A Java Collaborative Interface for Genetic Programming Applications: Image Analysis for Scientific Inquiry

Tommaso F. Bersano-Begey, Jason M. Daida, John F. Vesecky and Frank L. Ludwig
The University of Michigan Artificial Intelligence Laboratory and Space Physics Research Laboratory
2455 Hayward Avenue Ann Arbor, Michigan 48109-2143
tombb@engin.umich.edu,daida@eecs.umich.edu, jfv@umich.edu and fludwig@unix.sri.com

ABSTRACT

This paper discusses several key issues involved in designing and using a Java collaborative interface for genetic programming applications over the World Wide Web. We present our implementation that has been used in a new system that assists scientists in classifying and extracting novel features in remotely sensed satellite imagery. This paper also identifies issues in developing a class library that facilitates rapid-prototyping of such collaborative graphical user interfaces for genetic programming, and suggests how other researchers could benefit from them.

1. Introduction

In the past several years, there has been an increasing interest in collaborative tools (e.g., [3] [5]). In the scientific community, collaborative tools and groupware have been used mostly in scientific domains in which the geographical distance between researchers and data or between various researchers has been an obstacle. Collaborative interfaces have already proven to be a successful contribution to space physics research, such as the Upper Atmosphere Research Collaboratory (UARC) [3], and to other fields, such as medical imaging [7]. Nevertheless, the use of collaborative interfaces has not yet been introduced to the field of Genetic Programming (GP), in part because many GP projects have not required close, or perhaps even simultaneous, interactions between different researchers.

In this paper, however, we discuss circumstances under which a close interaction between researchers from different backgrounds (e.g., various domain-specific backgrounds and computer programming) might be desirable (if not essential), subsequently motivating the use of a collaborative tool.

Consequently, we have implemented a collaborative interface designed with several key features in mind:

1. World Wide Web (WWW or Web) distribution, which enables collaboration through authentication by most any computer on the Internet.[1]
2. Graphical and interactive capabilities, which facilitates manipulation of complex data sets, often needed by domain applications, and allows for immediate display and analysis of run results.
3. Shared data, which allows all researchers in the team to work on the same data sets and execute runs remotely regardless of time and location.
4. A messaging system, which enables all users to send data and communicate remotely.
5. A flexible Object-Oriented (OO) design, which allows our interface to fit most needs and resources.

The focus of this paper is on the issues involved in our designing and using a collaborative tool that is build around a GP and image classification project and is shared over the Web. The rest of this paper is organized as follows. In Section 2, we introduce the concept of incompletely specified problems as a driver for collaboration, and other relevant concepts useful for our discussion. In Section 3, we then give an overview of the overall process involved in our research and discuss how it has improved with the use of our collaborative interface. In Section 4, we discuss important issues involved with the design, implementation, and performance of such interface. In Section 5, we present the separate components of our initial implementation. Then, in Section 6, we discuss how similar interfaces could be adapted to benefit other GP research projects. Finally, in Section 7, we state our conclusions.

2. Background and Relevant Concepts

In our investigations, we have applied genetic programming to problems in other domains outside of computer science (i.e., polar oceanography, meteorology). We have often encountered situations in which some of the investigators have a computer science expertise, while others have expertise in the scientific domain in which the problem at hand is posed.

Ideally, computer-science investigators are given a clear and accurate definition of a domain-science problem, so that they can independently and correctly set up and run a GP process for a specific application. The GP-evolved code is then often used as a finished product by the domain scientists, who often do need to be concerned about how such code was obtained.

In an ideal situation, the domain investigators would not have to participate in the process of GP evolution of code, and the computer-science investigators would not have to participate in the original formulation of the domain problem. Therefore, collaboration would not need to be a central issue.

In practice, however, a clear and accurate definition of the domain problem is often not forthcoming, and interaction between different researchers becomes a key concern. We have found collaboration helpful, where both computer- and domain-science investigators interact to modify training sets (e.g., fitness cases) or to better assess the results of evolved code.

