Algorithm Animator

A Master’s Paper in
Computer Science
by
Gretchen Keller

@2002 Gretchen Keller

Submitted in Partial Fulfillment
of the Requirements
for the Degree of
Master of Science

December 2002
# Table of Contents

Abstract iii

Acknowledgements iv

List of Figures v

List of Tables vi

1 Introduction 1

2 Design 3
   2.1 Framework ........................................ 5
   2.2 Interface ........................................ 6

3 Implementation 8
   3.1 Framework ........................................ 8
      3.1.1 The AnimAlg Class ............................ 9
      3.1.2 The Algorithm Class .......................... 10
      3.1.3 The GraphicAlg Class ........................ 11
      3.1.4 The TextAlg Class ............................ 12
   3.2 Interface ......................................... 13
      3.2.1 Menubar ...................................... 13
      3.2.2 Title Area .................................... 14
      3.2.3 Animation Panel ............................... 14
      3.2.4 Control Panel ................................. 14
   3.3 Implemented Algorithms .............................. 16
      3.3.1 Selection Sort ................................ 16
      3.3.2 Bubble Sort ................................... 17
      3.3.3 Prim’s Algorithm .............................. 19
      3.3.4 Kruskal’s Algorithm ........................... 20
4 Usage Guide

4.1 Program Installation and Setup .................................. 25
4.2 Using AnimAlg to Learn about Algorithms ..................... 25
4.3 Adding an Algorithm .............................................. 26

5 Conclusion ........................................................................ 32

References ........................................................................... 32
Abstract

This paper describes the Algorithm Animator program. Algorithm Animator is a program that was designed with two goals in mind: to be an extendable framework so that a moderately sophisticated Java programmer could add new algorithms easily, and to be a means for undergraduate computer science students to better understand various algorithms by presenting them visually.
Acknowledgements

I would like to thank:

- Kerry, Gabriel, and Miranda for your tolerance of your distracted wife and mother and for your gracious acceptance of your TV-dinner lifestyle over the past several months (years?). You mean the world to me.

- Mom and Dad and the rest of my family for encouraging me every step of the way. I am truly blessed to have grown up in such loving and supportive surroundings.

- Dr. Bui for generously giving of your time and knowledge. Your combination of intelligence and humility is rare.

- The rest of the committee members – Dr. Ding, Dr. Naumov, and Dr. Null – for not only reviewing this paper, but also enduring its defense.

- My teachers and the staff at Penn State Harrisburg, Penn State Main, Elk County Christian and Queen of the World – especially Mr. Sini-baldi and Mr. Snelick – I can only hope that someday I can inspire others like you have inspired me.

- My co-workers and managers at EDS and HRB. I would not have been able to accomplish this without your technical, emotional, and financial support.

- God for health, happiness, and all of the above!!!
List of Figures

1. High-level interaction diagram ........................................... 4
2. Interface areas ................................................................. 7
3. AnimAlg program interface .................................................. 13
4. AnimAlg program running the Selection Sort algorithm .......... 15
5. The selection sort algorithm .................................................. 17
6. The bubble sort algorithm ..................................................... 18
7. AnimAlg program running the Bubble Sort algorithm .......... 19
8. Prim’s minimum-cost spanning tree algorithm ......................... 21
9. AnimAlg program running Prim’s MST algorithm ...................... 22
10. Kruskal’s minimum-cost spanning tree algorithm ..................... 23
11. AnimAlg program running Kruskal’s MST algorithm .................. 24
12. The bubblesort.properties file ........................................... 27
13. The bubblesort.txt file ......................................................... 28
14. AlgImpl.java.hide – Page 1 ................................................ 29
15. AlgImpl.java.hide – Page 2 ................................................ 30
List of Tables

1 A Comparison of AnimAlg's features with those of similar programs 2
1 Introduction

The idea for the Algorithm Animator program (AnimAlg) is based on the premise that it would be nice for undergraduate computer science students to learn the basics of a given algorithm by seeing a real-time graphical depiction of what is happening within the algorithm as it progresses while at the same time seeing what line of the algorithm’s text is being executed. Such a program should be able to be extended as more algorithms are programmed so that any reasonably skilled Java programmer could add algorithms into the program for presentation in an undergraduate class.

