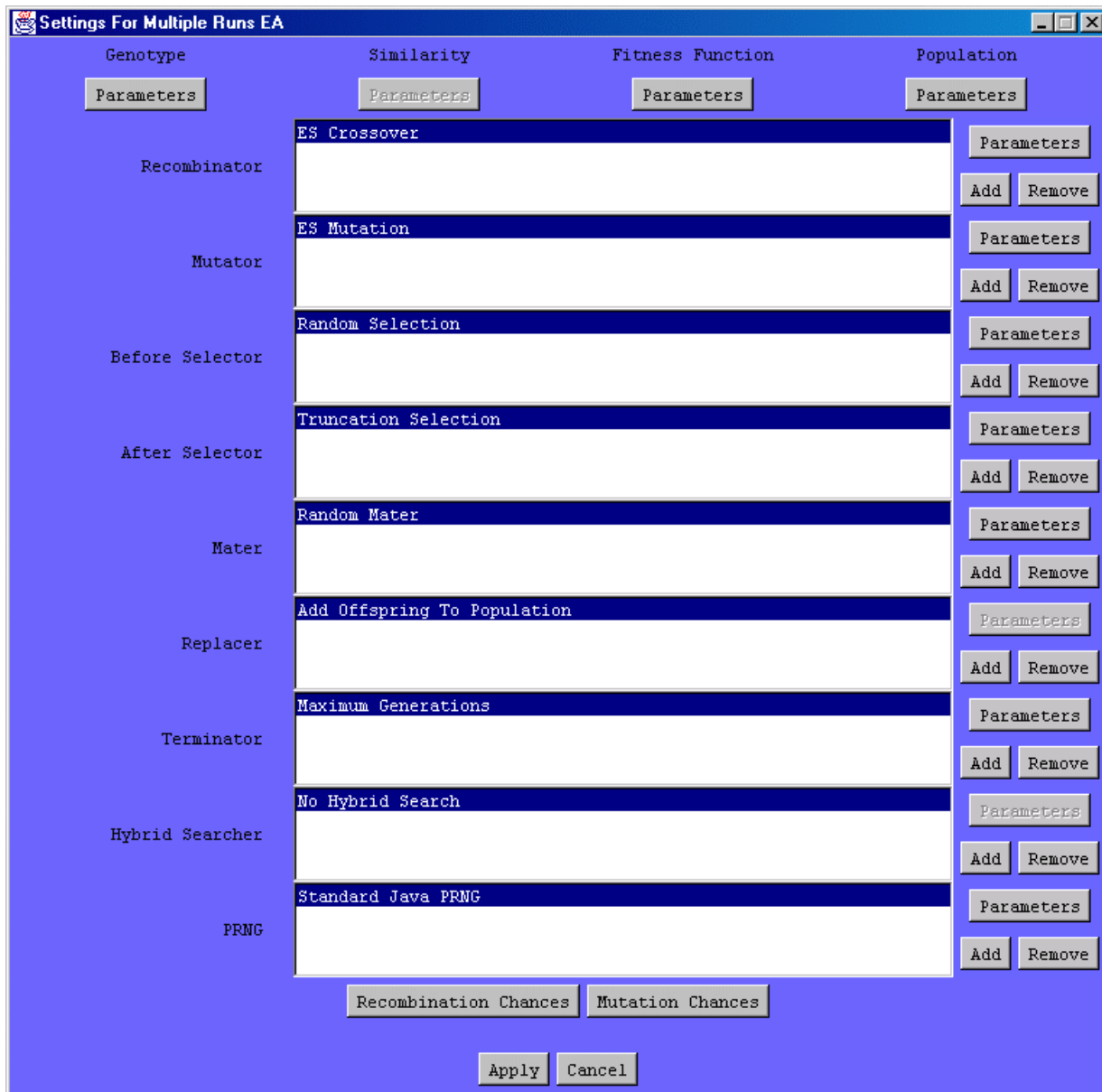


Figure 4.24: The settings for the multiple runs ES for the polynome problems.

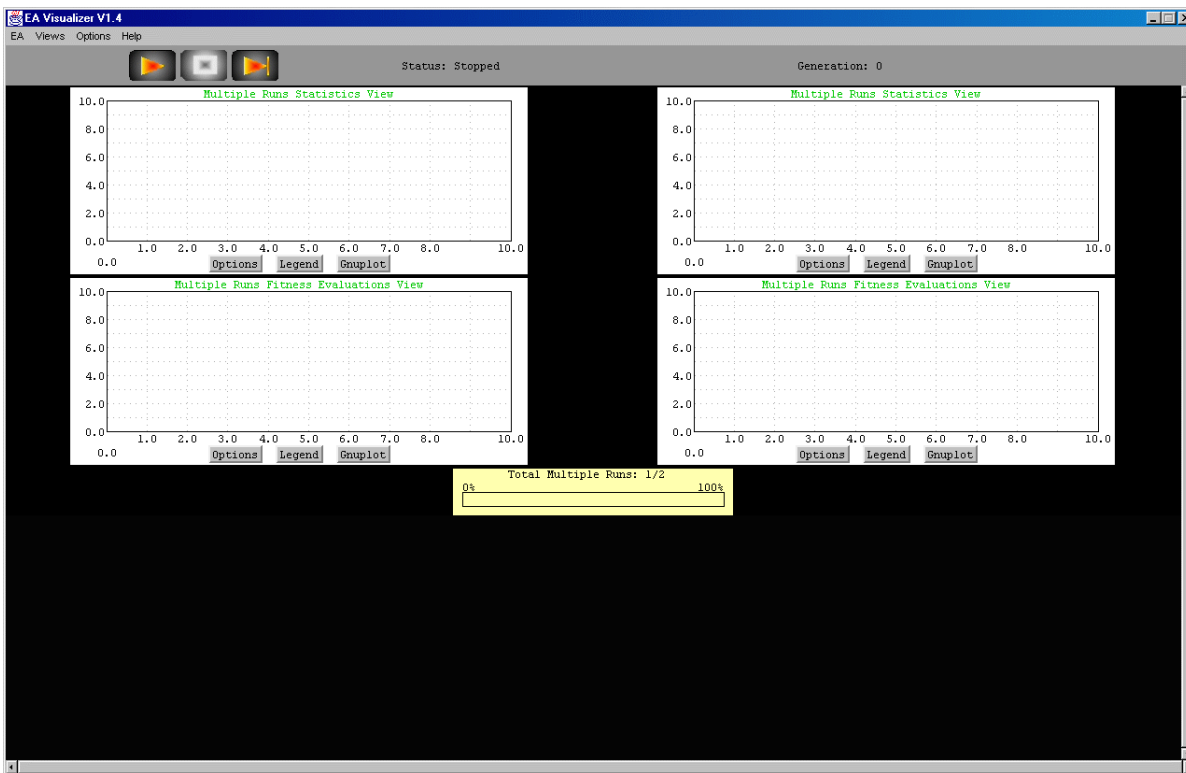


Figure 4.25: The system that is ready to run the multiple runs for the polynome problems.

the first example (section 3.4.1), namely the average fitness value of the best genome upon termination and the standard deviation over these values, as well as the average amount of fitness evaluations and the standard deviation over these values. Note that the last two views will not provide us with a whole lot of information as it should be the case that the standard deviation for all population sizes should be 0 because our idea was that we wanted to allow every algorithm an equal amount of fitness evaluations (given a certain population size). This means that the average fitness evaluations should be a straight line and that the standard deviation should be zero at all times. These two graphs are thus only a some sort of a check whether we did it right. Finally, for keeping track of how far we have come in running all combinations of settings, we add a progress indicator. The system that is ready to run on the polynome problems is shown in figure 4.25.

When the system terminates, each of the four graphs shows exactly 24 different lines. This is because for both the GA as well as the ES, three different strategies are tested on four different problems, leading to $2 \cdot 3 \cdot 4 = 24$ different evolutionary search procedures with associated search problem. Of course, having 24 lines in a single graph causes far too much clutter to still get any good ideas about the results as can be seen in figure 4.26. Note that we have altered the titles for the graphs at this point. Also note that the fitness evaluations plots show the results that we hoped for, namely a single straight line that shows that all

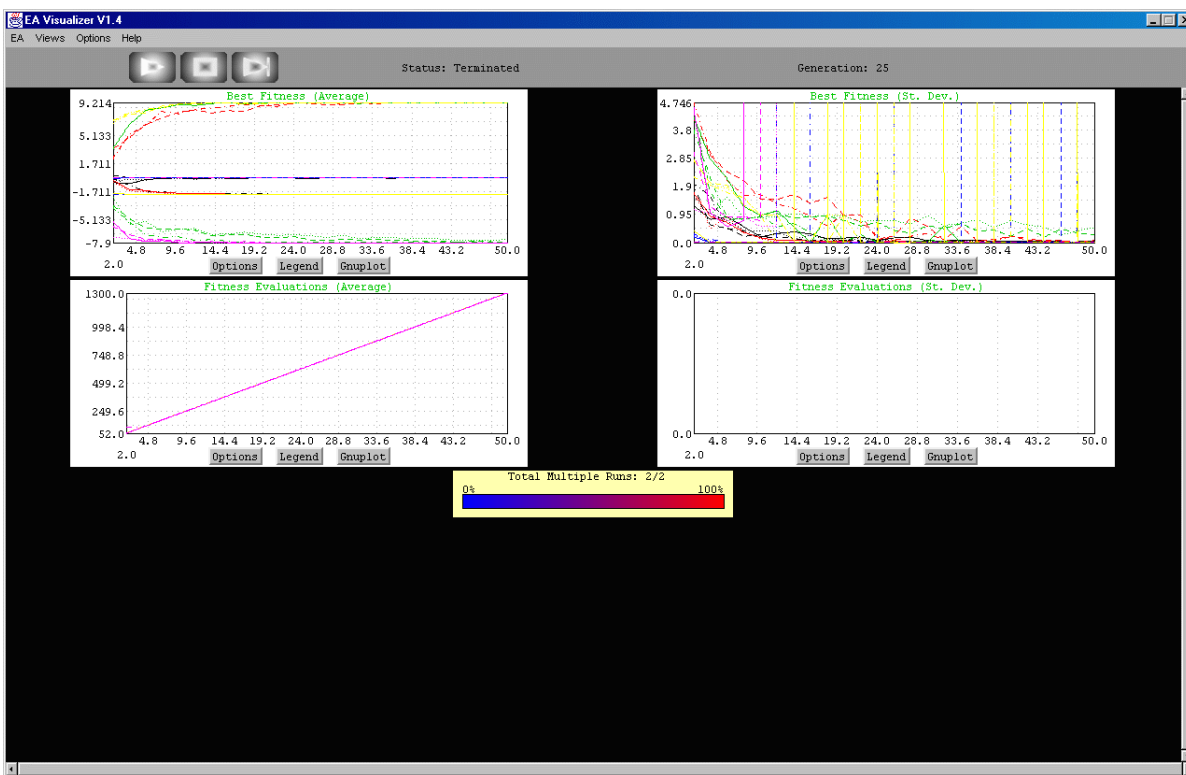


Figure 4.26: The system that has finished running the multiple runs for the polynome problems.

runs had the exact same amount of fitness evaluations, which can be stated because of the fact that the standard deviation graph always has a value of zero. Finally, the most peculiar result must be the straight lines that are pointing up to infinity in the standard deviation graph for the best fitness results. This is due to numerical errors. Values very close to zero cause problems when computing with them and result in such anomalies in the resulting graph. These can be filtered out once GNUPLOT is used as the datafiles can be edited. Note that GNUPLOT itself has the same problem as the *EA Visualizer* has here.

In order to present the results in an orderly fashion, we show that this is quite easily done using the *EA Visualizer* and the GNUPLOT option for the graphs. We show as an example how this can be done for the average best fitness graph and leave the standard deviation graph to the user as an exercise. First, we press the GNUPLOT button in the graph for the average best fitness and we save the results to disk. This causes 25 files to be created, amongst which 24 datafiles and a single plotfile. The datafiles contain the data for the lines in the graph. At this point, we should realize what datafile represents what results. The first 12 datafiles are for the GA and the second 12 are for the ES. For each of these batches of datafiles, the first three datafiles are for the first polynome problem with one—,

```

set terminal postscript eps color
set xlabel "Population Size"
set ylabel "Best Fitness for polynome problem 1 (Average)"
set xrange [2 : 50]
plot "BestFitnessAverage_datafile0" using 1:2 title "1PX" with lines,\
      "BestFitnessAverage_datafile1" using 1:2 title "2PX" with lines,\
      "BestFitnessAverage_datafile2" using 1:2 title "UX" with lines,\
      "BestFitnessAverage_datafile12" using 1:2 title "ES1" with lines,\
      "BestFitnessAverage_datafile13" using 1:2 title "ES2" with lines,\
      "BestFitnessAverage_datafile14" using 1:2 title "ES3" with lines

```

Figure 4.27: GNUPLOT code for the average best fitness value for polynome problem 1 over 20 runs.

two- and uniform crossover respectively. The second three datafiles are for the second polynome problem and so on. Equally, the first three datafiles from the second batch of 12 datafiles are for the three different strategies for the ES on the first polynome problem and the second three datafiles are for the second polynome problem. Knowing this, we can create new plotfiles that use these datafiles to group them in a way so that the resulting graphs are clear and useful.

The way we wish to create more clear graphs in this example is by dividing the 24 results over four graphs, all holding exactly six graph entries, namely the six different evolutionary algorithms. The four different graphs of course represent the four different polynome problems. To do this, we first create four new plotfiles that hold the GNUPLOT commands for the new graphs. These four datafiles hold nothing much more than references to the correct datafiles. With the information just given about how the *EA Visualizer* makes the datafiles, it should be clear to the reader what files are required for what plot. At this point, an exercise is to write down what datafiles go with which plot. The resulting four plotfiles that have to be run through GNUPLOT are shown in figures 4.27 through to 4.30 for polynome problems 1 through to 4 respectively. The first line in each plotfile tells GNUPLOT we want the result to be encapsulated postscript and in color. The next three lines define the labels along the axes and the range for the x axes that corresponds to the settings we had in the *EA Visualizer*. The remainder of each datafile is the reference to the right datafiles with the corresponding titles for the legend.

Finally, we are ready to create the final images. This is done by running GNUPLOT on the files. If we have named the files from figures 4.27 through to 4.30 `bfa1`, `bfa2`, `bfa3` and `bfa4` (where `bfa` stands for best fitness average), we could do this by executing the following commands:

```

gnuplot < bfa1 > bfa1GAVSES1.ps; gnuplot < bfa2 > bfa1GAVSES2.ps
gnuplot < bfa3 > bfa1GAVSES3.ps; gnuplot < bfa4 > bfa1GAVSES4.ps

```

```

set terminal postscript eps color

set xlabel "Population Size"
set ylabel "Best Fitness for polynome problem 2 (Average)"
set xrange [2 : 50]
plot "BestFitnessAverage_datafile3" using 1:2 title "1PX" with lines,\
     "BestFitnessAverage_datafile4" using 1:2 title "2PX" with lines,\
     "BestFitnessAverage_datafile5" using 1:2 title "UX" with lines,\
     "BestFitnessAverage_datafile15" using 1:2 title "ES1" with lines,\
     "BestFitnessAverage_datafile16" using 1:2 title "ES2" with lines,\
     "BestFitnessAverage_datafile17" using 1:2 title "ES3" with lines

```

Figure 4.28: GNUPLOT code for the average best fitness value for polynome problem 2 over 20 runs.

```

set terminal postscript eps color

set xlabel "Population Size"
set ylabel "Best Fitness for polynome problem 3 (Average)"
set xrange [2 : 50]
plot "BestFitnessAverage_datafile6" using 1:2 title "1PX" with lines,\
     "BestFitnessAverage_datafile7" using 1:2 title "2PX" with lines,\
     "BestFitnessAverage_datafile8" using 1:2 title "UX" with lines,\
     "BestFitnessAverage_datafile18" using 1:2 title "ES1" with lines,\
     "BestFitnessAverage_datafile19" using 1:2 title "ES2" with lines,\
     "BestFitnessAverage_datafile20" using 1:2 title "ES3" with lines

```

Figure 4.29: GNUPLOT code for the average best fitness value for polynome problem 3 over 20 runs.

```

set terminal postscript eps color

set xlabel "Population Size"
set ylabel "Best Fitness for polynome problem 4 (Average)"
set xrange [2 : 50]
plot "BestFitnessAverage_datafile9" using 1:2 title "1PX" with lines,\
     "BestFitnessAverage_datafile10" using 1:2 title "2PX" with lines,\
     "BestFitnessAverage_datafile11" using 1:2 title "UX" with lines,\
     "BestFitnessAverage_datafile21" using 1:2 title "ES1" with lines,\
     "BestFitnessAverage_datafile22" using 1:2 title "ES2" with lines,\
     "BestFitnessAverage_datafile23" using 1:2 title "ES3" with lines

```

Figure 4.30: GNUPLOT code for the average best fitness value for polynome problem 4 over 20 runs.

The resulting graphs are then stored in the postscript files denoted by the files with the extension `.ps`. These final results are shown in figures 4.31 through to 4.34.

4.3.3 Population sizing

A more interesting example than the simple test run for some recombination strategy as given in section 3.4.1 is that of the testing of the population sizing theory. Evolutionary algorithms have quite a lot of parameters and the question of course is what parameters are the best. One of the most prominent parameters is that of the population size. Clearly a population that is too small does not potentially hold enough information to solve a problem. G. Harik, E. Cantu-Paz, D.E. Goldberg and B.L. Miller [9] wrote a paper that concerns the problem of determining the required population size for a given optimization problem. They defined a function that predicts what size the population must be to converge with some probability to the optimum. We shall disregard that theory here but only use it in the *EA Visualizer* to test the soundness of that function. For starters we therefore need to do tests over a multiple of population sizes because we want to see the response for many population sizes (the empirical probability that the evolutionary algorithm converges to the optimum) within a specific range. Furthermore as we do not wish to rely on a single run, we want to average the results over a multiple of runs. This implies we require a multiple runs evolutionary algorithm in terms of the *EA Visualizer*. Because this is the case, we can enhance the tests we are about to run and test different recombination strategies at the same time.

The optimization problem that is inspected is the *trapfunctions* we introduced earlier in section 3.4.5. We shall in this example once more go over the multiple runs settings in detail as the settings are not always straightforward at first glance.

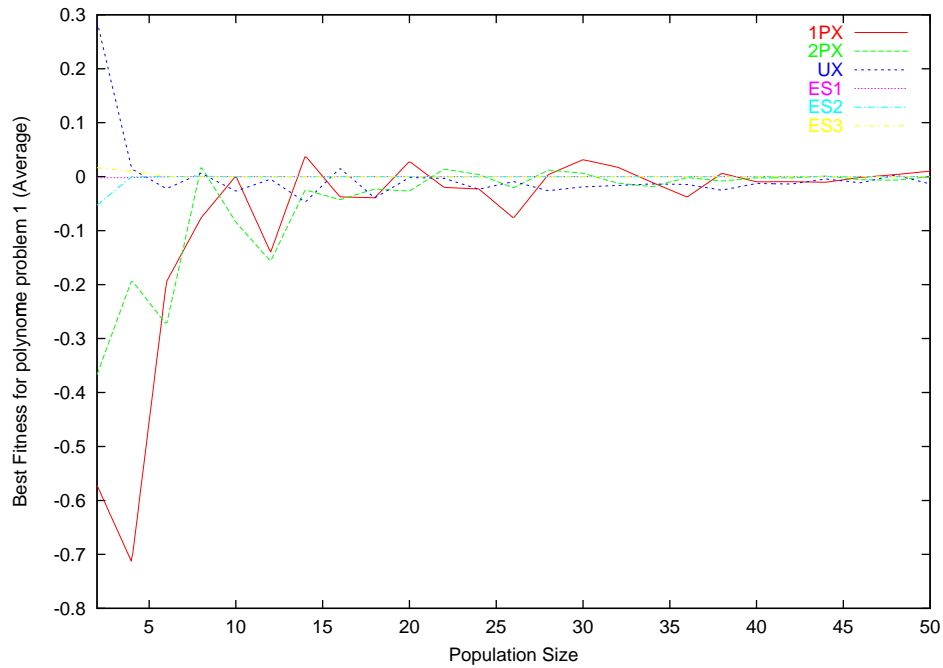


Figure 4.31: The average best fitness value for polynome problem 1 over 20 runs.

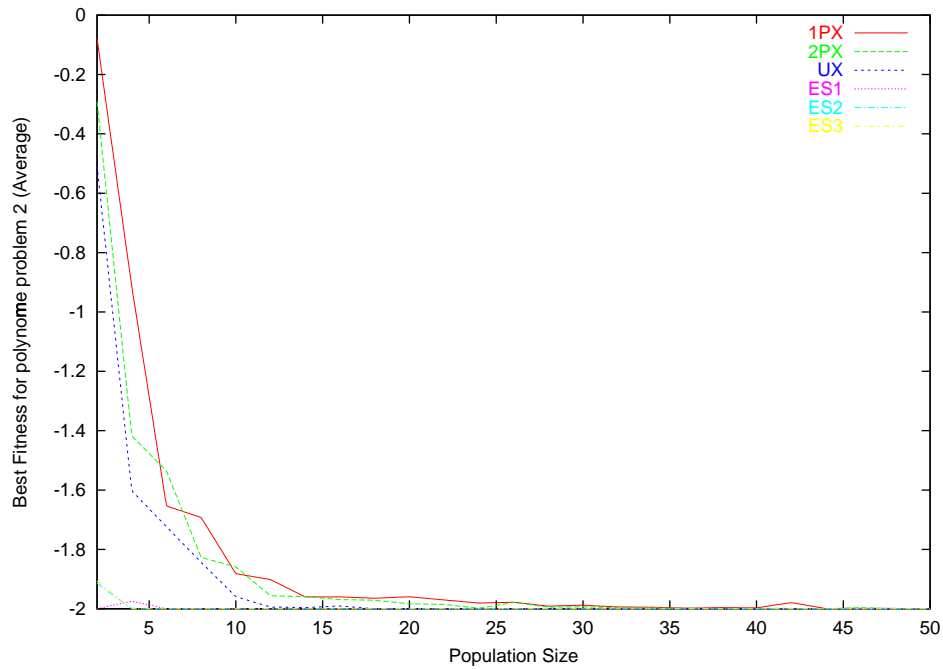


Figure 4.32: The average best fitness value for polynome problem 2 over 20 runs.

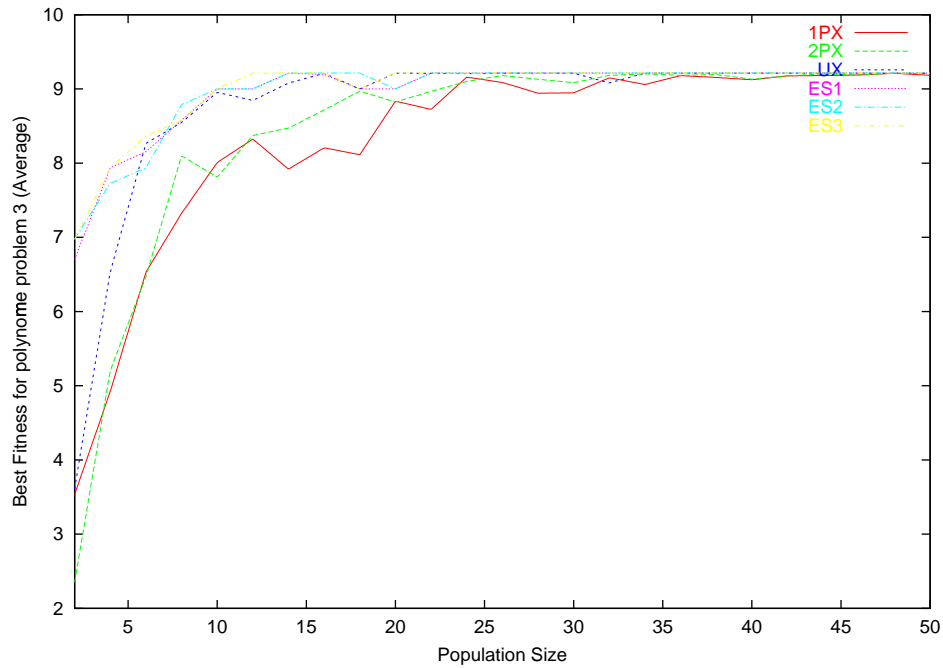


Figure 4.33: The average best fitness value for polynome problem 3 over 20 runs.

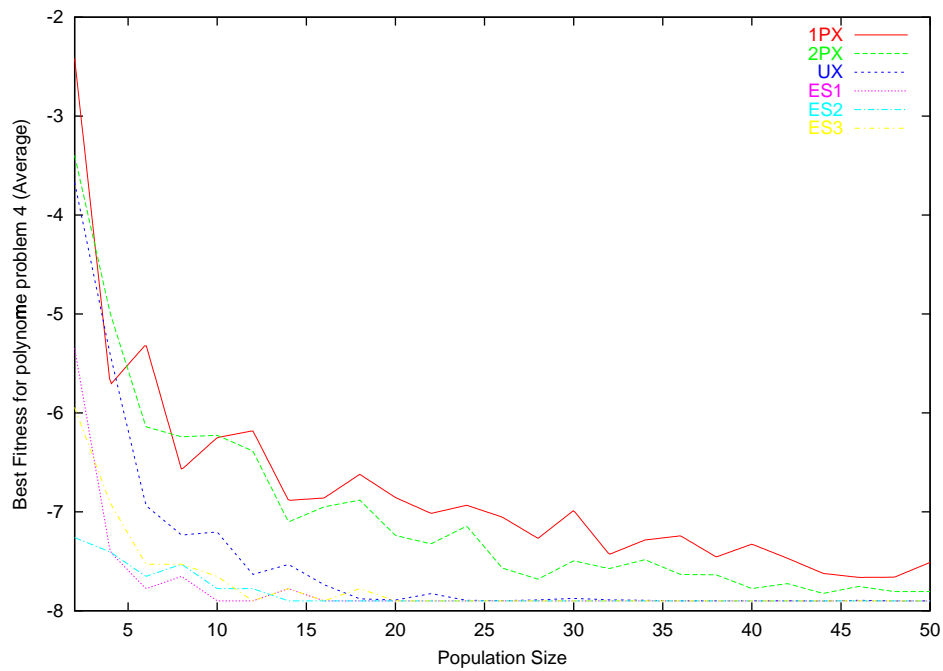


Figure 4.34: The average best fitness value for polynome problem 4 over 20 runs.

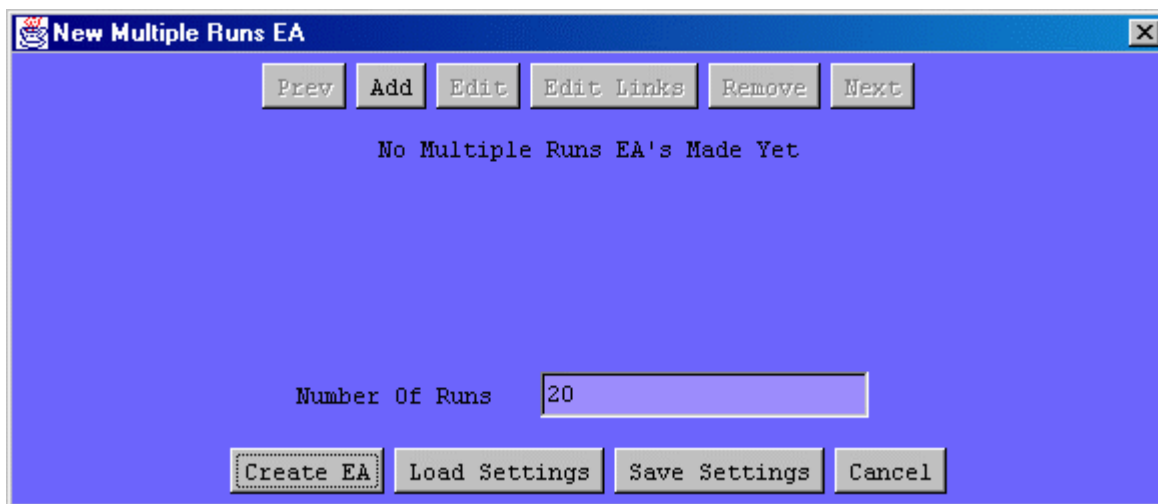


Figure 4.35: Creating a new multiple runs evolutionary algorithm.

By pressing F2 or by selecting **New Multiple Runs EA** from the EA menu in the main window of the system, we select to work with a multiple runs evolutionary algorithm. The window that appears requests from us that we enter the amount of runs we wish to average over. We set this amount to 30. The buttons at the top of the window indicate that we can create a multiple of multiple runs evolutionary algorithms. We shall only require one of such multiple runs evolutionary algorithms because we will have only one combination of dependency imposing components. The image of the current user interface is shown in figure 4.35.

