

2008

A framework for active learning

Sean Sharma
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_theses

Recommended Citation

Sharma, Sean, "A framework for active learning" (2008). *Master's Theses*. 3529.
DOI: <https://doi.org/10.31979/etd.7ap2-ayxe>
https://scholarworks.sjsu.edu/etd_theses/3529

This Thesis is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Theses by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

A FRAMEWORK FOR ACTIVE LEARNING

A Thesis

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Sean Sharma

May 2008

UMI Number: 1458144

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI[®]

UMI Microform 1458144

Copyright 2008 by ProQuest LLC.

All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

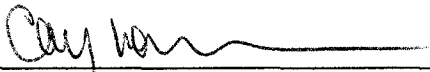
ProQuest LLC
789 E. Eisenhower Parkway
PO Box 1346
Ann Arbor, MI 48106-1346

© 2008

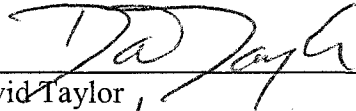
Sean Sharma

ALL RIGHTS RESERVED

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE



Dr. Cay Horstmann



Dr. David Taylor



Dr. Mark Stamp

APPROVED FOR THE UNIVERSITY



ABSTRACT

A FRAMEWORK FOR ACTIVE LEARNING

by Sean Sharma

Understanding of most algorithms in Computer Science is usually aided with iterative, graphical representations. Traditionally, these representations are conveyed to students via a textbook, and only one problem instance is illustrated. Newer methods of learning involve animations of algorithm execution.

We propose a framework that can be used to demonstrate multiple problem types via a combination of animation and student interaction. The framework should support existing algorithm code with minimal modifications.

A prototype of such a framework is developed with an additional construct known as Show Me Mode that enables students to view animations of the execution of an algorithm.

Table of Contents

List of Figures	vii
List of Tables	viii
List of Listings	ix
1 Introduction.....	1
1.1 Overview	1
1.2 The Project	3
1.3 Goals.....	4
1.4 Report Overview	5
2 State of the Art.....	6
2.1 Interactive Learning Environments.....	6
2.2 TRAKLA2.....	7
2.3 Interactive Data Structure Visualizations.....	8
2.4 Additional Art	9
3 Accomplishments.....	10
3.1 The Framework	10
3.1.1 Prim's Algorithm.....	10
3.1.2 Heap Sort	11
3.1.3 Insertion Sort	12
3.2 Show Me Mode	13
3.3 Initial Usage	14
3.4 Contributors.....	15
3.5 Summary	16
4 Framework Overview	18
4.1 Architecture.....	19
4.1.1 Algorithms Package.....	19
4.1.2 Effects Package.....	21
4.1.3 Engine Package.....	23
4.1.4 Exceptions Package	28
4.1.5 Graph Package.....	29

4.1.6 Moodle Package	33
4.1.7 Threading Package	34
4.1.8 Tools Package.....	35
5 Instructor Guide	41
5.1 Adding a Problem.....	41
5.2 Adding a Tool.....	48
5.3 Adding a Layout.....	55
5.4 Adding an Effect	59
6 Conclusion and Future Work.....	61
6.1 Conclusion.....	61
6.2 Future Work	62
Works Cited	64

List of Figures

Figure 1: Prim's algorithm in TRAKLA2	8
Figure 2: Graph representation in Interactive Data Structure Visualizations	9
Figure 3: Prim's algorithm in the framework	11
Figure 4: Heap sort in the framework	12
Figure 5: Insertion sort in the current algorithm	13
Figure 6: The algorithms package.....	19
Figure 7: The effects package	21
Figure 8: The engine package	23
Figure 9: The exceptions package.....	28
Figure 10: The graph package.....	29
Figure 11: The moodle package.....	33
Figure 12: The threading package.....	34
Figure 13: The tools package.....	36

List of Tables

Table 1: Key metric comparison.....	7
-------------------------------------	---

List of Listings

Listing 1: Pseudo random Edge and Edge Weight generation	20
Listing 2: Pulsation effect implementation	22
Listing 3: Glow effect implementation	22
Listing 4: Random array generation	24
Listing 5: Random tree generation.....	25
Listing 6: Random position generation.....	25
Listing 7: Array position generation	26
Listing 8: Tool synchronization.....	27
Listing 9: Vertex pick state management.....	27
Listing 10: Put method on LPPVertex	30
Listing 11: Graph initialization.....	30
Listing 12: Node swapping	31
Listing 13: Adding an edge to a LPPTree.....	32
Listing 14: Set method in LPPList.....	32
Listing 15: Members of LPPTreeVertex.....	33
Listing 16: Execute method in SerialExecutor	35
Listing 17: ExpectSelection method in Tool	37
Listing 18: Condition for Show Next Step Mode	38
Listing 19: Vertex based edge selection	38
Listing 20: Validation of user input	39
Listing 21: Transform tool support	40
Listing 22: A sample run method.....	43
Listing 23: Problem generation method signatures	44
Listing 24: Adding and initializing a tool to a problem.....	45
Listing 25: Our implementation of Prim's algorithm	48
Listing 26: The expectSelection method.....	50
Listing 27: The expectPut method	52
Listing 28: A sample method for a new Tool	53
Listing 29: Show Me Mode Support in the expectSelection method	54

Listing 30: The implementation of PickEdgeTool.....	55
Listing 31: A extension of AbstractLayout.....	56
Listing 32: initializeLayout from LPPGraph	57
Listing 33: The implementation of ArrayLayout.....	59
Listing 34: The implementation of the straight line effect	60
Listing 35: Pulsating an item after selection.....	60

1 Introduction

1.1 Overview

The prevalent model of learning in computer science relies heavily upon the established learning models of mathematics in which interactive, instructor guided, in-class sessions are augmented with independent, student guided sessions. The effectiveness of the in-class sessions is a result of the interactive nature of an instructor presenting material to students. The independent, student guided sessions are not interactive by nature and are often not as effective given that textbook material tends to be dry.

While in-class explanation and demonstration of an algorithm is very helpful for a student, outside of class, a student's recourse is often limited to the class textbook. Although a textbook can provide detailed analysis of an algorithm, illustrations of intermediate steps are often limited to trivial or edge cases. A student wishing to apply an algorithm to a more difficult or involved problem instance has to do so without any mechanism for validation of their thinking. The optimal solution is for students to have access to a tool that provides illustrations of the intermediate stages of execution of an algorithm. A student's learning could be enhanced further if they could choose what they felt was the next step in the execution of the algorithm and their choices were immediately validated. A student's interaction with the tool could result in animation or some other visually interesting behavior. These types of interactions will promote active

learning as the student would be applying the algorithm and working toward a solution instead of being presented with a solution, or at best, the intermediate steps required to reach a solution. Such a mode of learning has shown to be effective for students (Sangwan, Korsh, and LaFollette 272). Creating such a tool is frequently not possible within an instructor's time constraints. Furthermore, tools available today promote passive learning that approximates learning from a textbook. Ideally, instructors could use a framework that deals with the issues of animation and user input and interaction to produce such a tool for their students.

A logical inclination might be to increase the instructor guided, in class sessions and reduce the independent, student guided sessions. This may not be financially feasible for the instructional facility or the student. Furthermore, some students are successful in the current model. For other students, a shortcoming of the current model is the ineffectiveness of the independent, student guided session when compared to the in-class, instructor guided session.

A potential enhancement to the current model can be achieved by altering the independent, student guided sessions. If these sessions were able to more closely approximate the interactivity of the instructor guided, in-class sessions, their effectiveness would be improved. The means of increasing interactivity are illustrative and animated examples and immediate feedback. Illustrative and animated examples would more clearly demonstrate the progression through a problem than static images in

a textbook. Immediate feedback will allow students to validate their thinking against every step in the progression of a problem.

The key to realizing this enhancement is to provide a mechanism for instructors to deliver illustrated and animated problems with immediate feedback to their students. The optimal vehicle is a highly extensible framework. Instructors must find the framework to be easy to extend as instructor adoption is correlated to ease of use (Lahtinen, Javinen, and Melakoski-Vistbacka 259; Naps et al. 126). The framework would provide the expected functionality of problem and solution creation, and student tracking. Its potential would lie in its extensibility as new problems and problem types could be added by instructors. The new problems and problem types could utilize any of the expected functionality of the framework as well as any of its graphical capabilities.

Building a framework which promotes illustrated problems and immediate feedback while allowing instructors to easily augment the problem space will replace the independent sessions that prove to be ineffective for some students with a more effective active learning environment. A similar effort at Ithaca College in New York that evolved from slide based representations to in-class visualizations led the inventors to a similar conclusion (Erkan, Scaffidi, and VanSlyke 305).

1.2 The Project

The project entails creating an extensible framework that promotes an interactive learning experience. The learning experience is further enhanced by exposure to multiple

problem instances. User interaction and feedback are animated to provide emphasis and encourage adoption by students. Although instructor adoption is not always based on the quality of the tool, we attempt to promote instructor adoption by ensuring that existing algorithm code can be easily combined with the framework to create problem instances (Ben-Ari and Levy 247). Additionally, instructors may choose to adopt the framework since it will track user action and report information that can assist grading (Helmick, Integrated 148; Helmick, Interface-based 66).

A bevy of applications provide an interactive student experience, an animated demonstration, or random problem generation. The uniqueness of our framework is that it combines the aforementioned features and incorporates a high degree of extensibility. Moreover, our framework includes the construct of Show Me Mode which allows for an entire problem or a single step to be animated without any user action. See section 2.3 for an overview of a similar implementation in the Interactive Data Structure Visualizations system.

The result is a reusable framework for which instructor adoption would be fueled by the ease of problem type integration, and student adoption would be fueled by interactive, more enjoyable nature of problem solving.

1.3 Goals

The goals of this project are to create a framework that enables active learning by: allowing for instructors to enable instances of a problem type by making a minimal

amount of modifications to existing algorithm code, allowing students to iterate through visual representations of the intermediate steps of algorithm executions, and allowing random generation of problem instances.

In the course of addressing these goals, an additional goal was added. The framework should support modes in which the entire execution of an algorithm or the execution of a step is shown without user interaction

1.4 Report Overview

The state of our art is presented in Chapter 2, and it includes an overview of two applications which accomplish goals similar to our own. Chapter 3 provides information regarding our accomplishments. The framework and initial use feedback are reviewed. An overview of the framework is discussed in Chapter 4. The overview focuses on the architecture and is intended to aid anyone considering future work on the framework. An instructor guide is presented in Chapter 5, and it depicts mechanisms for extending the framework by adding problems, tools, layouts or effects. Chapter 6 outlines our conclusions and some suggestions for future work on the framework.

2 State of the Art

2.1 Interactive Learning Environments

The proliferation of the Internet in the last decade has resulted in the availability of information regarding virtually any topic. The format of information has progressed from simple text to text with illustrations to text with animations. As pertaining to our domain, a large variety of tools are available today.

Erroneously, many of these tools are labeled interactive even though they provide no mechanism for user interaction. User interactivity is the key component of an active learning environment (Carlson et al. 292). Furthermore, many of these tools are only capable of showing a specific problem type and often, can only show the same instance.

As relevant to our goals, the state of the art can be examined from the following perspectives: extensibility and interactivity. The extensibility of a framework pertains to the relative ease of adding support for problem types and new types of user action. The interactivity of a framework pertains to its level of animation and user interaction.

We reviewed many systems during the initial investigation period. Among the systems, were TRAKLA2 and Interactive Data Structure Visualizations. An analysis of the two systems follows.