2.1 Incompletely Specified Problems

As an aid to the discussion of the topics ahead, we describe the notion of incompletely specified problems. In particular, we

[1] A few recent Java interfaces also allow users to run GP on the Web [8][9], but are not designed for collaboration.

focus on how such problems can result in incomplete fitness sets used in GP, where we define an incompletely specified fitness set as a fitness set that does not exhaustively or sufficiently cover all relevant cases of the domain problem.

This could be due to several reasons, such as an incomplete knowledge of the phenomenon under scrutiny, varying interpretations of what makes up the problem, or the need to reduce the number of fitness cases to increase computational speed. In an incompletely specified problem, a score might not be a reliable enough measure of success: equal scores might lead to substantially different results and even lower scores might yield better results than higher ones.

This idea can be understood through a simple example. Given ten fitness cases and two individuals that score eight and five respectively (see Figure 1), it is easy to determine that individual I1 is probably better than individual I2 (a score of 8/10 versus a score of 5/10).

Figure 1. Unreliability of score in incompletely specified problems

```
                 Ten fitness-cases:
Individual I1: o--ooooooo score 8/10
Individual I2: oooo---o-- score 5/10
                 Four fitness-cases:
Individual I1: o--oOOOOOO score 2/4
Individual I2: oooo---o-- score 4/4

Hits are represented with an "o", misses with a"-"
```

However, assume that, accidentally, the previous trial was repeated exactly, except only the first four cases were used. In that case, I1 would appear to be worse than I2 (2/4 vs. 4/4), because of the distribution of hits within the fitness set.

2.2 The Role of Scaffolding

In our previous work, we have used scaffolding as a metaphor in approaching our inability to completely specify our domain problem [1]. Ordinarily, in developing code for scientific computation, we can identify three main steps. In the first, the program's task is defined. In the second, the program is written and goes through a series of steps (debug, test, re-write, re-compile) until it is conforming to the initial task specifications. In the third step, the program is tested on more specific and complex sets of data. Results are then processed, analyzed, and depending on the program's performance, the initial task definition could be improved and the developing process repeated. In GP terms, step 1 (the task definition) is partially represented by the fitness case selection, and step 2 is done through GP runs.

Often domain scientists might decide to improve their original selection of fitness cases after observing the performance of an individual program over a larger set of data. For instance, in the example given in Section 2.1, a scientist could decide to add the remaining fitness cases (e.g., going from 4 to 10 cases). After a new fitness set is created, the process can be repeated by performing a new run and analyzing the result.

Other ways to expand or improve a fitness set (which partially defines the problem to the genetic programming process) are to eliminate the least useful or the most misleading fitness cases while adding the most differentiating ones.

2.3 Frameworks of collaboration in GP projects

Figure 2 displays three main schemes in the use of GP for scientific applications, which are presented in increasing order of complexity and collaboration, and decreasing order of occurrence. This distinction is introduced to discuss the role of GP within a larger scientific research project, and to identify different needs for collaboration within each.

The flow diagram on the left is perhaps the most common and generic one. It consists of three main consecutive steps. First, the domain scientist arrives to a clear definition of a problem. Then, a computer scientist is given such definition, translates it in GP terms (operators, terminals and fitness cases) and runs the GP system until a satisfactory s-expression (code) is obtained. Finally, the resulting code is given back to the domain scientist, which incorporates it in his or her research.

The diagram in the middle is very similar to the previous one, but the given definition is either too complex or too vague to be completely understood by the computer scientist (or the problem to be analyzed is not yet well understood, thus lacking a straightforward definition). Thus, within a similar framework, the computer scientist will need the domain scientist's active participation every time the fitness cases have to be re-selected or an s-expression performance has to be assessed.

The third type of process, displayed on the right of Figure 2, is less common, and is the one used by our image analysis application. This is similar to the second type just described, except that the domain scientist is not able to define the domain problem, and uses GP in the process of hypothesis creation and refinement, in the attempt to reach a correct definition through successive approximations. In this latter scheme, the need for collaboration between scientists is even greater than in the previous scheme.