By pressing the *Add* button in the current user interface, a new interface appears that allows us to specify the dependency imposing components as can be seen in figure 4.36. We select to use **Binary String** as the genotype, **No Similarity** as the similarity (which doesn't matter for the same reasons as the single run example), the trap functions as the fitness function and the **Vector Population** as the population component. By pressing the *Apply* button we confirm these settings after which the user interface from figure 4.37 appears. At the top of the interface four buttons are located that allow us to enter the settings for the dependency imposing components. We enter for the genotype to have 40 bits. The fitness function parameters are set to a signal of 0.1, 8 subproblems and a length of 5 for each subproblem. The additional parameters that can be set for the trapfunctions that are scaling constants deal with the scaling of the fitness contribution of the blocks themselves. We are disregarding these options in this case and thus leave the scaling type to uniform at a scale factor of 1.0 which does not alter the fitness response of a single building block and thus gives us the fitness function as described. The parameter structure of the *EA Visualizer* creates the multiple values parameter structure itself. As an example in figure 4.38 the parameters for the trap functions are shown set. We specify the population sizes to come from a range of 2 to 500 members with a stepsize of 4. Having

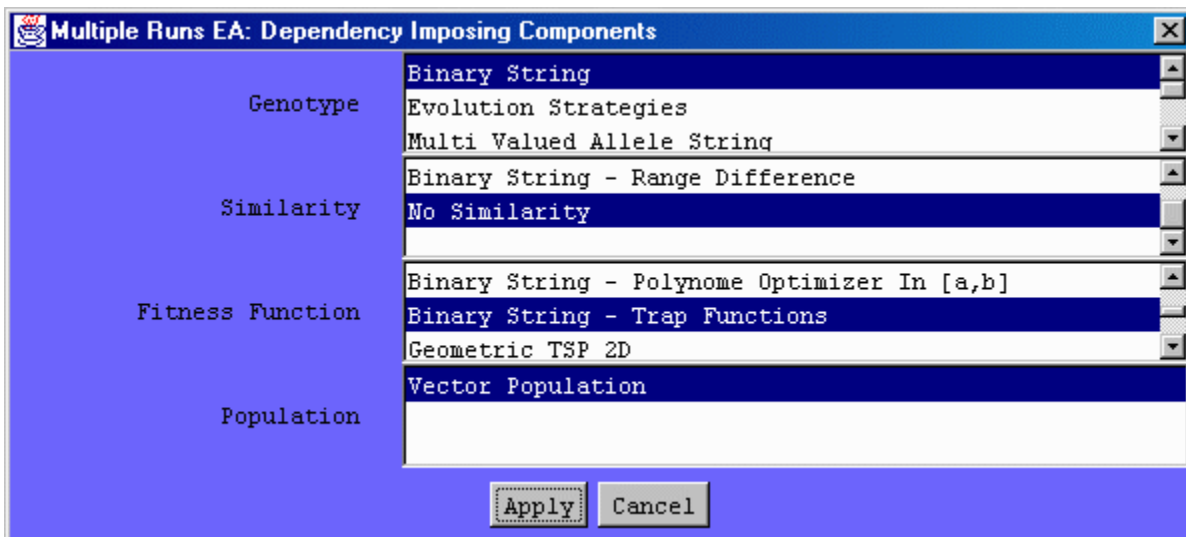


Figure 4.36: Selecting the dependency imposing components for multiple runs.

specified the parameters for the dependency imposing components, we continue with the remaining components and enter the recombinators we wish to use. By pressing **Add** on the right side of a list, we can add a new instance to use. We select to use both one-point, two-point and uniform crossover, the latter of which is deemed truly uniform crossover by employing a allele swapping probability of 0.5. The mutator we set to be **No Mutation** as we do not wish to use mutation anyhow (it causes the algorithm not to terminate based on the equality condition). The before selector is chosen to be tournament selection. We must remark that the settings for the selection size must be specified to be the same as the population size, namely from 2 to 500 with a stepsize of 4. Furthermore we employ no after selection strategy, a simple mater that mates the parents in groups of two, the new offspring only replacer to use the offspring only for the next generation, no hybrid search and the standard JAVA prng. The termination condition is specified to be the point when all genomes in the population are identical. We finish the specification of the parameters by entering that the recombination chance is always 1 and the chance at mutation is always 0. The resulting contents of the user interface can be seen in figure 4.37. By pressing *Apply* we once again agree to go with the specified settings and we return to the first user interface that was displayed. This interface now displays the selected four dependency imposing components for the multiple runs evolutionary algorithm we are editing. We can select to add a new multiple runs evolutionary algorithm, to edit the current one or to remove it. Finally we can select to edit the links. As we are still to connect the population sizes to the selection sizes so as to let the evolutionary algorithms select x genomes from a population of x members and not y genomes from a population of x members for all combinations of y and x from the numbers between 2 and 500 with stepsize 4, we select to edit the links by pressing the *Edit Links* button.

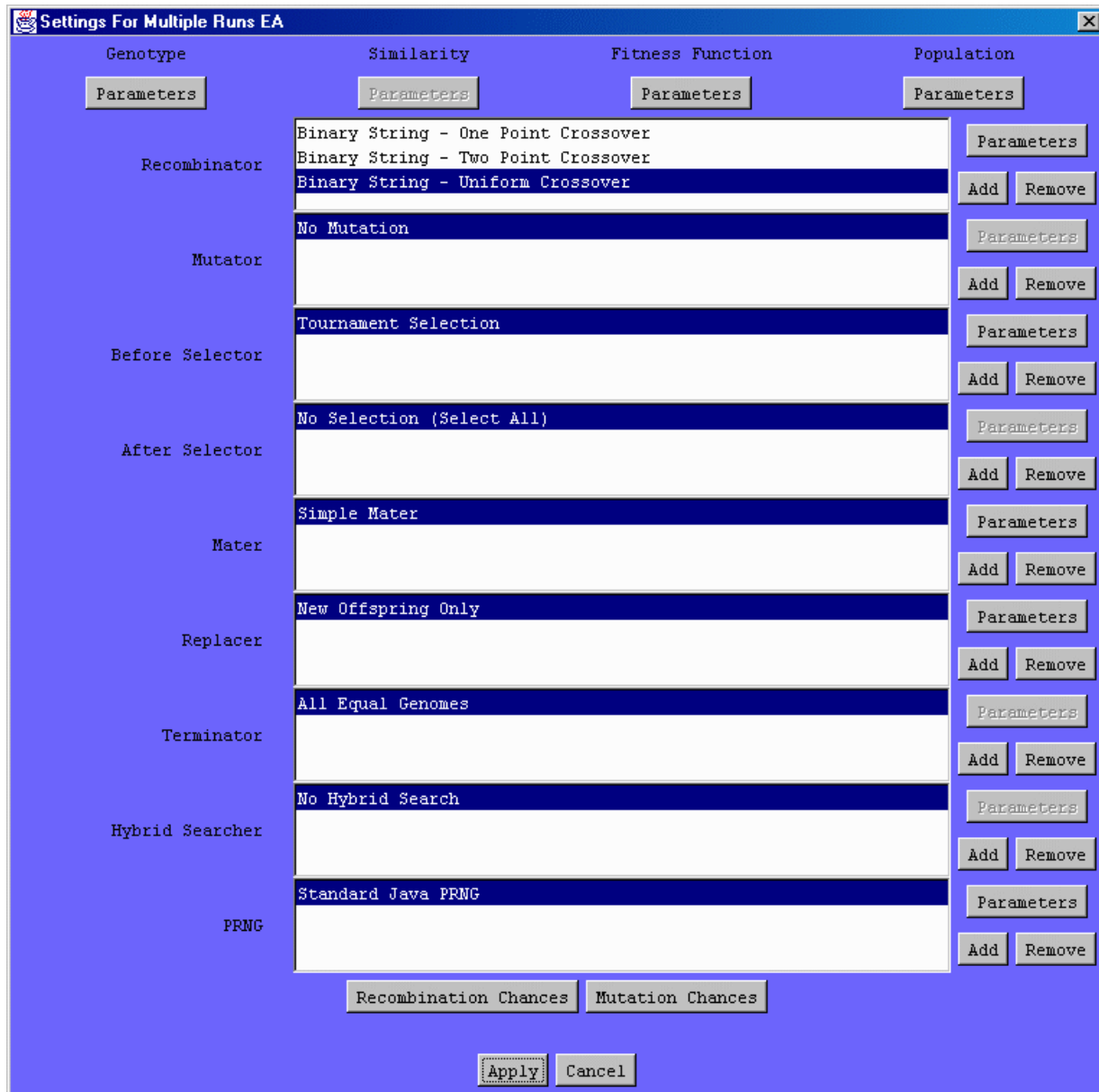


Figure 4.37: Entering the settings for one multiple runs evolutionary algorithm.

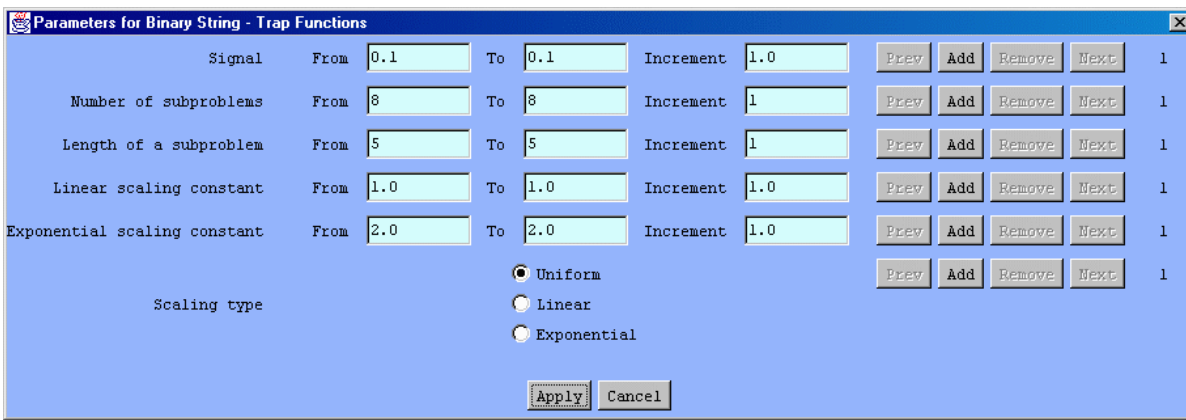


Figure 4.38: Multiple values parameter components for the trap functions.

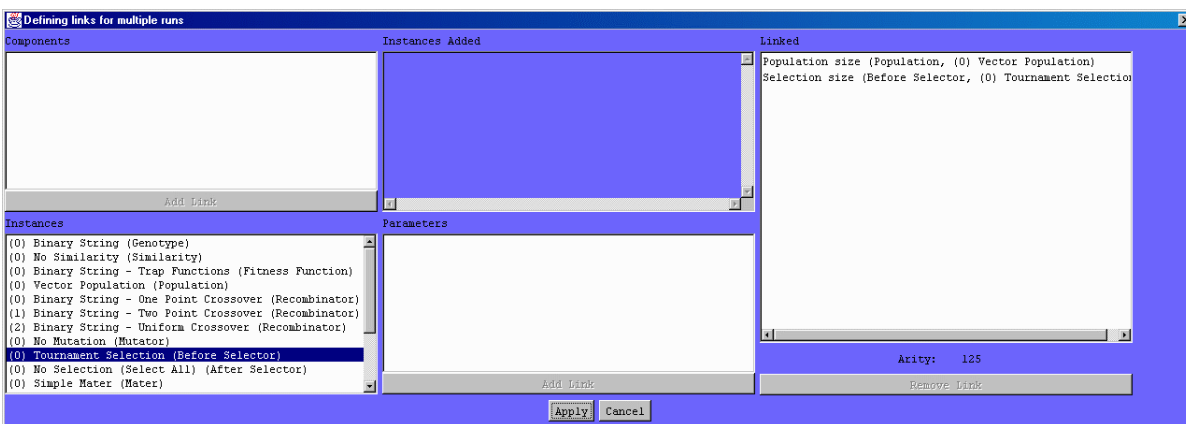


Figure 4.39: Linking the instances and the parameters to avoid crossproduct enumerations.

The complex user interface that appears allows us to specify the links between the components and/or parameters that are not to be enumerated separately. From the instances list we therefore select **Vector Population (Population)** and then select from the parameters list **Population size**. We then press the *Add Link* button below that list and see that the rightmost list now contains that parameter. The arity of the link is specified to be 125, which is the amount of numbers in the range between 2 and 500 with a stepsize of 4 ($\lceil \frac{500-2}{4} \rceil$). All parameters and components that do not have the same arity disappear when adding to a link for the first time. The only parameter that remains is the **Selection Size** of the tournament selector. We add this one to the link as well and end up with the configuration of the user interface as displayed in figure 4.39. Pressing *Apply* once again confirms the settings and returns us to the first interface. We can now press the *Create EA* button to finally create the evolutionary algorithm that was requested.

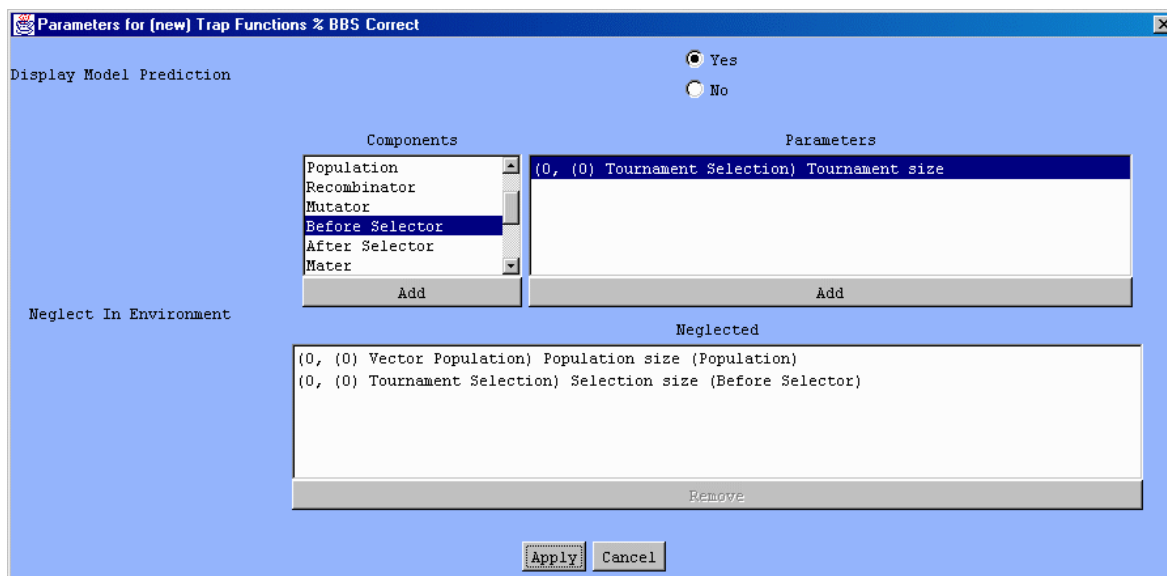


Figure 4.40: Entering the settings for a special multiple runs view for the trap functions.

As with the single run example, we have now reached the point where we are to decide what views to add to visualize the information about the evolutionary algorithm. Continuing to work analogously to the single run example, we wish to view some problem specific information along with some statistical information. Starting with the problem specific information, a special view has been developed that can be used to visualize the results and compare them with the population sizing function from the paper that was specified earlier. This view is named `Trap Functions % BBS Correct`. We specify to neglect both the population size of the population and the selection size of the tournament selector because we do not want an evolutionary algorithm with all the same settings except for the population size to be seen as a different evolutionary algorithm, especially because we are displaying the results as a function of the population size. The resulting settings can be seen in figure 4.40.

Second of all we wish to add some information about statistics. Such information is again added through the `Multiple Run Fitness Statistics` view. We want to display the resulting numbers under the same circumstances as with the specialized trap functions view, meaning we want to display them in a graph as a function of the population size and mapped by selecting what to neglect in the environment in the same way. The only thing that it is a little harder to select is the right combination of statistics as we are prompted to make a selection for the kind of `Statistic` we want to see and how we want to see it `Over All Runs`. We want to display the average over a multiple of runs as well as the standard deviation for we are doing the visualization analogous to the single run example. Plotting the standard deviation however is in this case rather superfluous as we do not let the algorithm terminate until all genomes are the same. They will therefore all have

the same fitness value, which will result in a standard deviation value of 0 over the board. So the only standard deviation value we would want to display that is meaningful is the standard deviation value over the fitness averages over the 30 runs per setting. We decide however to keep the amount of views for this example (it is still merely an example) low to avoid clutter in the example. As such we will select to display as the statistic type the `Fitness Average`. Over all runs we want to see this value `Averaged`.

Finally, because we are visualizing a multiple runs evolutionary algorithm that runs over a multiple of settings a multiple times, we want to know how far the system has come yet in doing all combinations when coming back to the computer at a certain time, so we add a `Progress Indicator` view. Once again we can press the *play* button when we have finished adding all the views. The system starts to enumerate all settings and combine the results in the views. As we have specified a very large number of combinations, the system will take some time before it completes the total computation. The reader can try out this example and see the results upon termination. In this example we present in figure 4.41 the system while it is still running after having computed a portion of all combinations. The names of the views allow the reader to deduce what is displayed where. We are once again not concerned with the implications of the (intermediate) results, but can see that uniform crossover (the dotted-dashed line) can only reach the deceptive attractors, certainly when the population size increases. When the population is small, the evolutionary algorithm might get lucky and find some good building blocks in most relatively fit strings. Furthermore, the solid line in the upper graph is the predictive function we discussed earlier. If it is to denote a good prediction for the population size, experimental results should show that the actual results must be above the solid line. The dotted line is one-point crossover and the dashed line is two-point crossover in the upper graph. In the lower graph, the solid line is one-point crossover, the dotted line is two-point crossover and the dashed line is uniform crossover. We can deduce that the theory seems to be rather sound for one-point crossover, but the two-point crossover operator is rather destructive after all as it seems to make the genetic algorithm perform worse and even below the predicted value.

4.3.4 Advanced: 1X-GA vs. UX-GA vs. MIMIC vs. FDA

Finally, as a more advanced issue, we wish to install different types of algorithms in the *EA Visualizer*. The problem here is not to see how a new fitness function can be easily explored through different algorithms, but how very differing selection and recombination techniques can still be tested at the same time. To this end, we wish to test two quite the same algorithms and two different algorithms. The main difference between the algorithms is the type of selection and recombination and especially the combination of instances for these two components. The two quite similar algorithms are a standard one-point crossover GA and a standard uniform-crossover GA that we already encountered in section 4.3.2. The other two algorithms are the original MIMIC and FDA that we introduced and explained

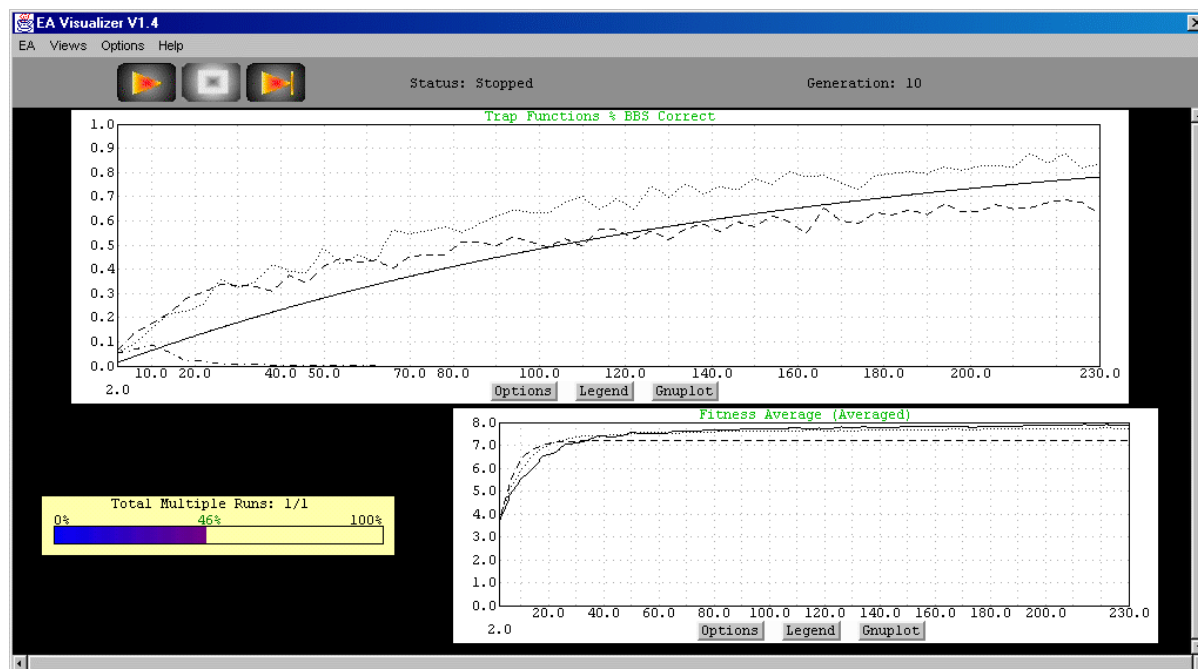


Figure 4.41: The multiple runs being run by the *EA Visualizer*.

for how to install in the *EA Visualizer* in section 3.4.5. This multiple runs example is in some sense a direct expansion to the single runs example from that section 3.4.5.

As the previous examples should by now have presented every aspect of installing and running multiple runs EAs in the *EA Visualizer*, we go swiftly over how the test we wish to run in this section can be run by selecting the right instances and such. Also, we present the results that were found through GNU PLOT images that were made by running GNU PLOT on the datasets that originated from the graphs in the *EA Visualizer*.

First, we must once again find out what the selection–recombination strategy is that all the algorithms that participate in the test make use of. In section 3.4.5 we already presented in tabular form the way in which MIMIC and FDA are to be installed in the *EA Visualizer*. The one–point and uniform crossover operators for the standard like GA have been described earlier already, particularly in section 4.3.2. Here, we present the instances for such a more or less standard genetic algorithm (with one point crossover) again, but now in the style of section 3.4.5:

Component	Instance
<i>Genotype</i>	Binary String
<i>Similarity</i>	Any
<i>Fitness Function</i>	Any
<i>Recombinator</i>	Binary String - One Point Crossover
<i>Mutator</i>	No Mutation
<i>Before Selector</i>	Tournament Selection
<i>After Selector</i>	No Selection (Select All)
<i>Mater</i>	Simple Mater
<i>Replacer</i>	New Offspring Only
<i>Terminator</i>	All Equal Genomes And Maximum Generations
<i>Hybrid Searcher</i>	Any
<i>Population</i>	Vector Population
<i>PRNG</i>	Any

The way in which we are going to setup the examples in this section is a way in which algorithms can *always* be combined when they share the *same* dependency imposing component instances. This is by selecting four instances for every other component given the instances for the dependency imposing components. It is clear that in such a case, if the instances for all other components are linked, the algorithms are always run correctly.

First, we press the F2 key to select that we want to create a new multiple runs evolutionary algorithm. We then select to have as dependency imposing components the binary string genotype and the trap functions fitness function. We shall use this fitness function again as we are able to specify exactly how to use FDA to perfectly match the distribution required to solve the problem as was already noted by Bosman and Thierens [7]. The instance for the *Population* component is again taken to be the **Vector Population**. The instance for the *Similarity* component may be chosen freely without implications. The resulting interface is depicted in figure 4.42.

After pressing the *Apply* button, the interface for the settings for multiple runs appears. We shall go over the most important aspects for entered these settings. The *Genotype* parameters required nothing more than to set the string length to 50 bits. The parameters for the *Population* are to let the population size grow from 50 individuals to 500 individuals with a step size of 50 individuals. The parameters for the *Fitness Function* are more extensive and are depicted in figure 4.43. We want to test a multiple of signal values for the trap functions, so we set the signal to increase from 0.2 to 0.8 with a stepsize of 0.2. The amount of subproblems always remains 10, whereas the subproblem length remains at 5 bits. The scaling is set to be uniform as can be seen in figure 4.43.

Next, we go over the variable components, starting with the *Recombinator* component. Just as all other variable components, this component is set to have four instances, namely the one point crossover and uniform crossover operators for the GA and of course the

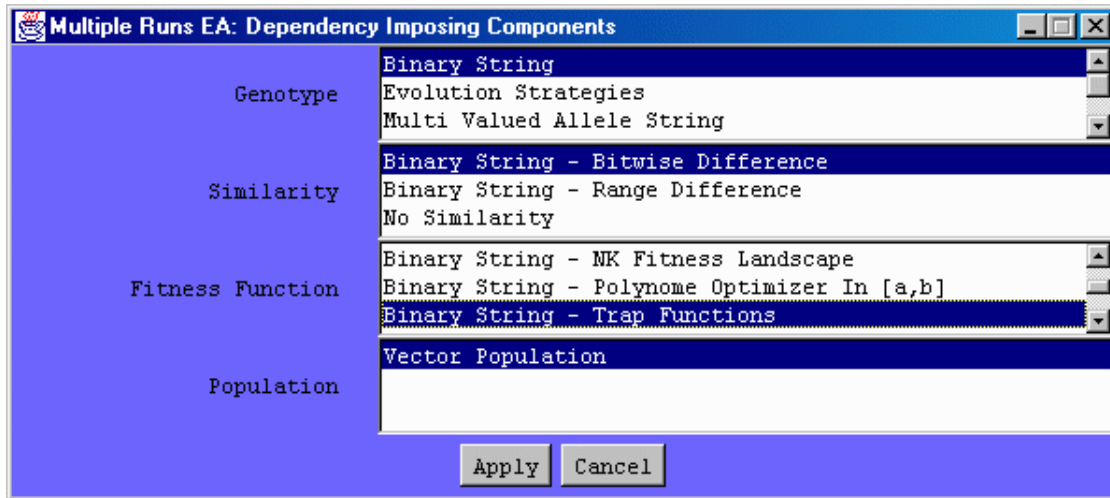


Figure 4.42: The dependency imposing components in the advanced example.

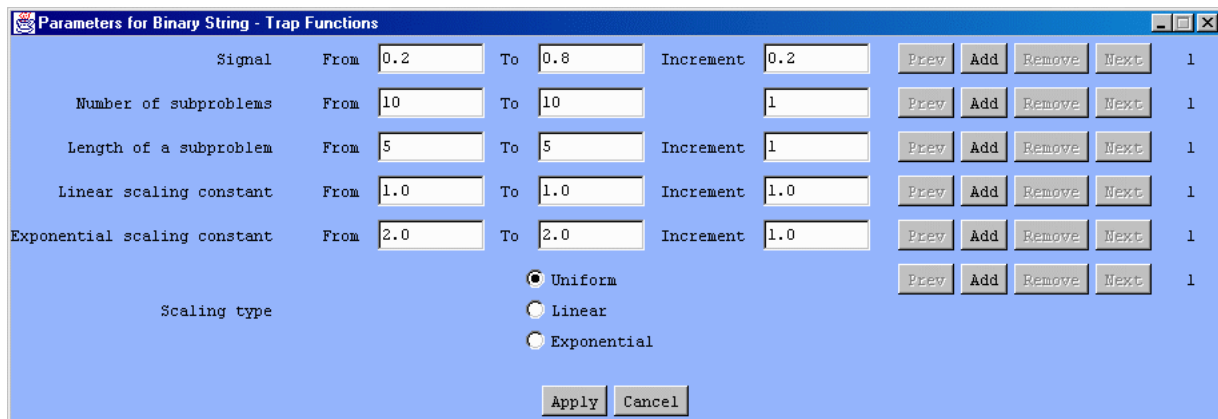


Figure 4.43: The parameters for the trap functions in the advanced example.

Figure 4.44: The parameters for MIMIC in the advanced example.

Figure 4.45: The parameters for FDA in the advanced example.

MIMIC and FDA recombination operators. The parameters for one point and uniform crossover are that they should have two offspring genomes. The parameters for MIMIC are slightly more involved. We must however specify the amount of samples this recombinator will produce. In the standard description of MIMIC [4], this amount is specified by the N -th percentile concept. This means that the amount of samples that are used to update the distribution is $(N \cdot 100)\%$ of all samples each generation. We set N to 0.2 as displayed in figure 4.44. The parameters for *FDA* that we use in this example is also merely one parameter, namely again the amount of samples to generate each generation. Here however, we must specify this amount directly through an integer number. As the strategy of FDA is to replace all of the population with the newly generated samples (with the exception of the best sample of the previous generation), we set the amount of new samples for FDA to increase from 50 to 500 with steps of 50, just as we did for the *Population* instance. This is depicted in figure 4.45.

In all four algorithms, we wish to *not* use any mutation. To this end, we specify to have four equal instances for the *Mutator* component. All of these components are the `No Mutation` instance.

The *Before Selector* for the GAs is clearly the `Tournament Selection` instance. Therefore, we add to of such instances and set the selection size for these components to increase from 50 to 500 with steps of 50. For MIMIC, which is the third algorithm we are installing, no samples are selected on beforehand. All samples are used to generate new offspring, where

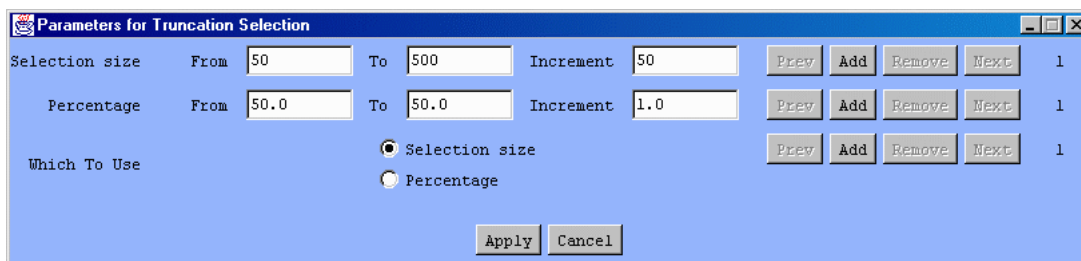


Figure 4.46: The parameters for the *Before Selector* for FDA in the advanced example.