The comparison in Table 1 reveals that TRAKLA2 is the superior software when evaluated against our criteria. Figures 1 and 2 illustrate both user interfaces.

Table 1: Key metric comparison

	TRAKLA2	Interactive Data Structure Visualizations
Animated/Interactivity	Yes	No
Extensible	No	No
Graphical Representation	Good	Poor

2.2 TRAKLA2

The TRAKLA2 project is an on-going effort at the Helsinki University of Technology. Many of the project's goals align with our goals. The key divergence occurs with the notion of an exercise. In our notion, a user is provided with a visualization of a problem and the tools to apply an algorithm to the visualization. In TRAKLA2's notion of an exercise, a user is provided with a visualization, some ancillary information and the algorithm itself. Our framework assumes previous exposure to the algorithm, while TRAKLA2 does not.

Measured against our definition of interactivity, TRAKLA2 is exceptional. As shown in Figure 1, it provides effective visualizations and animations and it can even support simultaneous display of a static capture of the progression of an algorithm and an animation of the execution of the algorithm. Additionally, TRAKLA2 provides feedback regarding user action (Korhonen, Laakso, and Myller 252).

Although TRAKLA2 satisfies one of our design goals, it falls short of our goal of extensibility as it appears that any enhancements or modifications to the system are strictly the domain of the effort at Helsinki University of Technology. Our framework bests TRAKLA2 with respect to our definition of extensibility.

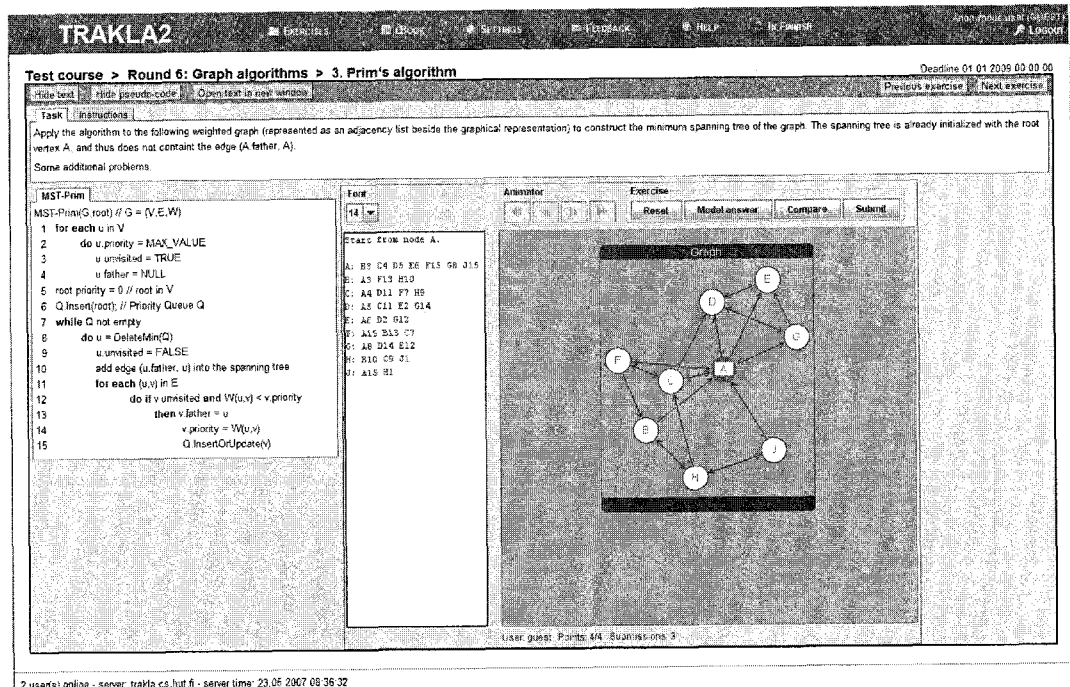


Figure 1: Prim's algorithm in TRAKLA2

2.3 Interactive Data Structure Visualizations

Interactive Data Structure Visualizations originated as a graduate project at The George Washington University. Its primary goal was to be a vehicle for measuring student performance with and without the use of a tool. It includes a novel Show Me capability that displays the final result of each step in an algorithm's execution.

Gauged against our definition of interactivity, Interactive Data Structure Visualizations measures poorly. While the interface may more closely approximate the internal representation of data, it may not be intuitive to some students. In fact, contrary to many other studies about other tools, students who used this tool performed worse than

students who did not use the tool (Jarc, Feldman, and Heller 380). Figure 2 contains the representation of a graph.

Additionally, Interactive Data Structure Visualizations does not provide any means of making modifications or enhancements. Extensibility was not a design goal. Our framework bests Interactive Data Structure Visualizations with respect to our definitions of interactivity and extensibility.

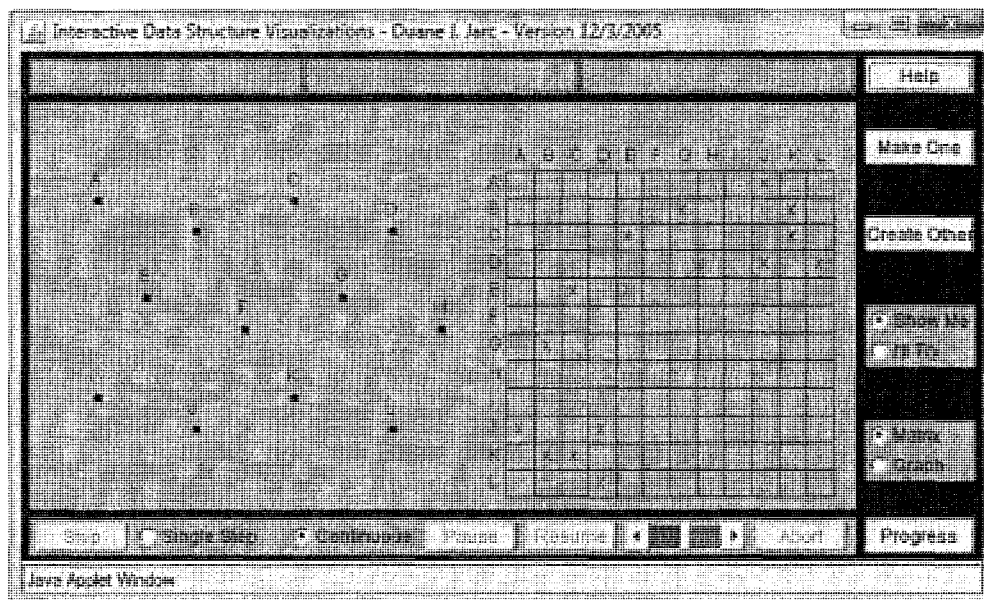


Figure 2: Graph representation in Interactive Data Structure Visualizations

2.4 Additional Art

An extensive overview of additional art has been assembled by members of the Department of Computer Science at Virginia Tech, and is provided at the following website: <http://web-cat.cs.vt.edu/AlgovizWiki> (Shaffer, Cooper, and Edwards 151).

3 Accomplishments

3.1 The Framework

To address our goals of extensibility and interactivity, we determined that the framework should support multiple problem types, multiple problem instances and be easily extended. Over two semesters, we developed a framework that supports multiple problem types, multiple problem instances, and has an extensible tool architecture. The resulting framework provides an interactive experience to users and is easily extended by instructors.

Much of the visualization in the framework is powered by the JUNG framework (“Overview”) and effects are enabled via the Timing framework (Hasse). JUNG provides layout management of items in the visualization, and the Timing framework manages threading and timing issues as they pertain to animation effects. An in-depth overview of the architecture of the framework is given in Chapter 4, and an instructor guide targeted at users who want to extend the framework is given in Chapter 5. The framework supports the following problem types: Prim’s algorithm, Heap sort and Insertion sort. An overview of each problem type is provided.

3.1.1 Prim’s Algorithm

The framework supports an implementation of Prim’s algorithm. A randomly generated instance is pictured in Figure 3. The Animator displays problem and tool

sensitive directions, and each user action is validated. At each Set π value step, after the user selects a vertex, a dialog box appears and allows for the entering of a π value. The value is validated against the expected the π value and the algorithm does not progress until the correct value is entered. This algorithm features an effect that animates a vertex selection by pulsating its border.

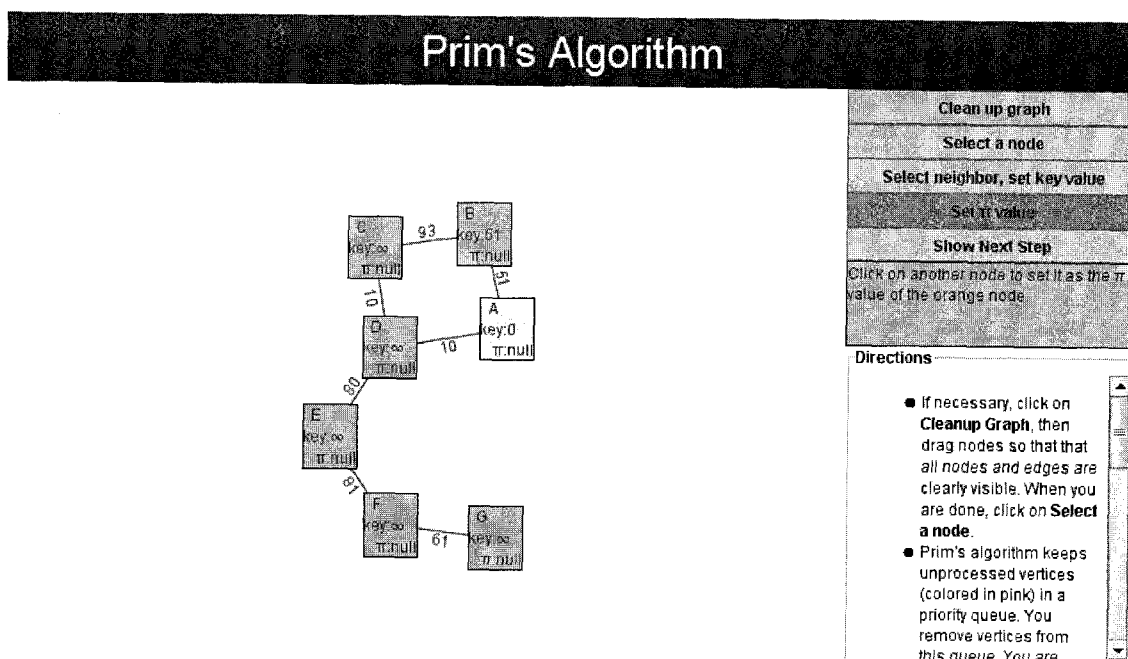


Figure 3: Prim's algorithm in the framework

3.1.2 Heap Sort

The framework supports an implementation of the Heap sort algorithm. As shown in Figure 4, a user is presented with a randomly generated instance of a heap and must perform swap and percolation operations until the resulting heap is a min heap.

Each user action is validated against the action expected by the algorithm and execution does not progress until the user has taken the correct action.