3. Application Overview

Our specific application has involved the design and implementation of an image analysis and pattern recognition tool, which uses GP to extract and classify novel features in satellite images [1]. Through successive hypothesis and runs, our system has been designed to produce custom-tailored algorithms for image pattern recognition.

Because the novel features to be extracted are often difficult to identify and lack a specific and complete definition [4], GP is used within a scaffolding system in the attempt to arrive to an accurate definition through successive approximations. In other words, after a tentative definition of a pattern is given (partially by selecting and classifying a small part of an image into a fitness set), the genetic programming system produces code to reflect such description. The code obtained is then used over a larger image, and the domain scientist visually inspects the resulting output, to decide how to improve the original hypothesis or definition (and thus the fitness set) and reiterate.

The collaborative interface described here was built around this image analysis problem, and one of its goals was to simplify a complex process, which has required a close interaction between researchers with different backgrounds and often in different locations and has been complicated by the issues previously described. Although our collaborative interface is still in its first revision, we were able to significantly improve the general process of our research project. This was accomplished in mainly by reducing dead time between different steps of the overall process and by simplifying the complex task of fitness cases selection. Shared data, for instance, eliminated the need for accessing and moving data from different computers, platforms or locations. Refer to [1] for details on the classification task and GP agent involved, and to [10] for an evaluation of the classification results.

3.1 Steps within the Core GP process

The gray areas in Figure 2 represent the core of the genetic programming system itself. In this context, we further divide

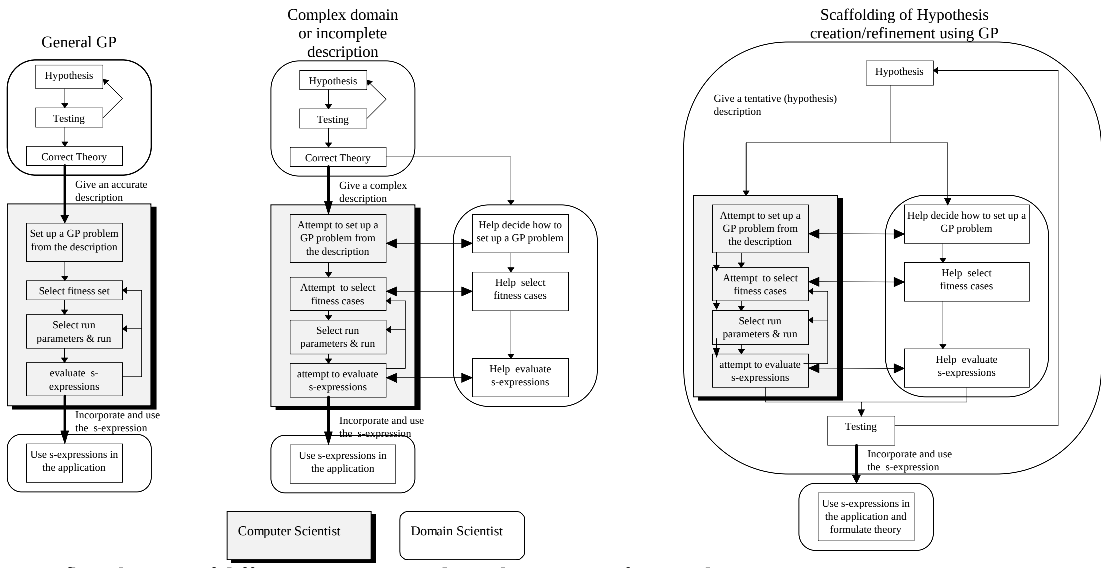

General GP
Hypothesis
Testing
Correct Theory
Give an accurate description
Set up a GP problem from the description
Select fitness set
Select run parameters & run
evaluate s-expressions
Incorporate and use the s-expression
Use s-expressions in the application
Complex domain or incomplete description
Hypothesis
Testing
Correct Theory
Give a complex description
Attempt to set up a GP problem from the description
Attempt to select fitness cases
Select run parameters & run
attempt to evaluate s-expressions
Help decide how to set up a GP problem
Help select fitness cases
Help evaluate s-expressions
Incorporate and use the s-expression
Use s-expressions in the application
Computer Scientist
Domain Scientist
Scaffolding of Hypothesis creation/refinement using GP
Hypothesis
Give a tentative (hypothesis) description
Attempt to set up a GP problem from the description
Attempt to select fitness cases
Select run parameters & run
attempt to evaluate s-expressions
Help decide how to set up a GP problem
Help select fitness cases
Help evaluate s-expressions
Testing
Incorporate and use the s-expression
Use s-expressions in the application and formulate theory

