AlgoExplorer* : a dynamic data structures viewer for Java

Pier Paolo Ciarravano†
ppciarravano@gmail.com

ABSTRACT

The correct programming of algorithms and data structures is very important for the execution of a program and it allow you to analyze complex algorithms; the visual approach through animated representation simplifies the study, understanding and debugging of source code. In this context the research is called *Data Structure and Algorithm Visualization*. We present an application called *"AlgoExplorer"* which allows the visualization of data structures that play a role in the execution of a program in Java. The application offers an educational approach to understanding the algorithms operations, allowing you to see how objects, handled by the algorithms, interact with each other and how they are related and linked. Furthermore, the application allows an easy visual debugging of algorithms and if the data structures and variables used accomplish the task set by the program.

1. INTRODUCTION

The visual representations are very useful for understanding the mechanisms and functioning of many types of algorithms for quickly understand the behavior of them. It is universally recognized that the graphical display of data structures and algorithms is a powerful alternative to textual description or verbal explanation: the visualization improves understanding and it is able to capture the attention of even better users thus enhancing the educational purposes. The study and testing of the visualization of data structures and algorithms begins as early as 1981 with the movie "Sorting out Sorting" [23] [24] created by Ronald Baeker and thereafter in 1984 was developed BALSA [18], the first system to create animations of algorithms; there are currently many well-known algorithm animation [5] and many systems for displaying runtime, many of which require that programs are developed implementing some specific rules or libraries.

*AlgoExplorer is released under GNU General Public License Version 3(GPLv3)

†Author web site: http://www.larmor.com

In addition to research and development of systems for displaying data structures and algorithms, there was systems also allow the *"Visual Debugging"*, these systems therefore add the ability to inspect the variables at different time of operation and if the values of variables and the behavior of the algorithms coincide with the expected values.

The goal of our application, called *AlgoExplorer* (a name chosen to highlight the ability to "explore" the operation of an algorithm), is to display in graphic and intuitive, and not simply in tabular way as most debugging applications, structures data that come in the execution of a program. More precisely we want to see now graphic instances of the classes used by a given program are related and connected to each other in the execution. In many fundamentals of computing textbooks the instances of classes are often represented as rectangles containing the values that represent the attributes of it, with arrows representing pointers or variables that reference other instances, in this way, for example, a linked list is often represented as a long chain of rectangles linked sequentially together by arrows. AlgoExplorer uses this method of visualization, thus it have a debugging function and also a teaching function because it allows to represent data structures in the way we are accustomed to imagine objects and pointers. Application is developed in Java and it is capable of analyzing programs written in that language.

The methods for the visualization of algorithms and data structures can be divided into two types:

1. methods through the execution of a program properly configured generate a not interactive animation of the algorithm, in which case the animation can also be exported and viewed with a viewer, which is limited exclusively to show the output of the animation, regardless of the system that created it;
2. methods that execute the algorithm and are able to visualize in real time the behavior, allowing end-users to display animation and to interact with different input values in the algorithm.

Both of these methods often require preliminary implementation of the source code of the algorithm that respects some rules of the application that execute the program; AlgoExplorer is an application that belongs to the second type described but the developer of algorithm don't use any particu-

lar rule for the creation of the algorithm or external libraries and utilities in addition to those for the algorithm. AlgoExplorer can be used as an application for debugging Java programs and for educational purposes for the study of algorithms well known in the literature, allowing interaction with several inputs in order to facilitate understanding; it also allows the export of the animation appears in the execution of the algorithm, so that you can display it using a standard browser equipped with Adobe Flash Player plug-in [2].

2. RELATED WORK

AlgoExplorer, of course, is not the only application that deals with the graphical display of data structures manipulated by the algorithms. It is therefore considered appropriate, at this point, analyze and compare with AlgoExplorer three other applications that deal with the similar type of visualization. The applications are:

- The Lightweight Java Visualizer (LJV) [21] [15]
- jGRASP [20] [14]
- JAVAVIS [22]

LJV is a simple tool for displaying data structures in Java. The application uses the Java Reflection for introspection of the data and the Graphviz library [7] for graphical display of the graph representing examined data structures. LJV does not have any graphical user interface and developer of the algorithm integrate the library calls of LJV to allow interaction. The output of LJV is a file text description of the graph, which then will be draw from the Graphviz library.