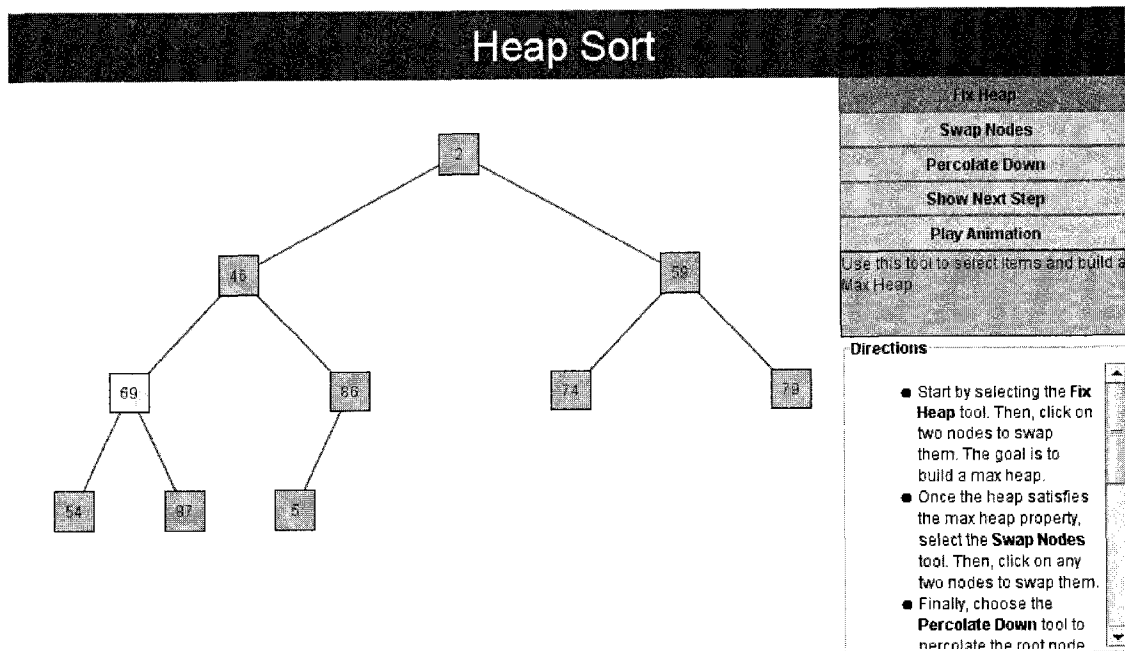


Figure 4: Heap sort in the framework

3.1.3 Insertion Sort

The framework supports an implementation of the Insertion sort algorithm. As shown in Figure 5, a user is presented with an array of randomly generated values. At each iteration, the user must select the value that the algorithm needs to be inserted, the item's location as it moves through the array, and the item's final insertion location.

Each user action is validated, and the algorithm does not progress until the user takes the expected action. This algorithm features an effect that swaps the values of elements after an insertion.

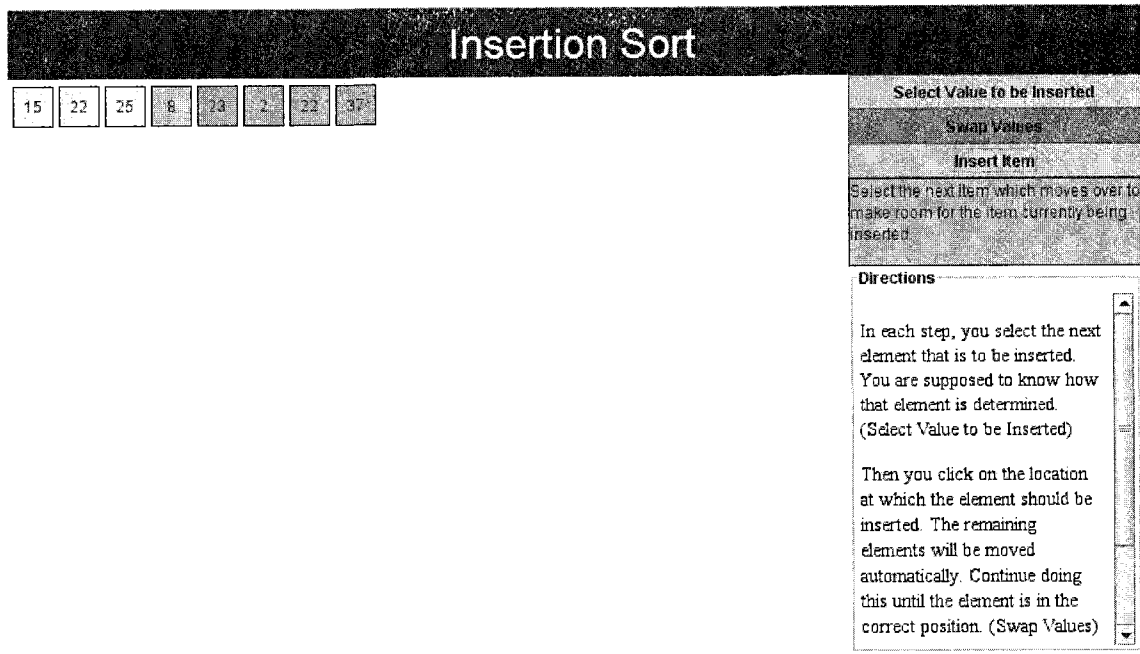


Figure 5: Insertion sort in the current algorithm

3.2 Show Me Mode

The framework supports Show Me Mode and Show Next Step Mode. In Show Me Mode, the Animator progresses through the entire algorithm without any user action. Each step is illustrated with an image of a cursor icon. Show Next Step Mode is initiated by the user and when invoked, only illustrates the next step in the progression of the algorithm. Similar to Show Me Mode, the image of a cursor icon is used to illustrate the step.

Show Me Mode is achieved by adding a guard clause to each user actionable method in a tool. Specifically, if algorithm code calls a specific method in a tool, that method must have a branch of execution that is taken when the system is in Show Me

Mode. This branch often includes calls to two methods in the Animator. The first moves the cursor icon image to a specific location and the second causes the cursor icon image to fire a click event at its current location.

The mechanism for Show Next Step Mode heavily utilizes the implementation of Show Me Mode. When a user invokes Show Next Step Mode, a flag in the Animator is set to Show Next Step Mode. Once the user actionable method in the tool is called, execution branches to the Show Me Mode logic. At the end of execution, the Show Next Step Mode flag is toggled. The effect is that the Animator is in Show Me Mode for only one step.

3.3 Initial Usage

The program was used in November of 2007, by Dr. David Taylor of San Jose State University and the students his CS 46B class. The initial usage consisted of students completing an instance of Insertion sort and then providing feedback regarding the tool. The deployment mechanism for the initial usage was the moodle system.

In the course of preparing the problems and tools for the initial usage, our efforts to meet our design goals of interactivity and extensibility were validated by Dr. Taylor. Dr. Taylor was able to extend the framework and implement Bubble sort. Additionally, Dr. Taylor was able to augment the interactivity of the framework by adding some color transitions to his implementations of Bubble sort.

Immediately after using the framework, participants were asked for feedback. The feedback indicated that overall experience was positive. Although most of respondents had not used a tool similar to our framework, many found our framework to be easy or very easy to use. Additionally, all respondents felt that the framework improved their understanding of the algorithm.

We feel that it is a significant accomplishment to have had participants recognize the benefit of our tool even though most had never used something similar. This claim is justified by the response of the respondents, most of whom would use our framework to augment their studying for a class. The initial usage validated our efforts to achieve our design goals. Dr. Taylor proved the extensibility of the framework, and the respondents proved its interactivity.

3.4 Contributors

The design and development of the framework involved students working under the guidance of Dr. Cay Horstmann and Dr. David Taylor of San Jose State University.

Working under Dr. Horstmann's supervision were Sean Sharma, Kristen Mori, Shiro Sakurai, Alexander Ljungberg. Dr. Taylor supervised Edward Yin and Andrei Lurie.

Dr. Horstmann was responsible for the problem model in the framework, a refactoring that included a new tool model, and integration with the timing framework

that is used to power effects in the framework. He also reviewed the design of the framework through its several iterations. Dr. Taylor created the problems for the initial usage and reviewed the interaction model of the framework throughout the project.

Sean developed the initial framework, and was responsible for a refactoring that included new threading and problem models which allowed multiple problem instances to be run in a single run of the Animator. Kristen developed an implementation of Heap sort and the new tools required. Kristen was instrumental in identifying several shortcomings in our initial design. Shiro added several animation effects to the framework and made improvements to the APIs used by the Animator to invoke animations and effects. Alexander facilitated the initial setup of the source repository and wiki, aided in the initial design of the framework and provided support for the wiki and source repository throughout the project. Edward aided in the design of the framework and implemented several versions of Prim's algorithm with different interaction models. Ed aided in the initial investigation of the JUNG framework and the examination of existing art. Andrei configured moodle to utilize the framework and added APIs that allow external systems to gather problem information. Additionally, Andrei performed much of the setup work for the initial usage.

3.5 Summary

Our efforts resulted in a framework that supports multiple problem types, multiple problem instances, and includes a tool architecture that enables tools to perform action

validation independent of the framework. Additionally, we developed the construct of Show Me Mode and Show Next Step Mode which allow a user to view an animation of an algorithm's execution.

We were able to validate our efforts against our goals via the initial usage. The development of the problems and tools for the initial usage and subsequent feedback indicated that our framework was both extensible and interactive.

Assessed against our goal of extensibility, the framework performs satisfactorily. New problems can be added without modification to the framework, and can leverage existing tools. If new tools are needed, they can be added without modification to the framework.

The framework also performs satisfactorily when measured against our goal of interactivity. The separation of the visualization element and the animator resulted in a more robust effects model that allows for the addition of new animation effects. Each user action is still validated against the algorithm, and the algorithm still does not progress unless the correct action is taken.

4 Framework Overview

The architecture of the framework is best reviewed by examining its major components. The major components are the problem model, the tool model, the threading model and the effects model.

All problems are derived from the Problem base class. The Problem class provides methods that can manipulate the animator and reduce the redundant code in different problem implementations.

The tool model moves the processing of user actions to the tool classes. The framework allows tools to define their own methods for processing and user actions are dispatched to tools for processing.

The threading model provides support for the tool model, and the ability to run more than one problem in an instance of the Animator. The algorithm and user action events occur on separate threads and tools serve as the mechanism for processing user actions and advancing the algorithm thread.

The effects model enables problems and tools to trigger effects and animations on items in the visualization. Several effects are packaged with the framework and are meant to be templates for any new effects. The effects in the framework are powered by the Timing framework.

These components were designed so instructors could easily add problems and tools and students would find the resulting experience to be interactive and visually interesting.

4.1 Architecture

4.1.1 Algorithms Package

The algorithms package houses definitions of problems and their accompanying drivers. It contains the following classes: Heapsort3, InsertionSorter, Prim4, AnimatorApplet, InsertionSorterAnimator and Prim4Harness.

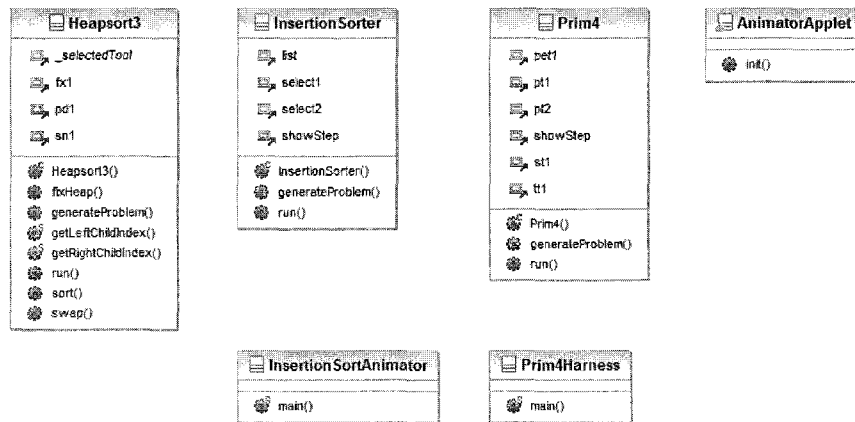


Figure 6: The algorithms package

The class Heapsort3 provides an implementation of a Heap sort algorithm. It is derived from the Problem class. This problem utilizes the FixHeap, SwapNodes and PercolateDown tools

The class InsertionSorter provides an implementation of the Insertion sort algorithm. It is a subclass of Problem. Generation of the item values is random. This problem utilizes the Select and ShowNextStep tools.