Other programs that have similar functionality include Zeus [2], XTango [4], and Leonardo [3]. A summary of these programs’ features compared to AnimAlg is given in Table 1.

Zeus is an algorithm animation system that was written at Compaq’s Systems Research Center. It was written in Modula-3 in the early 1990s. One of the primary differences between Zeus and AnimAlg is that Zeus does not appear to show a textual version of the algorithm being animated or to correlate a certain piece of the animation with a particular line of the algorithm’s text. Another difference between Zeus and AnimAlg is that it seems to be much more complicated to add new algorithms to Zeus (of course, this complexity probably means that Zeus is capable of much more than AnimAlg). A “simple” bin-packing algorithm example given in [1], required seven input files written in five different languages in order to implement the algorithm. Another important difference between AnimAlg and Zeus is that Zeus does not display the algorithm’s text on the screen or correlate it in any way with the animation.

XTango is another general purpose algorithm animation system that was written at Georgia Tech in the early 1990s [4]. While one of its focuses is on ease of use, it is written for a UNIX system and uses the X11 Windows System. It also requires that the algorithms to be added be implemented in C. Thus, XTango seems to be less portable than AnimAlg. Another important difference between AnimAlg and XTango is that XTango does not display the
algorithm’s text on the screen or correlate it in any way with the animation. Leonardo is an “integrated environment for developing and animating C programs” [3]. A developer using Leonardo is able to animate her C programs by embedding calls to the ALPHA visualization programming language directly into the C code. Leonardo provides the capability of stepping backwards through C code and was written primarily as a tool for helping its users to learn the C programming language. What makes AnimAlg different from Leonardo is that AnimAlg correlates the text of the running algorithm with its animation and Leonardo does not.

This paper specifies the design and implementation of the Algorithm Animation program. The remainder of this paper is organized as follows. In Section 2 we discuss briefly the design of AnimAlg’s framework and interface. In Section 3 we discuss in some detail the implementation of the framework and the interface and also discuss the algorithms that were implemented as part of the proof of concept for this program. Next, in Section 4 we provide a usage guide for people wishing to use AnimAlg or add algorithms to it. Finally, in Section 5 we draw conclusions and offer ideas for future work on the AnimAlg project.

<table>
<thead>
<tr>
<th>Easily Extensible</th>
<th>XTango</th>
<th>Leonardo</th>
<th>Zeus</th>
<th>AnimAlg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supports Recursion</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Coordinates Graphics</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Animation in more than two dimensions</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Step backwards</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Portability</td>
<td>No</td>
<td>No</td>
<td>Somewhat</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 1: A Comparison of AnimAlg’s features with those of similar programs
2 Design

There were two primary design goals for the AnimAlg project:

1. To be an extendable framework to which new algorithms could be readily added as the need arises.

2. To be a tool that facilitates students’ understanding of basic algorithms.

In this and future sections we will refer to two different types of users of the AnimAlg program: the end user — the computer science student who will use this program in order to learn the specifics of a given algorithm, and the intermediate user — the programmer who will be extending the framework by implementing new algorithms to fit in to it.

This section describes the design of AnimAlg from the viewpoints of both types of users: the framework (from the point of view of the intermediate user) and the interface (from the point of view of the end user).

Figure 1 shows an abstract depiction of the interactions among the major components of AnimAlg. The Interface and the Framework depicted in this diagram are the same ones discussed in this and other sections. The “Algorithm Information” bubbles depicted in the diagram refer to files that the intermediate user must supply with every new algorithm that is implemented. The arrows in the diagram represent the flow of information within AnimAlg. The interface communicates to the framework such things as requests to load, run, pause, stop, or change the speed of an algorithm’s animation. The framework communicates back to the interface the current state of the algorithm and information about how to draw the animation. The interface also gets information directly from the intermediate user’s input, such as the algorithm’s title and the algorithm’s explanation. The framework looks to the information supplied by the intermediate user for the details on how to animate any given algorithm as well as the text to display and highlight for the algorithm.
Figure 1: High-level interaction diagram
2 DESIGN

2.1 Framework

The primary design goal for AnimAlg was that it be an extendable framework such that algorithms could be added to it as the need arises. This framework consists of a set of mechanisms for controlling the flow of an algorithm, drawing algorithm graphics, and coordinating the drawing of the graphics with the highlighting of text, etc.