JGRASP is a lightweight development environment, implemented entirely in Java and designed for the automatic display of data structures for educational purposes. JGRASP allows the development of programs directly from its GUI (quite complex) and the visualization intuitively the traditional data structures such as stacks, queues, linked lists, trees and hashtable. jGRASP use JDI for introspection of data structures.

JAVAVIS is a program useful to the understanding and teaching of object-oriented programming, it allows you to monitor the execution of a program and to see its behavior through two UML diagrams: sequence diagrams and object diagrams. It uses JDI and the source code to allow the display as a normal debugging of an IDE for development.

AlgoExplorer, however, is an application specifically designed for displaying data structures that play a role in program implementation; it uses a custom graphical environment and it does not force the programmer to comply with specific rules or to use special libraries. However jGRASP is also a fairly complex development environment in the first approach and the visualization is restricted to certain types of data structures; instead AlgoExplorer is simple, usable and limited to the visualization of data structures, it is capable of visualize all memory status even if it does not visualize in an intuitive way traditional data structures such as arrays.

In Table 1 we compare the characteristics of the examined programs and AlgoExplorer.

3. KEY FEATURES

The application designed for the Java language, must visualize, in a graphic display, the state of variables and objects managed and manipulated by the algorithms. The following examples show, with some assumptions for the application, what are the views we want to achieve :

- Example 1:

  We have an algorithm that operates an order on a linked list, we want visualize how the algorithm operates on the elements of the list and how the elements of the list are related to each other in sequential steps of the execution.

- Example 2:

  We have an algorithm that handles AVL trees, we want visualize how the algorithm operates on the nodes and how occur any rotational at insertion or deletion of a node.

- Example 3:

  We have a program that implements the Dijkstra's algorithm on a graph; the graph is displayed in graphic and it display at any significant interaction of the algorithm the choices made by the algorithm on the same graph.

The basic requirement is that the algorithms are not written using specifically rules or API, the developer don't implement any particular creation rule or use external libraries as well as instructions and utilities for the algorithm itself. The application must allow the visualization of relations between all objects and variables in the algorithms execution points chosen on the source code, so as to allow examination the values of the variables chosen and the examination of relationships between objects of classes that you want to view. The application must allow the study of algorithms using a GUI (Graphical User Interface) easily usable. The portability of the application is granted by the Java SE 5 (Sun JDK 1.5 or later) [11], without using native libraries, so AlgoExplorer can be run on any operating system which has this version of Java. It would be desirable to be able to export animations generated and then display them in a web based, accompanied by explanatory text that describes the steps.

4. IMPLEMENTATION

AlgoExplorer consists of two distinct phases of user interaction: the first phase include the classes import of the program you want to perform, the choice of the class you want view and the choice of the variables to be inspected; the second phase include the control of the program you want to perform and visualization of data structures that have been chosen for display.

In the first phase, AlgoExplorer allows the import of the binary and optionally source Java classes. The imported

Table 1: Compare of examined programs and AlgoExplorer

	LJV	jGRASP	JAVAVIS	AlgoExplorer
Interactive animation that follows the runtime behavior		X	X	X
Basic instances visualization of data structures	X			
Ability to export the animation				X
Algorithm must follow strict rules and implement specific APIs	X			
Displaying of the animation with a Web browser				X
Needs the source code of the algorithm	X	X	X	

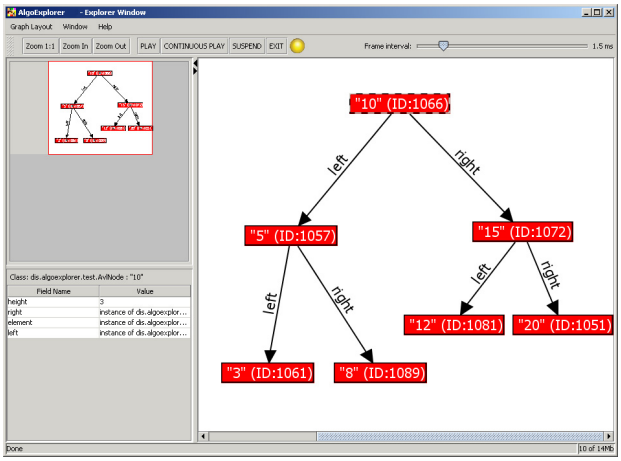

Figure 1: Execution and monitoring GUI

classes are displayed in a tree structure that highlights packages; the classes that contain a *main* method are highlighted in a different way, so they can be chosen for the execution. For each class you can view the source code if imported, set any breakpoints in which you want AlgoExoplorer displays the status of objects and variables, choose whether instances of the class will be shown and which attributes will be the label of the instance and choose the attributes that can be inspected in order to know the value breakpoint identified in different source code. If you have not entered any breakpoints in the source code, the visualization of the data structures will be done only after the *main* method of the executed program. Once you have made your choices you can execute a class containing the *main* method. You can set program parameters to pass to *main* method and any options for the Java Virtual Machine that will run it.