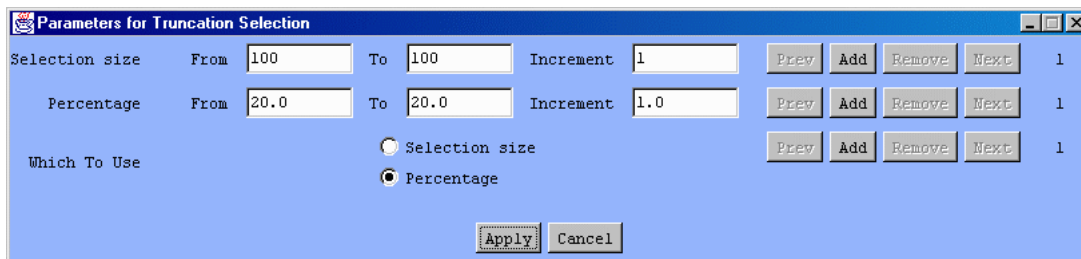


Figure 4.47: The parameters for the *After Selector* for MIMIC in the advanced example.

the samples that are used equal exactly that N -th percentile we were referring to earlier. This means that the third instance for the *Before Selector* is a **No Selection (Select All)** instance. For the FDA recombination operator, we also wish to use only 20% of all samples, but in this case the 20% refers to the current population, so on beforehand the best 20% of all genomes is to be selected. This is done by using a **Truncation Selection** instance that selects just that percentage as can be seen in figure 4.46.

The *After Selector* for the GAs is of course the **No Selection (Select All)** instance because after recombination no selection is applied. The offspring genomes are taken to be the population of the next generation through usage of the *Replacer* component. For MIMIC, the offspring genomes will be added to the population. To once again create a population of the right size, we require truncation selection to truncate the population to the required size, so we set the parameters for this third *After Selector* instance to select 50 to 500 genomes with a stepsize of 50 just as we did for the population size. This is depicted in figure 4.47.

The *Mater* is a simple component in this example. The GA variants require to mate the genomes in groups of two because the crossover operators require two parents. The other two algorithms base their recombination on all of the selected genomes, so all genomes need to be mated into one large group in this case. This means that the first two instances for the *Mater* are to be **Simple Mater** instances and the second two instances need to be **Mate all genomes in one group** instances. The parameters for the simple maters are

clearly to have a grouping size of two. Note that the use of the tournament selection for the GA versions justifies the use of the simple maters.

As we noted earlier, the GA variants use all of the offspring genomes as the genomes of the next generation. To this end, the first two instances for the *Replacer* component are to be the **New Offspring Only** instances. The MIMIC algorithm uses truncation selection afterwards to select the genomes that are to be part of population of the next generation and adds the offspring to the population. Therefore, the third instance for the *Replacer* is the **Add Offspring To Population** instance. Finally, we just noted that FDA uses the offspring genomes and only the single best offspring genome of the previous generation and so the fourth instance for the *Replacer* component is the **Best Sample Elitist Replacing** component.

Because of the fact that we know that GAs are sensitive to genetic drift (and because in this problem all alleles define some part of the fitness function), we use the **All Equal Genomes** instance as a termination condition for the GA variants. For the other two variants we terminate on this ground or on the ground that a maximum of 1000 generations has been reached by using the **All Equal Genomes And Maximum Generations** instance.

Just as we allow no mutation, we don't use any form of hybrid search. For this problem, such searching doesn't really apply. This means that we have four the exact same instances for the *Hybrid Searcher* component, namely the **No Hybrid Search** instance.

The final component that completes the interface with respect to components as can be seen in figure 4.48, is the *PRNG* component. In all cases we use the standard JAVA prng and set that the system is to pick a seed at runtime. The recombination probability is always set to 1 whereas the mutation probability is always set to 0. The resulting interface is depicted in figure 4.48. Note that every component has exactly *four* instances that together define four different algorithms once we have specified the links, which we will do next.

The links we require to specify are twofold. First of all, it is clear that in order to create the four different evolutionary algorithms, we require to link the instances for each individual component to the instances of all other components. This will cause instance i for component j to be selected only with instance i for component k for every j and k . This can be done by selecting one by one the components from the top left list in the links interface and by subsequently pressing the *Add Link* button underneath this list. The result of defining this first link can be seen in figure 4.49.

The second link we require to specify is that the selection sizes for the different instances of selection parts in the algorithms need to be linked to the population size. In addition however, the amount of samples generated by FDA needs to be linked to this parameter as well. The amount of samples generated by MIMIC needs not to be linked, as this amount is determined by a percentage and remains equal over all runs. The amount of samples that FDA generates however is a number and must equal the population size at any time. The second link therefore consists of five entries (can you guess which ones exactly?). The

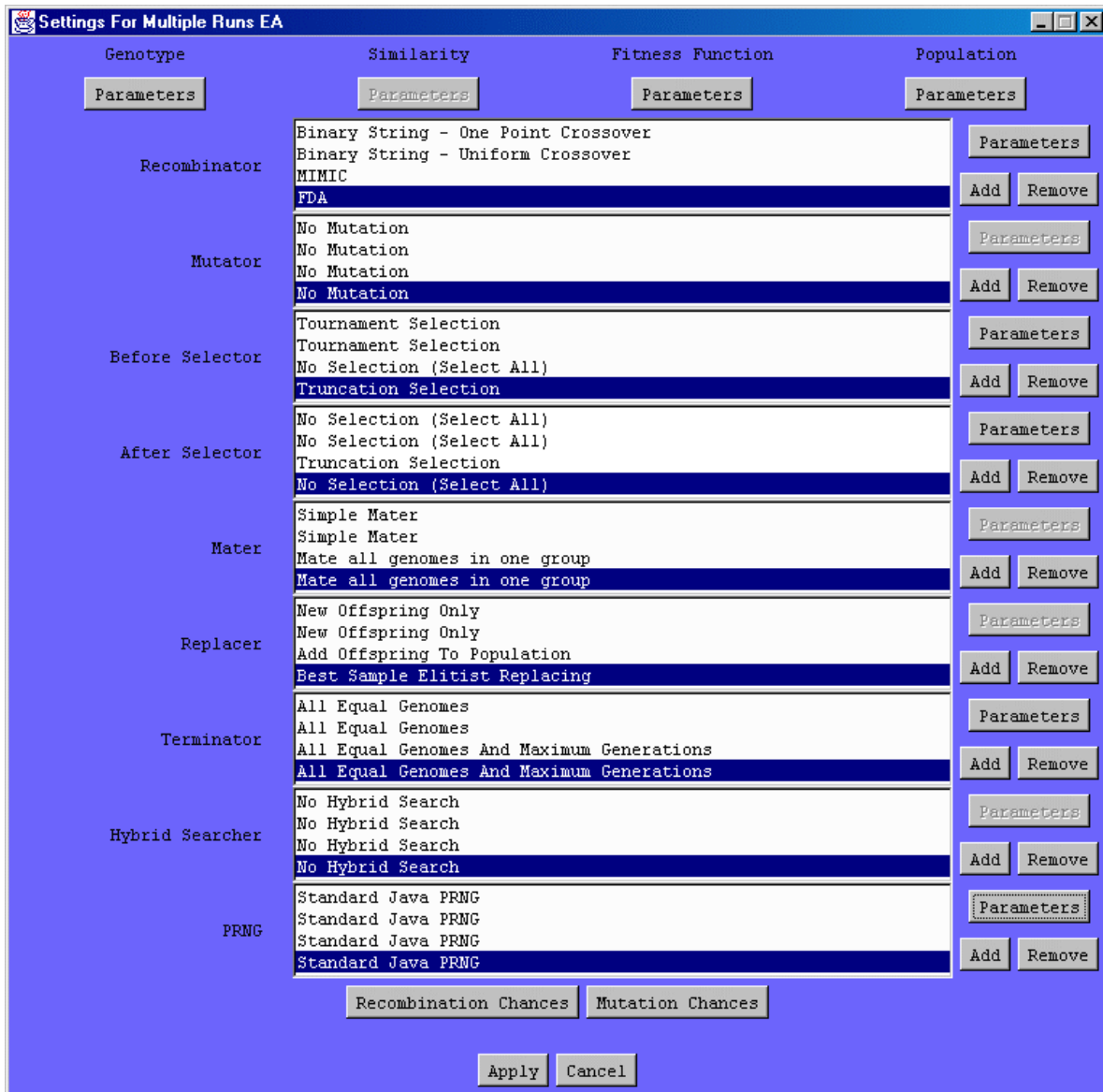


Figure 4.48: The resulting multiple runs interface after setting all parameters.

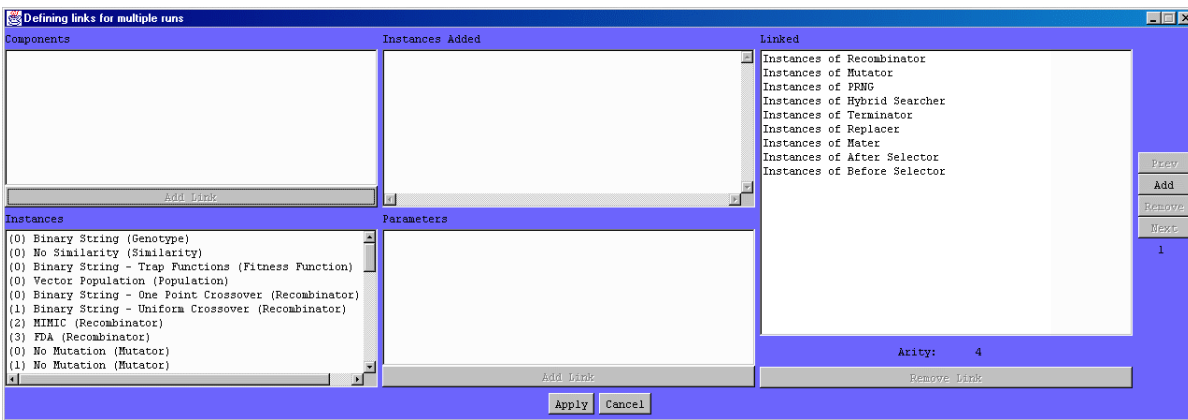


Figure 4.49: The first link for the the advanced example.

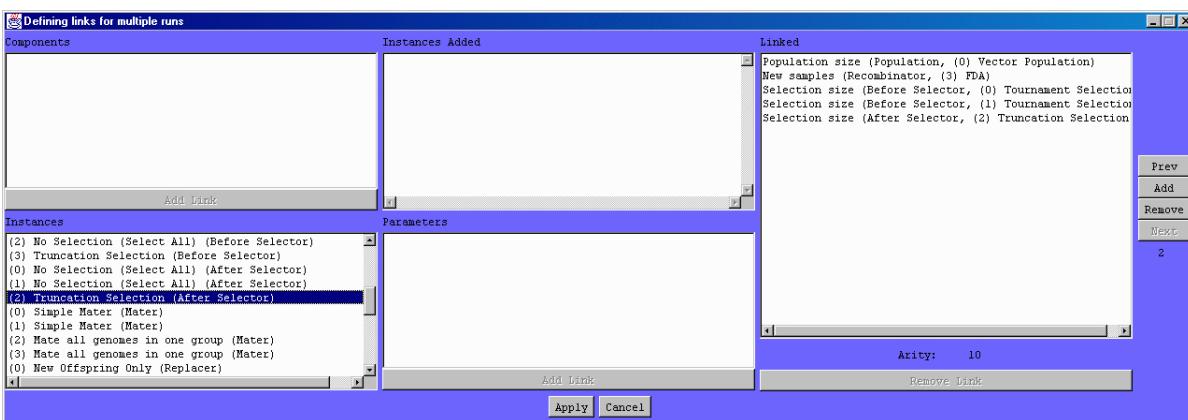


Figure 4.50: The second link for the the advanced example.

link can be specified after having pressed the *Add* button on the far right of the interface. The final result that can be seen in the interface is depicted in figure 4.50.

Now we are ready to actually run the algorithms and find the results. To this end, we place five more or less standard views on the screen in the *EA Visualizer* and one additional one for the problem at hand. Two of the five standard views are based on the best fitness upon termination. The values that are actually displayed are the average and standard deviation over the 25 runs. In a similar fashion, we wish to display the fitness evaluations. To be exact, we visualize the average and standard deviation over 25 runs. The fifth standard view is the progress indicator that tells us how far the system has progressed in running the different algorithms. The additional view that tells us neatly how the problem at hand has been solved, is the view that displays the percentile of building blocks correct upon termination in the entire population. This should of course move towards 1 so that the optimum is always found.

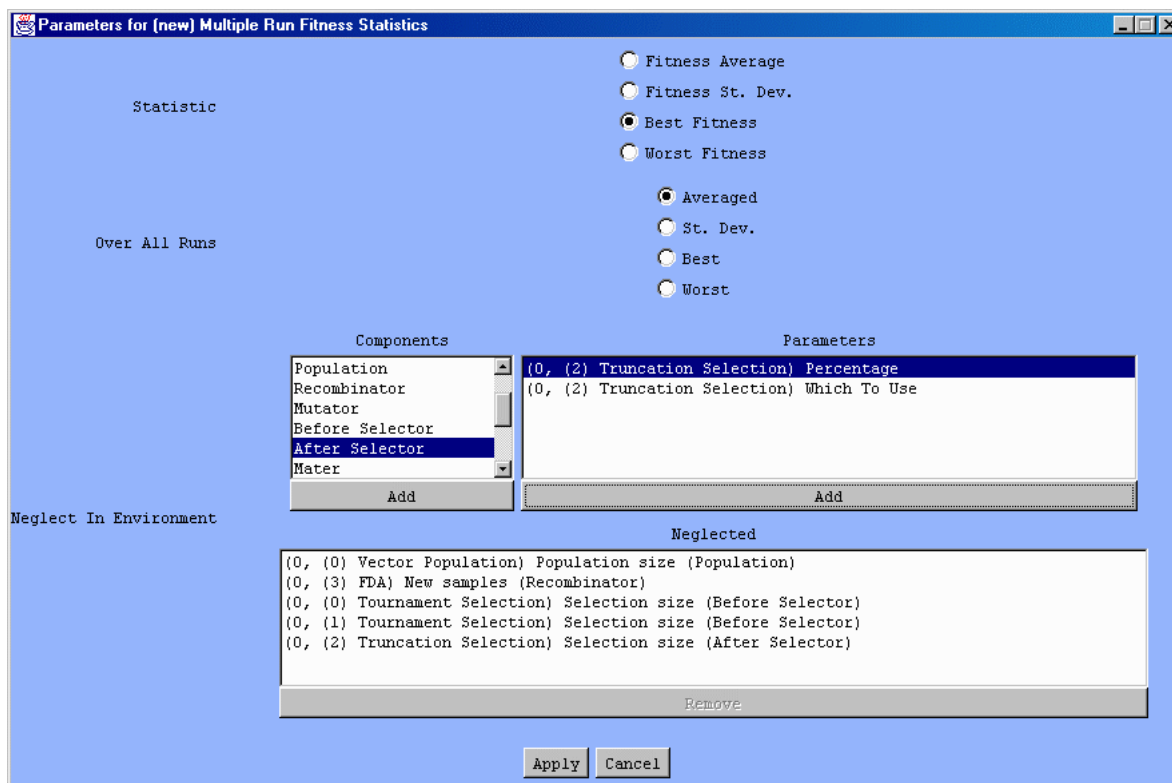


Figure 4.51: The items to neglect in the environment for the statistics views.

Before showing the results found, we first note that all views except for the progress indicator are views that show their statistical value on the y axis with on the x axis the population size. To properly identify the different evolutionary algorithms and display them as individual lines (graph entries) in the resulting graphs, the parameters that are to be neglected have again to be specified. Because the population size is to be neglected of course, all parameters that are linked to it must be neglected as well. This needs to be specified when adding a new view. For the average value statistic over the best values over the 25 runs this is shown in figure 4.51.

We conclude this example by presenting the results. We only present the results here in graphs as they resulted from GNPLOT. The datasets for GNPLOT were created by running the system with the six views we just mentioned and by pressing the *Gnuplot* button for each graph view after having altered the title for the graphs using the *Options* button for each graph. The resulting datafiles were all plotted by running GNPLOT on the plotfiles and the results are shown in figures 4.52 through to 4.56. Note that the graphs contain far too many lines to still be readable, but it is only to show the versatility of the system. Also note that using GNPLOT and the separate datafiles, the results can be easily combined in another manner so as to create different graphs that *do* represent the test results in proper manner.

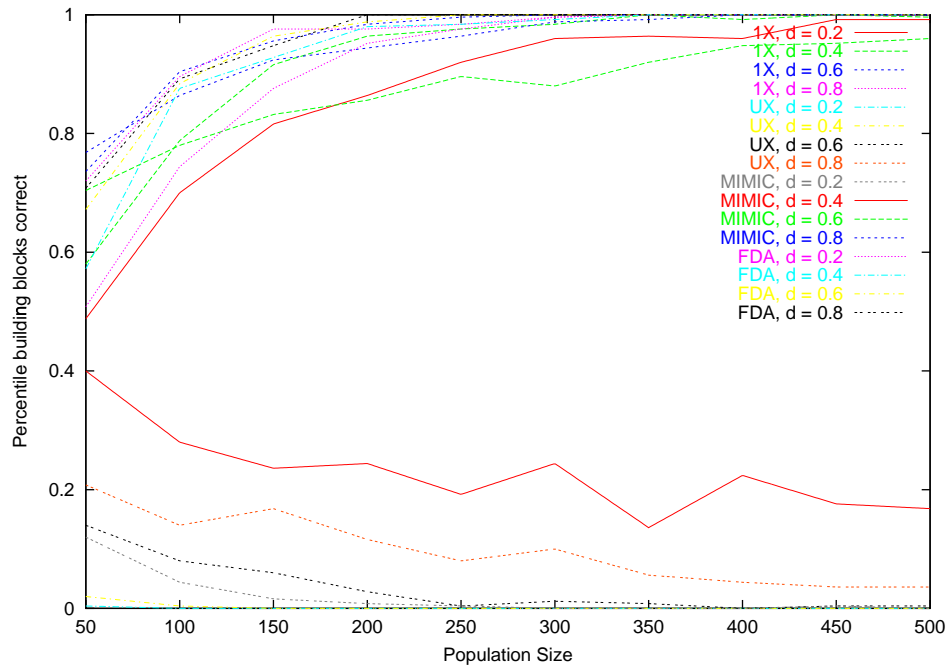


Figure 4.52: The percentile building blocks correct.

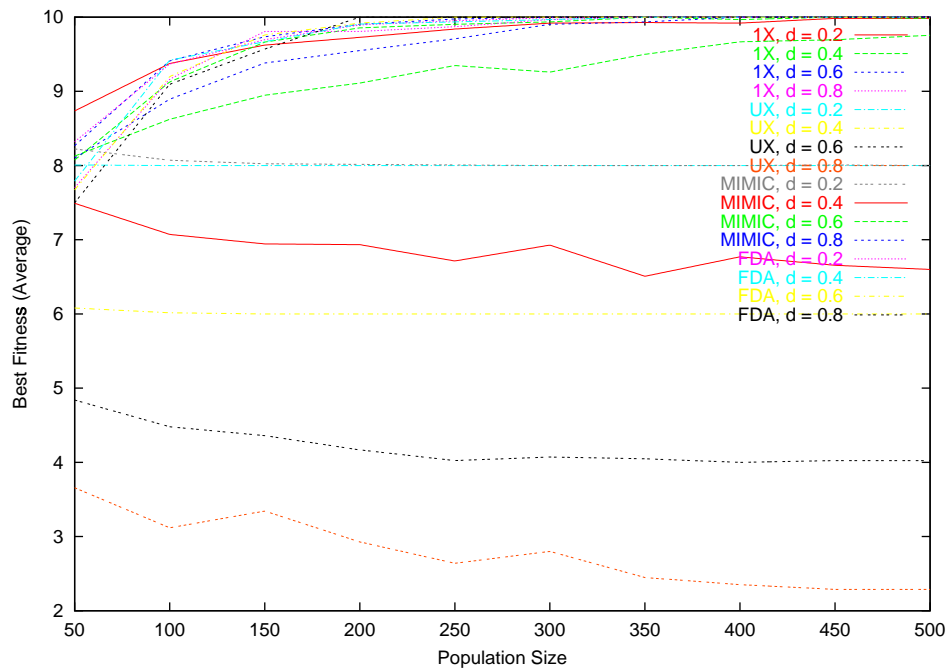


Figure 4.53: The average best fitness value over 25 runs.

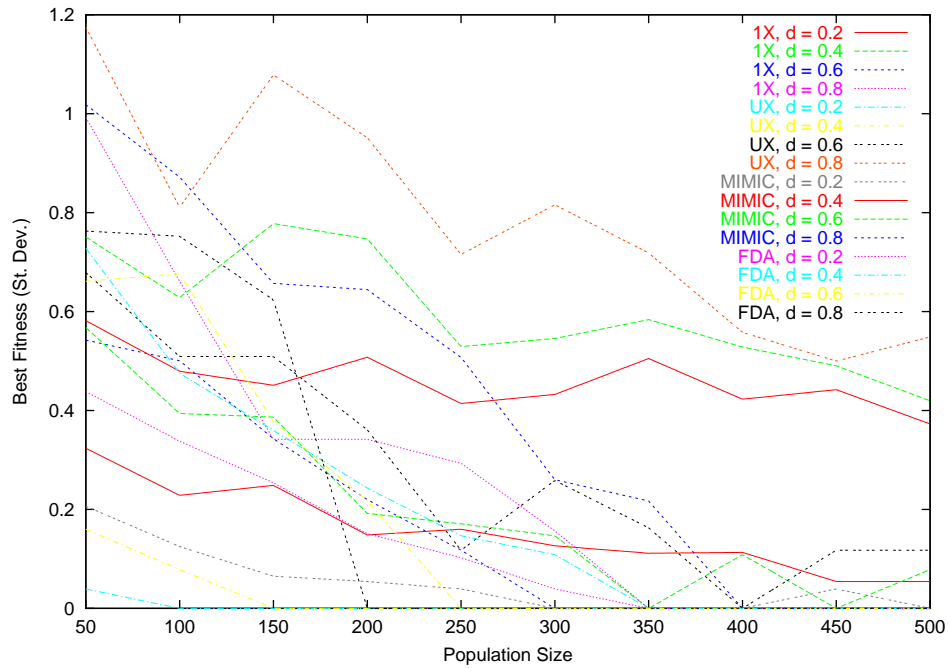


Figure 4.54: The standard deviation over the best fitness values over 25 runs.

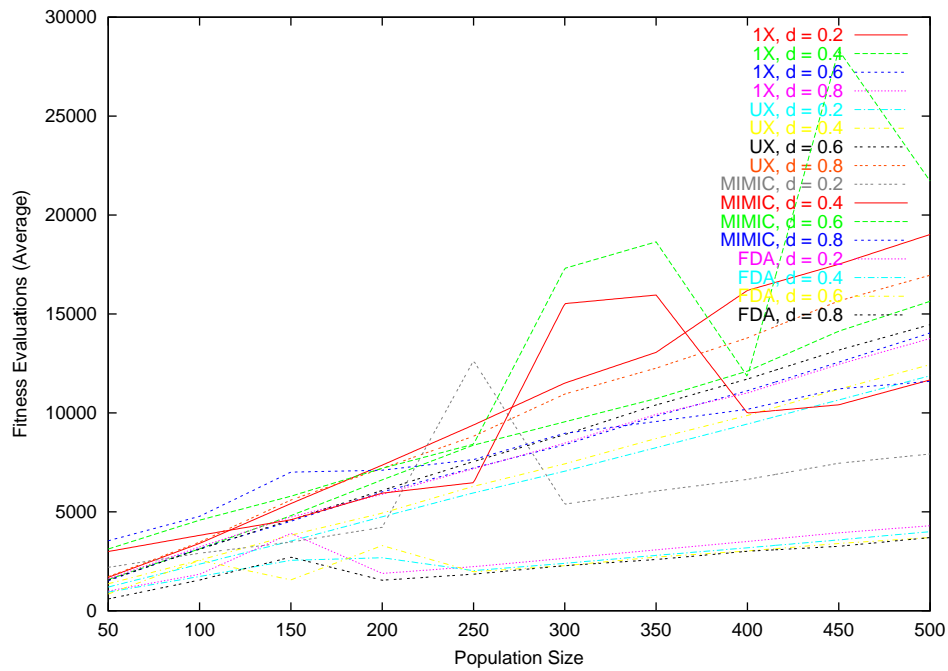


Figure 4.55: The average fitness evaluations over 25 runs.

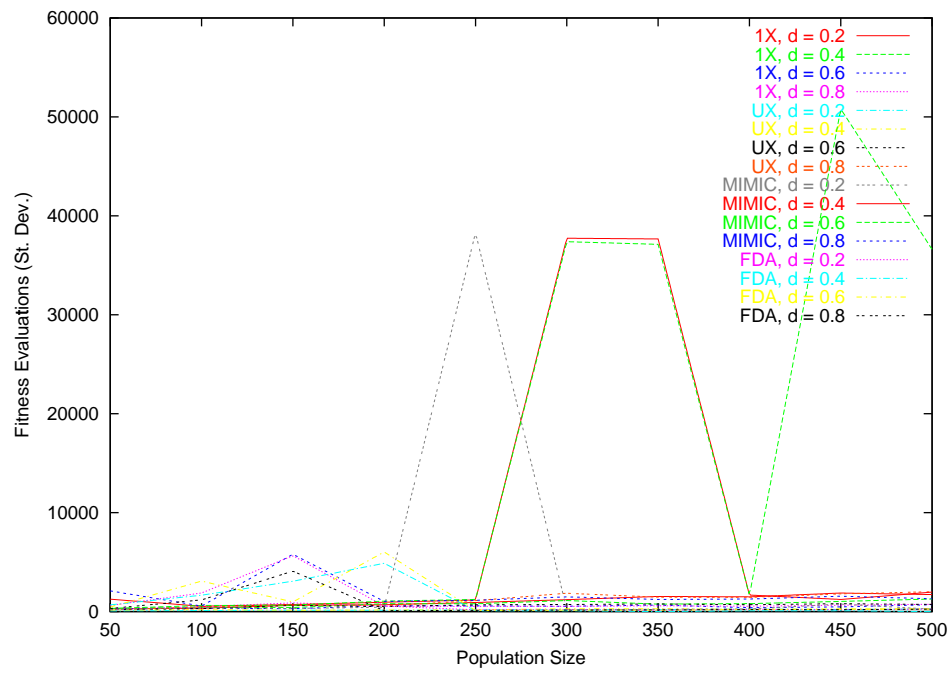


Figure 4.56: The standard deviation over the fitness evaluations over 25 runs.

Chapter 5

Using the editor

In this chapter, the editor version of the system is described. Using the *EA Visualizer Editor* allows you to expand the system with as little system details as possible for you to be concerned with. Through the editor for instance new instances for all components can be created that will directly be available to you in the system once restarted. This argument is equally valid for views as well as parameter components, general utilities and view utilities. For each of such parts in the system, the editor can be used to create new JAVA classes that are compiled into the system and incorporated into the system on your command. The only thing that is required is that you generate the class file with the functionality you wish to have. The editor will take care of adding your part to the system in the right way.

Before starting off with describing the editor, we wish to point out that we refrain as much as possible from giving technical details on how operations are established. For such information, the interested reader is referred to different readings on the *EA Visualizer* [5, 6]. Only the details that are of importance to use the editor properly are incorporated here and will be introduced when required. Most of this chapter is of a descriptive kind, meaning that all possible operations in the editor are described, such as menu items and different GUIs. Because editing can be very versatile and a description of how which parts work is enough for the user to expand the *EA Visualizer*, to keep up the operational like description. To give the user a good idea of how to actually expand the system, an example is described in which a new *Recombinator* is created and added to the system.

5.1 Starting up the editor

In the first chapter, we already described what happens when you start up the *EA Visualizer* system. When starting up the editor, the startup sequence is slightly different. Indeed, once again a very similar startup frame becomes visible once the startup is initialized. It

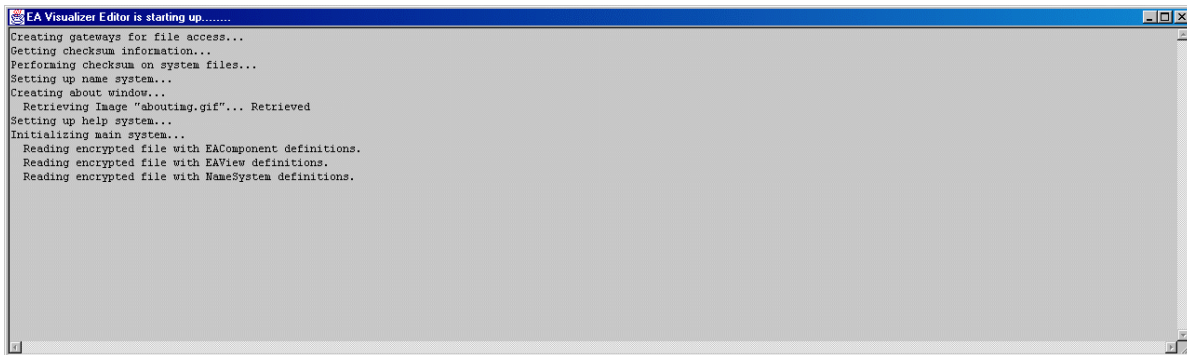


Figure 5.1: Startup frame at the end of editor initialization.

quickly becomes evident however that the information in this frame is different from when the running system was starting up. The list of operations is shorter and is most different when the main system is initialized. This is the last step in starting up the system as we already saw when starting up the *EA Visualizer* itself. In the editor however, once the main system is being initialized, three more times a encryption/decryption progress pop-up window is displayed. Next to performing a checksum on the system files for which encrypted information is required, definitions files that have been stored encrypted, have to be read. These definitions files tell the editor about the internal structure of the instances for the components, the actual views and the names of all of these parts of the system. This information is required to regenerate some parts of the *EA Visualizer* once the system is updated with the information you wish to add. Also this information itself is at such a point of course updated. It is thus therefore that during the startup of the editor, three more times decryption of data takes place. Otherwise, the startup of the editor is very similar to the startup of the system, which we already saw with possible errors during startup in the first chapter. In figure 5.1 the startup frame at the end of the initialization process is displayed, showing the intermediate steps that have all successfully been executed during startup.