Some of the questions that arose during the design of the framework were:

- How do we make the framework generic enough to avoid a re-compile every time that a new algorithm is implemented?
- What mechanism do we use to inform the program that there are new algorithms available?
- How do we correlate the algorithm’s text with its animation?
- How do we coordinate the animation of an algorithm with the highlighting of the corresponding line of text, e.g., How do we handle conditional statements and loops in the algorithm’s text?
- Should we allow more than one algorithm to run simultaneously?
- How do we add the ability to stop a running algorithm and reset it so that it could be run again from the beginning?
- How do we add the ability to pause a running algorithm so that it can be discussed and then resumed at a later time?
- How do we control the speed of the algorithm animation and accommodate for faster or slower processors?

We will show how these framework design questions were answered in Section 3.1.
2 Design

2.2 Interface

The interface of AnimAlg is the graphical piece with which the end user interacts in order to get a better grasp of the workings of a given algorithm. It must provide the end user access to all of the functionality of the framework without bogging her down with unnecessary detail. An attempt was made to make the interface flexible, yet intuitive, so that a highschool-aged end user could easily make sense of it.

The two main components that were needed in the graphical user interface were: an area on which to draw the algorithm’s animation, and a separate (though very close in terms of location on the screen) area on which to draw the algorithm’s text so that the end user can easily track the animation’s progress. Secondarily, we needed an area that the end user could use to issue commands in order to interact with the running algorithm (e.g., loading, starting, stopping, or changing the speed of the algorithm) or to keep tabs on the algorithm’s progress by viewing the output. This interaction area needed to be able to support both generic functions that should be available for all algorithm implementations, as well as algorithm-specific interfaces which the algorithm implementors could add at their discretion.

Figure 2 shows the various areas mentioned above: the animation area (1) outlined in blue, the text area (2) outlined in green, the generic control area (3) outlined in red, and the implementation-dependent control area (4) outlined in black.
Figure 2: Interface areas
3 Implementation

In this section we discuss the implementation of both the framework and the interface of the AnimAlg program and describe the algorithms that were implemented as part of the proof-of-concept for this program. The AnimAlg program itself was written and tested using Java 1.4 on a Windows 2000 machine, and though Java is supposedly cross-platform, no guarantee is given as to its behavior on other systems.

3.1 Framework

The intermediate user must be familiar with the framework (in addition to being familiar with the interface) in order to be able to implement new algorithms to be presented with this program.

The end user specifies a particular algorithm to study by choosing the corresponding configuration file. The configuration file (a Java .properties file) specifies the algorithm’s name, the name of the file where the algorithm’s text can be found, and the name of the primary Java .class file that implements the algorithm. This configuration file is also the mechanism used in order to “advertise” all the algorithms that are currently available to be animated. By putting a configuration file in the appropriate directory, the intermediate user is effectively saying that the algorithm it represents is ready for use within the framework.

The intermediate user is responsible for creating all of the pieces found in the configuration file (the properties file, the algorithm’s text file, and the implementation files) and putting them together. More will be said about these pieces in Section 4. AnimAlg uses Java’s reflection mechanism so that intermediate users can add algorithms (in the form of one or more Java classes) to the program without having to recompile in order to notify the framework that new algorithm is available to be loaded and run.

Though the animation itself was implemented as a thread, only one animation can be running at a particular time. This design decision was made because we felt that multiple animations running simultaneously would not
be a very useful feature for the amount of time it would take to implement
and the overhead that would be needed to keep track of several running algo-
rithms. As currently designed, the framework is able to keep track of requests
for stopping or pausing an algorithm by setting a generic flag (as opposed to
a specialized one on a per-algorithm basis). These flags – one for pause and
one for stop – are checked each time a new line is to be highlighted and the
appropriate action is taken if one of them is set.

The speed of the algorithm is controlled within the framework by making
a call to the \texttt{Thread.sleep()} method each time a new line of text is to
be highlighted. Currently, at the “default” speed, each animation pauses
for 0.5 seconds on each line of the algorithm’s text. This was done so that
as processors get faster, the animation won’t happen so quickly that it is
impossible for the end user to follow what is taking place. The speed can be
raised or lowered using the interface.

The framework itself was implemented primarily using the following four
Java classes:

\textbf{AnimAlg} The main class for the Algorithm Animator (AnimAlg) program.