The class Prim4 provides an implementation of Prim's algorithm. It is derived from the Problem class. Generation of the vertices, edges and edge weights is pseudo random, occurs in the Problem class and a portion of the algorithm is outlined in Listing 1. This problem utilizes the TransformTool, SelectTool, PutTool and ShowNextStep tools.

These tools are reviewed in section 4.1.8.

```
int rand = (int) (Math.random() * g.numVertices());
LPPVertex v_rand = (LPPVertex) i.getVertex(rand);
LPPEdge dse = new LPPEdge(v1, v_rand);
if (v1 != v_rand)
{
    if (!v1.isPredecessorOf(v_rand)
        && !v1.isSuccessorOf(v_rand))
    {
        g.addEdge(dse);
        v1.addIncidentEdge(dse);
        v_rand.addIncidentEdge(dse);
        v1.addAdjacentNode(v_rand);
        v_rand.addAdjacentNode(v1);
        ewl.setWeight(dse, labelNumber);
        dse.put("w", labelNumber);
    }
}
```

Listing 1: Pseudo random Edge and Edge Weight generation

The class AnimatorApplet enables an instance of Prim4 to be run as an Applet. The class InsertionSorterAnimator enables an instance of InsertionSorter to be run as an Applet. The class Prim4Harness enables an instance of Prim4 to be run as an Application.

4.1.2 Effects Package

The effects package houses resources that power some of the visual effects in the framework. It contains the following classes: VertexDecorator, TextDecorator and Effects. It also contains the following interfaces: Drawable and Locatable.

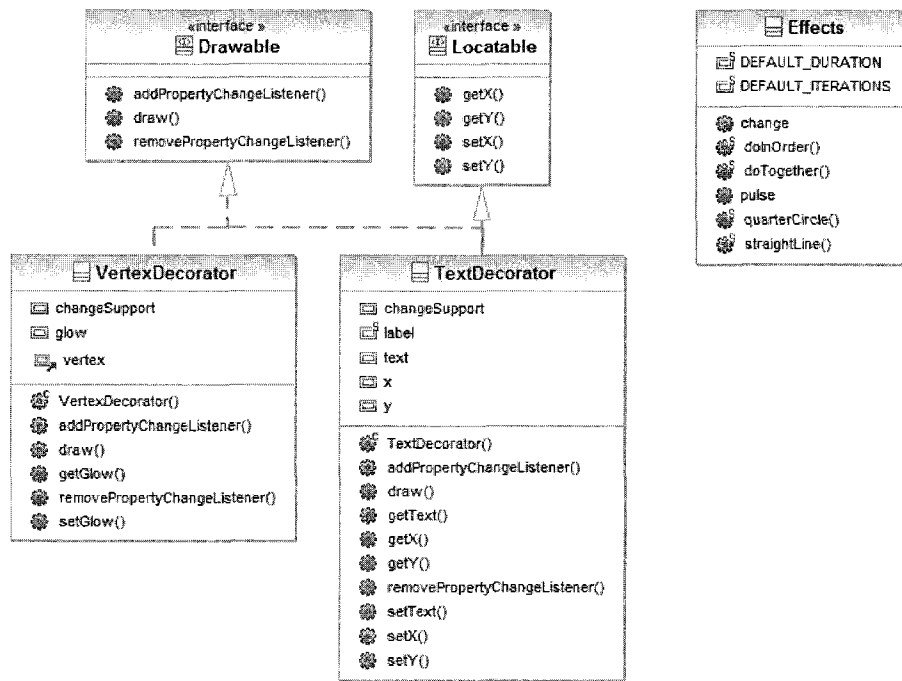


Figure 7: The effects package

The interface **Drawable** is implemented by the **TextDecorator** and **VertexDecorator** classes. It outlines the `draw` method and mechanisms for attaching and detaching a property change listener.

The interface **Locatable** is implemented by the **TextDecorator** class. It outlines getter and setter methods for x and y coordinates.

The class `Effects` defines several animations that can be invoked on items in the current visualization. It leverages the timing framework to perform positional change or other animations. The method for item pulsation is outlined in Listing 2.

```
public static <T> Animator pulse(Object object, String
    propertyName, T... values)
{
    return new Animator(DEFAULT_DURATION /
        (DEFAULT_ITERATIONS / 2),
        DEFAULT_ITERATIONS,
        RepeatBehavior.REVERSE,
        new PropertySetter(object,
            propertyName, values));
}
```

Listing 2: Pulsation effect implementation

The class `VertexDecorator` facilitates animation effects on vertices in the visualization. As demonstrated in Listing 3, `VertexDecorator` implements the `Drawable` interface to provide a glow effect to vertices.

```
g2.setPaint(Color.RED);
final double MAXGLOW = 5;
final double thickness = glow * MAXGLOW;
g2.setStroke(new BasicStroke((float) thickness));
Dimension size = vertex.getPreferredSize();
if (glow > 0)
    g2.draw(new Rectangle2D.Double(vertex.getX() - size.width / 2
        - thickness / 2, vertex.getY() - size.height / 2 -
        thickness / 2, size.width + thickness,
        size.height + thickness));
```

Listing 3: Glow effect implementation

The class TextDecorator provides a means for text based effects in the visualization. TextDecorator implements the Drawable and Locatable interfaces. It is primarily used in the Effects class for positional change animation of text.

4.1.3 Engine Package

The engine package provides the primary interface between the Animator and JUNG. It contains the following classes: Problem, LPPArrayToTreeLayout, LPPGraphLayout, ArrayLayout, VisualizationPanel and AnimatorPickedState. It also contains the ProblemType and AnimatorMode enumerations.

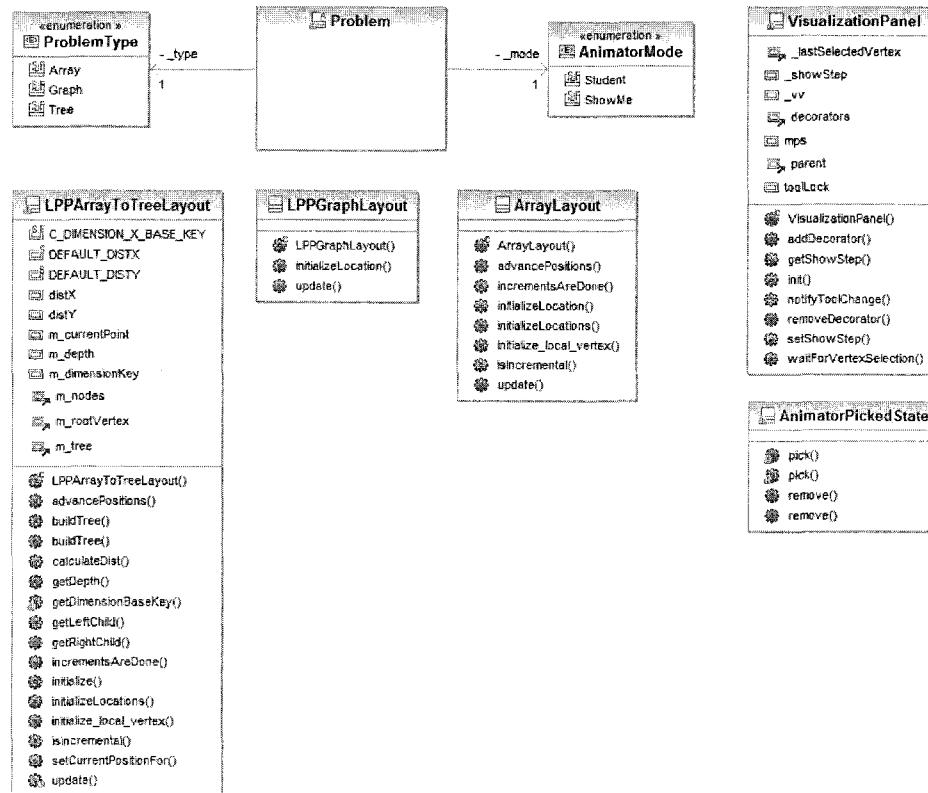


Figure 8: The engine package

The class `Problem` serves as the base class for all problems in the framework. In addition to tracking all user actions, it provides several methods to its subclasses that abstract interaction with the `Animator`. The goal was to eliminate the need for subclasses to directly interact with the framework. Additionally, the `Problem` class can generate random instances of visualizations for each of the enumerations in `ProblemType`. An excerpt of the `generateRandomArray` method is shown in Listing 4. This method is not guaranteed to generate a connected graph.

```
LPPGraph g = new LPPNetwork();
LayoutMutable layout = new ArrayLayout(g);
StringLabeller _labler = StringLabeller.getLabeller(g);
LPPVertex[] vertices = new LPPVertex[numItems];
ArrayList<Integer> values = new ArrayList<Integer>(numItems);
Random generator = new Random();
// get an array of unique random integers
for (int i = 0; i < numItems; i++)
{
    int randomNum = generator.nextInt(100);
    while (values.contains(randomNum))
        randomNum = generator.nextInt(100);
    values.add(Integer.valueOf(randomNum));
}
```

Listing 4: Random array generation

The class `LPPArrayToTreeLayout` provides a mechanism for converting an array of elements to a tree representation. The representation is encapsulated in JUNG graph objects so the layout can be applied directly to a JUNG graph. This layout is applicable for representations of heaps, binary trees, etc. The means of tree generation are shown in Listing 5.

```
private void buildTree()
```

```

{
    distX = DEFAULT_DISTX;
    distY = DEFAULT_DISTY;
    m_rootVertex = m_tree.getRoot();
    Point temp = new Point(this.getCurrentSize().width / 2, 50);
    m_currentPoint = temp;
    initializeLocations();

    m_nodes = (ArrayList<LPPVertex>) m_tree.nodes();

    if (m_rootVertex != null && getGraph() != null)
    {
        m_depth = getDepth();
        calculateDist();
        buildTree(m_rootVertex, 0, m_currentPoint.x,
                 m_currentPoint.y, m_depth);
    }
}

```

Listing 5: Random tree generation

The class LPPGraphLayout supplies a method for creating a graph whose nodes are positioned randomly. The representation is encapsulated in JUNG graph objects so the layout can be applied directly to a JUNG graph. This layout is applicable graph representations and is used in Prim's algorithm. Random position generation is outlined in Listing 6.

```

protected void initializeLocation(Vertex v, Coordinates coord,
    Dimension d)
{
    double x = 20 + Math.random() * (d.getWidth() - 40);
    double y = 20 + Math.random() * (d.getHeight() - 40);

    coord.setX(x);
    coord.setY(y);
    ((LPPVertex) v).setLayout(this);
}