In the second phase (Figure 1), AlgoExplorer allows the visualization of data structures. In a special area the user interface displays the instances of choices classes and you can choose the display scale and automatic layout that will be used to display the data structures (tree or graph). You can also control the program execution: a program is initially suspended and using a special command it can be started; for each breakpoints on the source code, execution stops itself to view the status of instances of the classes selected for display and when you want you can continue to run. When execution is suspended, you can click the visualized instances to inspect the values of the attributes selected for inspection as the first phase of AlgoExplorer.

4.1 Module of workspace configuration

On AlgoExplorer starting you just get a window that shows the configuration interface of the workspace. The *workspace* is a binary set of classes and their source code, so the first required thing is the binary Java class loading and source code of the algorithms (or programs) that want to be analyzed.

The inclusion of the directory containing the binary bytecode file is mandatory, while the inclusion of the file folder containing the source file is optional. AlgoExplorer can work even on programs that haven't their source code, in this case, however, you will not be able to set breakpoints on source code and the visualization of chosen data structures will be only at the end of the program. AlgoExplorer visits recursively the specified directory to search for binary class; this search is not simply limited to identify a class when it is a binary file with *"*.class"* extension, but it is actually verified if the file is compliant with Java bytecode files. AlgoExplorer uses the *Apache Byte Code Engineering Library API* [4] that allow the analysis, creation and manipulation of binary files representing compiled bytecode classes; the bytecode files are analyzed to search for public and private attributes, for the name of the source file references the bytecode file and for the integers represents the numbers of lines of source code actually executive of the source file. We use this procedure rather than the Java Reflection API *(java.lang.reflect)* [10], for many reasons: first because by using Java Reflection you have to read the class that you want to consider using the static *Class.forName* method; in this way, the class is loaded in the running Java Virtual Machine (JVM), which reads the bytecode and makes it available for execution as well as any other class used by the JVM and any external reference or dependence of the class is loaded. Then, using this method, if you look at a class that has a reference to a class not in JVM classpath, you might be incurred in a *ClassNotFoundException* exception type; this side effect is not generated by the Apache Byte Code Engineering Library API, which instead simply read the class bytecode just like any file, without investigating the dependencies. Furthermore the Java Reflection was not used because this technique can't know the private attributes names or private class methods names; the Apache Byte Code Engineering Library API have the ability to know these names and also it is capable of knowing the integers representing the numbers of lines of source code actually executive of the source file and also the name of the source file: this is possible on any class compiled using the option *"-g: none"* in the *javac* command. These abilities are of considerable interest for AlgoExplorer because we are interested also to analyze these references.

AlgoExplorer stores in instances of class *ClassDescriptor* the attributes names and their types and in instances of class

SourceDescriptor the name of the source and the integers representing the numbers of lines in executive source code; AlgoExplorer uses these information to allow you the choice of variables inspect at run time. Afterwards AlgoExplorer visit the files bytecode of the classes, it presents the packages tree, where the main classes are highlighted appropriately.

When you select a class in the packages tree, AlgoExplorer presents the class source code, if it is available, and it also presents the visualization settings of the class instances. You can choose to visualize class instances in different color and you can chose also the class attributes to display; if you don't choose any attribute for the instance label, it will be generated using the *toString* method. You can insert or remove breakpoints in the source code, you can insert breakpoints only on lines of code marked as executable; they are identified by the Apache Byte Code Engineering Library API using *getLineNumberTable()* method in class *org.apache.bcel.classfile.LineNumberTable*. AlgoExplorer show the differences of the state of instances only at breakpoints execution and at the end of main method. The settings you made are saved on instances of *ClassDescriptor* and *SourceDescriptor* classes.

Once you have made the visualization choices you can start a main method using "Run" command in context menu of the packages tree: you can specify program arguments and JVM options, such as memory allocated for the heap or classpath setting of external libraries.