\textbf{Algorithm} The class that handles the coordination between the graphical
and textual pieces of the algorithm.

\textbf{GraphicAlg} An abstract class that the intermediate user must extend in or-
der to implement a new algorithm. It provides access to the Algorithm
class.

\textbf{TextAlg} The class that is responsible for all things having to do with reading
in, displaying, and highlighting the algorithm’s text on the screen.

\subsection{3.1.1 The AnimAlg Class}

This is the highest level class of the AnimAlg program. It contains the “main”
routine from which all other classes and methods are ultimately called. This
method extends the \texttt{JFrame} class and initializes many of the components for
both the framework and the interface. The AnimAlg constructor is responsible for reading in the master.properties file and creating the menubar and the main panel. The main method of the AnimAlg class simply instantiates the AnimAlg class and then loads the specified algorithm. The user can specify the initial algorithm to be loaded in one of three ways (options are listed in order of precedence):

1. By including the name of the algorithm’s .properties file on the command line when the AnimAlg program is executed.

2. By specifying the name of the algorithm’s .properties file in the animalg.algorithm field of the master.properties file.

3. By choosing the algorithm’s .properties file from the dialog box that comes up if no default algorithm is specified.

The AnimAlg class is implemented using the singleton pattern so that only one instance of the class exists in any one execution of the program.

3.1.2 The Algorithm Class

This class is the true heart of the framework. Like the AnimAlg class, it too is implemented using the singleton pattern so that only one instance of any Algorithm can exist at any given time.

Two very important attributes are the myGraphicAlg and myTextAlg, which provide “hooks” into both the animation and the display of text for the algorithm. It is these hooks that enable the coordination between the visual display of the animation and the highlighted line of the algorithm’s text. Other attributes of the Algorithm class include the name of the algorithm, an explanation for the algorithm, its state, its preferred speed, some control-flow flags, and a pointer to the thread object which displays the animation.

The two main methods of the Algorithm class are the load() and run() methods. The load() method is overloaded. One of the two overloaded methods takes no arguments, determines the default algorithm (if one exists), and passes it on to the other load(String fn) method. The second of
3 IMPLEMENTATION

the overloaded methods takes a String argument that specifies the name of the .properties file of the algorithm to be loaded. The load(String fn) method uses Java’s reflection mechanism in order to be able to load new algorithms without ever having to recompile the framework’s code. The run() method of the Algorithm class creates and starts the Thread which executes the algorithm’s animation. This can only happen when the Algorithm’s state is LOADED (as opposed to NOT_LOADED or RUNNING).

3.1.3 The GraphicAlg Class

This is the class with which the intermediate user must be the most familiar. It is an abstract class and must be extended in a certain way in order for a new algorithm to be implemented. For example, each animation runs as a separate Thread, so the class that extends GraphicAlg must provide a run() method that will be called when the end user requests that the algorithm be run. The class that extends GraphicAlg also must provide a paintComponent(Graphics g) method so that the framework knows how to draw the animation. The specific details of how the GraphicAlg class must be extended are provided in Section 4. Like all other classes in the Framework, the GraphicAlg class is implemented using the singleton pattern. The GraphicAlg class extends the JPanel class to enable the animation to be drawn on the interface. It also implements the Runnable interface, so that the AnimAlg interface can be responsive while the animation is being displayed.

The GraphicAlg class has only two attributes: myGraphicAlg, which is the singleton instance of this class, and myUserPanel, which provides a way for the end user to enter input and receive output specific to a given algorithm. The constructor for the GraphicAlg class sets up the panel where the animation is to be displayed and its borders, as well as creating a generic new UserPanel for this particular algorithm. The UserPanel can (and perhaps should) be overridden in the subclass. Other methods of the GraphicAlg class include accessors for the myUserPanel instance variable, warmup(), cooldown(), pause(), nextLine(), nextLoop(), and endLoop(),
which simply provide encapsulation of the `Algorithm` class by calling similar methods there, and a `paintComponent(Graphics g)` method, which is meant to be overridden in the subclass.

### 3.1.4 The TextAlg Class

The TextAlg class is responsible for all things relating to the text of the algorithm. This includes, but is certainly not limited to: reading it in from the text file, tracking its structure and progress (through loops, etc.), drawing the text on the screen, and highlighting the appropriate line as the algorithm executes.