```

Listing 6: Random position generation

The class `ArrayLayout` enables a layout suitable for array-like representations of data. As shown in Listing 7, as requests are made to add data, the position of each element is calculated.

```
for (LPPVertex node : ((LPPGraph) getGraph()).nodes())
{
    int d1 = maxx / 2;
    x += d1;
    getCoordinates(node).setLocation(x, maxy / 2);
    x += maxx - maxx / 2 + DISTANCE;
    node.setLayout(this);
    node.setMinimumSize(new Dimension(maxx, maxy));
}
```

Listing 7: Array position generation

The class `VisualizationPanel` encapsulates the underlying visualization components of JUNG and exposes them as a `JPanel` to the framework. Additionally, `VisualizationPanel` contains a key component for the framework and tool interface strategy. The method `waitForVertexSelection` is invoked from virtually every tool. It enforces selection of the correct tool and it returns the selection made by the user. Synchronization with tools is achieved via the `toolLock` object. A portion of the implementation is shown in Listing 8.

```
synchronized (toolLock)
{
    while (true)
    {
        toolLock.wait();

        // "Show Next Step" was clicked
        if (_showStep)
        {
            return null;
        }

        if (parent.getSelectedTool() == tool)
```

```

    {
        return _lastSelectedVertex;
    }
    else
    {
        javax.swing.JOptionPane.showMessageDialog(null,
            "Incorrect Tool Selected, expected: " +
            tool.getName());
        tool.addWrongMove();
    }
}
}

```

Listing 8: Tool synchronization

The class `AnimatorPickedState` extends the standard pick state mechanism provided by JUNG to allow for multiple items to be in a picked state. By default, the mechanism provided by JUNG removes the currently picked item when another item is selected. The extended implementation of the pick method is featured in Listing 9.

```

public boolean pick(ArchetypeVertex v, boolean picked)
{
    Set<Vertex> sv = getPickedVertices();
    Object[] o = sv.toArray();

    boolean result = super.pick(v, picked);

    for (int i = 0; i < o.length; i++)
    {
        super.pickedVertices.add((ArchetypeVertex)o[i]);
    }

    return result;
}

```

Listing 9: Vertex pick state management

The enumeration `ProblemType` contains the three types of layouts supported by the framework. Each `Problem` has a `ProblemType`, and this at problem generation time, the problem type can be used to determine which type of layout to use for random instance generation.

The enumeration AnimatorMode includes the two types of modes supported in the framework. The mode Student progresses through a Problem only when the user has taken an action. In Student mode, each action is validated. In ShowMe mode, the algorithm progresses autonomously without and user action. The correct action is always taken.

4.1.4 Exceptions Package

The exceptions package maintains definitions of the custom exceptions thrown by the framework. It contains the following classes: InvalidProblemClassException, ProblemCreationException and InvalidOperationException.



Figure 9: The exceptions package

The class InvalidProblemClassException is used to convey a condition in which the framework failed to find the problem class passed to it. The class ProblemCreationException relays information when the framework has attempted to create a problem class, and the instantiation has failed. The class InvalidOperationException indicates that an attempted operation is invalid. Most likely occurrences pertain to mutations of the visualization.

4.1.5 Graph Package

The graph package maintains definitions of the visualizations represented by the framework. It contains the following classes: LPPVertex, LPPEdge, LPPGraph, LPPArray, LPPNetwork, LPPTree, LPPList, LPPTreeVertex. It also contains the LPPGraphElement interface.

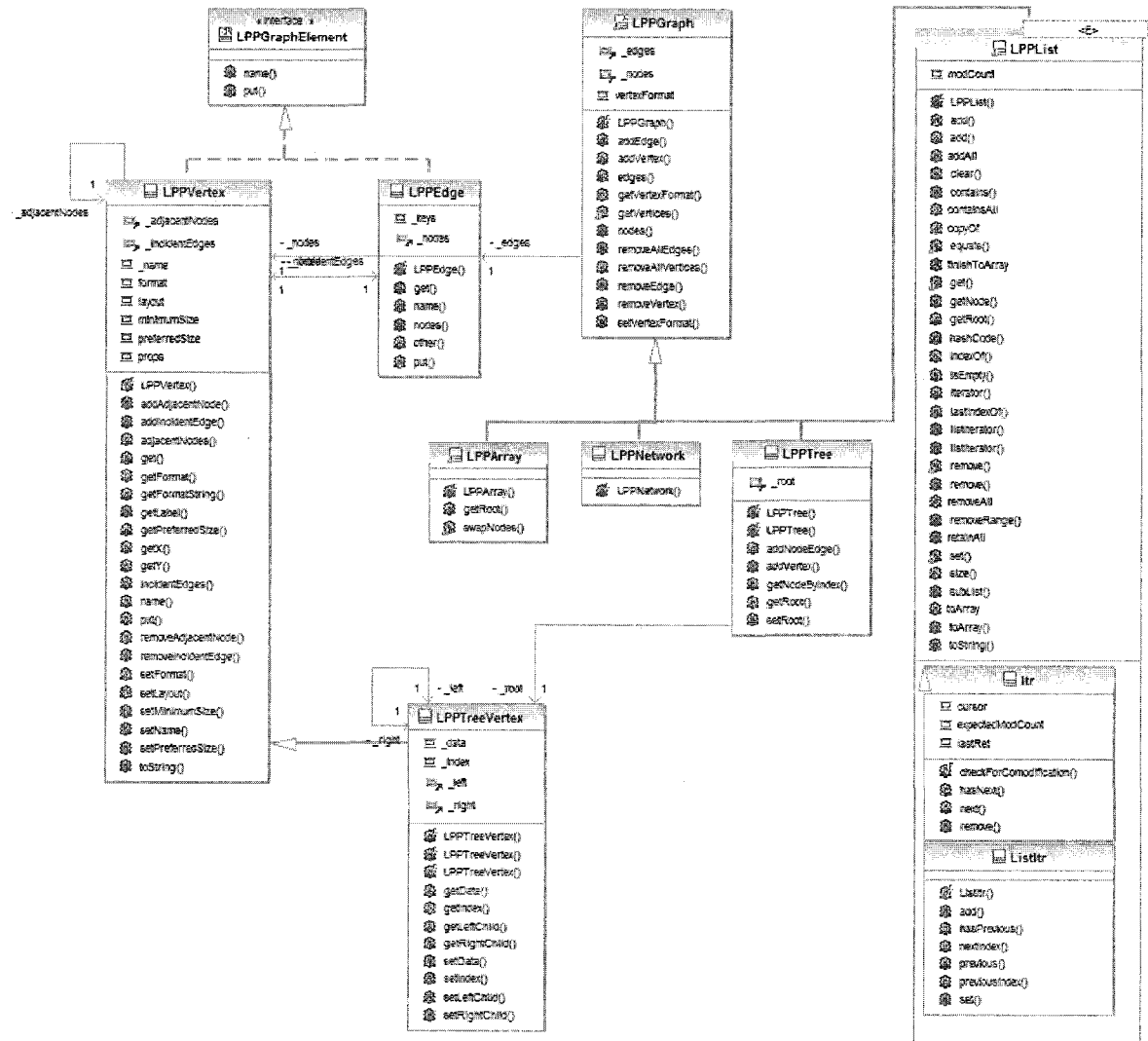


Figure 10: The graph package

The class LPPVertex extends the JUNG Vertex object to provide a mechanism for subclasses of Problem to interact with graph elements without having knowledge of JUNG. Additionally, as shown in Listing 10, LPPVertex maintains a key, value pair that can be used to store arbitrary data as required by problems and algorithms.

```
public void put(String key, Object value)
{
    props.put(key, value);
}
```

Listing 10: Put method on LPPVertex

The class LPPEdge extends the JUNG Edge object to provide a mechanism for subclasses of Problem to interact with graph elements without having knowledge of JUNG. Similar to the LPPVertex class, LPPEdge maintains an arbitrary key, value pairs.

The class LPPGraph encapsulates an underlying JUNG graph and LPPVertex and LPPEdge classes. It provides an abstraction of the graph to any subclasses of problem and any algorithms. Listing 11 contains an excerpt from the initialization code of LPPGraph in which existing vertices and edges are used to seed a new graph.

```
super ();

_nodes = vertices;
_edges = edges;
for (LPPVertex v : vertices) super.addVertex(v);
for (LPPEdge e : edges) super.addEdge(e);
```

Listing 11: Graph initialization

The class LPPArray encapsulates JUNG graph elements that are intended to be used in a tree representation. LPPArray bridges the gap between the visualization and the

array representation that may be used by a subclass of Problem or an algorithm. A key component of LPPArray is the swapNodes method. A portion appears in Listing 12.

```
for(int i = 0; i <= (_nodes.size() - 2) / 2; i++)
{
    LPPEdge left = new LPPEdge(_nodes.get(i), _nodes.get(2 * i +
        1));
    this.addEdge(left);
    if((2 * i + 2) < _nodes.size())
    {
        LPPEdge right = new LPPEdge(_nodes.get(i), _nodes.get(2 * i
            + 2));
        this.addEdge(right);
    }
}
```

Listing 12: Node swapping

The class LPPNetwork extends LPPGraph and is meant to be a more generic representation of an instance visualization. By default, it initializes with an empty set of vertices and edges.

The class LPPTree provides a mechanism for representation of tree structures in the framework. The method addNodeEdge provides a means for node insertion and is outlined in Listing 13.

```
public void addNodeEdge(LPPTreeVertex v, LPPTreeVertex newNode)
{
    if(v.getData() > newNode.getData())
    {
        if(v.getLeftChild() == null)
        {
            v.setLeftChild(newNode);
            LPPEdge e = new LPPEdge(v, newNode);
            this.addEdge(e);
        }
        else addNodeEdge(v.getLeftChild(), newNode);
    }
    else
    {
```

```

        if(v.getRightChild() == null)
        {
            v.setRightChild(newNode);
            LPPEdge e = new LPPEdge(v, newNode);
            this.addEdge(e);
        }
        else addNodeEdge(v.getRightChild(), newNode);
    }
}

```

Listing 13: Adding an edge to a LPPTree

The class LPPList exposes an underlying LPPGraph as a List object that provides convenient methods for iteration and comparison. Additionally, as illustrated in Listing 14, LPPList provides a set method that uses a LPPVertex's put method to save arbitrary data.

```

public E set(int index, E element)
{
    ArrayList<LPPVertex> nodes = (ArrayList<LPPVertex>)
    super.nodes();
    LPPVertex vertex = nodes.get(index);
    E r = (E) vertex.get("value");
    vertex.put("value", element);
    vertex.setName("" + element);
    return r;
}

```

Listing 14: Set method in LPPList

The class LPPTreeVertex extends LPPVertex to provide the functionality required of elements in a tree representation. LPPTreeVertex is used in the Heap sort algorithm. The member variables of LPPTreeVertex are shown in Listing 15.

```

private int _data;
private int _index;
private LPPTreeVertex _left;
private LPPTreeVertex _right;

```

Listing 15: Members of LPPTreeVertex

The interface LPPGraphElement is implemented by both LPPVertex and LPPEdge. It defines the signature of the put method. The put method is used by tools, visualizations and the Animator.

4.1.6 Moodle Package

The moodle package provides a mechanism for interfacing with Moodle. It contains the following classes: LPPResult and LPPMoodleError. It also contains the LPPAppletCallback interface.