4.2 Programs introspection with the use of Java Debug Interface (JDI)

We chose to use the Java Debug Interface API(JDI) [8] to allow AlgoExplorer to examining instances of classes at execution time. JDI is a Java API (part of Java Platform Debugger Architecture (JPDA) [9]) that allows introspection of the states of the JVM, classes, arrays, interfaces, and instances of primitive types; it also provides control of the JVM execution, the threads control and creation, the brakpoints control, the exceptions notify and the class loader control. All application variables (classes instances, primitive types, arrays) can be inspected through JDI interface, which automatically assigns to each instance a unique ID; furthermore you can also know all references instances of some instance so you can know whether instances are contained in array, Vector, List, Hashtable or an application class. In Figure 2 there is the JPDA diagram.

Looking at this interface we show the follow features:

- Easy to use, although poorly documented by SUN (Oracle) (there is only just Javadoc API and few examples of application code.)
- Portability on any platform (it is present on SUN JRE and each JVM issued by other vendors must implement it).
- It allows the introspection of any variable, even arrays.
- It identify each instance with a unique ID and it returns in a simple way the objects that reference some instance.

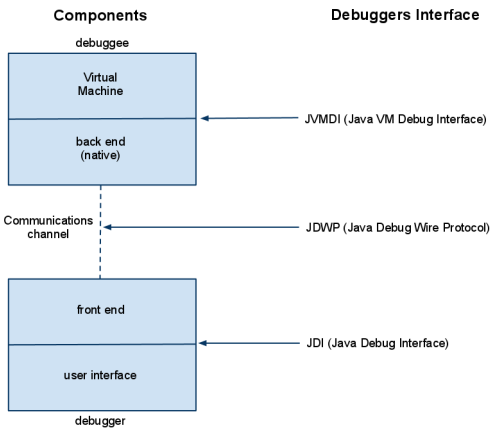

Figure 2: Java Platform Debugger Architecture (JPDA) diagram

4.3 Module of programs introspection

Afterwards the run of a main method, AlgoExplorer presents a new window that represents the visualization environment of execution, monitoring and control of the program that you want analyze. In AlgoExplorer the *Tracer* class and the *InspectThread* class deal with the execution and control of the programs. In Figure 3 we present the AlgoExplorer architecture diagram and in Figure 4 we present the introspection logic class diagram.

The class *Tracer* is responsible to initialize and prepare the JDI interface and it instances the *com.sun.jdi.VirtualMachine* class that will create a new JVM under the JDI control. The *Tracer* class also deals with initialize an instance of the *InspectThread* class, this is the true heart of AlgoExplorer introspection programs logic.

JDI uses event-driven architecture, so the *InspectThread* constructor initializes the events listeners you want to go to observe; first of all it initializes an event listener on *com.sun.jdi.event.MethodExitEvent* and on *com.sun.jdi.event.ClassPrepareEvent*. The *ClassPrepareEvent* event sets a exclusion filter for the event *MethodExitEvent* if the class is not chosen for visualization. The implemented *MethodExitEvent* event catchs all constructors of new instances: in this way AlgoExplorer can trace all initialized instances, it stores this informations in a *InstanceDescriptor* object.

The *InspectThread* class manages the events queue handled by JDI. JDI permits to suspend the execution of analyzed program when event occurs; we has chosen to manage all the events in this suspended mode to prevent the overlap of the events, furthermore, using this mode, is possible manage the play and suspend of the analyzed program as required in the specifications.

The *ClassPrepareEvent* event is executed for all new instances, the action associated with this event sets the listener for breakpoints in source code *(BreakpointEvent)* and

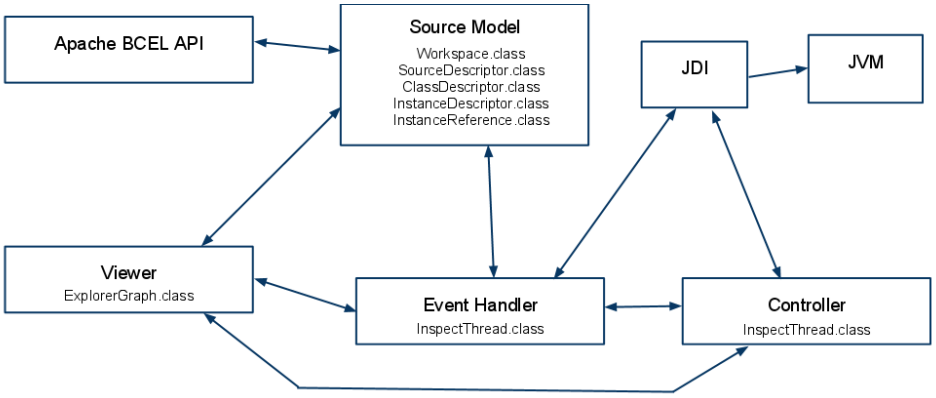

Figure 3: AlgoExplorer architecture diagram

sets the listener for attributes that you want to go to inspect *(ModificationWatchpointEvent)*.