Figure 2. A flow diagram of different processes involving the creation of GP applications

this process in three steps, to help in the analysis and discussion of specific steps in our application. We examine the different tasks required by each step both before and after the introduction of our collaborative interface, to allow an assessment of the utility of such interface to our project. This division is introduced merely to simplify the discussion of the topics presented in this paper, and ignores other important factors such as the creation of problem-specific code and the successive possible revisions of it.

3.2 Pre-Processing and Fitness set creation

The task of the GP s-expression in our application is to build a classification algorithm from a set of texture filters of a given image (where it would be too lengthy or difficult for a human to do). Consequently, the training or fitness set must reflect closely these conditions. However, due to memory and speed issues, only a few pixels out of the entire image can be used [1]. Thus, the most significant and least controversial pixels must be respectively extracted from each filter.

Without a specific graphical interface, fitness selection involved observing an image, marking *x* and *y* positions of each pixel and locate their value in large text versions of each image.

A graphical interface allowed us to load, convert and zoom in and out of an image, select and deselect a pixel in all filters by simply clicking on it, and, in the same way, enabled any of the domain scientists collaborate directly and instantly in the fitness set selection from remote locations.

3.3 GP processing and runs

An advantage of using a graphical interface over the Web is that any user that is authenticated to the system can also execute a GP run remotely, observe the best resulting s-expression and other run parameters, and share the results with others. The interface allows different GP parameters and options to be changed, and results can be displayed in a graphical interactive format (e.g., graphs, editable text, etc.)

3.4 Post-Processing and Evaluation of results

Post-Processing is a critical part of the above process. Indeed, as the standard fitness score is no longer an entirely reliable measure of success, it is necessary to evaluate the s-expression on a whole out-of-sample subimage (rather than just on few pixels). Then, an on-site domain scientist has to visually inspect the subimage and find correlation between geographical features and the output of the individual.

Again, the use of a collaborative interface allows the domain scientists to see the graphical output of an s-expression a few minutes after the fitness set is created, and enables them to compare it with the original image (and perhaps with an annotated map of the location in question), discuss it with others and decide how to improve either the fitness set selection or the GP run parameters.

4. Implementation & Performance

4.1 Platform Independence

One of the advantages of using the Java language to implement a collaborative interface is its platform independence. The interface operates in essentially the same way on PC, Macintosh, and UNIX and should do the same on any other computer that has a Java interpreter or uses Netscape2.x or higher.

For a research team, platform-independence means that each researcher could use a different platform, and perhaps even work at home. Also, different steps of a research are often divided over many platforms, and require data to be moved and converted from one platform to another. Our interface can help eliminate such time-consuming intermediate steps.

4.2 OO & Flexible Implementation

The current implementation is a Java Applet, which uses URL connections to receive data, and uses Common Gateway Interface (CGI) scripts to send data and execute commands. The minimum requirement for this implementation is a CGI directory. Each user should have access to Netscape2.x or higher and be connected to the Web.

However, because of Java flexibility and an OO implementation, the same code can work in several different modes and with several different requirements. We are building a set of generalized classes which implement useful inheritance of classes and allow for a transparent, OO design.

Main groups of classes are responsible for Input/Output (I/O) connections, for graphic display of data (charts, graphs, and interactive bar charts), customized windows, etc.

Low-level classes, such as the I/O class or the customized windows class, are either referenced, instantiated or inherited by other higher level classes (i.e., chat windows, GP shells, etc.).

Furthermore, the I/O class implements a broader protocol of communication with other classes than conventional I/O. This allows to specify or disregard additional parameters specific to a connection type (i.e., CGI, file-system, etc.) while hiding the actual connection implementation. Thus, all classes requesting some form of I/O will interface in essentially the same way regardless of the available connection type.