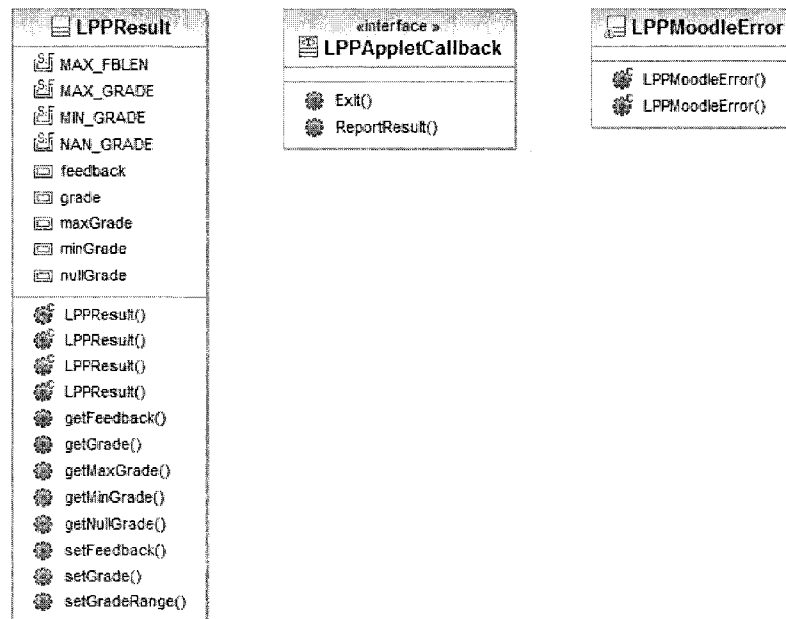


Figure 11: The moodle package

The class `LPPResult` serves as a container for all reporting and result data for a series of problems run in the framework. The class `LPPAppletCallback` enables initiation and shutdown of the framework from an external source. The class `LPPMoodleError` signals that an error condition has occurred in the communication between the framework and Moodle.

4.1.7 Threading Package

The threading package supports threading management in the framework. It contains the following classes: `SerialExecutor` and `ThreadPerTaskExecutor`.

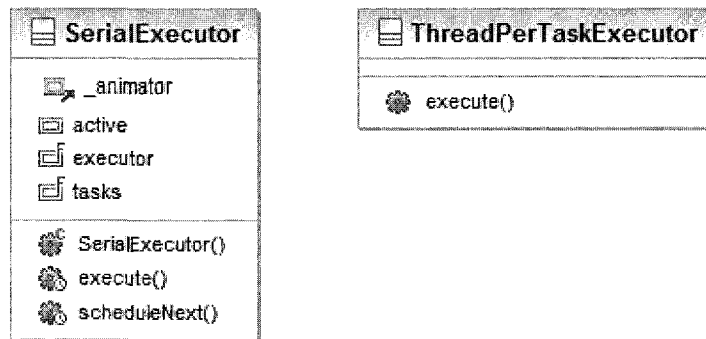


Figure 12: The threading package

The class `SerialExecutor` enqueues a series of execution requests from the `Animator` and uses a `ThreadPerTaskExecutor` to ensure that each is executed on its own thread. A portion of the `execute` method appears in Listing 16.

```

tasks.offer(new Runnable()
{
    public void run()
  
```

```

    {
        try
        {

            r.run();
            ((Problem)r).complete();

            _animator.stopCurrentProblem();

        }
        finally
        {
            scheduleNext();
        }
    }
});

if (active == null)
{
    scheduleNext();
}

```

Listing 16: Execute method in SerialExecutor

The class ThreadPerTaskExecutor consumes a Runnable object and starts it on its own thread. It is utilized by SerialExecutor to ensure that each problem is run on its own thread.

4.1.8 Tools Package

The tools package provides the infrastructure for tool support in the framework. It contains the following classes: Tool, ShowNextStepTool, SwapNodes, SelectTool, PickEdgeTool, PercolateDown, PutTool, FixHeap and Transform Tool. The Tool class serves as the base class for all tools. It abstracts much of the interaction with problems that most tools require.



Figure 13: The tools package

One of the principal methods of the framework, `expectSelection`, is detailed in Listing 17. The method is invoked by algorithm code and it causes the algorithm thread to block until the UI thread fires a notification event on a common lock object.

```

protected void expectSelection(LPPVertex v, boolean compoundStep)
{
    if(_animator.getMode() == AnimatorMode.ShowMe ||
        _animator.getShowStep())
    {
        _animator.moveCursorTo(v);
        _animator.clickAtCurrentLocation();

        wait(1500);

        // toggle show step
    }
}
  
```



```

        if(_animator.getShowStep() && !compoundStep)
            _animator.setShowStep(false);
    }
    else
    {
        while (true)
        {
            LPPVertex selected =
                _visual.waitForVertexSelection(this);

            // "Show Next Step" was clicked
            if(_animator.getShowStep())
            {
                expectSelection(v, compoundStep);
                break;
            }

            if (v == selected)
            {
                addCorrectMove();
                return;
            }
            else if(selected != null)
            {
                addWrongMove();

                javax.swing.JOptionPane.showMessageDialog(null,
                    "Incorrect Item Selected\n\nExpected: " +
                    v.name()
                    + "\nSelected: " + selected.name());
            }
        }
    }
}

```

Listing 17: ExpectSelection method in Tool

The ShowNextStepTool class toggles the framework's showStep flag to true and then attempts to re-call the current method. The result is that path of execution follows the path of execution for Show Me Mode for the current step. After the current step has completed, the showStep flag is toggled to false. The processing of the ShowNextStepTool is unique in that it does not occur in the tool definition itself. Rather,

as illustrated in Listing 18, the animator has a special case to handle mouse events on the ShowNextStepTool.

```
if (t.getClass().equals(ShowNextStepTool.class))
{
    _visualizationPanel.setShowStep(true);
    _currentProblem.nextStepShown();
}
```

Listing 18: Condition for Show Next Step Mode

The SwapNodes class provides a tool that allows for the selection of two items. The actual swapping of the items in the visualization is not performed by the tool.

The SelectTool class can be used whenever an arbitrary LPPVertex selection is required. SelectTool's only method, expectSelect primarily wraps the expectSelection method in the Tool base class.

The PickEdgeTool class supplies a method for selection of an edge based on the selection to LPPVertex objects. The order in which the LPPVertex objects are selected is not enforced. The mechanism for this is outlined in Listing 19.

```
LPPVertex selected1 = _visual.waitForVertexSelection(this);
LPPVertex selected2 = _visual.waitForVertexSelection(this);

if (selected1 == v1 && selected2 == v2 || selected1 == v2
    && selected2 == v1)
{
    addCorrectMove();
    return;
}
```

Listing 19: Vertex based edge selection

The PercolateDown class is identical to the SwapNodes class. It allows for order insensitive selection of two LPPVertex objects, and does not perform any changes to the visualization of the items.

The PutTool class serves a key role in mapping user's action with action expected by the algorithm. A portion of the principal method of PutTool, expectPut is shown in Listing 20. This portion of expectPut pertains to situations in which the algorithm expects a user to input a string or numeric value for an item. Every response is validated, and the algorithm does not continue until the correct value is received. Once the correct value is received, it is stored in the LPPVertex via a call to its put method.

The FixHeap class is identical to the SwapNodes and PercolateDown classes. It allows for order insensitive selection of two LPPVertex objects, and does not perform any changes to the visualization of the items.

```
String response = null;

do
{
    response = JOptionPane.showInputDialog(null,
        "Enter the new " + key + " value:", this.getName(),
        JOptionPane.QUESTION_MESSAGE);
}
while (null != response && 0 == response.length());

if (response.equalsIgnoreCase("" + value))
{
    v.put(key, value);
    addCorrectMove();
    return;
}
```

Listing 20: Validation of user input

The TransformTool class provides a means for a user to manipulate the visualization. Selection of the TransformTool causes all mouse activity in the visualization to be suppressed before any selection activity is detected. Listing 21 reveals a guard clause in VisualizationPanel's mouse event processing logic that permits the desired behavior.

```
if (parent.getSelectedTool() instanceof TransformTool)  
    return;
```

Listing 21: Transform tool support

5 Instructor Guide

The following chapter provides instructions for extending the framework and is targeted to instructors who need to add to any of the four major components. For the purposes of this guide, the four major components of the framework are problems, tools, layouts, and effects.

A convenient abstraction is to consider problems to be main component of extension since each problem has a layout and one or more tools, and most effects are triggered by problems. The suggested pattern for extending the framework follows this abstraction. An instructor should add a problem, associate a layout to the problem, associate any tools to the problem, and invoke any effects from the problem. Depending on the instructor's requirements, it may be necessary to add a new layout, tool, or effect. The addition of each component is covered in the following sections.

5.1 Adding a Problem

The most common type of framework extension will involve adding support for a new problem type. If existing tools, layouts, and effects are used, the only requirement will be to create a subclass of the Problem class. Adding tools, layouts, and effects is covered in subsequent sections.

The Problem class serves as base for all problems in the framework. Upon initialization, the framework queues and executes a series of problems. Each problem is

derived from the Problem class and it maintains its own collection of tools and is responsible for its own initialization and layout.

The Problem class provides several methods that can be overridden by subclasses, however it is only required that two methods be implemented.

The first method to be overridden, run, is intended to house the algorithm code and is called by the Animator once the problem has been initialized. The Animator interprets a return from the run method as completion of the problem instance. Listing 22 outlines the run method of our implementation of Prim's algorithm.

```
public void run()
{
    // ===== Initialization of vertices =====
    for (LPPVertex u : getNodes())
    {
        u.put("key", Integer.MAX_VALUE);
        u.put("pi", null);
    }

    final Comparator<LPPVertex> vertexComparator = new
        Comparator<LPPVertex>()
    {
        public int compare(LPPVertex a, LPPVertex b)
        {
            int d=(Integer)(a.get("key"))- Integer) (b.get("key"));
            return d != 0 ? d : a.name().compareTo(b.name());
        }
    };

    // ===== Processing of vertices =====
    PriorityQueue<LPPVertex> q = new
        PriorityQueue<LPPVertex>(getNodes().size(), vertexComparator);
    q.addAll(getNodes());
    LPPVertex r = q.peek();
    r.put("key", 0);
    while (q.size() > 0)
    {
        final LPPVertex u = q.remove();
        selectTool(st1);
        st1.expectSelect(u);
    }
}
```

```

u.put("color", Color.YELLOW);
glow(u);
ArrayList<LPPEdge> incidentEdges = new ArrayList<LPPEdge>(u
    .incidentEdges());
Collections.sort(incidentEdges, new Comparator<LPPEdge>()
{
    public int compare(LPPEdge a, LPPEdge b)
    {
        return a.other(u).name().compareTo(b.other(u).name());
    }
});
for (LPPEdge e : incidentEdges)
{
    LPPVertex v = e.other(u);
    int w = (Integer) e.get("w");
    if (q.contains(v) && w < (Integer) v.get("key"))
    {
        selectTool(pt1);
        pt1.expectSelectAndPut(v, "key", w);
        v.put("color", Color.ORANGE);
        glow(v);
        selectTool(pt2);
        pt2.expectPut(v, "pi", u);
        v.put("color", Color.PINK);
        q.remove(v);
        q.add(v);
    }
}
}
}
}

```

Listing 22: A sample run method

The method can be reviewed in two phases. In the first phase, the key and pi values of each vertex in the graph are initialized by calling the put (String keyName, Object value) method. Since our implementation of Prim's algorithm will break ties of key values alphabetically, a Comparator is defined. The second phase involves the progression of the algorithm through the vertices. Of special interest are the statements contained in the for loop. These statements set the expected tool selection, the expected vertex selection, the subsequent input, and invoke some effects. The expected tool

selection can be set via a call to the `selectTool` method. This method expects a `Tool` as an argument. The expected vertex selection can be set via a call method call on the `Tool`. In this instance, a call is made to `expectSelectAndPut` which takes the vertex to be selected, the name of the key value to be entered, and the expected value as arguments. Finally, an effect is invoked via a call to the `glow` method. The `glow` method expects a vertex as an argument.