5.2 The main GUI

Once the editor has started, the main GUI is shown on the full desktop area as shown in figure 5.2. There is only one frame with a sandy yellow background for a very large textarea. It is this textarea that is used to report system information to the user on the events that have taken place in the editor. For starters, the user is welcomed to the editor of the *EA Visualizer*. Just as in the running system, more urgent errors are directly displayed to the user in popup windows, meaning that the editor textarea is mostly used for warnings or progress information. The editor in itself seems to be not much more

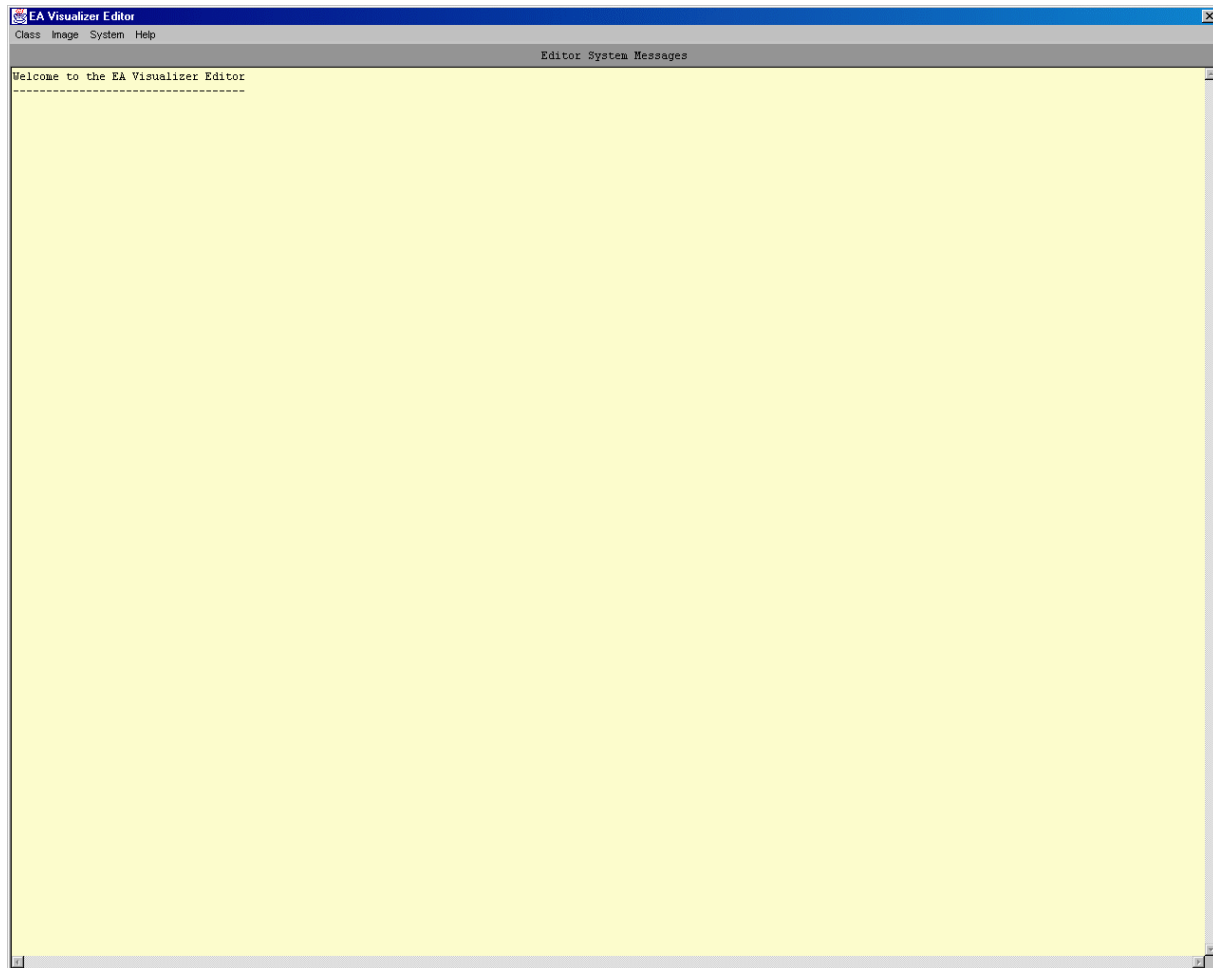


Figure 5.2: The main GUI for the editor right after startup.

than this textarea and indeed the frame contains nothing more next to this textarea than a menubar and a bar above the textarea that states that the textarea below indeed is used for editor messages to the user. As the menubar is the start of all operations in the editor, we have nothing more to describe about the main GUI of the editor but the menubar and more specifically all selectable menuoptions, which is what we do in the remainder of this section.

- *Class*. This is the main menu for manipulation of items in the system. New classes can be generated or removed as well as can current ones be edited or browsed. This menu corresponds to the **File** menu found in most standard applications and the **EA** menu in the *EA Visualizer* in non-editor mode.
 - *Add*. Alternatively, the F2 key can be pressed on the keyboard to activate this menu option. It displays a class selector interface in order to select in what

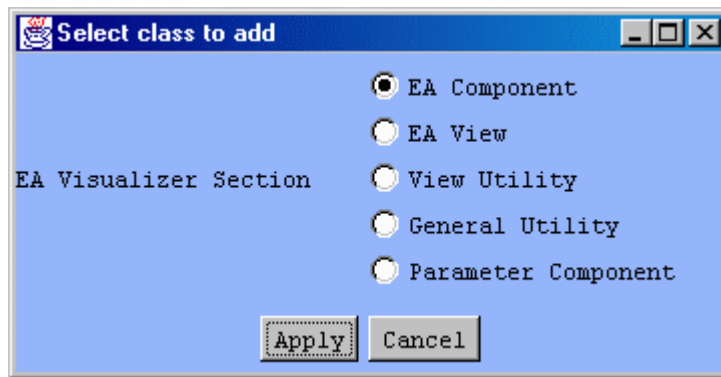


Figure 5.3: Selecting the class type.

category a new class is to be added. This interface is depicted in figure 5.3. After having specified the category and possible subcategory, the name for the class is to be entered. This is the JAVA classname, not the name that will be used in the system. After having entered this, the system automatically generates a general frame class in which the methods that are to be coded are already present. The editing itself is done with a editor for classes as we shall see more extensively in section 5.5. Having added a class causes the system to be out of synchronization. This can be set right by selecting menu option *Synchronize* as we shall see shortly. More on this menu option can be found in section 5.4.

- *Remove*. Alternatively, the F3 key can be pressed on the keyboard to activate this menu option. It displays a class selector interface in order to select what class to remove from the system. This is the same interface as before and can be seen in figure 5.3. After the category and possible subcategory have been selected, the class to be removed is to be selected. If there are no other classes than system classes, the list to select from will be empty. After having selected the class to be removed and acknowledging to actually remove the class, the class is removed from the internal administration of the system. Having added a class causes the system to be out of synchronization. This can be set right by selecting menu option *Synchronize* as we shall see shortly. More on this menu option can be found in section 5.4.
- *Edit*. Alternatively, the F4 key can be pressed on the keyboard to activate this menu option. It displays a class selector interface in order to select what class is to be edited. This is again the same interface as before and can still be seen in figure 5.3. After the category and possible subcategory has been selected, the class to be edited is to be selected. If there are no other classes than system classes, the list to select from will be empty. After having selected the class to be edited, a class editor window appears in which the class properties and code contents can be entered. More on this menu option can be found in section 5.5.

- *Browse*. Alternatively, the F5 key can be pressed on the keyboard to activate this menu option. It displays a class selector interface in order to select what class is to be browsed. This is again the same interface as before and can be seen in figure 5.3. After the category and possible subcategory has been selected, the class to be browsed is to be selected. If there are no other classes than system classes, the list to select from will be empty. After having selected the class to be browsed, a class browser window appears in which the class properties and code contents can be browsed. More on this menu option can be found in section 5.5.
- *Exit*. Alternatively, the F10 key can be pressed on the keyboard to activate this menu option. This opens a popup window in which a question is posed whether you really want to quit the system. Upon a confirmation, the system is terminated. Upon a negative response, the system resumes its tasks.
- *Image*. Through this menu, images can be added and removed in a way so that the system will know where to find them during runtime. This is useful for instance when adding images for the help system, because the help system will search for its images in the directory where they are placed through the menu options here described.
 - *Add*. Alternatively, the F6 key can be pressed on the keyboard to activate this menu option. It displays an interface through which images can be added to the system. The interface is displayed in figure 5.4. By pressing the *Open* button, an image can be selected to be loaded. The image is read using default JAVA components, so GIF and JPG files can for instance be read. If the selected file cannot be read because of an error or because the format is not understood, a timeout occurs, resulting in that no image is loaded. If the file *is* readable and in a format that can be understood, the loaded image is displayed in the interface for adding images as can be seen in figure 5.4. At this point, the *Save* button is enabled and can be pressed to save the image in the right directory for the system to find and use it during runtime. Saving the image comes down to specifying a filename and pressing the *Apply* button in a trivial interface. By pressing the *Close* button, the interface is dismissed.
 - *Remove*. Alternatively, the F7 key can be pressed on the keyboard to activate this menu option. It displays an interface containing a list with all files that can be removed (non system images). By pressing the *Remove* button and after acknowledging you actually want to remove the selected file, the file is removed from the system and from the list you can select the images from. Alternatively, you can press the *Display* button, which will trigger an external display to appear in which the image is displayed so you can check on beforehand whether the image you selected is one you really wish to remove. By pressing the *Close* button, the interface is dismissed.
- *System*. In this menu, items can be selected that will allow you to execute system

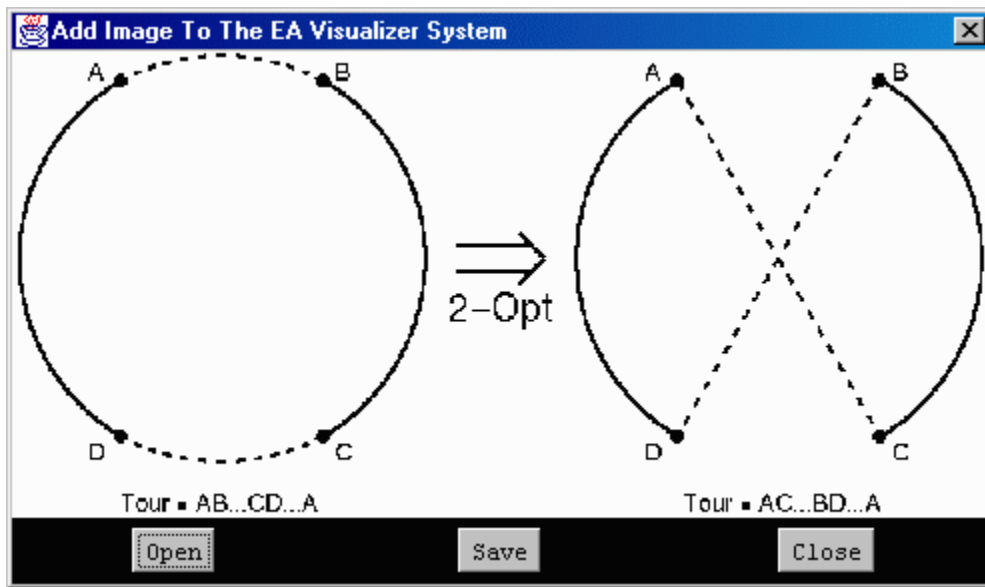


Figure 5.4: Adding an image to the *EA Visualizer* system.

related operations. This means for instance that the *EA Visualizer* with your latest changes can be started. Most importantly, through this menu, the system can be synchronized with your latest additions so that they are *automatically* incorporated in the system, requiring no further actions on your behalf.

- *Run EA Visualizer.* Alternatively, the F8 key can be pressed on the keyboard to activate this menu option. It starts the *EA Visualizer* system as a separate process, using the current settings as they are (settings since the last synchronization action) and the code as you of course last compiled it.
- *Clear Messages.* Alternatively, the F9 key can be pressed on the keyboard to activate this menu option. By selecting this menu option, the text in the main body of the editor is cleared. This is useful when a lot of messages have already been generated which are of no importance any longer and clutter your view of what is happening in the editor.
- *Synchronize.* Whenever this option is enabled, this means that the information that is currently known to the *EA Visualizer* Editor has not been written to the system files and classes so that the entire system is aware of the changes. Selecting this menu option flushes the current system details to disk in the right directions, regenerating classes and definitions files. As the synchronization of the system takes some time, this option does not have to be performed for every minor detail, but ***must not be forgotten*** to carry out before leaving the editor. If the editor is interrupted anyhow without synchronization, the system could be unstable and inconsistent when running the *EAVisualizer* a next time. In any which way, your entered information with the exception of your JAVA code

will be lost (such as name changes and parameters information) if you have not performed a synchronization operation. The course of action taken by the editor is shown in the textarea that is the body of the editor and the user is notified of success or failure of the different steps taken during synchronization. More on this can be found in section 5.4.

- *Help*. In this menu, help information on the *EA Visualizer* can be found. The self contained help system can be opened and general information on the system can be found. All information here is identical to the information found in the *EA Visualizer* runtime system as described in section 1.
 - *Help Index*. Selecting this item opens the help system on the index page. The help system is described in section 1.3.
 - *About EA Visualizer*. Selecting this item opens information on the *EA Visualizer* such as version, copyright and author as can still be seen in figure 1.11. Pressing the *Close* button at the bottom of the interface dismisses the information and allows you to operate the system again as displaying the information blocks your input to the system.

5.3 About parameter components

Before going into details about how to use the editor version of the system and present examples of how to do so, at this point we should first discuss the so called *parameter components*. We have seen the parameter components in a way when we presented the single runs and multiple runs EAs in sections 3 and 4 respectively. The way in which we were then confronted with the so called parameter components was through the entering of parameter values for the parameters for different instances for evolutionary components. In the former of this tutorial we have already seen that we have both components and views. The components make up the evolutionary process and the views define what the visualization of the EA will be. Next to these two important classes, a third type of class that can be separately edited and added through the editor and is of importance is the parameter component class. We shall first provide some insight into to what extent the parameter component class is used in the *EA Visualizer* and subsequently show how it is implemented in the system.

Many parts in evolutionary algorithms are parameterized. If we regard for instance selection strategies, the amount of individuals to select is a parameter. For a method such as tournament selection, the tournament size is another parameter. Other examples of parameters are string length for binary string genomes, fitness function parameters, recombination probability and prng seed. It is a common known issue to tweak parameters for certain problems or to create so called *robust* algorithms that do not depend very heavily on fluctuations in actual parameter values. At this point, we should first separate the

notion of *parameter* from *parameter value*. A parameter is the abstract concept of the type of input we desire. It can be compared to a typed variable in programming languages. A parameter value is a value for the parameter and can thus be compared for a value in a computer language that can be assigned to a variable. In terms of EAs, we could for instance as a parameter take the string length for binary string genomes and as a parameter value the value 50, meaning a string of length 50. Another example is the parameter that specifies we want a set of cities for a TSP, whereas the parameter value is the actual list of coordinates for the cities. These notions are important to consider for the parameter component concept.

As the entire setup of the *EA Visualizer* has been to create a modular structure so that the resulting system is highly expandable, the idea is to separate the parameter structure from the views and the components. We require some structure that views and components can use to specify in a uniform manner the required parameters so that the system can turn to the parameter structure and present the desired parameters to the user separately so that the values can be collected. After having collected the parameter values, these are then transferred to the component instance or view so that the values can be actually used. This dictates the form of the parameter component.

First of all, a parameter component *defines* and holds the structure for the parameter value as we noted above. This is done by taking the user input in some manner (usually a string of characters) and by parsing the input so that only specialized formatted texts are accepted. Alternatively, other input mechanisms than text based can be used, such as by drawing images or listwise selection. This immediately brings us to the second part of a parameter component, which is the user interface side. Next to defining and parsing user input, the parameter component itself determines what it will look like when presented in a window. This can directly be done by using JAVA components. Finally, the resulting parameter value can be returned as an object.

The system can now make use of such parameter components very easily. When the user is required for instance to specify the parameter values for tournament selection, the system looks up what parameter components are required for this *Selector* instance. It then finds out that it requires to use two parameters components that allow for the specification of integer numbers in some range (strictly positive range) for the selection size and tournament size. Subsequently, a window can be created on which amongst other things the designated parameter components can be placed, because the parameter component structure, as we saw above, also tells us what display components to use. Finally, after applying the user input, the parameter components are issues to parse the input and return the actual parameter values and the window is dismissed. The results are gathered in a list of objects and send back to the instance or view for which the parameter values needed to be specified (in this case tournament selection). In this way, the parameter structure is nicely separated from the component and view structure and can be expanded uniformly and separately so that presentation of parameters both internally and externally is uniform and modular.

Finally, it is nice to note that the system can immediately use this to specify parameters for multiple runs. This can directly be done by allowing the user to specify a list of parameter values by presenting the display component for the parameter component along with a list that specifies the how many parameter value we are entering. This can however be quite inconvenient as we can imagine that we desire to test all values between 10 and 500 with steps of 10 for the population size. This will require 49 single specifications of parameter values, which is far too time consuming. To this end, the *EA Visualizer* facilitates in single value parameter components and multiple values parameter components. The single value parameter component is the parameter component we introduced above. The multiple values parameter component is an extension over the single run parameter component in that it can be used to directly specify a multiple of single parameter values. What happens inside the system is that after having specified in a more easy manner a multiple of values, such as a lower boundary, an upper boundary and an increment value for instance for the problem of population sizes just described, the multiple values parameter component is request to generate all single values so that these can be used individually for the creation and setting of parameters for evolutionary component instances.

The implementation of the parameter component structure is found in the creation of the superclass for the abstract parameter component and of subclasses thereof for the single value parameter component and the multiple values parameter component. In the editor, these two different parameter components can both be created. The resulting class information is stored in the system and can be used the next time the *EA Visualizer* system is started. In section 5.6.1 we shall see how the parameter components can be specified for a new instance or view and how the resulting parameter values can be used.

5.4 Adding and removing classes

The *EA Visualizer* is expandable in many ways. The most important of such ways is by adding classes. By adding classes, new functionality is added to the system, making it possible to create new types of EAs at runtime, to view information differently or to have new ways to enter parameter values. In this section we describe what classes can be added and what purpose they serve. After classes have been added to the system, they can be edited. Also, they can be removed from the system, contrary to the classes that are already present when the *EA Visualizer* is first installed. Editing and browsing classes is discussed in section 5.5. Here, we only concern ourselves with adding classes to the system and the consequences thereof.

By selecting *Add* from the *Class* menu in the main window of the editor or by pressing F2, a class can be added to the system. The user is presented with a window in which the class type must first be determined. The first selection that has to be made is the section at top level that concerns the *EA Visualizer*. The corresponding window can be seen in figure 5.5. The **EA Component** selection stands for the components in the decomposition that make up

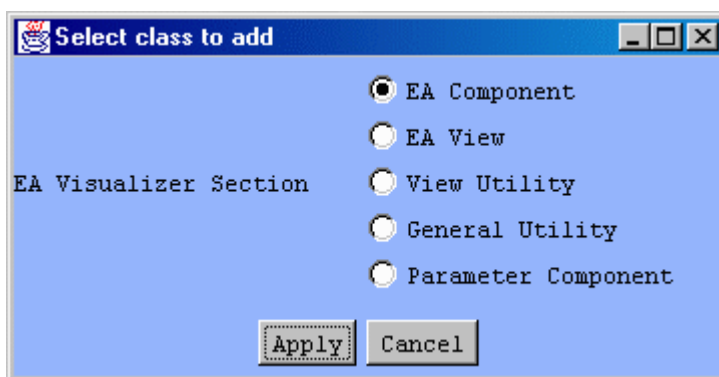


Figure 5.5: Selecting at top level the type of class to add.

the EAs in the *EA Visualizer*, such as *Genotype* and *Recombinator*. The *EA View* selection stands for the views that implement the visualizations, both internally and externally. At the bottom of the list, the *Parameter Component* selection is found. As noted in the previous section, the parameter components are a uniform manner in which parameters can be specified, both in use and in value. Both the single value and multiple values versions can be added to the system. On top of these type of classes, the *View Utility* and the *General Utility* classes can be added. It is common practise to have general utilities that can aid any program. In the *EA Visualizer*, this is no different and a package of such utilities is already present in the system. You can add your own general utilities to the system by selecting to add a class here. Likewise, but in a little bit more detailed manner, you might have visualization primitives or utilities for visualizations that you want to use in many cases in the system. To this end, you can select to add a *View Utility*. These two types of utility classes will end up in the JAVA packages `eavctrl.genutil` and `eavview.viewutil` respectively.

After selecting at top level which type of class to add to the system, a subtype is mostly required to be entered. This is the case for all but the utility classes. Once this is done also, the name for the class needs to be entered. This is the JAVA classname, which must not be mistaken for the actual name that is used in the system. The actual name can be entered when editing the class, as explained in section 5.5.

Once the entering of the classname has been applied, a file is generated at the right location in the file structure of the *EA Visualizer* and for all but the utility classes, information is added to the internal structure for the *EA Visualizer* that is currently stored in the memory of the editor. The file that is generated contains the methods that are *required* to be coded for the class to work. This is because the generated class is a subclass of some *abstract* class that defines what is required from any such class. For example, a new *Recombinator* will be defined as a subclass of class `Recombinator` from package `eavmodel.recombinator` and will be required to define the method `recombine(...)` so that the system can call this method when recombination is required. There are many

such superclasses available and all of them have different methods that need to be defined in their subclasses. More information on this can be found in a more technical report [5] or in the code of the abstract superclasses, where code comment is available to clearly explain what is required. An example is given in section 5.6.1.

After this, the editor returns to ready mode and seemingly nothing has happened. Behind the scenes though, a file has thus been created and the internal structure has been adapted. The information currently stored in memory has to be flushed to disk in the right classes for the *EA Visualizer* in order to use this information at runtime. In other words, the runtime system has to know that the new class is available. Just as much so, when class properties are edited, such as the actual name for a class in the system, the internal structure is changed and this needs to be written to disk in order to become permanent. To this end, the menu item *Synchronize* under the *System* menu is available. This item is selectable whenever the internal structure has changed. You will require to select this menu option at the least once before leaving the editor once it has become enabled. You might not want to select to perform synchronization each and every time the menu option becomes enabled, because when you start editing classes, many changes could very well be required, which can all be done one after the other, just as long as synchronization is performed in the end. If you abort running the system at any time before synchronization, all edited data is lost. Once synchronization is started, the editor keeps you updated on the progress in the textarea that is the main body of the editor window. A successful synchronization will result in the messages as displayed in figure 5.6.

Finally, it lies in the line of expectation that because classes can be *added*, they can also be *removed*. This can indeed be done by pressing the F3 key or by selecting *Remove* from the *Class* menu. This will once again display the class selector interface from figure 5.5, but now with a slightly different title. Once the full type of class to remove is selected, the classes that can be removed are displayed. All the classes that are provided as system classes upon the installation are removed from this list. This means you can only remove classes that you have added to the list yourself. Removing a class from the system does not mean the file is actually removed, because you might want to keep it. For instance you might want to rename the class itself, which requires to remove the class from the system and then to add a new class to which the old code has to be copied. The internal structure is once again updated, but this time by removing the associated information. After this, synchronization is again required to actually establish removal of the class from the system.

Adding and removing classes is of fundamental importance because new items can be introduced to the system and redundant ones removed. The next step is to actually fill the new classes with the desired functionality and parameters such as name and dependencies. This completes adding new classes to the system and is described in the next section. The full trajectory of adding new information to the system in the form of a new instance for a component, namely a new recombination strategy, is presented in an example in section 5.6.1.

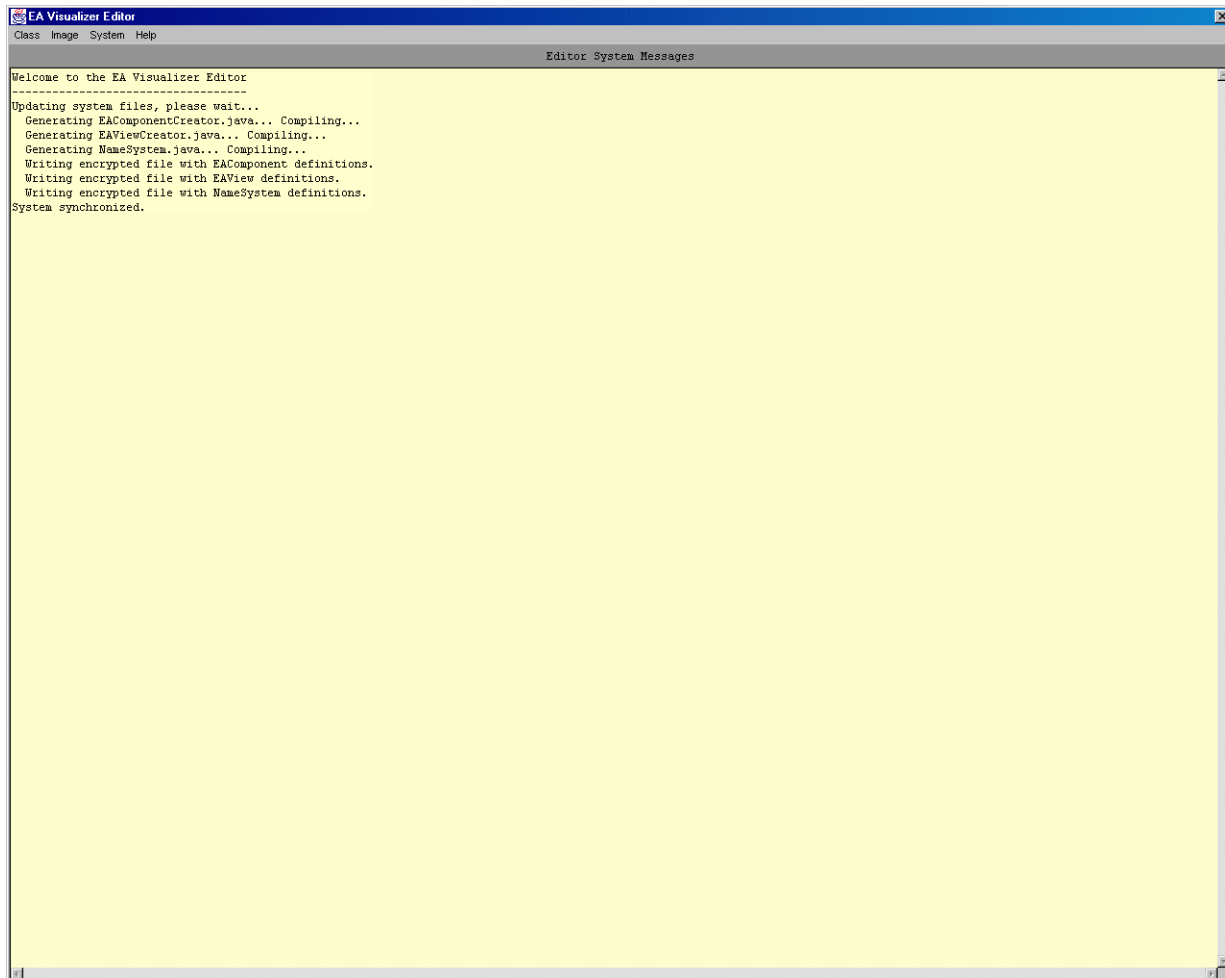


Figure 5.6: Successful synchronization of the *EA Visualizer* system.

5.5 Editing and browsing classes

Once classes have been added to the system, they can be edited so the actual JAVA code can be written for the functionality that is required. In addition, specialized items can be defined such as the parameter components and the actual name within the system. Alternatively of course, classes you already added and edited before and are perhaps currently active in the system, can be altered. In addition, when editing classes or perhaps just in general, it might very be very useful to browse classes without being able to edit them just to look up how certain things are established in other instances. In the *EA Visualizer* this can be done using the class editor and class browser respectively for editing and browsing. Note that the editable classes are only the ones you added yourself. The system provided classes cannot be altered. Contrary, for browsing, any class can be inspected. Browsing classes is very similar to editing classes, so we shall first go into detail in how classes can be edited in section 5.5.1 and subsequently briefly discuss the browse version in section 5.5.2.