On *BreakpointEvent* event AlgoExplorer executes the *updateInstancesReferencesAndLabel* method, which analyzes all instances of *InstanceDescriptor* type and for each instance, it updates the label to be displayed and also it updates all references of the instances using the recursive *findReferences* method. The method *findReferences* visits recursively, up to a preset and constant number of levels, looking for all objects that reference the given instance using the JDI *referringObjects* method of *ObjectReference* class; each found instance is stored in a new instance of the *InstanceReference* class.

Once the method *updateInstancesReferencesAndLabel* updated all instances, the *renderInstances* method of *ExplorerGraph* class is executed; it deals with the visualization of instances and links between them. We have chosen to use the JGraph [13] API for graphical visualization of object instances. This library is able to handle the visualization of complex graphs using the automatic positioning of nodes with different types of layout (tree or graph). The method *renderInstances* monitors changes made in instances and it makes any updates on the nodes of the graph, furthermore it check, using *isCollected* method, if the instance is under the garbage collection control; an asterisk appears at the end of the instance label if the instance is under the garbage collection control. However, you can disable the garbage collection via the AlgoExplorer settings file.

4.4 Animation export and web viewer

AlgoExplorer allow to generate, through the open source Adobe Flex SDK [1], animation of its graphical output; the exported animation allow to see the algorithms operations for educational purposes even on web pages. AlgoExplorer enables the user to photograph different instants of interest in the visualization and add a text comment to each screen, which can explain the algorithm working. A XML file is generated through the class *XMLExporter*, which describes instances and their relations as shown in the visualization; later this XML file is read by the Flex AlgoExplorer viewer *(AlgoExplorerViewer)* allowing viewing on any browser with Flash Player plug-in [2] (supported by most operating systems: Windows, Mac and Linux). Only AlgoExplorerViewer was built using Adobe Flex 3 language, which implies that in the future could be made a viewer who visualizes the XML file generated by AlgoExplorer using any technology.

4.5 Evaluation and performance

We used the Java profiling tool VisualVM [12] to analyze the performance of AlgoExplorer. We didn't see any performance problems due to design choices made and the use of the resources used by AlgoExplorer is directly proportional to the resources needed by analyzed programs.

The bottleneck could be the listener on *com.sun.jdi.event.MethodExitEvent* event (the applications under the JDI control are slow if there are frequent events run by JDI), but it isn't because it is called only on the methods of the classes you want to visualize, this procedure minimizes the slowdown of the analyzed programs.

5. EXAMPLES

Now we describe some examples of the use of AlgoExplorer for well-known algorithms, it permit to highlight the possible uses for the educational interactive understanding of algorithms.

5.1 Ordered list

We analyzed the *ordered list* algorithm derived from source code in [17] and [16], which operates the insertion of items in an ordered list. The algorithm visit the list until find a item higher than the item you are entering, so that the latter is inserted immediately before the element found. The Figures 5-7 are some screenshots of AlgoExplorer output visualization in order to highlight the steps performed in the insertion algorithm.

5.2 AVL balanced tree

We analyzed the algorithm of nodes insert on a AVL balanced tree from source code in [26] and [25]. The Figure 8 is the AlgoExplorer output visualization for the AVL tree.

5.3 Dijkstra's algorithm

We implemented an algorithm that solves the problem of the *Single Source Shortest Path (SSSP)* on a simple and connected graph and with unitary values of nodes represented