Like GraphicAlg, TextAlg extends JPanel, creating a place on the interface on which the algorithm’s text can be presented. It does not need to implement the `Runnable` interface since the drawing of the algorithm’s text is ultimately controlled by the GraphicAlg thread.

The attributes of the `TextAlg` class include the `filename` of the file that holds the text for the algorithm, the singleton instance variable `myTextAlg`, an array which holds information about each line of the algorithm’s text `algAry`, and an index into `algAry` to indicate the current line `currentLine`.

The constructor of the `TextAlg` class reads in the contents of the algorithm’s text file and uses it to populate the `algAry` array with information regarding where loops start and stop, how many lines of text are there, how much each line of text should be indented on the screen, and text itself of each individual line. The `paintComponent(Graphics g)` method uses the information in the `algAry` array to draw the algorithm’s text on the screen and highlight the appropriate line of text (if any). The `nextLine()`, `nextLoop()`, and `endLoop()` methods update the `algAry` array to ensure that the current line of the algorithm is unhighlighted and the next line (as appropriate) of the algorithm gets highlighted.
3 IMPLEMENTATION

3.2 Interface

The interface of AnimAlg is the graphical piece that the end user interacts with in order to get a better grasp of the workings of a given algorithm. The interface consists of a menu bar, a title area, an animation area, and a control area. A picture of the interface with no algorithm loaded is shown in Figure 3.

3.2.1 Menubar

The menubar sits at the very top of the interface and provides the user with the same functionality as the static part of the control panel, which is explained in more detail in Section 3.2.4.

Figure 3: AnimAlg program interface
3 IMPLEMENTATION

3.2.2 Title Area

The title area displays the algorithm’s title as well as providing some explanation or history of the algorithm. This information is different for each algorithm and as such must be obtained from the configuration file. The configuration file is provided by the intermediate user responsible for this particular algorithm.

3.2.3 Animation Panel

The animation panel could be considered the “main” part of the interface and is split into two parts. The left side shows the algorithm graphically as it is executing and the right side shows the algorithm’s text with the current statement being processed highlighted.

Both sides of this panel are algorithm-dependent — the information to be drawn on the animation panel comes from the `paintComponent()` method of the Java class provided by the intermediate user. The algorithm’s text itself is read from a text file provided by the intermediate user. The coordination between the two is achieved when the intermediate user places calls to the framework within the `run()` method of the primary Java class for the algorithm.

3.2.4 Control Panel

The control panel also consists of two parts: a static or generic part that the same no matter which algorithm is currently loaded or running, and a dynamic or algorithm-specific part that the intermediate user can program in order to get input from or display output to the end user as the algorithm is running.

The static part of the control panel provides a means of loading a new algorithm, running or stopping the currently loaded algorithm, pausing or resuming the currently running algorithm, and speeding up or slowing down the currently running algorithm. Although we call it the “static” portion of the control panel, it is not truly static in that it changes based on the
3 IMPLEMENTATION

Figure 4: AnimAlg program running the Selection Sort algorithm

state of the algorithm, e.g., only the load and exit buttons are active before any algorithm is loaded, the load button is inactivated while an algorithm is running, the pause button is only activated when an algorithm is running, and it changes to a “resume” button when the algorithm is paused, etc. As mentioned in Section 3.2.1, all of the functionality of the static portion of the control panel is duplicated in the menu bar.

The dynamic part of the control panel provides a mechanism for the intermediate user to get input from and display output to the end user. As shown in Figure 4, the dynamic portion of the control panel for the selection sort algorithm allows the user to enter the preferred number of items for the algorithm to sort.
3.3 Implemented Algorithms

AnimAlg’s primary motivation was as a framework to which new algorithms could be added, but we’ve implemented four algorithms in order to do a proof-of-concept for the program. The four algorithms that were implemented are described in the next several sections.

3.3.1 Selection Sort

In general, sorting algorithms solve the problem of turning an unordered array of values, or keys, into an array of keys sorted in increasing order. Selection sort is a very basic sorting algorithm.