5.5.1 The class editor

By selecting the *Edit class* option from the *Class* menu or alternatively by pressing F4, as stated in the overview in section 5.2, a class editor can be started for a certain instance in the *EA Visualizer*. The class editor interface itself is shown in figure 5.7. The main body of the interface is a textarea showing the JAVA code for the class that can be edited. Below this textarea, a smaller textarea is placed which is used by the system for feedback regarding this class. Just as in the main system, this is a message frame for system messages. Specifically, error messages from the compiler will be reported here. Over the main textarea, a textbar is placed which displays miscellaneous information during editing about the current state of the class editor (such as FILE SAVED). Finally, at the top of the interface, a menubar is located. As we have done in all other parts of this tutorial, we shall go over the menu options one by one to finish the description of the editor interface. In section 5.6.1 we show an example of how you can actually add a new *Recombinator* operator and how to specifically use the class editor interface. For now, a description of all possible menu items will suffice.

- *File*. This menu holds all actions that can be carried out on the entire classfile. This menu is generally found in most application programs and holds basic operations such as saving the written text.
 - *Save*. Alternatively, the keyboard combination **Ctrl+S** can be pressed. This causes the text which is the JAVA code for the file to be saved to disk. The filename is already determined when the class is selected, which is in turn already determined when the class is created.

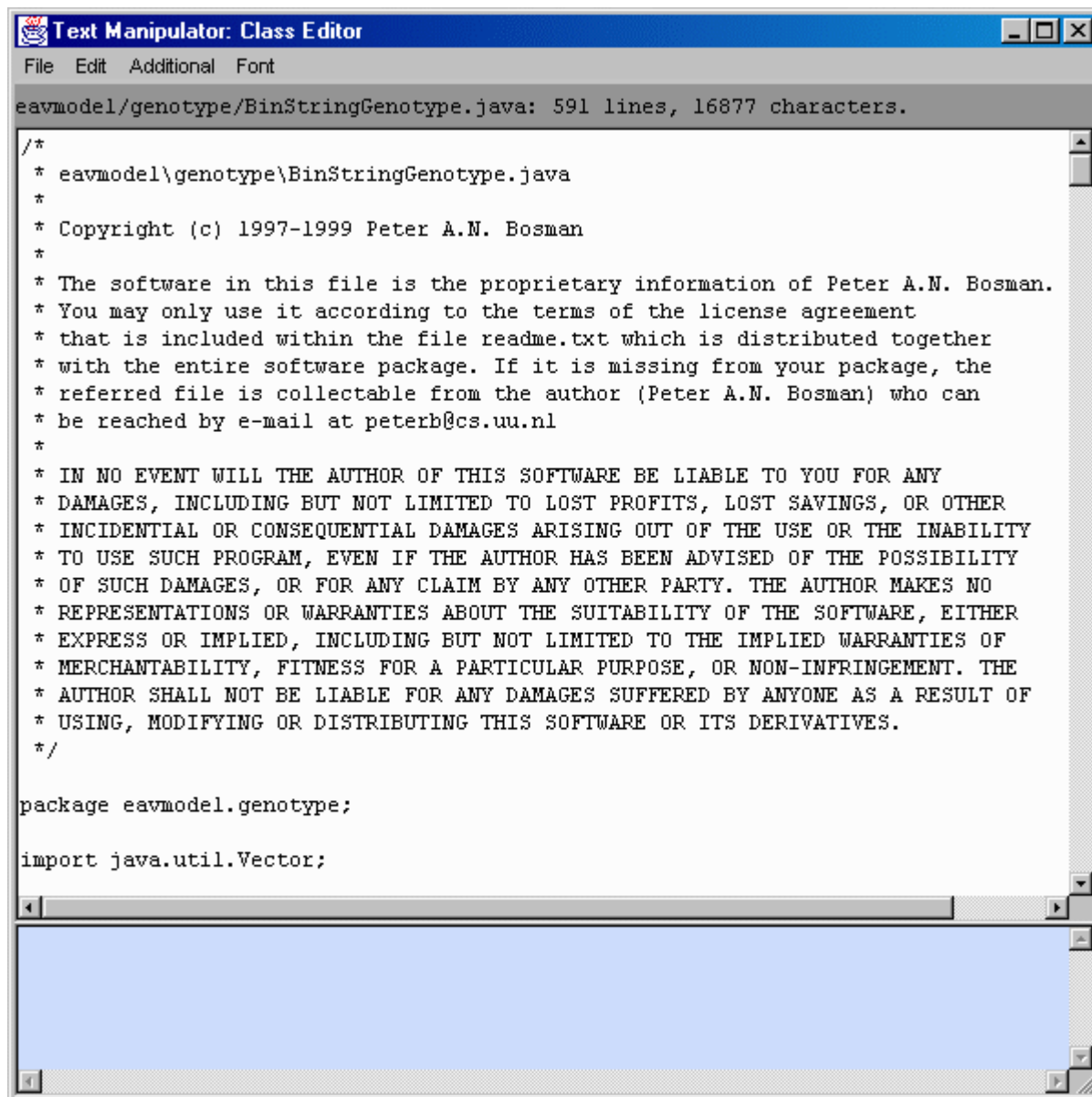


Figure 5.7: The interface for the class editor.

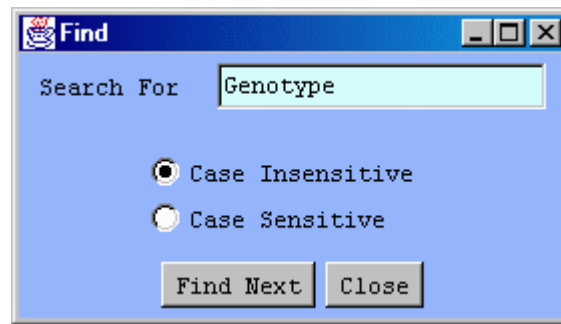


Figure 5.8: Finding substrings in the text.

- *Compile*. Alternatively, the F2 key can be pressed. After selecting this option, the text is first saved to disk, after which the JAVA compiler is called to compile the class file. This requires that the `javac` executable is in the path of the shell the system was started in. The compiler either terminates with error messages or without them, resulting in respectively unsuccessful or successful termination of the compile process. In case of any compiler errors, these messages are displayed at the bottom of the user interface in the text area. The message bar that is situated just under the menubar states that the compile process has terminated.
- *Close*. Alternatively, the F10 key can be pressed. Selecting this option causes the editor window to be closed. If any other windows that were opened from this editor are still open, the user is prompted to close these first. If no such windows are still open, the user is always prompted whether or not to save the text to disk before closing the window. In case of a confirmation, the text is first saved before closing the window.
- *Edit*. This is a standard menu that offers convenient and standard editing facilities such as finding substrings and jumping between lines.
 - *Find*. Alternatively, the keyboard combination `Ctrl+F` can be pressed. This causes a pop-up window to appear as can be seen in figure 5.8 in which a string can be entered to search for in the text. Furthermore, the search can be specified to be case insensitive or case sensitive. By pressing the *Find Next* button, the next occurrence for the entered string is found, starting at the current position of the cursor. If the string is found, it is highlighted through selection in the main text area of the user interface. If the string is not found when the end of the text is reached, the user is prompted whether the search should start again at the beginning of the text. The find frame itself is not closed. By pressing the *Close* button, the frame is closed. The string that was searched for however is still kept in memory in order for the next menu option to be plausible.

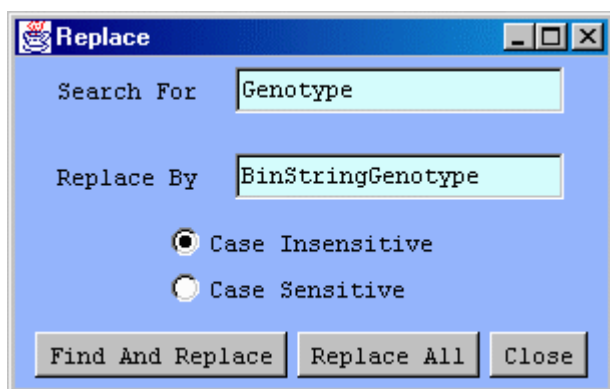


Figure 5.9: Finding and replacing substrings in the text.

- *Find Again*. Alternatively, the F3 key can be pressed. This causes the latest string that was searched for to be found again in a similar manner as is done by using the above menu option for finding strings in the text.
- *Replace*. Alternatively, the keyboard combination `Ctrl+R` can be pressed. This causes a pop-up window to appear as can be seen in figure 5.9. This frame is an extension over the find frame as just discussed. Using this menu option, substrings can be found and immediately be replaced. The top textfield in the frame can be used to specify the substring to be replaced in the text. The bottom textfield can be used to specify what that substring is to be replaced with. Again, the search (not the replacement) can be specified to be either case insensitive or case sensitive. By pressing the *Find And Replace* button, the next occurrence is found and highlighted through selection. Immediately a popup window is shown that prompts the user whether or not to replace this occurrence by the specified string in the replace frame. Upon a positive reply, the occurrence is replaced. Upon a negative reply, the occurrence is left unchanged, but the cursor is left there, causing the next time the *Find And Replace* button is pressed to actually find the next occurrence. Once the search has reached the end of the text, the user is prompted to continue the search at the beginning of the text just as is the case for the search feature. Pressing the *Find And Replace* button does not cause the frame to disappear. By pressing the *Replace All* button in the replace frame, the text is scanned from top to bottom, replacing all occurrences in the text. Pressing this button *does* cause the frame to disappear. Finally, pressing the *Close* button causes the frame to disappear.
- *Goto Line*. Alternatively, the keyboard combination `Ctrl+L` can be pressed. This causes a pop-up window to appear as can be seen in figure 5.10. This frame shows a single textfield in which a line number can be entered. By pressing the *Apply* button, the frame is disposed of and the cursor is placed at the specified

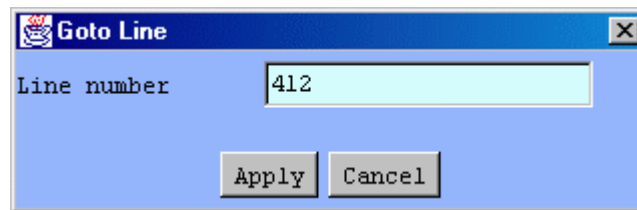


Figure 5.10: Jumping to a specified line in the text.

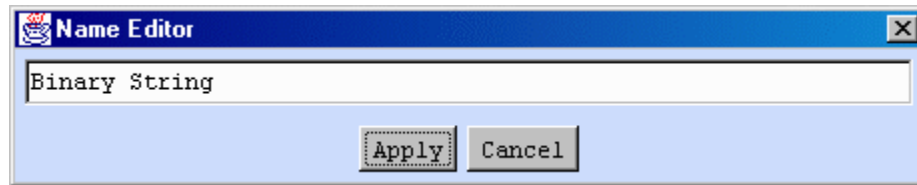


Figure 5.11: Editing the name for a class as to be used in the system.

line. Also, that line is highlighted through selection. If the line number that was specified is larger than the amount of lines in the text, the messagebar under the menubar displays this and no jump to a line is performed. By pressing the *Cancel* button, the frame is disposed of.

- *Additional*. This menu specifies additional options that are available only to the class editor. This is normally not found in text editors as these options concern settings specific to the *EA Visualizer*. This menu option may or may not be available, depending on the type of class that is edited. For components and views, this menu option is available. For utilities it is not and for parameter components it is only available in a limited version.
 - *Edit Name*. Selecting this menu option displays an interface with a single textfield as can be seen in figure 5.11. The textfield holds the name for the class as it is used within the system. As an example, the class that is written as `eavmodel.genotype.BinStringGenotype` in JAVA code, is named `Binary String` in the *EA Visualizer*. It is this latter name that is used as actual name and can be found here in the name editor when this class would be edited. Pressing the *Apply* button causes the editor to change its internal structure to hold the new or altered name. Pressing the *Cancel* button leaves the internal structure unaltered. Both buttons trigger the frame to be disposed of.
 - *Edit Help*. Selecting this menu option opens a new text editor window as can be seen in figure 5.12. This interface is quite similar to the class editor interface we are describing here and is displayed in figure 5.7. It is for this reason that we shall not go further into the details of the menu options for this interface, as they are all similar to the options described here. Furthermore, there are

no additional options to be found, merely a restricted set. The compile option from the *File* menu now causes a parser to parse the text that is contained in the help window. This parser is required as the text that specifies the help page has to be formatted according to a simple language that is quite similar to the \LaTeX language. The help files are displayed within the help system through some mechanism. This mechanism involves the reading of the structured text and the displaying of that text in some form. When at some point for a new version we decide to have a new look for the help files or have a better way to format the text, we do not wish to re-edit all the help files to adapt them to some structure, but rather only to rewrite the displaying mechanism so that the same files can still be read. Such is the case for instance with HTML documents for the WWW and with \LaTeX documents. It is for this reason that a \LaTeX kind of language has to be used in which text can be written with special environments to format the text. The commands that can be used in the language are the following:

- * Normal text.
- * `\emph{...}` for emphasizing.
- * `\bold{...}` for bold font.
- * `\color[r,g,b]{...}` for colored text.
- * `\seealso[subject]` for a crossreference (`subject` is fully specified class-name).
- * `\image[filename]` for an image.
- * `\\` for a forced newline.
- * `\backslash` for a backslash symbol.
- * `\{` for a curly brace open symbol.
- * `\}` for a curly brace close symbol.

In figure 5.13 a help page written in this language is shown as an example. The resulting output that is created by the *EA Visualizer* helpsystem is displayed in figure 5.14. You can of course try to find this page yourself in the system through the index page as a simple exercise.

- *Edit Parameter Components.* By selecting this menu option, a new interface appears as can be seen in figure 5.15. This interface contains a large list at the center and a series of buttons at the bottom. The list at the center is a list of the actual parameter components that are used for the class. The parameter components are identified by the name for the parameter that was given when a new parameter was added. These are the names in the list. By pressing the *Apply* button at the bottom of the interface, the settings for the parameter components are registered in the internal structure of the editor, causing the state of the editor to be altered. Alternatively, by pressing the *Cancel* button no updates are performed.

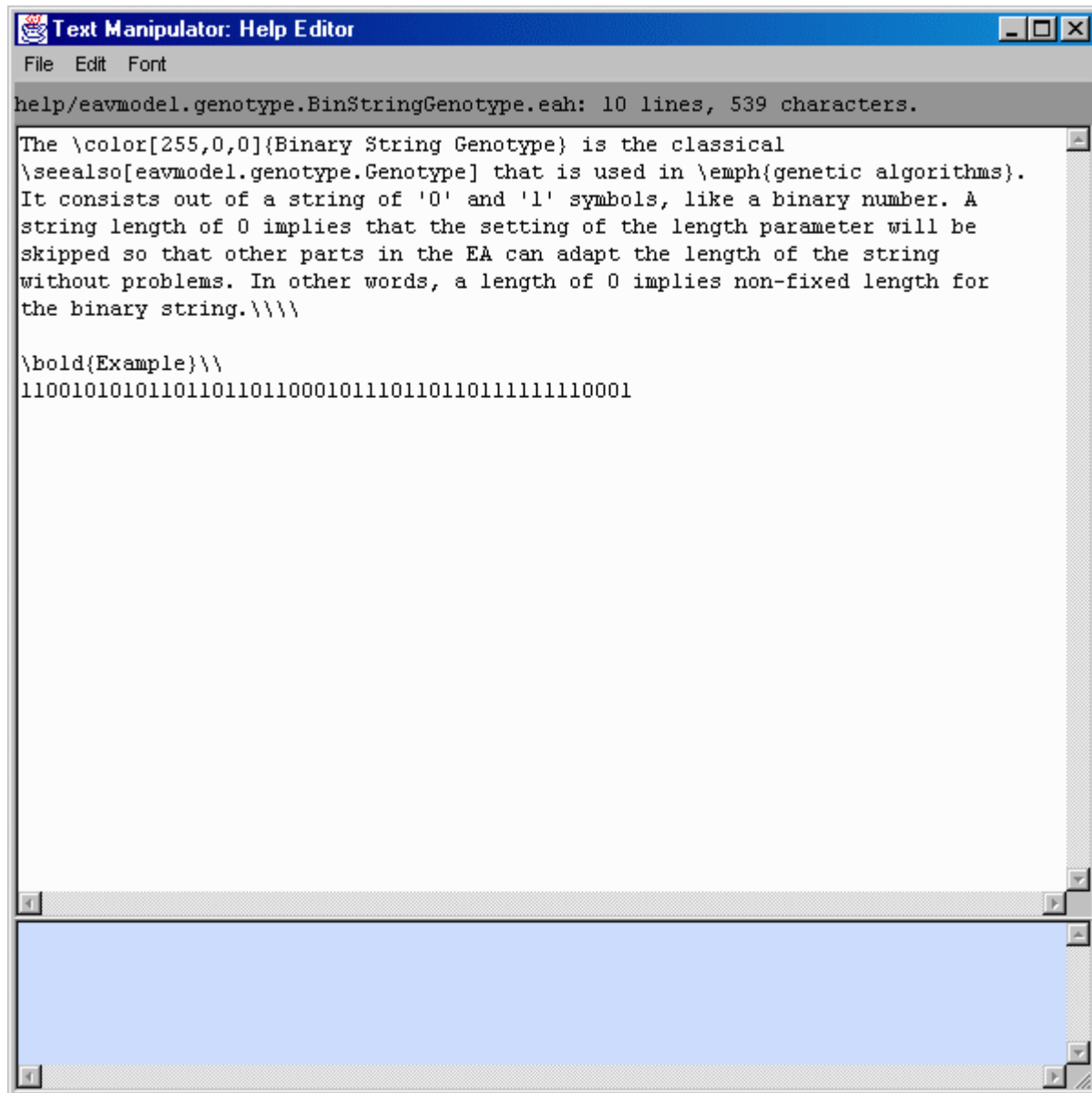


Figure 5.12: Editing the contents of the help page for a class.

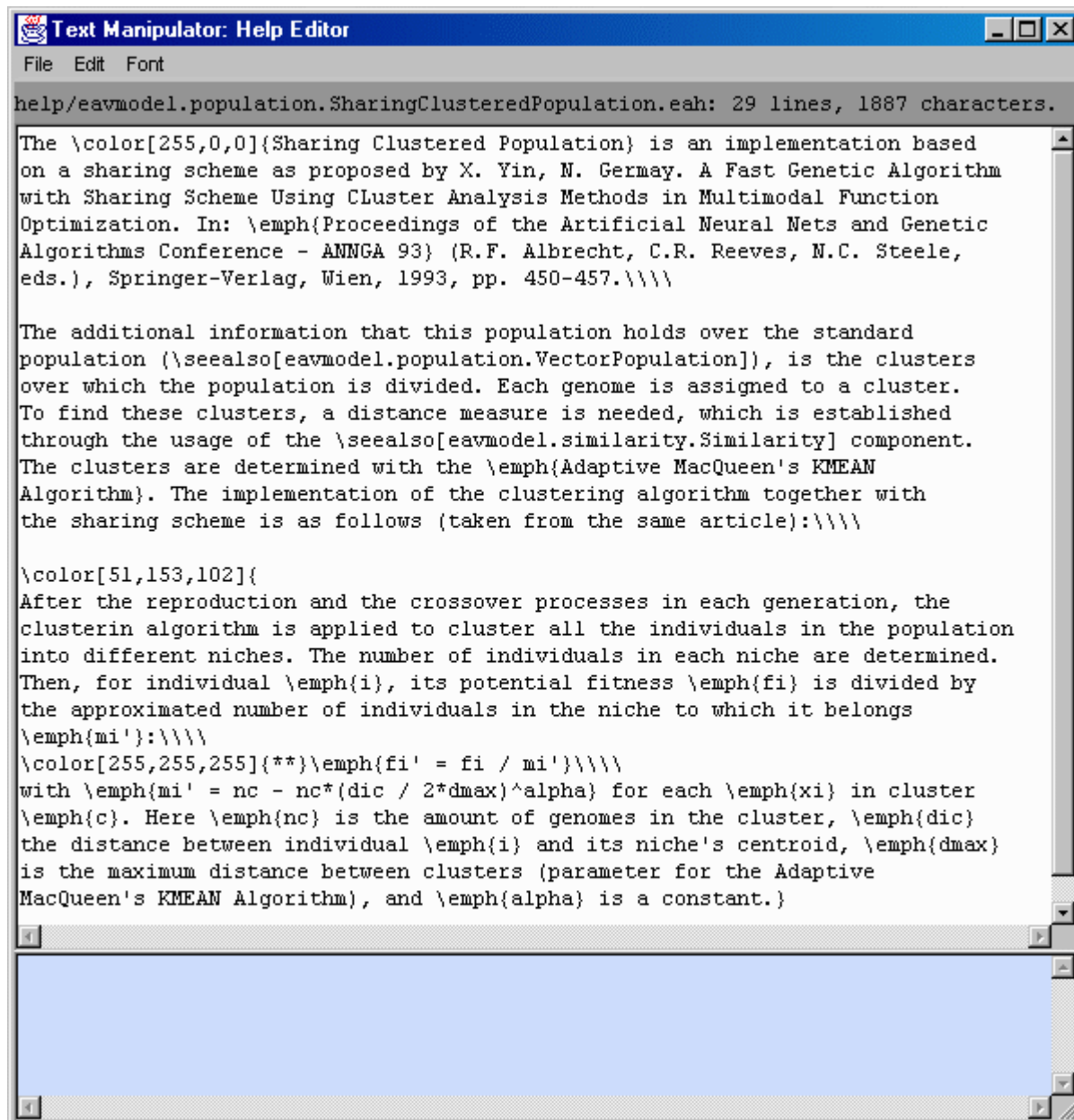


Figure 5.13: A help page written in the specified language.

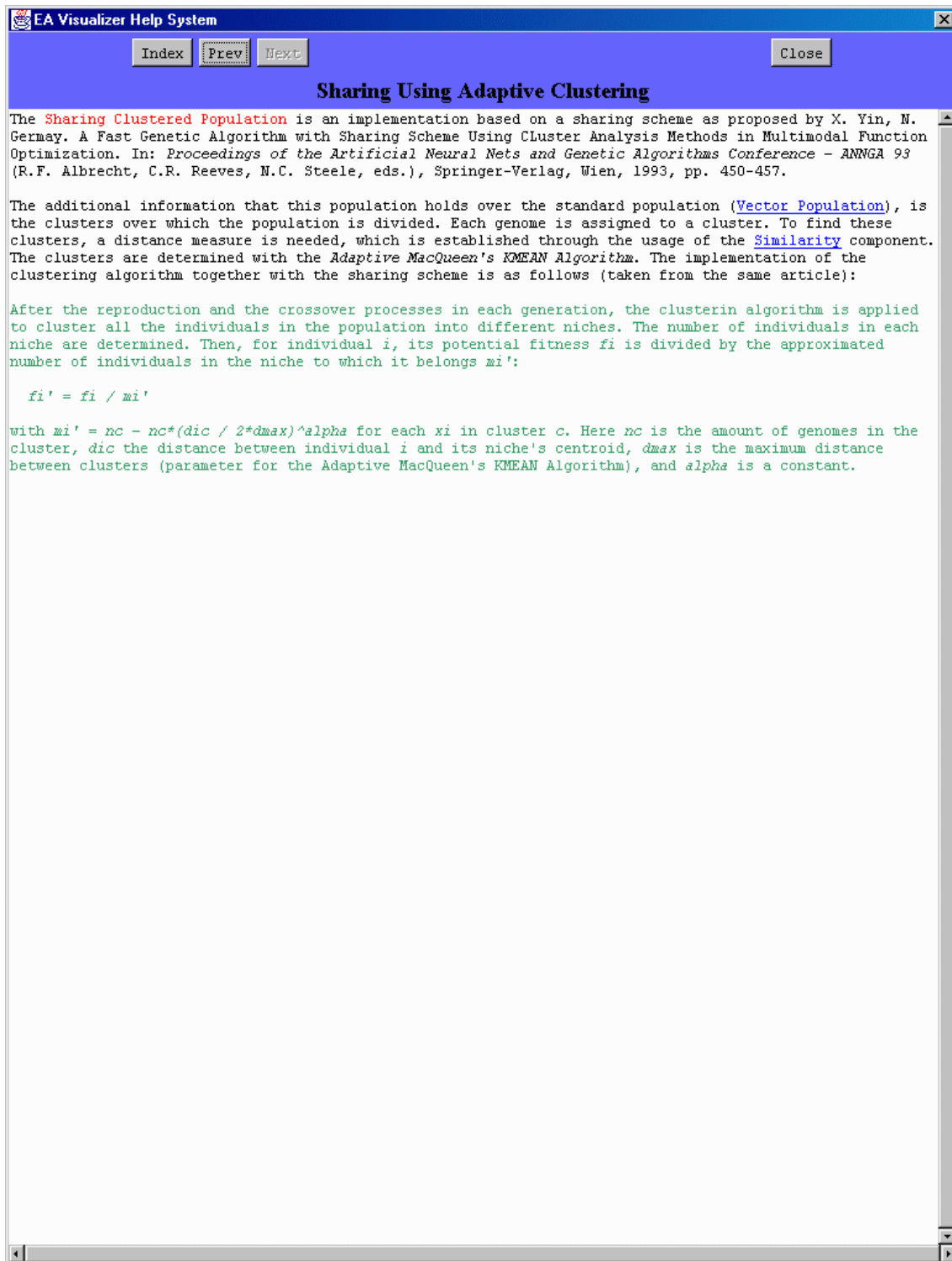


Figure 5.14: The resulting help page as the user finally gets to see it.

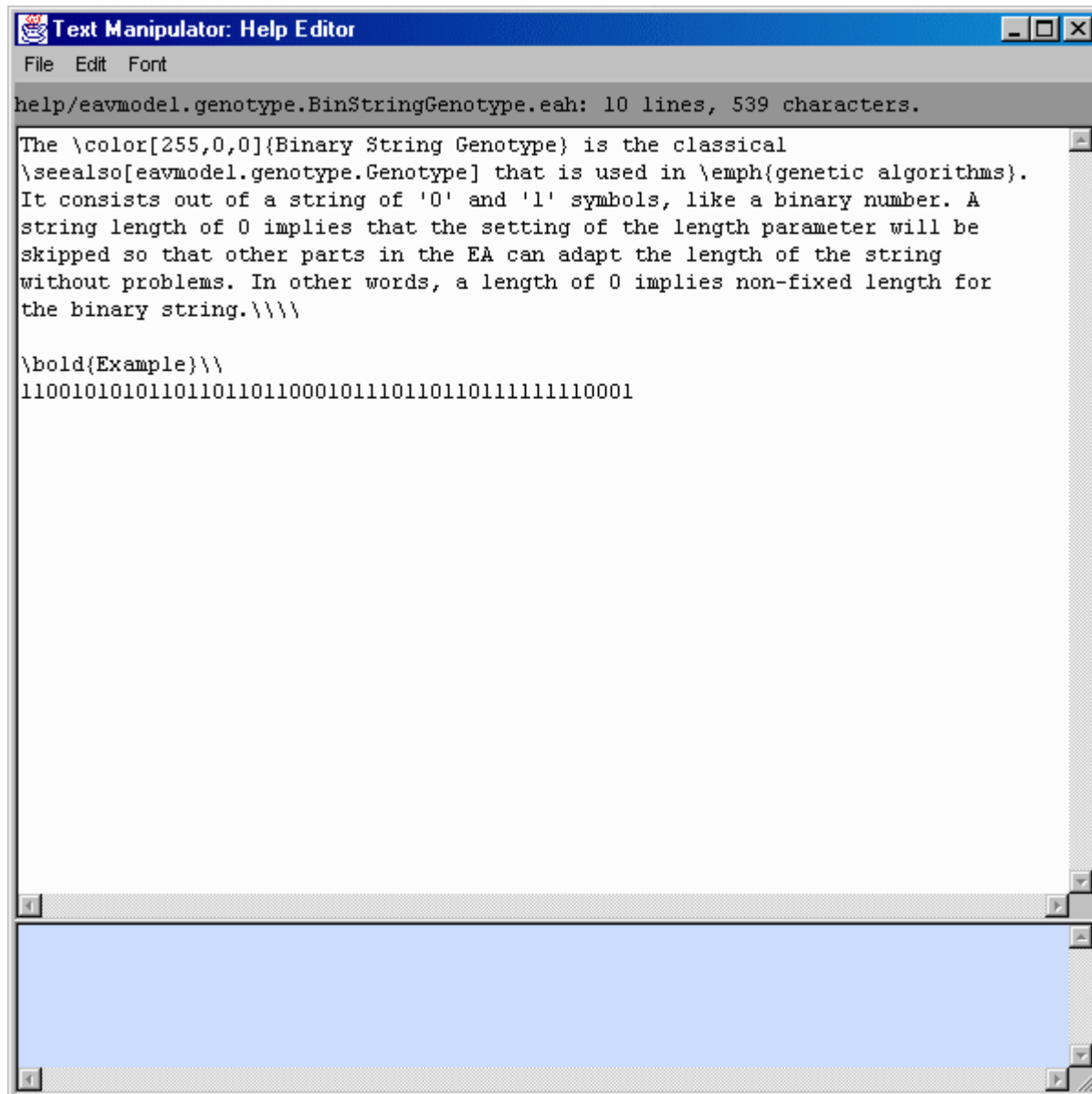


Figure 5.15: Editing the parameter components for a class.