4.2.1 Implementation Options

Due to an OO design and to Java flexibility, several implementation options are available. Depending on the implementers' means and needs, the same basic framework and code can be adapted to fit most circumstances. Table 1 displays the available options and their relative advantages.

Running Mode	Web page	Stand-alone	Local Applet	Compiled
Connection	CGI	Client/Server	File System	Client/Server
Platform-Ind.	YES	YES	YES	NO
Extension	WWW	WWW, LAN	LAN or Single User	WWW /LAN
Collaborative	YES	YES	only through F.S.	YES
Security	Secure Web Srv.	Network	LAN F.S	Network
Connection Type	URL / Post, Get	Sockets	File-System (r,w,x)	Sockets
Network Traffic	good for low tr.	Low-high	(none/LAN-FS)	low-high
User requirements	Netscape >=2.0	Network conn.	Netscape >=2.0	Network conn.
Server req.	CGI directory	Server program	(no Server)	Server Program
Speed	Interpreted	interpreted	interpreted	compiled (fast)

Table 1. Summary of Implementation options discussed in Section 4.

4.2.2 Java Running Options

Java offers several options besides Web applets.

Applets can run without Netscape or applet viewers, by simply adding a `main()` function to the original code. Our interface, for instance, contains both a `main()` and an `init()` function, and is thus both a Web Java applet and a stand-alone Java application. As a stand-alone application, it will lose its ability to make URL connections, but it will now be allowed to make any other network connection (e.g., client/server) and to load or save files to the local file system.

Another relevant distinction is that between local applets and remote applets. An applet becomes local when it is loaded from the local file system rather than from the network. Local applets can use the local file system, but cannot make URL or network connections. Remote applets, instead, cannot access the local file system, but are allowed to make URL or network connections with the server they are loaded from.

Furthermore, Java applications can be left as byte-code and run from a Java interpreter, thus retaining their platform independence, or be compiled (if the compiler is available) to become faster, platform-dependent executables.

4.2.3 Connection Type

As mentioned above, different types of applets allow different connection options. A remote Web applet can be shared over the WWW, and can make both URL and socket connections with the server from which it originates. A local applet can use the local file system to load, save and execute files. And while a stand-alone application cannot make URL connections, it can use both socket network connections and the local file system.

Again, all these options can be incorporated in the same OO design, or even left as run-time choices, so that the same code can be used in any of the above modes. In an OO design, for instance, a custom I/O class, which currently implements URL connections, can be substitute with (or instantiated as) another I/O class that implements the client end of a client/server connection, or a local file I/O connection.

Using URL connections has the advantage of using the existing Web server to implement the serving side of the network communication, but message broadcasting has to go through CGI and perhaps even the file-system, which due to the high overhead is not ideal for high traffic communication.

Client/server connections, both from Netscape or as a stand-alone application, are preferable for higher message traffic, but they require writing a custom server that must continuously run on a computer and on the network, which is often an undesirable consequence (and might not be allowed by the Internet provider). Local applets can work essentially in the same way as remote ones, by using the local file system in the same way as a CGI directory (or use commands like the UNIX talk to communicate).

4.3 Collaborative Design

While the interface does not share objects, it provides shared data and messaging. Both are currently implemented by accessing common files through CGI. The data is then copied locally to minimize network traffic, and is updated by request.

While some operating systems provide means to easily share distributed objects, these means are still platform dependent. Also, sharing an object could place a substantial overhead on the network, since objects are larger than the data they display.

Our current implementation was based on the premise of low traffic needs and considered request-for-updates an acceptable mean for sharing data. However, a client/server implementation would allow for direct broadcasting, and eliminate the need for requested updates. The collaborative interface that was developed for this project implements a chat interface and messaging system on the same structure of request-for-updates and shared data.

4.4 Security

Depending on the resources available and the implementation chosen, security might become a very important issue. Sharing an application over the Web means that anyone in the world could possibly use it and have access to the shared data.