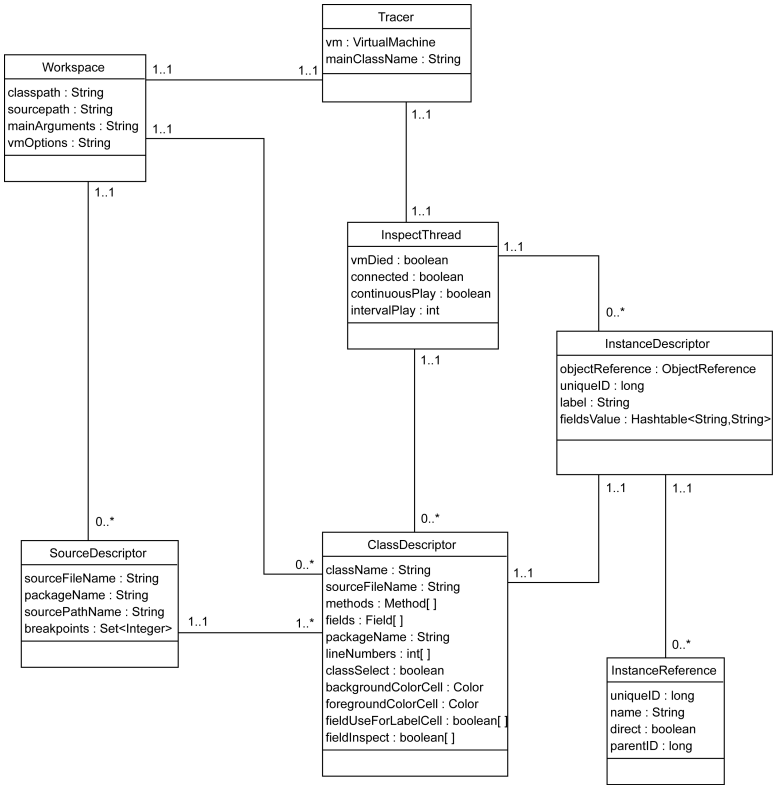

Figure 4: Introspection logic Class diagram

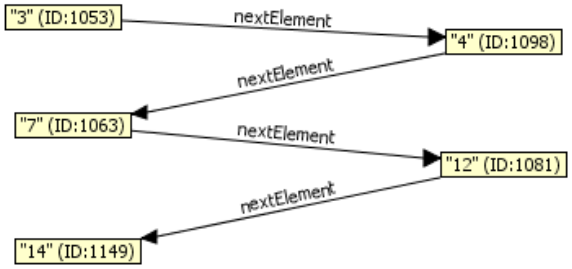

Figure 5: Ordered list with 5 elements

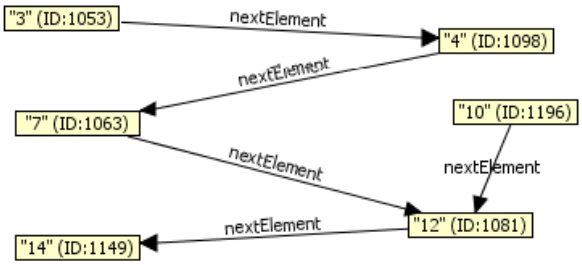

Figure 6: Creation of an element with value set to "10"

by adjacency matrix. We used the classical Dijkstra's algorithm [19]; the algorithm returns the shortest path tree from

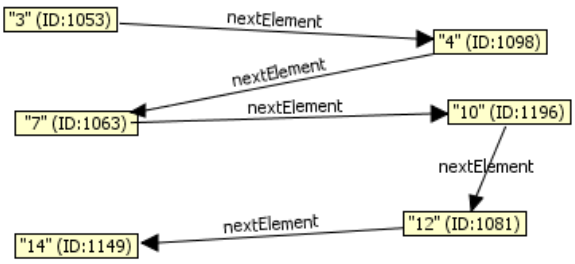

Figure 7: Completing the insert of new element

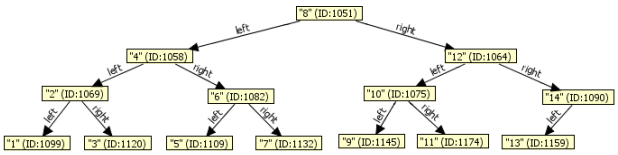

Figure 8: Visualization of a complete AVL tree

a given node, there was also developed a data structure that transforms an adjacency matrix in a graph with objects, it allows intuitive visualization in AlgoExplorer. The Figures 9-12 are some screenshots of AlgoExplorer output visualization using different inputs.