The second method to be overridden, `generateProblem`, is intended to assist the `Animator` in initialization of the visualization of a problem instance. The primary purpose of this method is to initialize and return an object of type `LayoutMutable`. Presently, the framework supports three types of layouts and each type is represented by a different subclass of `LayoutMutable`. Listing 23 illustrates the signatures of the currently available generation methods:

```
public static LayoutMutable generateRandomGraph(int numVertices)
public static LayoutMutable generateRandomHeap(int numVertices)
public LayoutMutable generateRandomArray(int numItems)
```

Listing 23: Problem generation method signatures

Each of these methods is defined in the `Problem` base class, and should be invoked from the `generateProblem` method of any `Problem` subclasses.

After implementing these methods, it is suggested that an instructor add any tools to the problem. The recommended pattern is to add tools as member variables and

perform initialization in the constructor. Listing 24 illustrates how this pattern was applies to our implementation of Prim's algorithm.

```
private TransformTool ttl = null;

public Prim4()
{
    ttl = new TransformTool();
    ttl.setName("Clean up graph");
    ttl.setToolTip("Use this Tool to reposition vertices.");
}
```

Listing 24: Adding and initializing a tool to a problem

The final step in addition of a new problem is the integration of tools in the run method. Without tool integration, the run method will run to completion without any user action. In the course of execution of the run method, whenever user action is expected, a call to a tool's method should be added. As an example, if the user was expected to select an item, a call to the SelectTool's expectSelection method would be added. As a reference, the implementation of Prim's algorithm is provided in Listing 25.

```
import java.awt.Color;
import java.net.URL;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.PriorityQueue;
import java.util.Scanner;

import edu.sjsu.cs.lpp.engine.*;
import edu.sjsu.cs.lpp.tools.*;
import edu.sjsu.cs.lpp.graph.*;
import edu.uci.ics.jung.visualization.LayoutMutable;

public class Prim4 extends Problem
{
    private TransformTool ttl = null;
    private SelectTool st1 = null;
    private PutTool pt1 = null;
}
```

```

private PutTool pt2 = null;
private PickEdgeTool pet1 = null;
private PlayAnimationTool play = null;
private ShowNextStepTool showNext = null;

public Prim4()
{
    setName("Prim's Algorithm");
    tt1 = new TransformTool();
    tt1.setName("Clean up graph");
    tt1.setToolTip("Use this Tool to reposition vertices.");
    st1 = new SelectTool();
    st1.setName("Select a node");
    st1.setToolTip("Select the next vertex to be removed
        from the queue.");

    pt1 = new PutTool();
    pt1.ValueType = Integer.class;
    pt1.setName("Select neighbor, set key value");
    pt1.setToolTip("Click on the next neighbor (in alphabetical
        order), then enter its new key value.");

    pt2 = new PutTool();
    pt2.ValueType = LPPVertex.class;
    pt2.setName("Set \u03C0 value");
    pt2.setToolTip("Click on another node to set it as the
        \u03C0 value of the orange node.");

    play = new PlayAnimationTool();
    play.setName("Play Animation");
    play.setToolTip("Select this Tool to play and pause an
        animation of the algorithm.");

    showNext = new ShowNextStepTool();
    showNext.setName("Show Next Step");
    showNext.setToolTip("Select this Tool to see the next step
        in the animation of the algorithm.");

    addTool(tt1);
    addTool(st1);
    addTool(pt1);
    addTool(pt2);
    addTool(showNext);
    addTool(play);
}

public LayoutMutable generateProblem()
{
    return Problem.generateRandomGraph(7);
}

```

```

public void run()
{
    for (LPPVertex u : getNodes())
    {
        u.put("key", Integer.MAX_VALUE);
        u.put("pi", null);
    }
    final Comparator<LPPVertex> vertexComparator = new
        Comparator<LPPVertex>()
    {
        public int compare(LPPVertex a, LPPVertex b)
        {
            int d = (Integer) (a.get("key")) - (Integer)
                (b.get("key"));
            return d != 0 ? d : a.name().compareTo(b.name());
        }
    };
    PriorityQueue<LPPVertex> q = new
        PriorityQueue<LPPVertex>(getNodes()
            .size(), vertexComparator);
    q.addAll(getNodes());
    LPPVertex r = q.peek();
    r.put("key", 0);
    while (q.size() > 0)
    {
        final LPPVertex u = q.remove();
        selectTool(st1);
        st1.expectSelect(u);
        u.put("color", Color.YELLOW);
        glow(u);
        ArrayList<LPPEdge> incidentEdges = new
            ArrayList<LPPEdge>(u
                .incidentEdges());
        Collections.sort(incidentEdges, new
            Comparator<LPPEdge>()
        {
            public int compare(LPPEdge a, LPPEdge b)
            {
                return
                    a.other(u).name().compareTo(b.other(u).name());
            }
        });
    }
    for (LPPEdge e : incidentEdges)
    {
        LPPVertex v = e.other(u);
        int w = (Integer) e.get("w");
        if (q.contains(v) && w < (Integer) v.get("key"))
        {
            selectTool(pt1);
            pt1.expectSelectAndPut(v, "key", w);
            v.put("color", Color.ORANGE);
        }
    }
}

```


selected items are displayed to the user and the tool waits for the correct item to be selected.

```
protected void expectSelection(LPPVertex v, boolean compoundStep)
{
    if(_animator.getMode() == AnimatorMode.ShowMe ||
        _animator.getShowStep())
    {
        java.awt.Point p = _animator.getToolLocation(this);
        _animator.getProblem().moveCursor(p.x, p.y - 75,
            (int)v.getX() - 15, (int)v.getY());

        wait(1500);

        // toggle show step
        if(_animator.getShowStep() && !compoundStep)
            _animator.setShowStep(false);
    }
    else
    {
        while (true)
        {
            LPPVertex selected =
                _visual.waitForVertexSelection(this);

            // "Show Next Step" was clicked
            if(_animator.getShowStep() || _animator.getMode() ==
                AnimatorMode.ShowMe)
            {
                _animator.getProblem().selectTool(this);
                expectSelection(v, compoundStep);
                break;
            }

            if (v == selected)
            {
                addCorrectMove();
                return;
            }
            else if(selected != null)
            {
                addWrongMove();
                javax.swing.JOptionPane.showMessageDialog(null,
                    "Incorrect Item Selected\n\nExpected: " +
                    v.name()
                    + "\nSelected: " +
                    selected.name());
            }
        }
    }
}
```

```

    }
}

```

Listing 26: The expectSelection method

In the case of expectPut, behavior varies depending on the type of the value parameter. If it is a String or an Integer, the user is presented with a dialog prompt and must enter the expected value. Otherwise, the tool assumes that the expected value is a vertex selection and the user must selected the expected vertex.

```

public void expectPut(LPPVertex v, String key, Object value)
{
    if(_animator.getMode() == AnimatorMode.ShowMe ||
        _animator.getShowStep())
    {

        if (ValueType == Integer.class || ValueType ==
            String.class)
        {
            _animator.getProblem().showInformation("In the dialog
                that appears, enter " + value);
        }
        else
        {
            // assume vertex
            LPPVertex v2 = (LPPVertex)value;

            java.awt.Point p = _animator.getToolLocation(this);
            _animator.getProblem().moveCursor(p.x, p.y - 75,
                (int)v2.getX() - 15, (int)v2.getY());
        }

        v.put(key, value);

        // toggle show step
        if(_animator.getShowStep())
            _animator.setShowStep(false);
    }
    else
    {
        while (true)
        {
            // "Show Next Step" was clicked
            if(_animator.getShowStep() || _animator.getMode() ==

```

```

        AnimatorMode.ShowMe)
    {
        _animator.getProblem().selectTool(this);
        expectPut(v, key, value);
        break;
    }

// int or string?
if (ValueType == Integer.class || ValueType ==
    String.class)
{
    String response = null;
    do
    {
        response = JOptionPane.showInputDialog(null,
            "Enter the new " + key + " value:",
            this.getName(),
            JOptionPane.QUESTION_MESSAGE);
    }
    while (null != response && 0 ==
        response.length());
    if (response.equalsIgnoreCase("" + value))
    {
        v.put(key, value);
        addCorrectMove();
        return;
    }
    else
    {
        addWrongMove();

        javax.swing.JOptionPane.showMessageDialog(null,
            "Incorrect Value Entered\n\nExpected:
            " + value.toString() + "\nEntered:
            " + response);
    }
}
else
{
    // TODO: assume vertex
    LPPVertex lastSelected =
        _visual.waitForVertexSelection(this);

    // "Show Next Step" was clicked
    if(_animator.getShowStep())
    {
        expectPut(v, key, value);
        break;
    }

    LPPVertex vrtx = (LPPVertex) value;

```


Listing 28: A sample method for a new Tool

Any new tool should support Show Me Mode. Support for Show Me Mode usually entails the addition of a block of code at the beginning of any public methods. An abstraction for Show Me Mode is to consider two components. The first is the visualization, and it involves showing the user the expected action, and the second is the data seeding which involves setting values that allow algorithm code to proceed. Support for Show Me Mode in existing tools follows this abstraction. First, the correct action is shown to the user and second, and variables are set to the expected values. An example is outlined in Listing 29.

```
protected void expectSelection(LPPVertex v, boolean compoundStep)
{
    if(_animator.getMode() == AnimatorMode.ShowMe ||
        _animator.getShowStep())
    {
        java.awt.Point p = _animator.getToolLocation(this);
        _animator.getProblem().moveCursor(p.x, p.y - 75,
            (int)v.getX() - 15, (int)v.getY());

        wait(1500);

        // toggle show step
        if(_animator.getShowStep() && !compoundStep)
            _animator.setShowStep(false);
    }
    else
    {
        while (true)
        {
            LPPVertex selected =
                _visual.waitForVertexSelection(this);

            // "Show Next Step" was clicked
            if(_animator.getShowStep() || _animator.getMode() ==
                AnimatorMode.ShowMe)
            {
                _animator.getProblem().selectTool(this);
            }
        }
    }
}
```

```
        expectSelection(v, compoundStep);  
        break;  
    }  
}
```

Listing 29: Show Me Mode Support in the expectSelection method

The first if block checks if the system is in Show Me Mode or Show Next Step Mode. If this is the case, the cursor is moved to the location of the expected selection. Additionally, the Show Next Step Mode flag is toggled if this is not part of a compound step. A compound step is anything that involves more than one user action. An example would be if the user was expected to select an item, and enter a new value for it. At this point, the selection would be complete. The else block (which is the execution path if the first if block is not entered) also checks for the Show Next Step Mode and calls itself if the system is in Show Me Mode or Show Next Step Mode. This code may seem duplicitous, but it is needed since the system could enter Show Me Mode or Show Next Step Mode in two ways. First, the system could enter either mode before the call to expectSelection. In this case, the code in the if block will be executed and the method will return. Second, if the user is in the middle of an algorithm and the system is currently waiting for a selection, the system would continually execute the while loop in the else block. The seemingly duplicitous check for Show Me Mode or Show Next Step Mode will allow the method to exit the loop and return.