Idea – Starting with an unsorted array, we scan the array from left to right looking at element 1 through \( n \) to find the smallest element in the array. We then swap this element with the element in the first position of the array. We can consider the elements that have already been swapped to the left side of the array to be sorted. Next, we start at the second position of the array (the first position of the unsorted array since element 1 has already been sorted) and scan elements 2 through \( n \) to find the smallest element in the unsorted array. This element is exchanged with the element in position 2 of the array (i.e., position 1 of the unsorted array). We continue this process, starting each search at position \( i \) – the first position of the unsorted array – for the smallest remaining element and swapping the smallest element with the element at position \( i \). We do this until the unsorted array is empty. At this point the array is sorted. The algorithm is shown in Figure 5.

Implementation – In AnimAlg, this algorithm is animated using vertical bars of differing lengths to represent the elements in the array. Originally, the bars are unsorted and in this state they are colored black. A yellow bar indicates the current position of the algorithm - all bars before it in the array are blue to indicate that they have been sorted. A green bar is used to indicate the element that’s currently under scrutiny to find out if it is the smallest one left in the unsorted array. A red bar is used to indicate that
3 IMPLEMENTATION

SelectionSort(A, n)  
for i := 1 to n - 1  
    smallest := i;  
    for j := i to n  
            smallest := j;  
    swap(A[i], A[smallest])  

Figure 5: The selection sort algorithm

d this bar is the smallest one that’s been found (thus far) in the remaining portion of the array. With the selection sort algorithm, the end user has the option of indicating the number of items to be sorted (between 2 and 20 for purposes of clarity on the screen).

3.3.2 Bubble Sort

As stated above in Section 3.3.1, sorting algorithms solve the problem of turning an unordered array of values, or keys, into an array of keys sorted in increasing order. Bubble sort is another very basic sorting algorithm.

Idea – For the bubble sort algorithm, we start again with an array A of n unsorted items, and split our array into a sorted side on the left and an unsorted side on the right. Starting at the n\textsuperscript{th} element and working backwards, we compare A[n] with element A[n - 1]. If A[n] < A[n - 1], then we swap A[n] and A[n - 1] and move on to the next pair of adjacent elements, A[n - 1] and A[n - 2]. We continue in this way to the beginning of the array.
Figure 6: The bubble sort algorithm

when we compare elements $A[2]$ and $A[1]$ and swap them if necessary. After this swap, the element in $A[1]$ should be the smallest element in the whole array and becomes the first member of the sorted array. At this point, we start the process over again at element $A[n]$, comparing it with its neighbor, and swapping if necessary, all the way down to $A[2]$ (we need not compare $A[1]$ since it was sorted in the last round). In this way, in every iteration of the loop, the smallest element left in the unsorted array is “bubbled” backwards to become the largest element in the sorted array. At the end of $n - 1$ such iterations, we have a sorted array. The algorithm is shown in Figure 6.

Implementation – This algorithm is implemented in AnimAlg and reuses much of the code that was written for the selection sort implementation. This is one of the beauties of the AnimAlg program – code written to implement one algorithm can often be reused in the implementation of another similar algorithm. As in the selection sort algorithm, the array of elements to be sorted is represented by a series of vertical bars. The bars are black when they are unsorted, a yellow bar indicates the “current” item position being sorted, and a blue bar indicates that this element has already been sorted.

```plaintext
BubbleSort(A)
for i := 1 to length[A]
    for j := length[A] down to i + 1
            swap(A[j], A[j-1])
```
Figure 7: AnimAlg program running the Bubble Sort algorithm

and is in its final place in the array. In addition to being able to choose the number of items to be sorted in the bubble sort algorithm, a status window is displayed in the control panel to indicate what two elements have just been swapped. The bubble sort algorithm is shown being animated in Figure 7.

3.3.3 Prim’s Algorithm

Before we describe Prim’s Algorithm, we will give a few definitions to support our discussion:

Definition: Let $G = (V, E)$ be a connected undirected graph. A subgraph $T$ is a spanning tree of $G$ if it is acyclic and connects all vertices of $G$.

Definition: Let $G = (V, E)$ be an undirected graph, and let $w : E \to \mathbb{R}^+$ be a positive weight function, then $T$ is a minimum-cost spanning tree (MST) of $G$ if $T$ is a spanning tree of $G$ and $w(T) = \sum_{e \in T} w(e)$ is minimum among
all spanning trees of $G$.