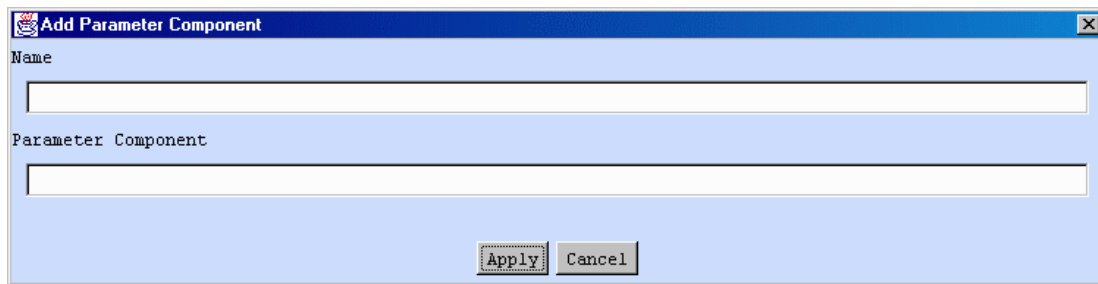


Figure 5.16: Adding a single parameter component.

Pressing the *Add* button adds a new parameter component. This causes a new window to appear with two textfields as can be seen in figure 5.16. The top textfield in this interface holds the name for the parameter component. This name will be used in the list that holds the defined parameter components as shown in figure 5.15 as well as when upon displaying the parameters for this class to the user. In the lower textfield the actual parameter component must be specified. More on how to format the entry in this textfield can be found in the example that is described in section 5.6.1. Having specified the parameter component, pressing the *Apply* button actually adds the parameter component to the list of parameters. Pressing the *Cancel* button disposes of all entered information. Both buttons trigger the interface to disappear.

After having selected a parameter component from the list of defined parameter components, three more buttons are enabled to be selected. These are the *Edit Name*, *Edit PC* and *Remove* button. The *Remove* button does no more than removing the selected parameter component from the list. Pressing the *Edit Name* button triggers an interface to appear that is identical to the interface that appears when pressing the *Edit PC* button. The former interface holds a textfield through which the name for the parameter can be altered. The latter interface holds a textfield through which the parameter component can be changed. This is done in exactly the same manner as when defining a new parameter component, albeit that now the two textfields from the interface in figure 5.16 are now taken care of separately. Finally, we wish to note that how the result of the parameter components can be used when writing JAVA code, can be found when an example of adding a new class is discussed in section 5.6.1. Also, this menu option is only available to class editors invoked on instances for components or on views.

- *Edit Dependencies*. By selecting this menu option, a new interface appears as can be seen in figure 5.17. This interface contains two large lists at the center and a four buttons at the bottom. The leftmost list contains the components that this class can be dependent on. For new instances for components, these are only the four dependency imposing components from section 3.3. For new

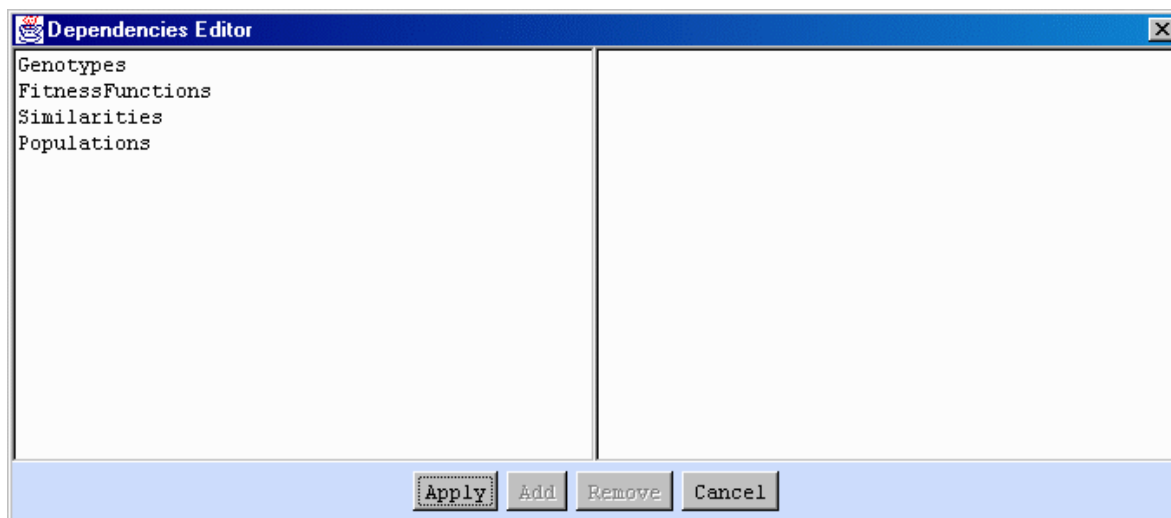


Figure 5.17: Editing the dependencies for a class.

views, these are all of the available components. By pressing the *Apply* button at the bottom of the interface, the settings for the dependencies are registered in the internal structure of the editor, causing the state of the editor to be altered. Alternatively, pressing the *Cancel* button, no updates are performed.

After having selected one of the dependency imposing components from the leftmost list in the interface, the *Add* button becomes enabled. Pressing this button brings forward a list of all instances for that component as can be seen in figure 5.18. Pressing the *Cancel* button in that window causes no additional dependency to be specified. Selecting an instance from the list in this interface, followed by pressing the *Apply* button *does* result in a new dependency if the selected instance was not already part of the dependencies. The result will be that the dependency instance will now appear in the rightmost list in the interface in figure 5.16. Finally, selecting an instance from this rightmost list enables the *Remove* button. Pressing this button removes the selected instance from the dependencies list, removing the dependency for the class that is being edited. This menu option is only available to class editors invoked on instances for components or on views.

- *Font*. Through this menu, personal preferences for the appearance of the text editor can be set. This comes down to changing the style of the text through type, size and font. The default setting is the `Monospaced` font set as *plain* with a font size of 12 points.
 - *Type*. By selecting this menu option, an interface with a list is shown as displayed in figure 5.19. In this list, the available fonts are displayed that can be used for the text. By selecting a font and pressing the *Apply* button, the text is displayed

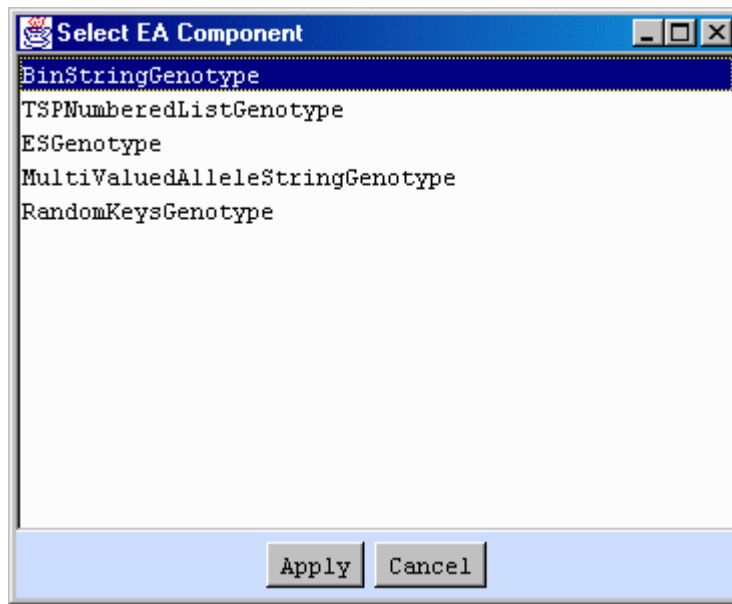


Figure 5.18: Adding a new dependency for a class.

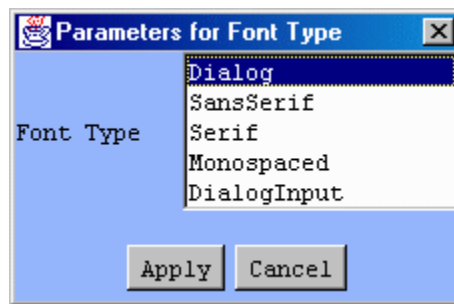


Figure 5.19: Selecting the font to use for the text.

with that font. By pressing the *Cancel* button, the interface disappears, leaving the display of the text unaltered. Both buttons cause the interface to be disposed of.

- *Style*. By selecting this menu option, an interface with a list is shown as displayed in figure 5.20. In this list, the four available style are displayed that can be used for the text (plain, bold, italic or both bold and italic). By selecting a style and pressing the *Apply* button, the text is displayed with the selected style adjustment. By pressing the *Cancel* button, the interface disappears, leaving the display of the text unaltered. Both buttons cause the interface to be disposed of.
- *Size*. By selecting this menu option, an interface with a textfield is shown as displayed in figure 5.21. Using the textfield, the fontsize can be specified in

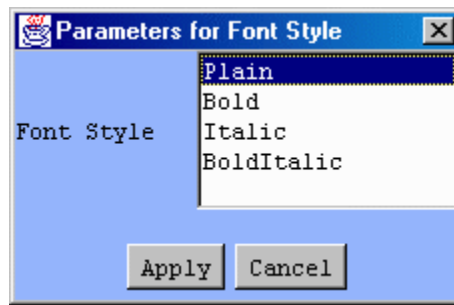


Figure 5.20: Selecting the style adjustment to use for the font.

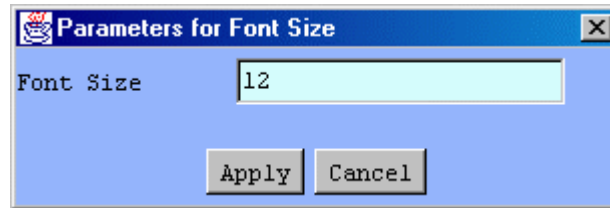


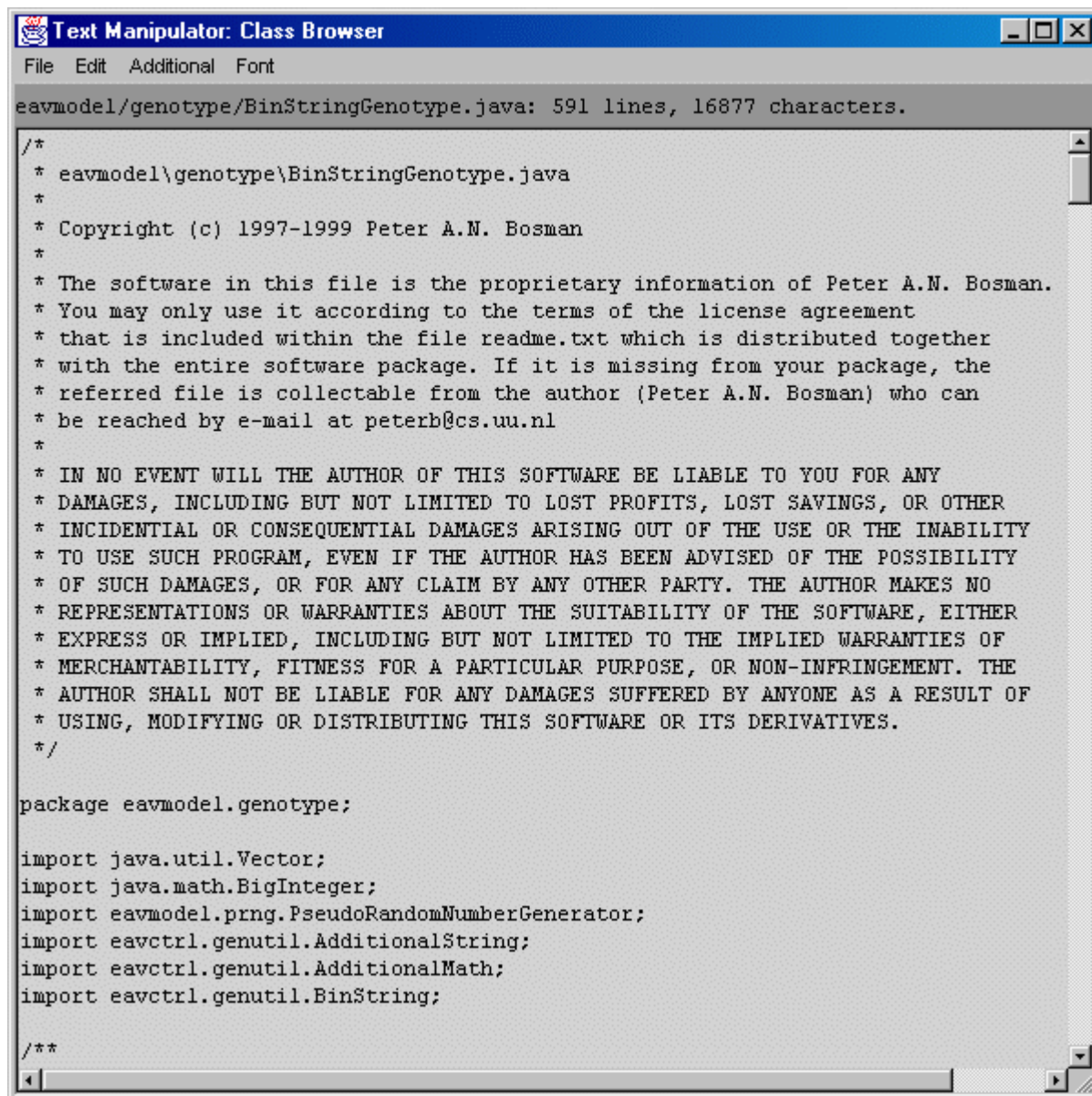
Figure 5.21: Selecting the size to use for the font.

points. The font size is an integer number between 1 and 200. By entering a number as font size and pressing the *Apply* button, the text is displayed with the selected size for the font. By pressing the *Cancel* button, the interface disappears, leaving the display of the text unaltered. Both buttons cause the interface to be disposed of.

5.5.2 The class browser

As was stated in section 5.5, the class browser that can be launched from the editor by pressing the F5 key or by selecting **Browse** from the **Class** menu, differs only slightly from the class editor that is fully described in section 5.5.1. In figure 5.22, the class browser user interface is shown. At first glance, the difference with the interface for the class editor as shown in figure 5.7 is virtually absent. The main differences are however found in the contents of the menus, which we shall shortly go over in the following.

- *File*. The difference is that files cannot be saved or compiled. The browser window can only be closed by selecting **Close** or by pressing F10 on the keyboard. Also, selecting to close the interface does not result in prompting the user to close other windows that were opened by this browser window first. Instead, the system closes all such additional browsing windows itself.



The screenshot shows a window titled "Text Manipulator: Class Browser". The window has a menu bar with "File", "Edit", "Additional", and "Font". Below the menu bar, it displays the file path "eavmodel/genotype/BinStringGenotype.java: 591 lines, 16877 characters." The main area of the window contains the following Java source code:

```
/*
 * eavmodel\genotype\BinStringGenotype.java
 *
 * Copyright (c) 1997-1999 Peter A.N. Bosman
 *
 * The software in this file is the proprietary information of Peter A.N. Bosman.
 * You may only use it according to the terms of the license agreement
 * that is included within the file readme.txt which is distributed together
 * with the entire software package. If it is missing from your package, the
 * referred file is collectable from the author (Peter A.N. Bosman) who can
 * be reached by e-mail at peterb@cs.uu.nl
 *
 * IN NO EVENT WILL THE AUTHOR OF THIS SOFTWARE BE LIABLE TO YOU FOR ANY
 * DAMAGES, INCLUDING BUT NOT LIMITED TO LOST PROFITS, LOST SAVINGS, OR OTHER
 * INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR THE INABILITY
 * TO USE SUCH PROGRAM, EVEN IF THE AUTHOR HAS BEEN ADVISED OF THE POSSIBILITY
 * OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY. THE AUTHOR MAKES NO
 * REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF THE SOFTWARE, EITHER
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. THE
 * AUTHOR SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY ANYONE AS A RESULT OF
 * USING, MODIFYING OR DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.
 */

package eavmodel.genotype;

import java.util.Vector;
import java.math.BigInteger;
import eavmodel.prng.PseudoRandomNumberGenerator;
import eavctrl.genutil.AdditionalString;
import eavctrl.genutil.AdditionalMath;
import eavctrl.genutil.BinString;

/**
```

Figure 5.22: The interface for the class browser.

- *Edit.* The only difference with the edit menu from the class editor is that the *Replace* menu option is absent. This is of course due to the fact that the class text cannot be altered in a browser, so substrings in the text may of course neither be replaced in this way.

- *Additional.* The largest difference between the browser and the editor can perhaps be found in this menu. The options available are equal, but have a different functionality. Therefore we shall separately discuss the functionality of these options for the browser briefly.
 - *Browse Name.* Selecting this menu option triggers an interface to appear that is similar to the editing version. The only difference is that the text for the name cannot be altered and that the window can only be closed.

 - *Browse Help.* Just as the class editor has a help editor equivalent for editing the help files, the class browser has a help browser equivalent for browsing the help. The interface for browsing the help page differs from the edit version just as the class browser which we are presenting right now differs from the class editor. It is for this reason that we need not specify any further the menu options of the help browser.

 - *Browse Parameter Components.* Selecting this menu option brings forward an interface with a single text area and a single button. As opposed to the edit variant, this interface is very simple. The text displays an overview of the parameter components that have been defined for the class. This means that for each defined parameter component two lines of text are displayed. The first line is the parameter name and the second line (which is indented by two spaces) is the definition for the parameter component. The textarea itself contains a multiple of such pairs of strings.

 - *Browse Dependencies.* Just as is the case for the browsing of parameter components, selecting this menu option brings forward an interface with a single text area and a single button. As opposed to the edit variant, this interface is also very simple. For each component that dependencies can be specified on, a list is presented of all instances that this class is allowed to be selected with. Each such list is indented by two spaces and is preceded by the name of the component (followed by an additional character `s`) that is not indented. For views, this list consists of all evolutionary components whereas for instances for components this list only consists of the dependency imposing components that we first mentioned in section 3.3.

- *Font.* There are no differences with the font menu from the class editor.

5.6 Examples

In this section we present various examples. The two examples are chosen so that the user is guided directly in using the editor, namely by adding a new instance for the *Recombinator* component. After this, we present a more complicated recombination operator, namely FDA and show how this operator has been implemented in the *EA Visualizer*. The first example presented is perhaps described in more detail than the second example, but nevertheless the reader is guided in the use of the editor.

5.6.1 Adding the n -point crossover operator

In this section we provide some experimental information for using the *EA Visualizer* editor. We show how the editor can be used to actually create and add a new recombination operator to the system. In this section we implement a simple recombination operator, which is a generalization of the one and two point crossover operators. In section 5.6.2, we show how the *Factorized Distribution Algorithm* or FDA is implemented in the system as a different type of recombination operator.

As a generalization of the most common known recombination operators in genetic algorithms, being one-point and two-point crossover, we desire to implement an n -point crossover operator that randomly chooses n crossover points (with n smaller than the string length l) and swaps all even parts between these crossover points. This operator works on two parents that are both binary strings. Enumeratively stated:

1. Select n distinct crossover points within the range $[0 \dots l - 1]$
2. For each bitposition, if an even number of crossover points have been passed (with 0 being even), copy bits from that location from parent one to offspring one and from parent two to offspring two. If an odd number of crossover points have been passed, perform the copy with the offspring reversed.

To implement this recombination operator in the *EA Visualizer*, we first start the editor version of the system. From the *Class* menu, we select the *Add* option and are faced with the interface from figure 5.23. It is clear that we wish to add a new instance for a selected component, so we select *EA Component* by clicking on the right radiobutton and press the *Apply* button.

Having done so, we are faced with an adapted interface with a new set of options from which we have to choose one. We have to select what type of component it is we wish to add as can be seen in figure 5.24. As we wish to add a new crossover operator, we select the *Recombinator* option and once again press the *Apply* button.

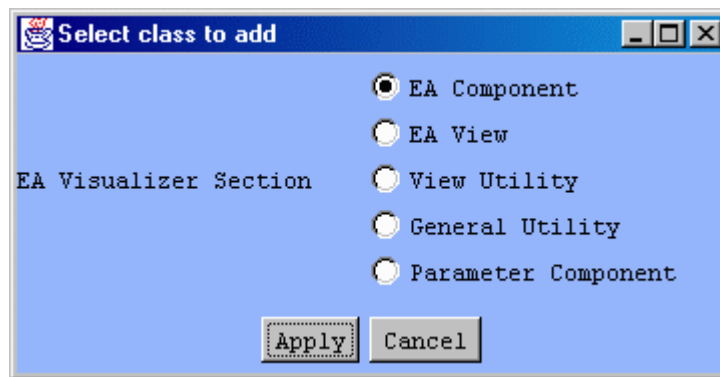


Figure 5.23: Selecting the type of class to add.

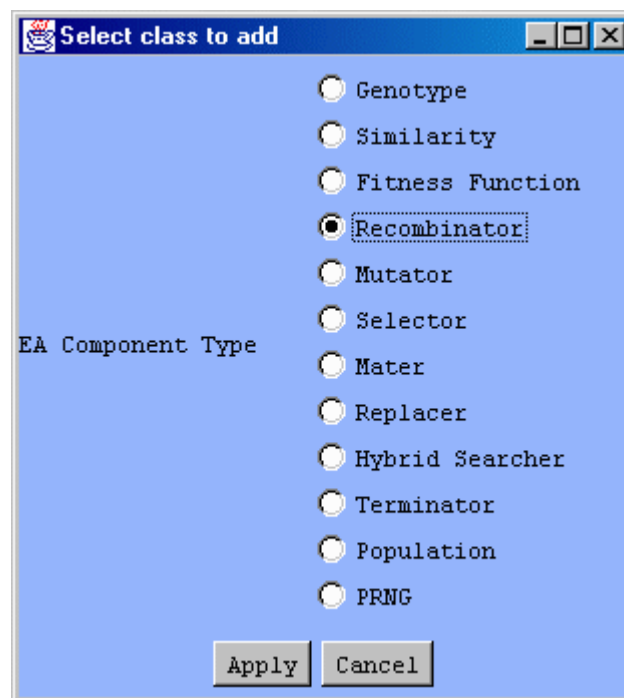


Figure 5.24: Selecting the type of component to add.

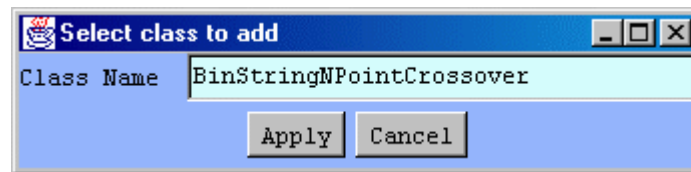


Figure 5.25: Entering the name for the new crossover operator.

Now the interface shows only a single textfield in which we have to specify the JAVA classname for the new recombination operator as is shown in figure 5.25. We enter the name `BinStringNPointCrossover` and press the *Apply* button one more time. We are now again faced with the editor window, but can now observe that the title for the window has been altered. Behind the title the bracketed word *modified* can now be found. This means that the system must now be synchronized because the version on disk is inconsistent with the version in memory because of our addition to the system. Selecting to exit the system will therefore cause the *EA Visualizer* to present a popup window in which the warning is given that the system has been changed but not synchronized. At this point however, we do not wish to update the records on disk as we first want to complete the specification of the new class, so we first continue to do exactly that.

We now need to edit the class we have just added. To this end, we select *Edit* from the *Class* menu. At first hand, we are required to select again what type of class it is we wish to edit. To this end, we of course select again to edit an *EA Component* and in the next phase to edit a *Recombinator*. Next, we are given a list of all available recombination operators¹. As is shown in figure 5.26, we select the class we just added and press the *Apply* button.

We are now presented with a class editor interface in which the classbody and the required methods for the new class are already specified. This interface is shown in figure 5.27. Before we aim ourselves at writing the code for the new operator, we first move to edit the additional aspects such as parameter components and dependencies. At this point, we note that the user could open a browser window by selecting *Browse* from the *Class* menu and browse either the one–point or two–point crossover entry in the system, because those entries will of course be quite similar to the one we wish to implement. We shall refrain from that in this tutorial and leave it as an exercise.

We first select to edit the name for the new class by selecting *Edit Name* from the *Additional* menu. By doing so, we edit the name that will be used throughout the system for this new recombination operator. We select to use the same naming that was used for the other crossover operators and enter the name `Binary String - N Point Crossover` as can be seen in figure 5.28.

¹The ones available for editing. For the author, these are all operators, but for the user, only the self-defined classes will be available.

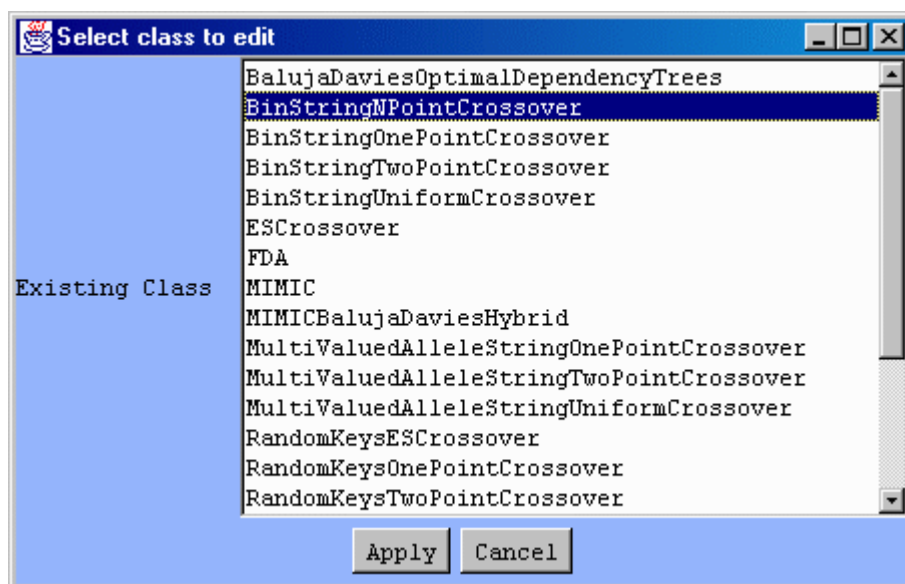


Figure 5.26: Entering the name for the new crossover operator.

Next we go to the *Edit Help* option in the *Additional Menu*. Selecting this menu option brings forward a help editor interface. As noted before in section 5.5.1, the help editor is very similar to the class editor. Therefore, working with this second editor window is quite straightforward. In the text area we enter the help information we wish to make available to the user. We enter the following text²:

```
The \color[255,0,0]{N Point Crossover} operator is a generalization of
the \seealso[eavmodel.recombinator.Recombinator] operators that are commonly
used in \emph{genetic algorithms}, being
\seealso[eavmodel.recombinator.BinStringOnePointCrossover] and
\seealso[eavmodel.recombinator.BinStringOnePointCrossover]. If we define
\emph{l} to be the length of the string, positions \emph{p0} through to
\emph{pn} are distinctly chosen within the string. Between each pair of
crossover points, bits are copied from the two parents to the two
offspring genomes. After an odd number of crossover positions, the
bits are copied with the parents switched.\\

\bold{Example}\\
Parents:\\
\color[0,0,255]{11}|\color[0,0,255]{001}|\color[0,0,255]{10}|\color[0,0,255]{101}|
\color[0,0,255]{01}\\
\color[255,0,0]{00}|\color[255,0,0]{101}|\color[255,0,0]{00}|\color[255,0,0]{011}|
\color[255,0,0]{10}\\\\
```

²Each line of colors should actually not be written over multiple lines, but it is done in this tutorial due to the linewidth and readability issues.

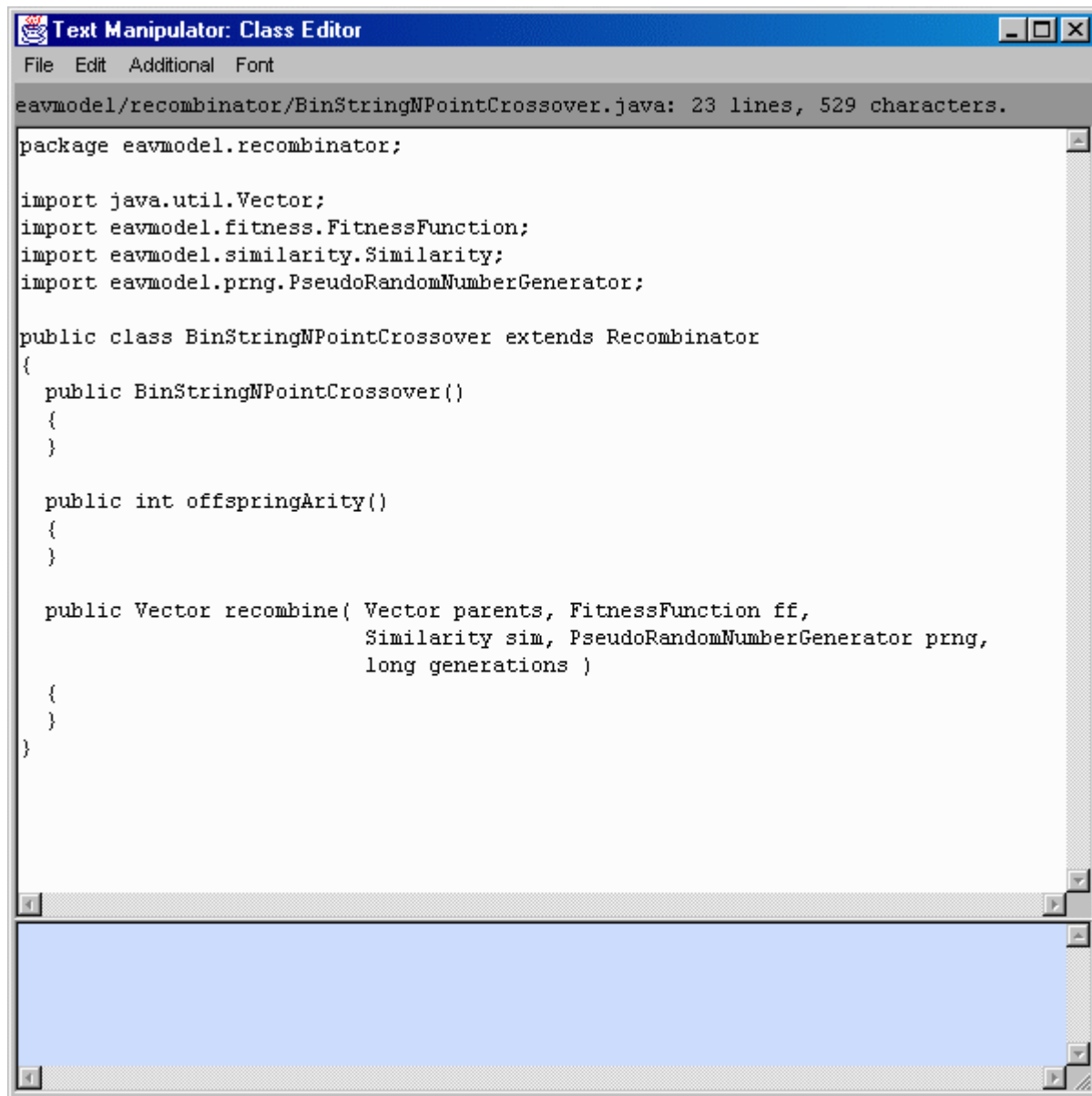


Figure 5.27: The initial class editor for the new crossover operator.