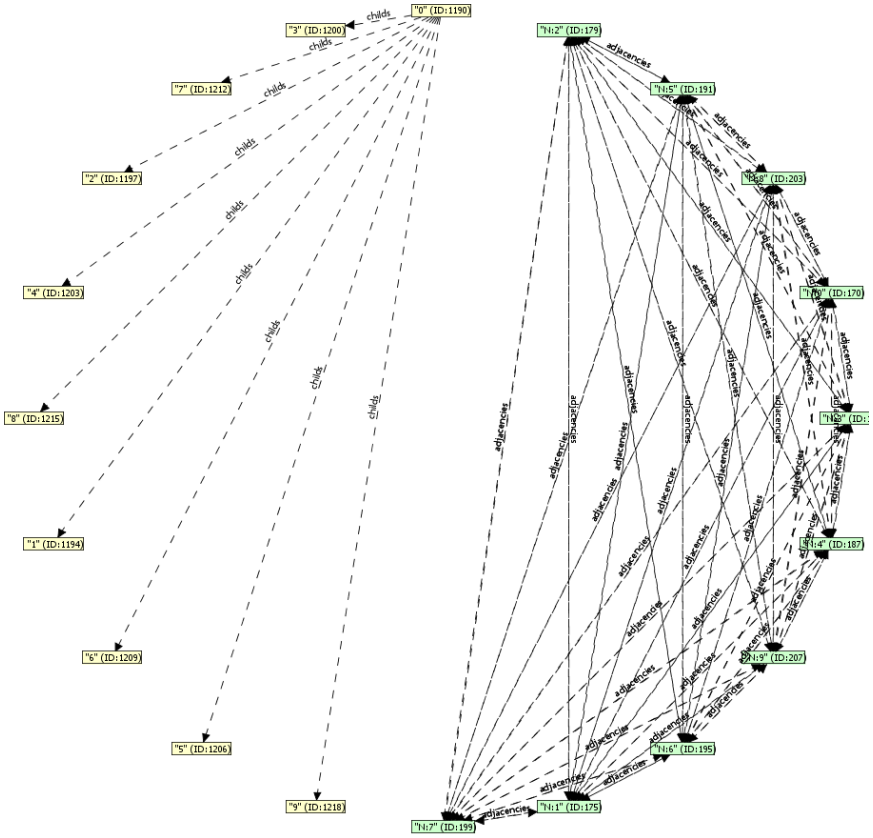

Figure 12: Example of a graph with 10 nodes, fully connected and a shortest path tree extracted from a node

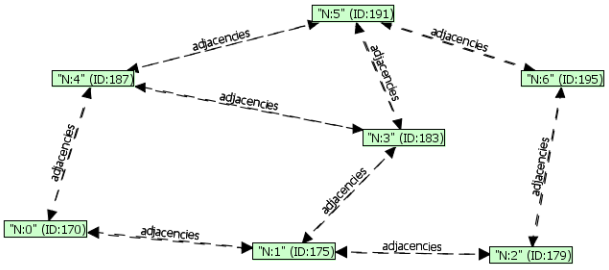

Figure 9: The graph used in the example

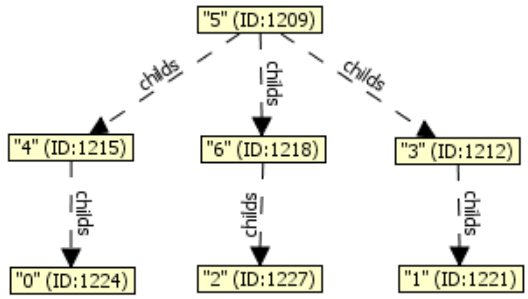

Figure 11: Shortest path tree generated from the node "5"

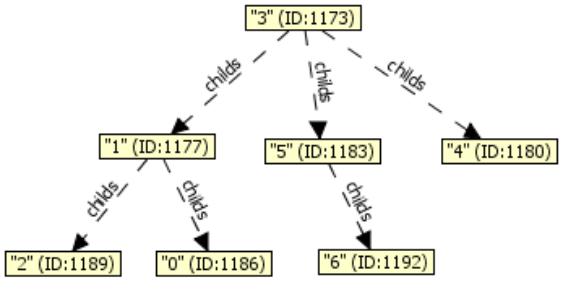

Figure 10: Shortest path tree generated from the node "3"

6. SUMMARY AND FUTURE WORK

AlgoExplorer highlighted the strengths of the approach used: it is very simple, usable and it fully satisfies all key features required and described at the beginning of this document. From the analysis of the examples, we can assign a teaching validity to AlgoExplorer because it allows you to easily study and understand complex algorithms.

The use of JDI has made possible a interference-free of analyzed programs; this procedure may in later versions of AlgoExplorer also be used to visualize in an intuitive and automatic way some data structures that aren't currently

displayed in intuitive graphic (array).