There are several ways to get around this. The easiest and most effective one is to use a secure server that requires authentication to access a given address (or only accepts given IP addresses). Another option is to include authentication in the CGI scripts and in the Java code, and encrypt the data when needed. Also, Requesting IP addresses can be observed and either the CGI scripts or the Java code can decide to respond only to selected IP addresses. Since we did not have access to a secure Web server, we have implemented security through using Java code and CGI scripts.

4.5 GP Performance

In implementing the interface for our project, we also decided to use a GP kernel written in C,[2] to improve performance and to

[2] The C kernel used is lilgp. See [6].

avoid dealing with an interpreted language remotely. Previously, in fact, we worked with a LISP GP kernel, which was slow in dealing with large arrays. The GP kernel could also have been implemented in Java, but because Java is an interpreted language (thus slower than a compiled language) and performance is a central issue in genetic programming, we chose not to.

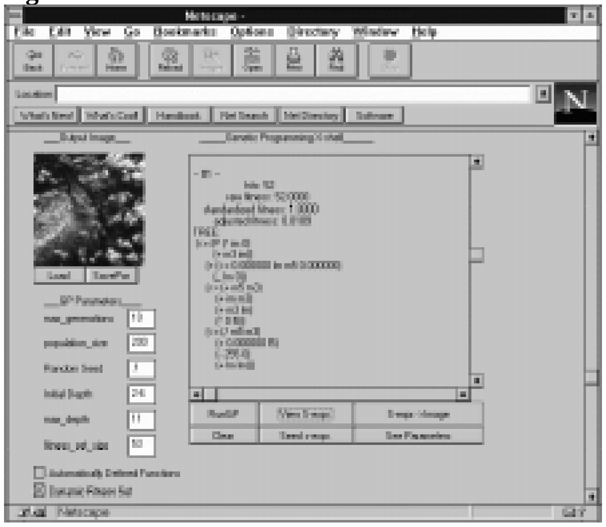
Unfortunately, while LISP does allow insertions of specific s-expressions (LISP programs) in a population by simply typing them, the version of the C kernel we had on hand did not. Consequently, we implemented a program that allows users to type any s-expression and introduce it (seed it) in the GP population of a given run. Besides the advantages of such an option, this modifications gave us the ability to test specific GP operators and other sections of problem specific code. Furthermore, with this program, s-expressions can be stored as text and re-used or later re-tested on different fitness sets.

The overall time required to evaluate a s-expression over an image containing 16,384 pixels (128 x 128) is approximately 15 seconds, and a generic run using a training set of 53 fitness cases, 20 generations and 200 individuals takes less than 1 min.

5. Interface Components

The first version of our collaborative interface consists of three parts: an interface for remote GP runs, a graphical interface for creating fitness cases and manipulating images, and a chat window. This section briefly covers the functionality of each part, and how they relate to the process described in Section 2.

Figure 3. The Remote GP Shell.


Netscape
File Edit View Go Bookmarks Options Directory Window Help
Output Image
Genetic Programming Shell
Load SavePic
GP Parameters
max_generations
population_size
Random Seed
Initial Depth
max_depth
fitness_set_size
Automatically Defined Functions
Dynamic Fitness Set
RunGP View S-exp S-exp->Image
Clear Seed s-exp See Parameters