Idea – Prim’s algorithm is an algorithm that finds the MST of a graph $G = (V, E)$. It works by adding one vertex at a time to the tree until the MST contains all the vertices $V$ that are in $G$. The first vertex of the MST is selected at random from all vertices. Subsequent vertices are selected in a greedy fashion. First, the minimum-cost edge of all edges connected to any vertex already in the MST is selected and if adding that edge to the MST does not create a cycle in the MST, then the edge and the vertex that it connects to are added to the MST. As mentioned above, this process repeats until all vertices in $G$ are accounted for in the MST. The algorithm is given in Figure 8.

Implementation – In AnimAlg, this algorithm is implemented with unfilled blue circles representing vertices in the original graph. Blue lines with collocated numbers represent the edges in the original graph and their associated weights. The graph can be rearranged for ease of visibility by dragging a vertex to another part of the animation section of the screen with the mouse. A filled red circle represents a vertex that was added to the tree. A green edge represents an edge on the heap, and a red edge represents an edge in the final tree.

In the control panel, the user can choose the number of vertices and/or edges the graph should have. The end user can also generate a new graph, save the currently-displayed graph, or load a graph that had been previously saved by using the various tools in the control panel for this particular algorithm. The control panel also displays a status indicating the total weight of the minimum-cost spanning tree for this graph once the algorithm is done running. Prim’s MST algorithm is shown being animated in Figure 9.

3.3.4 Kruskal’s Algorithm

Idea – Unlike Prim’s MST algorithm where we grow the MST $T$ one vertex at a time, with Kruskal’s MST algorithm we grow the MST $T$ one edge at a time.
Prim(G)

\[ v \leftarrow \text{random vertex from } G \]

\[ T \leftarrow v \quad // \text{Add } v \text{ to the tree} \]

for each vertex adjacent to \( v \) such that \((v,w)\) is an edge with weight \( w \)

\[ H \leftarrow H + (v,w,w) \]

while \( \text{size}(G) > \text{size}(T) \)

\[ (x,y) \leftarrow \text{min}(H) \text{ where } \text{min}(H) \text{ is the edge in } H \text{ with the least weight} \]

if \( y \) is not already in \( T \)

\[ T \leftarrow T + y; \quad // \text{Add the vertex to the tree} \]

\[ T \leftarrow T + (x,y) \quad // \text{Add the edge to the tree} \]

for each vertex adjacent to \( y \) such that \((y,z)\) is an edge with weight \( z \)

if \( z \) is not already in \( T \)

\[ H \leftarrow H + (y,z,z) \]

**Figure 8:** Prim’s minimum-cost spanning tree algorithm
Figure 9: AnimAlg program running Prim’s MST algorithm
Kruskal(G)
for each vertex Vi in graph G
    put Vi in its own tree in the forest
for each edge <Vx, Vy> in G taken in nondecreasing
    order of weight
    if (Tx != Ty)
        MST = MST union <Vx, Vy>
        Tx = Tx union Ty

**Figure 10:** Kruskal’s minimum-cost spanning tree algorithm

We start by considering each vertex as its own tree and sorting the edges by weight. Then we consider each edge in nondecreasing order of weight to see if it can be added to the MST \( T \). If the edge under consideration joins two separate trees (i.e., it does not create a cycle within a tree), the edge can be added to the MST \( T \) and the two trees that the edge previously connected can be joined into one tree. Otherwise, the edge under consideration cannot be added to the tree and the algorithm moves on to evaluate the next edge. This process continues until each edge has been evaluated and either discarded or added to the MST \( T \). The algorithm is given in Figure 10.

**Implementation** – In AnimAlg, this algorithm is animated using blue lines to represent edges that haven’t yet been chosen to be in the minimum cost spanning tree (they’re still on the heap). Red edges represent edges that have been chosen for the minimum-cost spanning tree. Green edges represent edges that have been removed from the heap, but not chosen for the minimum cost spanning tree.

Like the sorting algorithms, much of the code used in Prim’s MST algorithm was reused in the Kruskal algorithm. Therefore, the graph can be
Figure 11: AnimAlg program running Kruskal’s MST algorithm

manipulated in the same way (by dragging a vertex to another location on the animation screen), and the same functionality exists on the control panel as for the Prim’s algorithm. It is an interesting exercise to generate a tree using one of the two MST algorithms, save the graph, and then reload it under the other algorithm to verify that the two algorithms do indeed generate the same cost for the minimum-cost spanning tree as indicated by the Status bar in the control panel. Krusal’s MST algorithm is shown being animated in Figure 11.
4 Usage Guide

In this section we show how to install and setup the AnimAlg program, how to use the program to learn about already-implemented algorithms and how to add another algorithm to the framework.