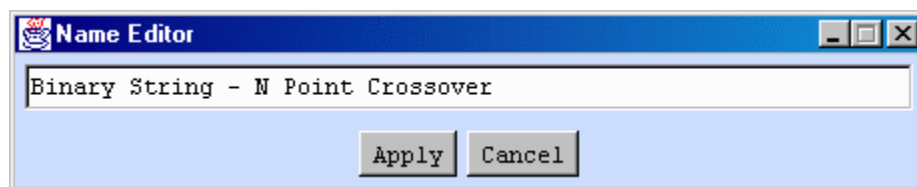


Figure 5.28: Entering the name to be used in the system for the new *Recombinator*.

```

Offspring:\\
\color[0,0,255]{11}|\color[255,0,0]{101}|\color[0,0,255]{10}|\color[255,0,0]{011}|
\color[0,0,255]{01}\\
\color[255,0,0]{00}|\color[0,0,255]{001}|\color[255,0,0]{00}|\color[0,0,255]{101}|
\color[255,0,0]{10}

```

After typing this text, we see if the *EA Visualizer* understands what we want to show by pressing F2 or selecting to *Compile* the text. The text is first saved and subsequently we are reported that the compile was successful, meaning that we have entered a text that is formatted correctly. This is depicted in figure 5.29. The resulting helppage can now be found at runtime by pressing F1 and going to the index page. There a link can be found to the newly added crossover operator. The page that is then displayed is depicted in figure 5.30.

The next step in defining the additional items is to edit the parameter components for the recombination operator. This means that we are to specify what parameters we require before we can actually apply n -point crossover to two parents. Basically, we only require the obvious value for n to know exactly how many crossover points we require. As we can see however in the implementations of one-point crossover, two-point crossover and uniform crossover that already exist in the system, another parameter all of these recombination operators define is the offspring arity. This is the amount of offspring this *Recombinator* produces. As the crossover operator can easily return one or two offspring, this can be set to be a parameter also. Concluding, we require to specify two parameters for this recombination operator, namely an integer larger than 0 that defines the amount of crossover positions and an integer that equals either 0 or 1 that defines the amount of offspring this *Recombinator* produces. In terms of the *EA Visualizer* and existing parameter components, we require to call on the `LongRangedParameterComponent`. Using the class browser in the editor, we can quickly punch up the code for this class to see how this parameter component is to be used in terms of JAVA code. The comment header for the class assures us that this is the parameter component we are looking for:

```

* A <code>ParameterComponent</code> that only accepts long values within a
* certain range.

```

A bit further we find the way to create one of such parameter components since this is coded in the constructor method of this class:

```

/**
 * Creates a new long ranged parameter component.
 * @param a    the lower limit of the feasible range.
 * @param b    the upper limit of the feasible range.
 * @param def  the default value.
 */
public LongRangedParameterComponent( long a, long b, long def )

```

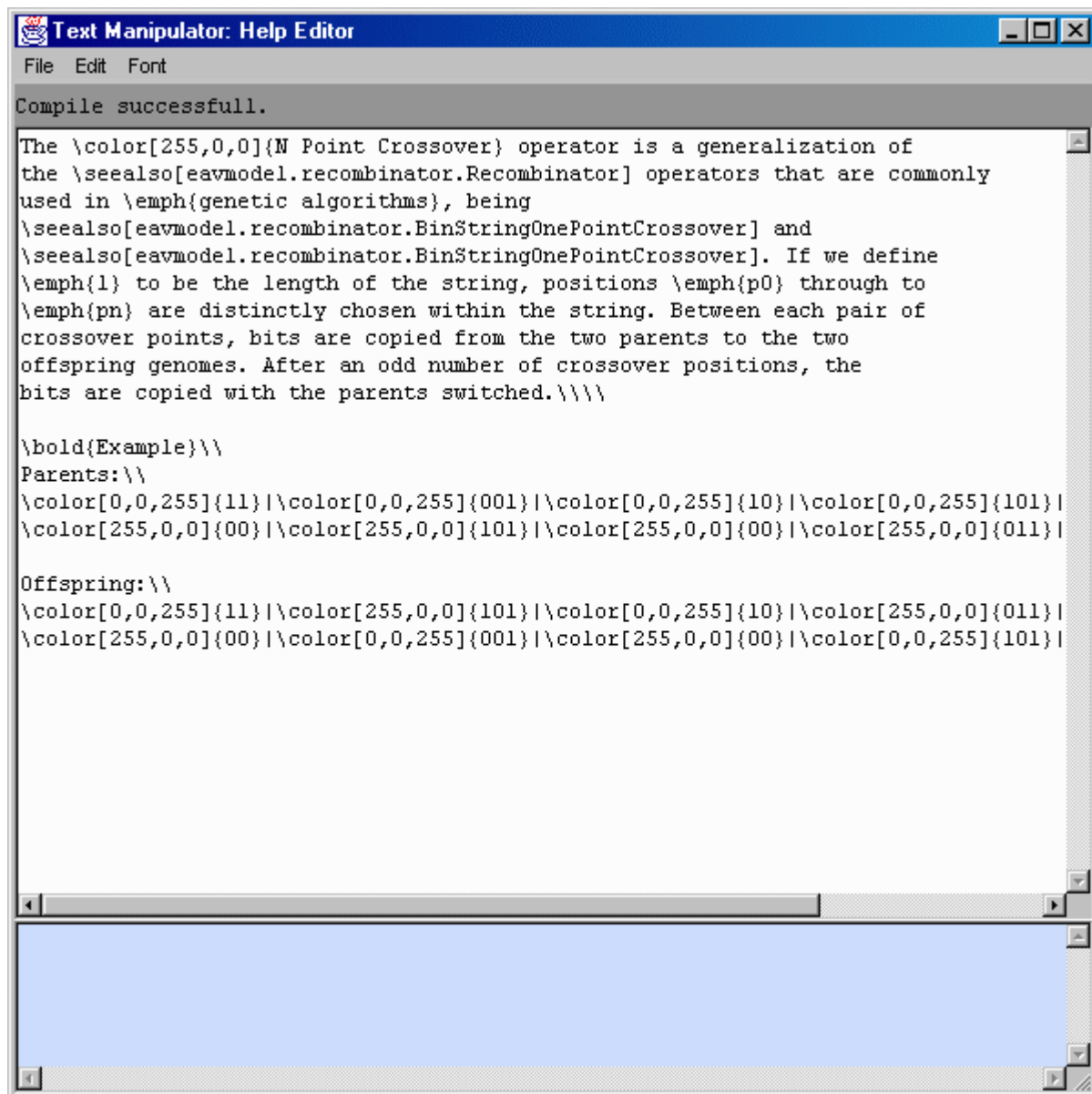


Figure 5.29: The help text as it is entered into the help editor and compiled without errors.

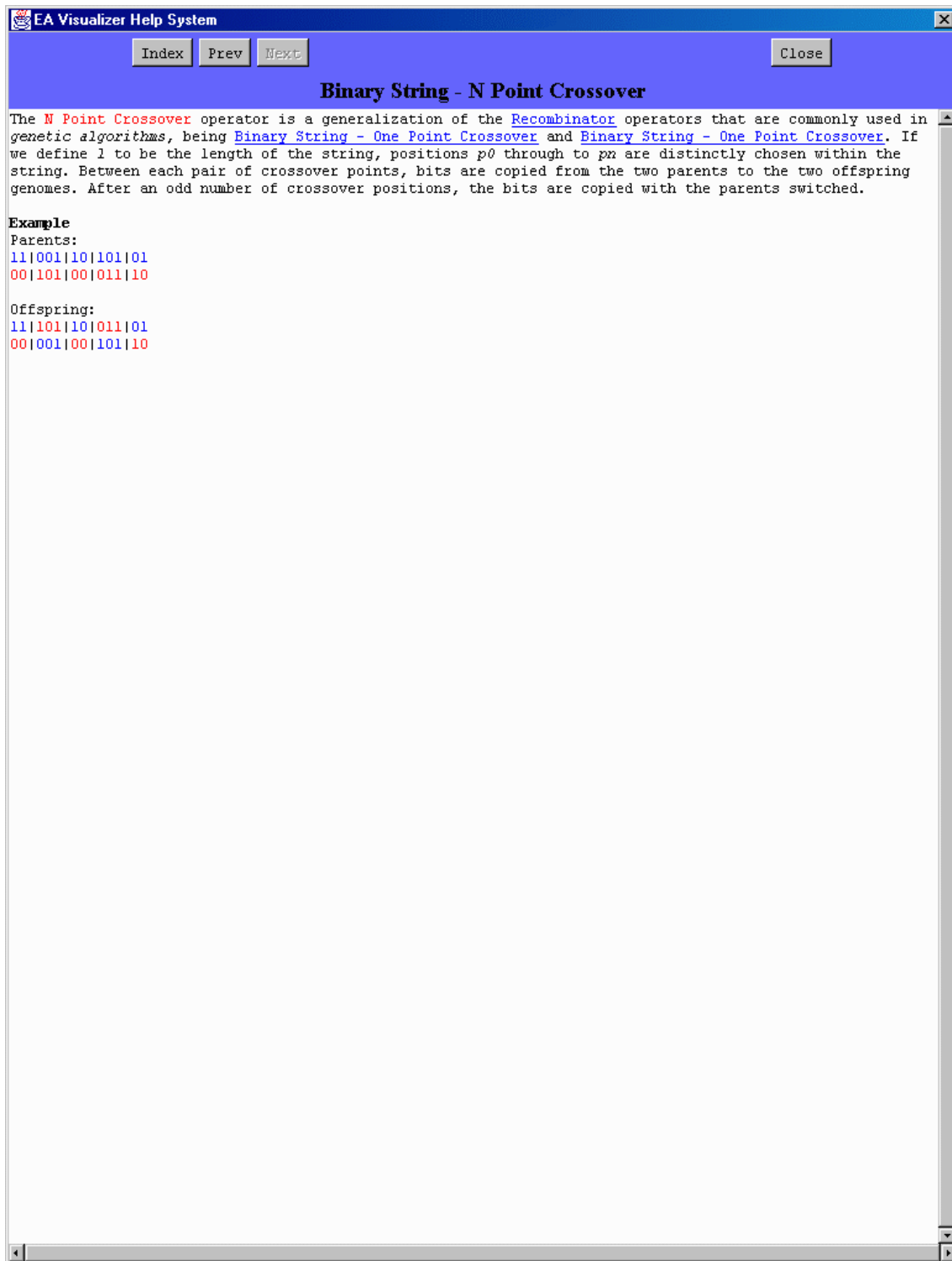


Figure 5.30: The help text as it finally appears as a help page in the system.

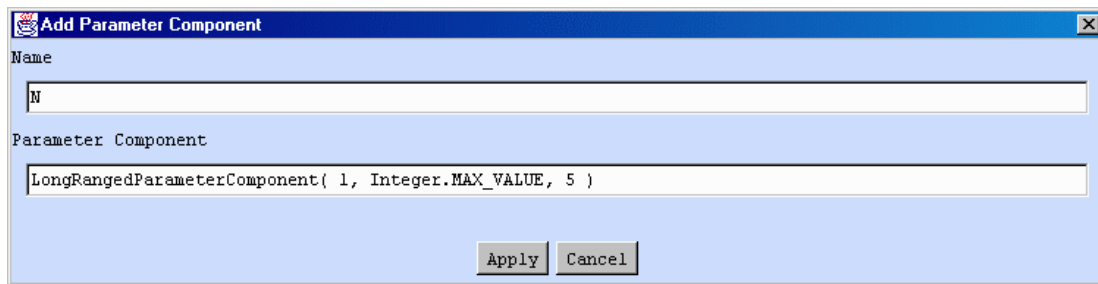


Figure 5.31: Defining the first parameter for n -point crossover.

This tells us that such a parameter component requires first the lower and upper limit of the range that the values have to come from and lastly the default value that has to be specified. We are now ready to actually specify the parameter components and select *Edit Parameter Components* from the *Additional* menu in our class editor for our new recombination operator. In the interface that appears we press the *Add* button. This causes the interface we saw earlier in figure 5.16 to appear. We enter as a name for the parameter in the upper textfield **N**, indicating that this is the parameter that specifies the amount of crossover points. As a parameter component we need to specify the constructor method as we would normally write it in JAVA code to create a new object, but now *without* the *new* word in front of it and without a semicolon after it. For this parameter, this results in the following text:

```
LongRangedParameterComponent( 1, Integer.MAX_VALUE, 5 )
```

This indicates that we allow integer values larger than 1 and use as a default the value 5. After having entered this information, we press the *Apply* button. Directly afterwards, we again press the *Add* button and enter the information for the second parameter, being the offspring arity. Once again we use a long ranged parameter component, but now with the values 1, 2 and 2, defining that for this parameter we only allow the values 1 and 2 and use as a default the value 2. The resulting interfaces for the definition of the parameter components can be seen in figures 5.31 and 5.32. Furthermore, the resulting interface for adding the parameter components is shown in figure 5.33

The final aspect that is to be defined before going into writing the code, are the dependencies. This is done by selecting *Edit Dependencies* from the *Additional* menu in the class editor. The dependencies for this recombination operator are of course only on the genotype. As this operator specifically crosses over parts of binary strings, the dependency is that this operator may only be selected when the genotype is set to be a binary string. In the interface that is shown when the *Edit Dependencies* option is selected from the *Additional* menu, we therefore select the **Genotypes** entry in the leftmost list and subsequently press the *Add* button. This causes a second window to appear with all possible

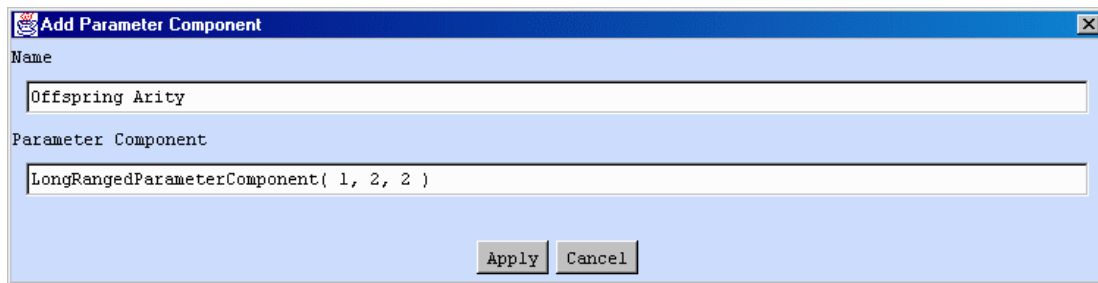


Figure 5.32: Defining the second parameter for n -point crossover.

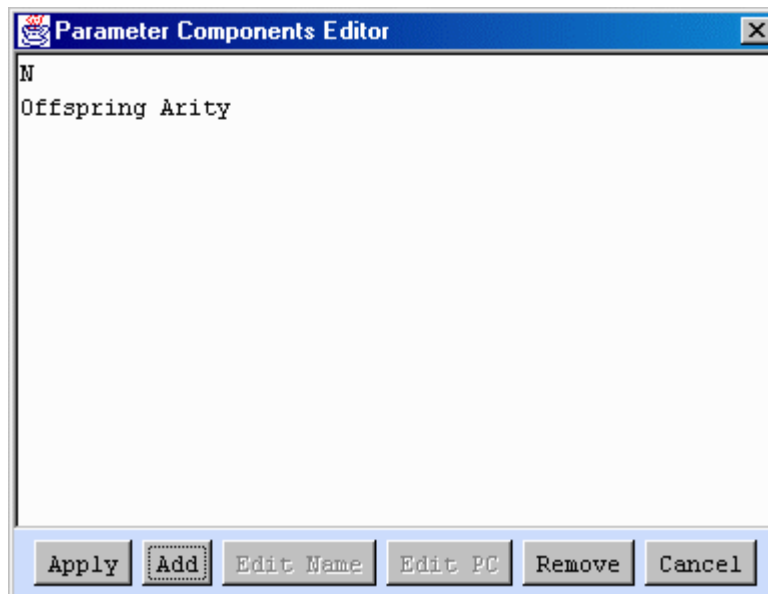


Figure 5.33: Added two parameter components for the new recombination operator.

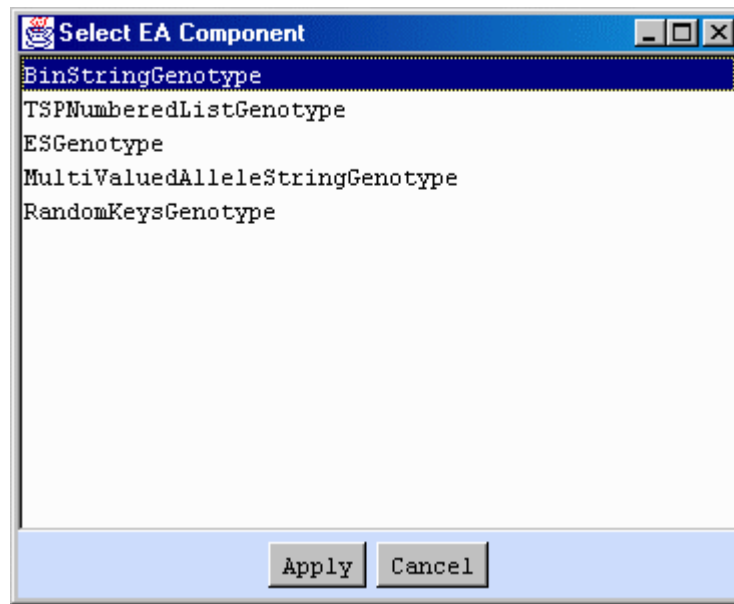


Figure 5.34: Selecting the binary string genotype as a dependency.

genotypes that we can add as a dependency. We select the `BinStringGenotype` as shown in figure 5.34 and press the *Apply* button. As a result, the `BinStringGenotype` is now displayed in the rightmost list of the dependencies editor as can be seen in figure 5.35. Having defined the dependencies, we press the *Apply* button to return to the class edit window for our new recombination operator.

We are now ready to edit the JAVA code for the n -point crossover operator. The *EA Visualizer* editor has already specified three methods that we need to write method bodies for. These methods are *required* and must therefore be defined. The most important of these is of course the *recombine* method. First of all though, we focus on the constructor method. Doing so makes us realize however that we need to do nothing in the constructor. All that we need to do is done when we are asked to perform recombination. At that point are required to know however what the offspring arity is and what the amount of crossover points are. To this end, we require class variables in which this information can be stored once the parameters are set. Concluding, the first few lines of our class become the following:

```
public class BinStringNPointCrossover extends Recombinator
{
    private int offar, n;
```

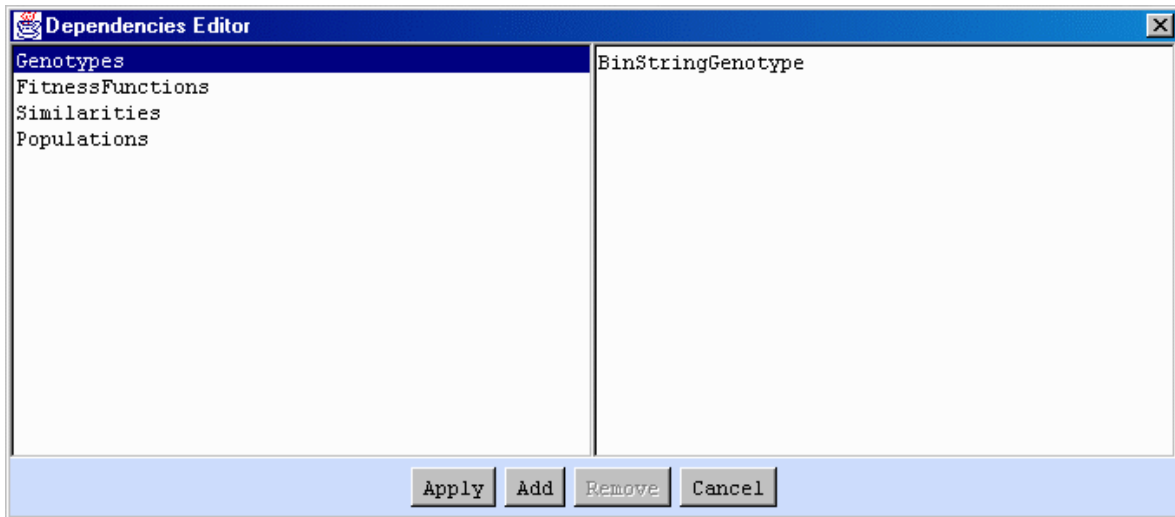


Figure 5.35: Defining the dependencies for n -point crossover.

```
/**
 * Creates a new n-point crossover operator.
 */
public BinStringNPointCrossover()
{
}
```

The next method is the method that returns the offspring arity for this recombination operator. We have specified however that we want the user to be able to define this number through a parameter component. To this end, we need to set the variable `offar` we defined as class variable. This needs to be done when the *EA Visualizer* ships the parameter values to this class. When we dig into the code of the `EAComponent` class, we find that this is done through a certain method:

```
protected void actuallySetParameters( Vector parameters )
```

The parameters are defined inside a JAVA datastructure called the `Vector`. This is a dynamic array of `Object` elements. The first element of this `Vector` is the result of our first parameter component. We already know that this parameter component is the parameter component that defines long values within a certain range. The result of that parameter component we can find in the class for that parameter component under the `getSelectedValue()` method. It turns out that the result is `Long` object, which is a wrapper class for long values. Noting this and keeping in mind that the second parameter is just such a long parameter as well, we can write code for unraveling the parameters at runtime:

```

/**
 * Sets the parameters. The parameters are
 * <ol>
 * <li> The value for N.
 * <li> The amount of offspring to generate (1 or 2).
 * </ol>
 * @param parameters the resulting values from the parameter components.
 */
protected void actuallySetParameters( Vector parameters )
{
    n      = ((Long) parameters.elementAt( 0 )).intValue();
    offar  = ((Long) parameters.elementAt( 1 )).intValue();
}

```

We can now continue with defining the next required method, namely the one returning the offspring arity. This is now a trivial method, because we need to return of course nothing else than the value inside the class variable `offar`:

```

public int offspringArity()
{
    return( offar );
}

```

This finally brings us to the method that actually performs the crossover operation. This is done in the `recombine` method. The header for this method in the `Recombinator.java` file tells us what this method actually needs to do:

```

/**
 * Recombines the parent genomes. The result should be a collection of
 * new <code>Genotype</code> objects and not the old genotypes that were
 * altered, because of information that might be extracted at a later
 * stage for visualization.
 * @param parents    the <code>Vector</code> of parents to recombine.
 * @param ff         the fitness function used by the EA.
 * @param prng       the PRNG used by the EA.
 * @param generations the generation counter.
 * @return a <code>Vector</code> of offspring.
 */

```

The parameters are a collection of parent genomes coded as `Genotype` objects, the fitness function, the random number generator and the generation counter. The result should be a collection of offspring genomes, where the collection is just as the collection of parent genomes, a `Vector`. We are now ready to write the code. Before doing so, we note that we wish to generate some warning messages when the input collection of parent genomes for instance doesn't contain two elements. Such warning messages can be placed

on the message frame in the *EA Visualizer* by using the global variable `mf`. The method `addMessage(String message)` can be used to add a message to that message frame at run time. Furthermore, determining the crossover points can be done using boolean variables, where for each crossover position a boolean variable is used. If the variable is set to `true`, the position is a crossover position, otherwise it is not. At first, n random positions are chosen to become crossover positions. If a position is selected twice, there is no need to worry, because the crossover position can simply be reset to become `false` again, because the set of bits between the crossover points is empty, resulting in that the parts before and after the crossover points are to be copied with the same ordering of parents. Concluding, the complete code for the newly generated class in the *EA Visualizer* is the following:

```
public Vector recombine( Vector parents, FitnessFunction ff,
                        Similarity sim, PseudoRandomNumberGenerator prng,
                        long generations )
{
    int          i, pos, len;
    boolean      cpoints[], p1_to_o1;
    Vector       result;
    BinStringGenotype p1, p2, o1, o2;

    if( parents.size() < 2 )
    {
        mf.addMessage(name + ": Must have 2 parents, received less.");
        mf.addMessage(AdditionalString.spacesInstead( name ) + " null returned.");

        return( null );
    }
    else if( parents.size() > 2 )
    {
        mf.addMessage(name + ": Must have 2 parents, received more.");
        mf.addMessage(AdditionalString.spacesInstead( name ) + " Ignoring superfluous parents.");
    }

    p1 = (BinStringGenotype) parents.elementAt( 0 );
    p2 = (BinStringGenotype) parents.elementAt( 1 );
    len = p1.getLength();

    if( len != p2.getLength() )
    {
        mf.addMessage(name + ": Parents have unequal length.");
        mf.addMessage(AdditionalString.spacesInstead( name ) + " null returned.");

        return( null );
    }

    if( len < n )
    {
```

```

mf.addMessage(name + ": Amount of crossover points larger than string length.");
mf.addMessage(AdditionalString.spacesInstead( name ) +
    " reduced crossover points to string length.");

    n = len;
}

/* Determine crossover positions, make offspring bitstrings and
** perform actual crossover */
cpoints = new boolean[len];
for( i = 0; i < len; i++ )
    cpoints[i] = false;

for( i = 0; i < n; i++ )
{
    pos          = (int) prng.randomNumber( len );
    cpoints[pos] = !cpoints[pos];
}

result = new Vector( offar );
o1      = new BinStringGenotype();
o2      = new BinStringGenotype();
o1.administerGenotype( p1 );
o2.administerGenotype( p2 );
o1.justSetLength( len, p1.getTwoPowerLength() );
o2.justSetLength( len, p1.getTwoPowerLength() );
p1_to_o1 = true;
for( i = 0; i < len; i++ )
{
    if( cpoints[i] )
        p1_to_o1 = !p1_to_o1;

    if( p1_to_o1 )
    {
        o1.setBit( i, p1.getBit( i ) );
        o2.setBit( i, p2.getBit( i ) );
    }
    else
    {
        o1.setBit( i, p2.getBit( i ) );
        o2.setBit( i, p1.getBit( i ) );
    }
}

result.addElement( o1 );

if( offar > 1 )
    result.addElement( o2 );

return( result );
}

```

Having written the code, this code can be compiled using the F2 key or by selecting *Compile* from the *File* menu in the class editor. The class is then compiled into a classfile that JAVA can read and the *EA Visualizer* will find that class at runtime. If no such classfile is generated, the *EA Visualizer* will give a warning message in the message frame. After compiling the class file, only one thing remains to be done in the editor, namely synchronization. This can be done by selecting *Synchronize* from the *System* menu in the editor frame. This will cause the editor to present the following messages if all is performed without problems:

```
Updating system files, please wait...
  Generating EAComponentCreator.java... Compiling...
  Generating EAViewCreator.java... Compiling...
  Generating NameSystem.java... Compiling...
  Writing encrypted file with EAComponent definitions.
  Writing encrypted file with EAView definitions.
  Writing encrypted file with NameSystem definitions.
System synchronized.
```

If there were any problems along the way, there is something wrong with the input you gave for the class you just edited, in one of the options from the *Additional* menu. In such a case, the editor will not leave the *modified* mode and the editor cannot be exited³. If all is right, there will be no problems and the synchronization will be successful. After this, the editor can be exited and the *EA Visualizer* system be started, or this can be done immediately from the editor by selecting to do so from the *System* menu. Using the resulting new *Recombinator* is an aspect that is more of the kind of applications we described in section 3.4.1, so we refer the user to that section in order to test the newly added *Recombinator* or of course to his or her own experience that by now may allow the user to use the new *Recombinator* without any external help.

5.6.2 The implementation of FDA

In this section take a quick look at how the FDA algorithm has been implemented in the *EA Visualizer*. In section 3.4.5 we already gave a theoretical description of FDA as well as a practical approach which showed how FDA could be incorporated in the system. At this point, this information together with the information found on the editor in the previous sections, should already be enough to understand how FDA can be implemented. We shall therefore go over the details swiftly as less extensive example than what we presented in section 5.6.1.