5.1 The Remote GP Shell

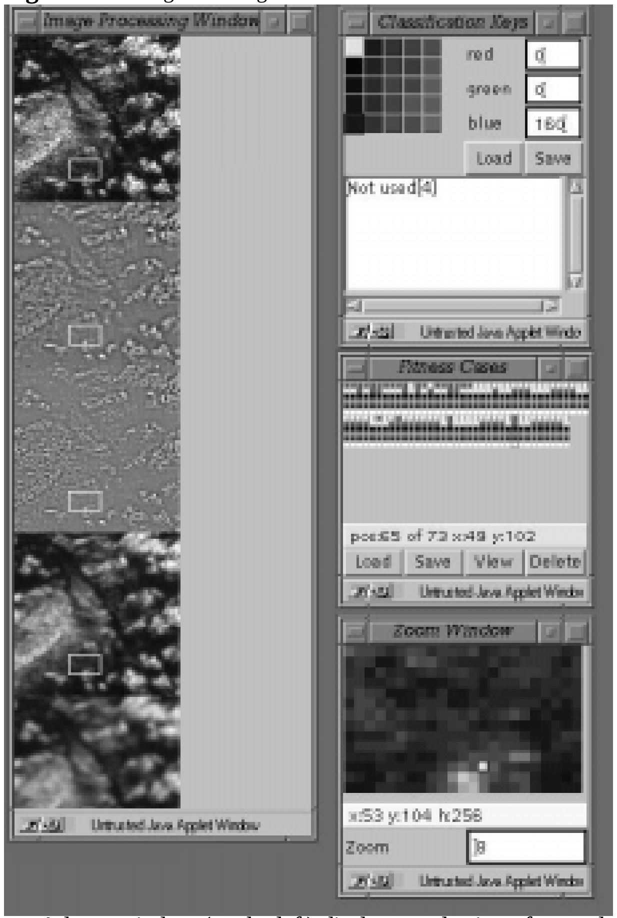
This component enables users to specify GP run parameters, execute GP runs, display a run's statistics (either the current user's or someone else's) and visualize outputs produced by an s-expression. Figure 3 shows how the interface appears on a PC running Windows NT. At the top left of Figure 3, a load button allows to read the shared output of a s-expression (e.g., a large text version of an image), convert it to an image format, and display it. Directly below the image, the user is allowed to modify various parameters for a GP run (we have implemented bounds to the number of generations, population size and tree depth to avoid attacks by resource exhaustion).

On the right side of Figure 3, a large text window displays the output of a run as it proceeds, display s-expressions, and can load the results of previous runs, and allow text to be copied to the local clipboard to be used by other applications.

5.2 The Image Processing Tool:

The Image Processing tool is the most problem-specific component of the interface, and was designed to manipulate and extract features for a fitness set specific to our domain problem. Figure 4 shows how the interface appears on a Sun Sparc 20 running Solaris.

Figure 4. The Image Processing Tool.



A large window (on the left) displays a selection of several filters from a satellite image, and allows users to move and resize the zoom area. A smaller window (on the bottom right) displays a selected portion of the image above, and additional information about the currently highlighted pixel..

A Fitness case consists of a set of pixels [1], each taken from the same relative coordinates in each of the filters). After their selection, pixels are marked on the image and zoom windows, and stored in the "Fitness Cases" window, which allows to undo and move any of the selections by a simple drag and drop mechanism. Highlighting a fitness case in this window highlights the respective pixels in the image and zoom window, to simplify deletion and selection processes.

Finally, the last window allows to specify how each fitness case (pixel) is to be classified (i.e., positive or negative example), as well as providing the option of annotations for each and any of the classification keys. These keys allow to associate different colors to different pixels in the image, and are called by reference, so that if the color of a classification key is changed, the respective pixels and fitness cases change accordingly.

5.3 The Chat Window:

The chat window has been implemented through CGI scripts, but we are planning to switch to a client/server implementation soon. Again, a request for update (using a thread that awakens at given time intervals) enables users to receive messages. The chat window allows users to communicate through messages, by accessing a common file. While this implementation is not very efficient, it only requires a CGI directory, does not assume any other specific resource and its performance is currently sufficient for our project.

6. Applications to a Generic GP Project

Through the previous sections, we have discussed how a Java collaborative interface has been flexibly implemented to adapt to most needs and resources. Furthermore, we have presented a custom interface built around our GP and image processing project as an example. In this section, we summarize issues justifying the use of similar interfaces in other GP projects, and how the interface presented in this paper can be changed to fit most needs.

The driving issues for a GP collaborative interface include the need for complex data manipulation (mainly for fitness set creation) and for collaborative data analysis. The main aspects of the interface presented in this paper are collaboration, accessibility, and graphical interactive display of data.