As a reference, the implementation of the PickEdgeTool is provided in Listing 30. The PickEdgeTool is intended for use when a user selected two vertices to indicate selection of this adjoining edge.

```

package edu.sjsu.cs.lpp.tools;

import edu.sjsu.cs.lpp.graph.LPPVertex;

public class PickEdgeTool extends Tool
{
    public void expectEdge(LPPVertex v1, LPPVertex v2)
    {
        while (true)
        {
            LPPVertex selected1 =
                _visual.waitForVertexSelection(this);
            LPPVertex selected2 =
                _visual.waitForVertexSelection(this);
            if (selected1 == v1 && selected2 == v2 || selected1 ==
                v2 && selected2 == v1)
            {
                addCorrectMove();
                return;
            }
            else
            {
                addWrongMove();
                javax.swing.JOptionPane.showMessageDialog(null,
                    "Incorrect Edge Selected\n\nExpected: " +
                    v1.name() + "-" + v2.name());
            }
        }
    }
}

```

Listing 30: The implementation of PickEdgeTool

5.3 Adding a Layout

A problem's visualization is driven by its layout. Each problem is required to associate itself with a layout. New problem types may require the addition of a layout not supported by the framework. As required, support for new layouts can be added to the framework.

This framework utilizes the JUNG framework for its visualizations. Accordingly, adding support for a new layout requires an extension of JUNG's `AbstractLayout` class. All layouts in the framework are derived from this class. Listing 31 depicts an extension of the `AbstractLayout` class.

```
package edu.sjsu.cs.lpp.engine;

import edu.uci.ics.jung.graph.Vertex;
import edu.uci.ics.jung.visualization.*

public class ArrayLayout extends AbstractLayout implements
    LayoutMutable
```

Listing 31: A extension of `AbstractLayout`

The addition of a new layout will most probably stem from a need to control the positional placement of items in the visualization. Varying problem types can require linear, random, or other placement of items in the visualization. To add a new layout, first, create a subclass of `AbstractLayout` and second, override the `initializeLocations` method.

The new implementation of `initializeLocations` will most likely enumerate the items in the visualization and set location information for each. An enumeration of the visualization elements can be obtained by calling the method `getGraph`, and positional information can be persisted via a call to the `setLocation` method. Listing 32 illustrates how `LPPGraph` enumerates the items and sets location information to generate random placement of vertices.

```
@Override
protected void initializeLocation(Vertex v, Coordinates coord,
```

```

Dimension d)
{
    double x = 20 + Math.random() * (d.getWidth() - 40);
    double y = 20 + Math.random() * (d.getHeight() - 40);

    coord.setX(x);
    coord.setY(y);
    ((LPPVertex) v).setLayout(this);
}

```

Listing 32: initializeLayout from LPPGraph

As a reference, the implementation of the ArrayLayout layout is provided in Listing 33.

```

package edu.sjsu.cs.lpp.engine;

import java.awt.Dimension;
import java.util.Collection;

import edu.uci.ics.jung.graph.Vertex;
import edu.uci.ics.jung.visualization.*;
import edu.sjsu.cs.lpp.graph.LPPGraph;
import edu.sjsu.cs.lpp.graph.LPPVertex;

/**
 * Lays out the nodes in a linear fashion.
 * @author cay
 *
 */
public class ArrayLayout extends AbstractLayout implements
    LayoutMutable
{
    public ArrayLayout(LPPGraph g)
    {
        super(g);
    }

    @Override
    protected void initializeLocations()
    {
        super.initializeLocations();
        final int DISTANCE = 5;

        int maxx = 0;
        int maxy = 0;

        for (LPPVertex node : ((LPPGraph) getGraph()).nodes())
        {

```

```

        Dimension dim = node.getPreferredSize();
        if (dim.width > maxx) maxx = dim.width;
        if (dim.height > maxy) maxy = dim.height;
    }

    int x = DISTANCE;

    for (LPPVertex node : ((LPPGraph) getGraph()).nodes())
    {
        int d1 = maxx / 2;
        x += d1;
        getCoordinates(node).setLocation(x, maxy / 2);
        x += maxx - maxx / 2 + DISTANCE;
        node.setLayout(this);
        node.setMinimumSize(new Dimension(maxx, maxy));
    }
}

public void update()
{
    initialize_local();
    initializeLocations();
}

protected void initializeLocation(Vertex v, Coordinates coord,
    Dimension d)
{
    double x = ((LPPVertex) v).getX();
    double y = ((LPPVertex) v).getY();
    coord.setX(x);
    coord.setY(y);
}

@Override
public void advancePositions()
{
}

@Override
protected void initialize_local_vertex(Vertex arg0)
{
}

public boolean incrementsAreDone()
{
    return true;
}

public boolean isIncremental()
{
    return false;
}

```

```
}  
}
```

Listing 33: The implementation of ArrayLayout

5.4 Adding an Effect

Effective visualizations of algorithm execution often require visual effects. The addition of a new problem type may require the addition of a new effect. Adding an effect requires two steps. First, the new effect must be added to the Effects class in the effects package. Second, the new effect must be invoked by the problem.

The Effects class is part of the effects package which contains a series of classes that power the visualization effects of the framework. The suggested manner of adding an effect is to add a method to Effects class. Additionally, this method should be invoked from a method in the Problem base class. Usually, a problem's run method invokes methods in the Effects class.

This framework uses the Timing framework to enable effects that occur over a time span. Pulsation and text movement are examples of such effects. Such effects require the addition of a method that is called at every interval in the duration of the effect. The straightLine effect provides an intuitive implementation. This effect moves an item along a linear path. The movement is accomplished by setting the position of the item at every interval in the duration. Listing 34 outlines how this is implemented in the straightLine effect. It may be helpful to use this as a template for any new effects.

```
public static Animator straightLine(final Locatable object, final  
    double x1, final double y1, final double x2, final double y2)
```

```

    {
        return new Animator(DEFAULT_DURATION, new TimingTargetAdapter()
        {
            @Override
            public void timingEvent(float t)
            {
                double dx = x2 - x1;
                double dy = y2 - y1;
                double x;
                double y;
                x = x1 + t * dx;
                y = y1 + t * dy;
                object.setX(x);
                object.setY(y);
            }
        });
    }
}

```

Listing 34: The implementation of the straight line effect

Once the new effect has been added to the Effects class, it is necessary to invoke it from the problem. Effect invocations often occur after a user action. A common scenario is to perform an effect after the user has selected an item. Listing 35 illustrates how to pulsate an item's borders after user selection. The effect name is Glow and it is invoked via calls to the glow method.

```

selectTool(putTool);
putTool.expectSelectAndPut(vertex, "key", value);
vertex.put("color", Color.ORANGE);
glow(vertex);

```

Listing 35: Pulsating an item after selection

As a reference, the implementation of Prim's run method is given in Listing 22 to illustrate the invocation of effects in the Effects class.

6 Conclusion and Future Work

6.1 Conclusion

Our initial goals were to create a framework that enabled active learning by: allowing for instructors to enable instances of a problem type by making a minimal amount of modifications to existing algorithm code, allowing students to iterate through visual representations of the intermediate steps of algorithm executions, and allowing random generation of problem instances. Additionally, it was decided that the framework should support modes in which the entire execution of an algorithm or the execution of a step is shown without user interaction

As pertains to the goal of enabling new problem instances with minimal modifications to existing algorithm code, the current framework performs satisfactorily. The changes to existing code in the implementations of Prim's algorithm, Insertion sort and Heap sort were minimal, and the majority of the required effort dealt with tool and direction setup.

Gauged against the goal of allowing students to iterate through visual representations of the intermediate step of algorithm execution, the current framework performs satisfactorily. A user can iterate through each step of the execution of Prim's algorithm, Insertion sort, and Heap sort.

With regard to the goal of allowing random generation of problem instances, the current framework performs satisfactorily. Every instance of Prim's algorithm, Insertion sort and Heap sort includes randomly or pseudo randomly generated data.

Finally, with respect to the goal of the framework supporting modes in which the entire execution of an algorithm or the execution of a step is shown without user interaction, the current framework performs satisfactorily.

6.2 Future Work

The uniqueness of the framework lies in how easily it can be extended or adapted to new problem types and tools. In general, the framework should be augmented to support a larger variety of animations and should interface with a larger variety of systems.

Currently, the framework supplies the infrastructure for the following three problem types: undirected graphs, trees and arrays. Support for directed graphs and hash tables should be added since they are extensions or existing supported problem types. Additionally, support for problem types involving representations like linked lists and B-trees will likely fuel adoption.

Several responses from the initial usage included suggestions for the framework. Most prevalent among the suggestions was an enhancement to the context sensitive information display. At present, information is displayed for the current problem and the

currently selected tool. A potential enhancement would be to display action sensitive information. Specifically, respondents requested a legend for any color coding, and a reminder about their last action.

Works Cited

- Ben-Ari, M., and Levy, R. "Why We Work So Hard and They Don't Use It: Acceptance of Software Tools by Teachers." SIGCSE Annual Conference on Innovation and Technology in Computer Science Education (2007): 246-250.
- Carlson, D., Guzdial, M., Kehoe, C., Shah, V., and Stasko, J. "WWW interactive learning environments for computer science education." SIGCSE Bull. 28, 1 (1996): 290-294.
- Erkan, A., Scaffidi, T., and VanSlyke, T.J. "Data Structure Visualization with LaTeX and Prefuse." SIGCSE Annual Conference on Innovation and Technology in Computer Science Education (2007): 301-305.
- Hasse, Chet. "Timing is Everyting." timingframework: Timing is Everything March 1, 2008 <<https://timingframework.dev.java.net/>>
- Helmick, M. "Integrated Online Courseware for Computer Science Courses." SIGCSE Annual Conference on Innovation and Technology in Computer Science Education (2007): 146-150.
- Helmick, M. "Interface-based Programming Assignments and Automatic Grading of Java Programs." SIGCSE Annual Conference on Innovation and Technology in Computer Science Education (2007): 63-67.
- Jarc, D. J., Feldman, M. B., and Heller, R. S. "Assessing the benefits of interactive prediction using Web-based algorithm animation courseware." SIGCSE Bull. 32, 1 (2000): 377-381.
- Korhonen, A., Laakso, M., and Myller, N. "Analyzing Engagement Taxonomy in

Collaborative Algorithm Visualization.” SIGCSE Annual Conference on Innovation and Technology in Computer Science Education (2007): 251-255.

Lahtinen, E., Javinen, H., and Melakoski-Vistbacka, S. “Targeting Program Visualizations.” SIGCSE Annual Conference on Innovation and Technology in Computer Science Education (2007): 256-260.

Naps, T., Cooper, S., Koldehofe, B., Leska, C., Rößling, G., Dann, W., Korhonen, A., Malmi, L., Rantakokko, J., Ross, R. J., Anderson, J., Fleischer, R., Kuittinen, M., and McNally, M. “Evaluating the educational impact of visualization.” In Working Group Reports From ITiCSE on innovation and Technology in Computer Science Education (2003): 124-136.

“Overview.” JUNG – Java Universal Network/Graph Framework March 1, 2008
<<http://jung.sourceforge.net>>

Sangwan, R. S., Korsh, J. F., and LaFollette, P. S. “A system for program visualization in the classroom.” SIGCSE Bull. 30, 1 (1998): 272-276.

Shaffer, C. A., Cooper, M., and Edwards, S. H. “Algorithm visualization: a report on the state of the field.” SIGCSE Bull. 39, 1 (2007): 150-154.