AlgoExplorer could be integrated into an IDE (Integrated Development Environment) such as Eclipse [6], to allow a direct visualization and execution of the algorithms at the same time the developer designing the code.

Finally the web viewer, for visualizations produced by AlgoExplorer, allows a valid educational approach to understanding the algorithms operations, this fully satisfies our goals also in a web context.

7. AVAILABILITY

AlgoExplorer is released under GNU General Public License Version 3(GPLv3). You can get the source code and binaries from the url:

http://todo [3]

8. REFERENCES

[1] World Wide Web electronic publication. http://opensource.adobe.com/wiki/-display/flexsdk/Flex+SDK.

[2] Adobe flash player. World Wide Web electronic publication. http://www.adobe.com/products/flashplayer/.

[3] Algoexplorer. World Wide Web electronic publication. http://todo/.

[4] Apache byte code engineering library. World Wide Web electronic publication.

[5] Data structure and algorithm visualization library for computer science education. World Wide Web electronic publication. http://wiki.algoviz.org.

[6] Eclipse ide. World Wide Web electronic publication. http://www.eclipse.org/.

[7] Graphviz - graph visualization software. World Wide Web electronic publication. http://www.graphviz.org.

[8] Java debug interface (jdi) api. World Wide Web electronic publication. http://download.oracle.com/javase/6/-docs/jdk/api/jpda/jdi/index.html.

[9] Java platform debugger architecture (jpda). World Wide Web electronic publication. http://download.oracle.com/javase/6/-docs/technotes/guides/jpda/index.html.

[10] Java reflection api. World Wide Web electronic publication. http://download.oracle.com/javase/-tutorial/reflect/index.html.

[11] Java se downloads. World Wide Web electronic publication. http://www.oracle.com/technetwork/-java/javase/downloads/index.html.

[12] Java visualvm. World Wide Web electronic publication. https://visualvm.dev.java.net/.

[13] Jgraph. World Wide Web electronic publication. http://www.jgraph.com/jgraph5.html.

[14] jgrasp. World Wide Web electronic publication. http://www.jgrasp.org.

[15] The lightweight java visualizer (ljv). World Wide Web electronic publication. http://www.cs.auckland.ac.nz/ j-hamer/LJV.html.

[16] D. A. Bailey. Java data structures package. World Wide Web electronic publication. http://www.cs.williams.edu/ bailey/-JavaStructures/Software.html.

[17] D. A. Bailey. *Java Structures: Data Structures in Java for the Principled Programmer*. McGraw-Hill, Inc., New York, NY, USA, 2001.

[18] M. H. Brown and R. Sedgewick. A system for algorithm animation. *SIGGRAPH Comput. Graph.*, 18(3):177–186, 1984.

[19] A. Creak. Edsger w. dijkstra. *SIGPLAN Not.*, 37(12):14–16, 2002.

[20] J. H. Cross, II, T. D. Hendrix, D. A. Umphress, L. A. Barowski, J. Jain, and L. N. Montgomery. Robust generation of dynamic data structure visualizations with multiple interaction approaches. *Trans. Comput. Educ.*, 9(2):1–32, 2009.

[21] J. Hamer. Visualising java data structures as graphs. In *ACE '04: Proceedings of the sixth conference on Australasian computing education*, pages 125–129, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[22] R. Oechsle and T. Schmitt. Javavis: Automatic program visualization with object and sequence diagrams using the java debug interface (jdi). In *Revised Lectures on Software Visualization, International Seminar*, pages 176–190, London, UK, 2002. Springer-Verlag.

[23] L. P. Programmers and R. Baecker. Appears in software visualization: Programming as a multimedia experience. mit press, 1998, 369-381. In *Software Visualization: Programming as a Multimedia Experience, chapter 24*, pages 369–381. The MIT Press, 1998.

[24] D. S. RM. Baecker. 30 minute colour sound film, dynamic graphics project. World Wide Web electronic publication, 1981. http://www.kmdi.utoronto.ca/rmb/-video/sos_recap.mov, http://www.kmdi.utoronto.ca/rmb/-video/sos_dotclouds.mov.

[25] M. A. Weiss. Code of data analysis and algorithm analysis in java. World Wide Web electronic publication. http://users.cis.fiu.edu/ weiss/dsaajava2/code/.

[26] M. A. Weiss. *Data Analysis and Algorithm Analysis in Java (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.