Under specific circumstances, a computer scientist might benefit from the expertise of specific domain scientists in defining the problem or analyzing the results. For instance, cases in which fitness set creation is particularly complex, or the domain problem is incompletely specified. Also, a collaborative interface would allow domain scientists to easily integrate in the hypothesis creation and refinement process (help in selecting fitness cases and analyzing the results). Since our interface is distributed and operates through the Web, it can provide all members of a research team easy access to all steps of the research process, regardless of their location or resources.

As discussed in Section 4, only one of the three main components of our interface could not be generalized and used in other GP project, namely the fitness set creator, or image processing tool. This is because fitness cases can be very specific to the type of data on which GP will act on. This component would most likely be customized and re-written for each application. The other components of the current implementation (the chat window and the remote GP shell) can be easily generalized and used in most other GP domains. Many GP applications share the same type of running parameters, and produce text s-expressions as output.

A fourth component, which in our case was incorporated in the remote GP shell, is a tool to help evaluate s-expressions over larger or different sets of data (for which a simple score might not be sufficient or available). In the more canonical GP examples of the artificial ant and the linear regression problems [2], for instance, this might involve visualizing the results in an animation and in a Cartesian graph respectively, while in our case it involved displaying an image. In any case, Java's audio, video and graphical capabilities can simplify the task of displaying and analyzing such results.

7. Conclusion

We have presented an interface which allows scientists to use GP collaboratively, and is designed to be accessible (platform-independent, Web distributed) and to work within several resource configurations. Furthermore, we have presented circumstances under which GP applications require or benefit from both collaboration between scientists and other features such as graphical and interactive interfaces.

The interface we have implemented allowed to speed up the process involved in our research significantly, mainly by reducing dead time between different steps of the overall process (collaboration) and by simplifying the complex task of fitness cases selection (graphical interface). Shared data, for instance, eliminated the need for accessing and moving data from different computers, platforms or locations.

We have then addressed issues in security, implementation and performance, and suggested how similar GP collaborative interfaces could benefit other research groups working on different types of GP projects. For further information, please see http://www-personal.engin.umich.edu/~daida/.

Acknowledgments

This research has been funded by a grant from the Office of Naval Research. We appreciate the contributions from the following people: E. Durfee, Q. Stout (U-M), R. Onstott (ERIM). We also thank these other individuals for their assistance: S. Ross, C. Grasso, S. Stanhope, R. Bertram, J. Polito, S. Frazier, D. Batey, and A. & M. Bersano-Begey.

Bibliography

[1] Daida, Jason. M., et al. 1996. Algorithm Discovery Using the Genetic Programming Paradigm: Extracting Low-Contrast Curvilinear Features from SAR Images of Arctic Ice. In , P. Angeline and K. Kinnear, Jr. (ed.). *Advances in Genetic Programming II*. Cambridge, MA: The MIT Press. Pp 417-442.

[2] Koza, John. R. 1992 *Genetic Programming: on the Programming of Computers by Means of Natural Selection*. Cambridge, MA: MIT Press.

[3] Prakash, Atul and H. S. Shim, 1994. DistView: Support for Building Efficient Collaborative Applications using Replicated Objects. In *The 1994 ACM Conference on Computer-Supported Cooperative Work*. The ACM Press, pp. 153-164.

[4] Congalton, R. G. 1991. A Review of Assessing the Accuracy of Classifications of Remotely Sensed Data. In *Remote Sensing of the Environment*, 37:1. pp. 35-36.

[5] Ellis, C.A., S. J. Gibbs, and G.L.Rein 1991. Groupware: Some Issues and Experiences. *Communications of the ACM*, pp. 38-51.

[6] http://isl.cps.msu.edu/GA/software/lil-gp/

[7] http://www.si.umich.edu/~weymouth/Medical-Collab/

[8] ftp://isl.cps.msu.edu/pub/GA/lilgp/viewer.txt

[9] http://www.ifh.ee.ethz.ch/~gerber

[10] Daida, J. M., et al. 1996. Ice Roughness Classification and ERS SAR imagery of Arctic Sea Ice: Evaluation of Feature-Extraction Algorithm by Genetic Programming. In *Proceedings of the 1996 IGARSS*, Washington, IEEE Press, pp. 1520-1522.