First of all, we should realize what parameters are needed and what dependencies are required. By specifying these, the reader is expected to be able to fill in the blanks in

³Unless this is done by brute force.

how these values are actually specified in the editor, using the example from the previous section. We shall now look closer at the implementation of FDA in the *EA Visualizer*⁴.

Looking at the parameters, we should realize that the input is twofold, namely the sets s_i that make up the factorization within FDA and the amount of samples that should be generated on the basis of the factorization and the actual samples. The selection factor is of no importance because this only determines the amount of parents that are used to base the probability distribution on, and this is simply runtime input to our new *Recombinator* called FDA. This means that we can specify the following to parameters as follows using the class editor:

Name	Distribution
Parameter Component	SetOfMultipleLongsParameterComponent()
Name	New Samples
Parameter Component	LongRangedParameterComponent(0, Long.MAX_VALUE, 200)

The `LongRangedParameterComponent` was already introduced in the previous section, where we extensively presented as an example the n -point crossover operator. In this tutorial however, the `SetOfMultipleLongsParameterComponent` was not mentioned before. This parameter component was devised because it was required for the implementation of FDA. Before going into the specifications of this parameter component, we point out that when creating your own instances components or views in the system and you reach the point where you must define what parameter components you desire to use, you should first check the parameter components that have already been created, because the most common and even some less common of types of input are already available in the current collection of parameter components.

The input that specifies the sets s_i for the FDA recombination operator defines the factorization that is used. This part of the input for FDA is thus a set of multiple integer numbers. As we already saw in section 5.6.1, the `LongRangedParameterComponent` can be used for single integers, or scalar values. The `MultipleLongParameterComponent` can be used as an extension over this to specify multiple numbers, or vector values. The next logical extension over this, is the `SetOfMultipleLongsParameterComponent` (the one we require for FDA), which can be used to specify a multiple of such multiple numbers. This is done by presenting a single textfield in which the user can enter a string. This string should be a list (set) of lists of long values, where the long values are separated by spaces. This is also the case for the `MultipleLongParameterComponent`. The sets themselves have to be bracketed (between '(' and ')'). Any non integer valued or wrongly bracketed input is not accepted and triggers a popup message with the exact error to appear. An example is thus as we presented earlier in section 3.4.5 the following input:

⁴The actual implementation in the *EA Visualizer* has an additional parameter called perturbation factor which is normally not used in FDA, but was used for research [7].

```
(0 1 2 3 4)(5 6 7 8 9)(10 11 12 13 14)(15 16 17 18 19)(20 21 22 23 24)(25 26 27 28 29)
(30 31 32 33 34)(35 36 37 38 39)(40 41 42 43 44)(45 46 47 48 49)
```

This input is the representation required in the *EA Visualizer* for $s_i = \{5i, 5i+1, \dots, 5i+4\}$ for $i \in \{0, 1, \dots, 9\}$, resulting in the following probability distribution to be used in FDA: $p(X) = \prod_{i=0}^9 p(X_{5i}, X_{5i+1}, \dots, X_{5i+4})$.

Next to the input for the recombination operator, we should realize what the dependencies are for this operator. As the FDA operator as a standard works on strings of discrete variables that have a binary domain, the implementation will require binary strings. To enforce this, we enter as a dependency that the *Genotype* be restricted to the `BinStringGenotype` instance. Having entered this information and having edited the name for the new *Recombinator* to be FDA and having written the help page, we are ready to go over the implementation of FDA.

The class variables that we require to implement the FDA recombination operator clearly contain the parameters we just defined. In addition, from the sets s_i , the sets b_i and c_i are derived, so we shall require data structures for those as well, since we shall want to compute them once when the parameters are set and not over again each time we have to recombine the input genomes. In addition, we could make a general structure for the probabilities that have to be empirically determined. We could create this datastructure over again at each recombination operation, but here we choose to create a class variable for the probability distribution that follows from the sets s_i so that every time we need to recombine, we can fill in the actual values in this distribution datastructure and don't have to request for new memory. The datastructure required for the probability distribution is a two dimensional array of doubles. For each set s_i , we require an array of doubles, where the length of that array is determined by the size of sets b_i and c_i together. For each possible assignation of bitvalues to the discrete variables from the sets b_i and c_i we have an entry in the array, so the required array as a first dimension the amount of input sets s_i and as a second dimension $2^{|b_j|+|c_j|}$ with $j = \max_i\{|b_i| + |c_i|\}$. This results in the following first few lines for the FDA recombination operator as a class in the *EA Visualizer* system:

```
public class FDA extends Recombinator
{
    private int          generate;
    private double       prob[][];
    private DynIntValues s[], b[], c[];

    /**
     * Creates a new FDA recombinator.
     */
    public FDA()
    {
    }
}
```

The arrays `s[]`, `b[]` and `c[]` for the three different type of sets contain variables of type `DynIntValues`. This is a general utility class in the *EA Visualizer* that codes nothing but a dynamic array for `int` values. Looking closer at the parameters, we already noted that we require two parameters for this recombination operator and noted that we required two different parameter components. The amount of samples to generate can simply be unfolded from the parameter component defining long values, but the result from the parameter component defining sets of multiple longs is a `Vector` of `DynLongValue` objects, that are dynamic arrays for `long` values. This structure is thus somewhat more complex, so we leave it another method to do the unraveling of the input and convert the result of the parameter component to the required internal structure in the `FDA` class conform to the specification of the sets s_i , b_i and c_i as presented in section 3.4.5:

```

/**
 * Sets the parameters. The parameters are
 * <ol>
 * <li> The sets <em>si</em>
 * <li> The amount of samples to generate
 * </ol>
 * @param parameters the resulting values from the parameter components.
 */
public void actuallySetParameters( Vector parameters )
{
    generate = ((Long) parameters.elementAt( 1 )).intValue();

    makeSets( (Vector) parameters.elementAt( 0 ) );
}

/**
 * Creates the sets s, b and c that are used by the FDA.
 */
private void makeSets( Vector sets )
{
    int          i, j, sz, sz2, sz3, max, value, len;
    DynLongValues values;

    max = 0;
    len = 0;
    sz  = sets.size();
    s   = new DynIntValues[sz];
    b   = new DynIntValues[sz];
    c   = new DynIntValues[sz];
    for( i = 0; i < sz; i++ )
    {
        s[i] = new DynIntValues();
        b[i] = new DynIntValues();
        c[i] = new DynIntValues();

        values = (DynLongValues) sets.elementAt( i );
        sz2    = values.size();

```

```

for( j = 0; j < sz2; j++ )
{
    value = (int) values.valueAt( j );
    s[i].addValue( value );
    if( !historyContains( i-1, sets, value ) )
        b[i].addValue( value );
    else
        c[i].addValue( value );
}

len = b[i].size() + c[i].size();
if( len > max )
    max = len;
}

max = (int) AdditionalMath.power( 2, len );
prob = new double[sz][max];
}

/**
 * Simple enumerative method that looks whether the history sets
 * contain a certain item.
 */
private boolean historyContains( int i, Vector sets, int value )
{
    int j, k, sz;
    DynLongValues values;

    for( j = 0; j <= i; j++ )
    {
        values = (DynLongValues) sets.elementAt( i );
        sz = values.size();
        for( k = 0; k < sz; k++ )
            if( ((int) values.valueAt( k )) == value )
                return( true );
    }

    return( false );
}

```

As we have seen before in the previous section, we have to define the method `offspringArity()` that returns the amount of offspring this recombination operator produces. In the case of section 5.6.1 where we defined the n -point crossover operator, this amount was equal to either 1 or 2, dependent on the input of the user. In this case we have a very similar result as the amount of offspring to generate comes immediately from the user input and as we just saw, this amount is stored in the variable `generate`. Concluding, we can write the following method:

```

public int offspringArity()
{
    return( generate );
}

```

The final point is performing the actual recombination operation. In the case of FDA, first the probability distribution structure has to be computed exactly, after which the samples can be generated. Determining the probability distribution is done based on the parent genomes and the length of the binary strings, whereas the generation of the samples is done based on the probability distribution structure (which is coded as a class structure) and the prng, which is required to generate samples which a certain probability. Concluding we can code this as follows:

```

/**
 * Recombines the parent genomes according to the above specified framework.
 * @param parents    the <code>Vector</code> of parents to recombine.
 * @param ff         the fitness function used by the EA.
 * @param prng       the PRNG used by the EA.
 * @param generations the generation counter.
 * @return a <code>Vector</code> of offspring.
 */
public Vector recombine( Vector parents, FitnessFunction ff,
                        Similarity sim, PseudoRandomNumberGenerator prng,
                        long generations )
{
    int          length;
    BinStringGenotype bgt;

    bgt    = (BinStringGenotype) parents.elementAt( 0 );
    length = bgt.getLength();

    computeProbabilities( parents, length );

    return( generateSamples( bgt, prng ) );
}

```

What remains is to define the methods `computeProbabilities()` and `generateSamples()`. Computing the probabilities is done based on the input sets s_i that make up the factorized distribution. As the required probabilities in FDA are mainly of the form $p(\prod_{X \in b_i} X \mid \prod_{X \in c_i} X)$, we realize we can compute this as

$$\frac{\prod_{X \in c_i} X \wedge \prod_{X \in b_i} X}{\prod_{X \in c_i} X}$$

By going over all possible $2^{|b_i|+|c_i|}$ combinations for all i , we compute the empirical probabilities for the underlying distribution:

```

/**
 * Determines the probabilities based on the distribution provided
 * by the user on beforehand. This is done by enumerating all assignable
 * combinations of bit values to the substructures <code>b[i]</code> and
 * <code>c[i]</code>. The probability <code>p(b[i]|c[i])</code> is computed
 * as <code>p(b[i] & c[i]) / p(c[i])</code>. The results are stored in a
 * two dimensional array of doubles, named <code>prob</code>. The main index
 * of this array is the set indicator and the subindex is the integer
 * representation of the bitvalue assignation to <code>b[i]</code>
 * concatenated to <code>c[i]</code>.
 */
private void computeProbabilities( Vector parents, int length )
{
    int          i, j, k, l, sz, sz2, sz3, len, szb, szc, bits[],
                countab, countb;
    boolean      match, match2;
    BinStringGenotype bgt;

    sz = s.length;
    for( i = 0; i < sz; i++ )
    {
        szb = b[i].size();
        szc = c[i].size();
        len = szb + szc;
        bits = new int[len];

        sz2 = (int) AdditionalMath.power( 2, len );
        for( j = 0; j < sz2; j++ )
        {
            countab = 0;
            countb = 0;
            sz3 = parents.size();
            for( k = 0; k < sz3; k++ )
            {
                bgt = (BinStringGenotype) parents.elementAt( k );

                // Does string match with bits in ci only?
                match = true;
                for( l = szb; l < len; l++ )
                    if( (bgt.getBit( c[i].valueAt( l - szb ) ) ? 1 : 0) != bits[l] )
                    {
                        match = false;
                        break;
                    }
                if( match )
                {
                    countb++;

                    // Does string match with bits in bi and ci?
                    match2 = true;
                    for( l = 0; l < szb; l++ )

```

```

        if( (bgt.getBit( b[i].valueAt( 1 ) ) ? 1 : 0) != bits[l] )
        {
            match2 = false;
            break;
        }
        if( match2 && match )
            countab++;
    }
}

if( countab > 0 )
    prob[i][j] = ((double) countab) / ((double) countb);
else
    prob[i][j] = 0;

    tickBits( bits, len );
}
}
}

/**
 * Increments the bit representation in an array by a single unit.
 */
private void tickBits( int bits[], int len )
{
    int i;

    for( i = len - 1; i >= 0; i-- )
    {
        if( bits[i] == 0 )
        {
            bits[i] = 1;
            break;
        }
        else
            bits[i] = 0;
    }
}
}

```

The second part of the functionality of the FDA recombination algorithm to code is the `generateSamples()` method that actually generates the samples. It is completely straightforward of course to generate a multiple of samples in one method as every sample is generated in the exact same way:

```

/**
 * Generates the samples, which is no more than repeating the generation
 * of a single sample a specified amount of times.
 */

```

```

private Vector generateSamples( BinStringGenotype bgt,
                               PseudoRandomNumberGenerator prng )
{
    int    i;
    Vector samples;

    samples = new Vector( generate );

    for( i = 0; i < generate; i++ )
        samples.addElement( generateSample( bgt, prng ) );

    return( samples );
}

```

Generating a single sample can be done by going over the sets b_i c_i . Inspecting the definition shows that going over these sets with increasing i will not encounter any impossibilities on bits to be set before they are used. The probabilities are stored in the array `prob`. The least significant part of the integer that is the second index to this array, are the bits for c_i . These bits have already been set, so these are looked up if the c_i set is not empty. This information is then given and the right probabilities need to be filtered from the remainder of the array. This is done by stepping through the second indices with a period of $2^{|c_i|}$ as $|c_i|$ bits are used in the integer representation that is the second index to the array. This is not required when the set c_i is empty. Having thus found the right probabilities and thus the right indices which in turn tell us the configurations for b_i , a single selection is done in which the chance at selection is of course directly the chance stated in `prob`. Having thus found the values for the bits pointed to in b_i , these values are used to partially build the (offspring) sample. This procedure is looped over all sets i , thus completing the total build of the (offspring) sample:

```

private BinStringGenotype generateSample( BinStringGenotype bgt,
                                          PseudoRandomNumberGenerator prng )
{
    int          i, j, sz, len, pos, szb, szc, low, high,
                period, two, mask;
    BinStringGenotype offspring;

    offspring = new BinStringGenotype();
    offspring.administerGenotype( bgt );
    offspring.justSetLength( bgt.getLength(), bgt.getTwoPowerLength() );

    sz = s.length;
    for( i = 0; i < sz; i++ )
    {
        szb = b[i].size();
        szc = c[i].size();
        len = szb + szc;
    }
}

```



```

    if( szc == 0 )
    {
        period = 1;
        low     = 0;
        high    = ((int) AdditionalMath.power( 2, szb )) - 1;
    }
    else
    {
        period = (int) AdditionalMath.power( 2, szc );
        low     = 0;
        two     = 1;
        for( j = szc - 1; j >= 0; j-- )
        {
            if( offspring.getBit( c[i].valueAt( j ) ) )
                low += two;
            two *= 2;
        }

        high = ((int) AdditionalMath.power( 2, len )) - 1;
    }

    pos = cumulativeSelection( i, low, high, period, prng );
    two = (int) AdditionalMath.power( 2, szc );
    for( j = szb - 1; j >= 0; j-- )
    {
        offspring.setBit( b[i].valueAt( j ), (pos & two) > 0 );
        two *= 2;
    }
}

return( offspring );
}

```

The above implementation of course requires to define the method `cumulativeSelection()` that goes over the array `prob` with the first index (in the first dimension) fixed and a maximum and a minimum for the second index (in the second dimension) as well as a period to step through that range. This stepping with a certain period (namely the period $2^{|c_i|}$) keeps the configuration for the bits in c_i intact. What is required is to pick one entry for the b_i configuration such that the choice is made based on the probabilities computed in the first part of the recombination operation. This can be done by cumulating the probabilities for the entries and by then drawing a random number between 0 and 1 that defines which entry is actually to be selected:

```

/**
 * Does the selection from the <code>prob</code> array at a given
 * main index, low and upper bound as well as stepping period and
 * a prng to do the actual selection. The probabilities are made
 * cumulative and then by choosing a number between 0 and 1, the actual

```

```

* selection is performed.
*/
private int cumulativeSelection( int which, int low, int high, int period,
                               PseudoRandomNumberGenerator prng )
{
    int    i, j, sz, poss[];
    double cumul[], value;

    sz      = ((high - low) / period) + 1;
    cumul   = new double[sz];
    poss    = new int[sz];

    cumul[0] = prob[which][low];
    poss[0]  = low;
    j        = 1;
    for( i = low+period; i <= high; i += period )
    {
        cumul[j] = cumul[j-1] + prob[which][i];
        poss[j]  = i;
        j++;
    }

    for( i = 0; i < sz; i++ )
        cumul[i] /= cumul[sz-1];

    value = ((double) prng.nextRandomNumber()) /
            ((double) Long.MAX_VALUE);

    for( i = 0; i < sz; i++ )
        if( cumul[i] >= value )
            return( poss[i] );

    return( poss[sz-1] );
}

```

Finally, we end with the contents of the help page coding to give yet another example of how to write help pages. As a very small exercise, the reader should start up the program and look up the FDA page and try to understand how the code in the help file results in the formatted graphical text on screen. The actual help page coding is the following:

```

This class is a recombinator conform the \color[255,0,0]{FDA} algorithm
proposed by Muhlenbein. Whereas algorithms such as
\seealso[eavmodel.recombinator.MIMIC] and the approach by Baluja and Davies
(\seealso[eavmodel.recombinator.BalujaDaviesOptimalDependencyTrees])
concentrate on finding a bivariate distribution over a set of samples,
\color[255,0,0]{FDA} (\emph{Factorized Distribution Algorithm}) assumes a
factorization of the probability distribution for an ADF
(\emph{Additively Decomposable Function}) is given and works with that
distribution to empirically compute probabilities. Based on these

```

probabilities and the a priori determined distribution, new samples are generated.\\

It is therefore that the user is prompted to provide a factorization which is the distribution to be used in `\color[255,0,0]{FDA}`. This specification is done by providing sets `\emph{si}`. These sets are used to compute sets `\emph{bi}` and `\emph{ci}` for `\emph{i = 1, 2, ..., l}` where `\emph{si}` is a set of loci, `\emph{di}` is the union of all sets `\emph{sj}` with `\emph{j <= i}`, `\emph{bi}` is set `\emph{si}` without the contents of set `\emph{d{i-1}}` and `\emph{ci}` is the intersection of set `\emph{si}` with set `\emph{d{i-1}}` (set `\emph{d0}` is set to the empty set). The probability distribution used is then `\emph{p(P(b1)p(P(b2)|P(c2))p(P(b3)|P(c3))...p(P(bl)|P(cl))}` with `\emph{P(s)}` the product of all elements in `\emph{s}`.\\

The dynamics of `\color[255,0,0]{FDA}` is as follows:

1. Initialization: generate `\emph{N}` samples
2. Selection: select `\emph{tN}` samples with `\emph{t} < 1`
3. Compute the probabilities `\emph{p(P(bi)|P(ci))}`
4. Generate a new population (`\emph{N}` samples) according to the distribution
5. If termination criterium is met, stop
6. Add the best sample of the previous generation to the generated points (elitist)
7. Go to the selection step.\\

Additional to this, the FDA algorithm can be perturbed by adding elements to the `\emph{ci}` sets from the `\emph{b{i-1}}` sets. This was originally programmed for testing with the trap functions. The perturbation factor equals the amount of elements to randomly add from the `\emph{b{i-1}}` sets for each `\emph{ci}`. The elements taken from the `\emph{b{i-1}}` sets are taken from index 0 to the amount desired, so not at random.\\

```
\bold{Example} (how to install this recombinator in the EA Visualizer):\\
\seealso[eavmodel.genotype.Genotype]\color[255,255,255]{*****}:
  \seealso[eavmodel.genotype.BinStringGenotype]\\
\seealso[eavmodel.similarity.Similarity]\color[255,255,255]{*****}:
  \color[0,155,0]{Any}\\
\seealso[eavmodel.fitness.FitnessFunction]:
  \color[0,155,0]{Any}\\
\seealso[eavmodel.recombinator.Recombinator]\color[255,255,255]{****}:
  \color[255,0,0]{FDA}\\
\seealso[eavmodel.mutator.Mutator]\color[255,255,255]{*****}:
  \seealso[eavmodel.mutator.NoMutator]\\
Before \seealso[eavmodel.selector.Selector]\color[255,255,255]{*}:
  \seealso[eavmodel.selector.TruncationSelector]\\
After \seealso[eavmodel.selector.Selector]\color[255,255,255]**:
  \seealso[eavmodel.selector.AllSelector]\\
\seealso[eavmodel.mater.Mater]\color[255,255,255]{*****}:
  \seealso[eavmodel.mater.AllMater]\\
\seealso[eavmodel.replacer.Replacer]\color[255,255,255]{*****}:
  \seealso[eavmodel.replacer.BestSampleElitistReplacer]\\
```

```
\seealso[eavmodel.terminator.Terminator]\color[255,255,255]{*****}:  
  \color[0,155,0]{Any}\\  
\seealso[eavmodel.hybrid.HybridSearcher]\color[255,255,255]{*}:  
  \color[0,155,0]{Any}\\  
\seealso[eavmodel.population.Population]\color[255,255,255]{*****}:  
  \seealso[eavmodel.population.VectorPopulation]\\  
\seealso[eavmodel.prng.PseudoRandomNumberGenerator]\color[255,255,255]{*****}:  
  \color[0,155,0]{Any}
```

Chapter 6

Miscellaneous

In this chapter we describe a few miscellaneous issues that are part of the *EA Visualizer*. These issues are not the main operations of the system, but may be required or may aid the user in learning about system details. A description is given of how to unpack and install the *EA Visualizer* so that it is set up to be used. Also, information is provided on how the author can be contacted for more information, help or feedback.

6.1 Installation

In order to install the *EA Visualizer* you should be in the possession of the zipfile containing the program files. This zipfile can be downloaded at the WWW site of the *EA Visualizer*, for which the address is given in section 6.2. Once this zipfile is downloaded, it must be extracted using an unzip program that can handle long filenames (WinZip under WINDOWS 95/98/NT). Lets assume the files are extracted to the following directory on your system:

`C:\EAVisualizer`

Alternatively, if you aren't running WINDOWS but LINUX or UNIX, lets assume the directory is:

`$HOME/EAVisualizer`

Next, you must have a JAVA version installed on your system. The required JAVA version is 1.1.x, which means any 1.1 version (or greater). The latest version of the original JDK (Java Development Toolkit) can be freely downloaded at the WWW site of the inventors of JAVA, namely SUN:

<http://www.sun.com/java>

Once this is installed on your system, we assume that you have the `bin` directory of the executables of `java` and `javac` in the path of the shell you are working in. When you are using `LINUX` or `UNIX`, this mostly amounts to updating your `.profile` file in your `$HOME` directory to include the line that sets the path to incorporate the `bin` directory of `JAVA`. When using `WINDOWS`, the best way to do this is to make a `bin` directory on your system such as `C:\bin` and to create a batch file there with for instance your own name: `johndoe.bat`. This file should contain some lines such as:

```
@echo off
set path=%path%;c:\bin;d:\Languages\Java\bin;d:\Languages\BorlandC\bin
c:\windows\command.com
```

The above lines ensure that the `JAVA bin` directory is in the path, as well as that of other languages (for example a `C` compiler by Borland). Now you can make a shortcut to this batch file that starts in the directory `c:\bin` and executes `c:\bin\johndoe.bat`. This will give you a `DOS` shell in which you can run Java by typing `java`.

Having installed `JAVA` as well as the *EA Visualizer*, the system can now be run (for the first time) by going to the directory where the *EA Visualizer* was installed into and by then typing:

```
java EAVisualizer
```

When using `WINDOWS`, this should thus be entered in the shell that was just created. If the editor version of the system is to be started, the suffix `-editor` should be appended to the startup line just presented:

```
java EAVisualizer -editor
```

In addition, in case of memory problems, you can ask `JAVA` to increase the amount of memory that is used. This can be done by adding some commandline parameters that specify the memory size. In `JAVA 1.1` versions:

```
-msXXXXX specifies the amount of memory to use initially.
           Here XXXXX is the amount of bytes to use. Append k for
           kilobytes, m for megabytes.
-mxXXXXX specifies the amount of memory to use at maximum. Use the
           same modifiers to change to kilobytes or megabytes.
```

In JAVA 1.2 the flags have changed to `-Xms` and `-Xmx` respectively. The default values in JAVA 1.2 are `1m` and `16m` respectively. Here's an example of using a different memory model than the standard model¹:

```
java -ms16m -mx64m EAVisualizer
```

For a full specification of the command line options of the *EA Visualizer*, we present the so called *usage* information that is printed when the user makes an error in the command line options for starting up the system:

```
Usage      : java EAVisualizer [-EDITOR] [-NOIMAGES]
-EDITOR    : Start up editor system
-NOIMAGES: Don't use images to speed up drawing (saves memory)
```

Note:
ALWAYS START THE *EA Visualizer* BY CALLING **JAVA** IN THE
DIRECTORY WHERE THE *EA Visualizer* HAS BEEN INSTALLED!

If you start the *EA Visualizer* from some place else, the system will not be able to locate certain files and will not start. So in our example you should start the system in the directory `$HOME/EAVisualizer` when using LINUX or UNIX and in the directory `C:\EAVisualizer` when using WINDOWS.

In case of any problems, you should contact the author by reading the next section that contains the contact information.

¹Also see the `readme.txt` file on this and other issues such as *Known Problems*.

6.2 Contacting the author

The author (Peter A.N. Bosman) can be reached by email at the following address:

peterb@cs.uu.nl

Alternatively, the author can be reached during office hours by normal mail, phone, fax or physically by using the following contact information:

Name	Peter A.N. Bosman (M. Sc.)
Mail Address	PO Box 80.089 3508 TB Utrecht The Netherlands
Visiting Address	Universiteit Utrecht Department of Computer Science Centrumgebouw Noord, Office A219 Padualaan 14, De Uithof 3584 CH Utrecht
Office	CGN-A219
Telephone	+31 - 30 - 253 7594
Faxnumber	+31 - 30 - 251 3791

More information on the *EA Visualizer* can be found at the WWW site of the system. A mailing list is maintained there, which is used for keeping users updated of the latest versions, tips and tricks. Also the program can be downloaded there:

<http://www.cs.uu.nl/people/peterb/computer/ea/eavisualizer/EAVisualizer.html>

Bibliography

- [1] T. Bäck, M. Schütz and S. Khuri. Evolution strategies: an alternative evolutionary algorithm. In *Artificial Evolution* (J.M. Alliot, E. Lutton, E. Ronald, M. Schoenhauer, D. Snyers, eds.), Springer-Verlag, Berlin, 1996, pp. 3–20.
- [2] T. Bäck and H-P. Schwefel, Evolution strategies I: variants and their computational implementation. In *Genetic Algorithms in Engineering and Computer Science, Proc. First Short Course EUROGEN'95* (G. Winter, J. Priaux, M. Galn, and P. Cuesta, eds.), Wiley, New York, 1995, pp. 111-126.
- [3] S. Baluja and S. Davies. Using optimal dependency-trees for combinatorial optimization: learning the structure of the search space. In *Proceedings of the 1997 International Conference on Machine Learning* (D.H. Fisher, ed.), Morgan Kaufman Publishers, 1997. Tech Report: CMU-CS-97-107.
- [4] J. S. De Bonet, C. Isbell and P. Viola. MIMIC: finding optima by estimating probability densities. In *Advances in Neural Information Processing* volume 9 (M. Jordan, M. Mozer, M. Perrone, eds.), MIT Press, Cambridge, Denver 1996.
- [5] P.A.N. Bosman. A general framework and development environment for interactive visualizations of evolutionary algorithms and using it to investigate recent optimization algorithms that use a different approach to linkage learning. Utrecht University graduation paper INF-SCR-98-15. <http://www.cs.uu.nl/people/peterb/computer/papers.html>, 1998.
- [6] P.A.N. Bosman and D. Thierens. On the modelling of evolutionary algorithms. Utrecht University technical report UU-CS-1999-11. <http://www.cs.uu.nl/people/peterb/computer/papers.html>, 1999.
- [7] P.A.N. Bosman and D. Thierens. Linkage information processing in distribution estimation algorithms. In *GECCO-1999: Proceedings of the Genetic and Evolutionary Computation Conference* (to appear). Utrecht University technical report UU-CS-1999-10. <http://www.cs.uu.nl/people/peterb/computer/papers.html>, 1999.

- [8] C. Chow and C. Liu. Approximating discrete probability distributions with dependence trees. In *IEEE Transactions on Information Theory*, Volume 14, 1968, pp. 462-467.
- [9] G. Harik, E. Cantu-Paz, D.E. Goldberg and B.L. Miller. The gambler's ruin problem. Genetic algorithms and the sizing of populations. IlliGAL Report 96004, 1996.
- [10] S. Mahfoud. Crowding and preselection revisited. In *Parallel Problem Solving From Nature II - PPSN II* (R. Männer, B. Manderick, eds.), Springer-Verlag, Berlin, 1992, pp. 27-36.
- [11] K. Mathias and D. Whitley. Genetic operators, the fitness landscape and the traveling salesman problem. In *Parallel Problem Solving From Nature II - PPSN II* (R. Männer, B. Manderick, eds.), Springer-Verlag, Berlin, 1992, pp. 221-230.
- [12] H. Mühlenbein, T. Mahnig and O. Rodriguez. Schemata, distributions and graphical models in evolutionary optimization. ftp://borneo.gmd.de/pub/as/ga/gmd_as_ga-98_02.ps, 1998.
- [13] D. Thierens. Selection Schemes, Elitist Recombination, and Selection Intensity. In *Proceedings of the Seventh International Conference on Genetic Algorithms (ICGA97)* (Bäck, T. ed.), Morgan Kaufmann, San Francisco, CA, 1997.
- [14] X. Yin and N. Gernay. A fast genetic algorithm with sharing scheme using cluster analysis methods in multimodal function optimization. In: *Proceedings of the Artificial Neural Nets and Genetic Algorithms Conference - ANNGA 93* (R.F. Albrecht, C.R. Reeves, N.C. Steele, eds.), Springer-Verlag, Wien, 1993, pp. 450-457.