4.1 Program Installation and Setup

To bring up the AnimAlg program, follow these steps:

1. Download the code from http://cs.hbg.psu.edu/~gml100/proj and follow the instructions for installation.

2. Choose a directory to be “home” for the AnimAlg program, and extract the files to that directory. For the purposes of this document, we’ll call this directory the $ANIMHOME directory.

3. Define $ANIMHOME to be the place where the AnimAlg program’s main directory is installed.

4. Edit the master.properties file as appropriate in the $ANIMHOME/config directory with the path name previously chosen for $ANIMHOME.

5. Add $ANIMHOME/classes to the $CLASSPATH.

6. Bring up a command prompt and type java AnimAlg <filename> where <filename> is an optional argument and is the name of the desired algorithm’s .properties file.

4.2 Using AnimAlg to Learn about Algorithms

Once the program is up, it will prompt for an algorithm to be loaded if one hasn’t already been chosen on the command line or if one wasn’t specified in the master.properties file. After the algorithm is loaded, any settings given on the user panel can be altered for the algorithm (located in the bottom right-hand corner of the screen), otherwise the default settings will be used.
when the algorithm’s animation is started. While the animation is running, it can be paused (which allows the option of being resumed) or stopped (which does not allow the option of being resumed) using the appropriate button or file menu option. The speed at which an algorithm runs can also be varied as it’s executing by adjusting the speed control slider in the static part of the control panel (on the bottom left-hand corner of the screen).

4.3 Adding an Algorithm

In this section, we show how the intermediate user would add an algorithm to the framework, using the bubble sort algorithm as an example.

To add an algorithm to the AnimAlg program, do the following:

1. Choose a name for the algorithm that works well as a filename (e.g., bubblesort). Substitute that name in this section wherever you see the text “<algorithm>”.

2. Create a new “properties” file called `<algorithm>.properties` in the $ANIMHOME/config directory (e.g., bubblesort.properties) as shown in Figure 12. Note that though it appears that the `alg.explanation` text is on several lines in Figure 12, in reality it must be all on one line in the `.properties` file. Line breaks that should occur when the explanation is displayed on the interface can be indicated by placing a \n at the appropriate place in the explanation text. The fields that should be placed in the `<algorithm>.properties` file are as follows:

- **alg.title** the title that’s displayed in the title bar (e.g., “Bubble Sort Algorithm”).
- **alg.textfilename** the name of the file that contains the algorithm’s text (e.g., bubblesort.txt).
- **alg.implfilename** the name of the file that holds the main implementation of the algorithm (extends the GraphicAlg class) with or without the .class extension (e.g., BubbleSort or BubbleSort.class).
alg.title = Bubble sort algorithm
alg.textfilename = bubblesort.txt
alg.implfilename = BubbleSort
alg.explanation = The bubble sort algorithm sorts items by iterating (backwards) through the array of n items to be sorted n times starting at item n and going back to item i + 1 where i is the number of the current iteration. It compares each item to its neighbor and if its neighbor is smaller than itself, it "bubbles" the neighbor toward the beginning of the array by swapping the two items. In this way, the smallest item is "bubbled" to the first position on the first iteration, the second smallest item is "bubbled" to the second position on the second pass, and so on.

Figure 12: The bubblesort.properties file
Figure 13: The bubblesort.txt file

alg.explanation an explanation of how the algorithm works, its history or author, initialization parameters, etc.

3. Create a new text file that holds the algorithm’s text in the $ANIMHOME/algorithms directory. Figure 13 shows the bubble sort algorithm’s text file.

   - Give it the same name as specified by the alg.textfilename entry in the properties file (e.g., bubblesort.txt).
   - Specify a loop (for/while/do) in the algorithm by putting the token “<LOOP>” on a separate line immediately before the beginning of the loop (loops can be nested).
   - Specify the end of a loop in the algorithm by putting the token “<ENDLOOP>” on a separate line immediately following the end of the loop.
   - In order for the algorithm text to display properly on the screen, there should be no tabs in the algorithm’s text file. HINT: Create the text file with tabs for indenting and then use emac’s “untabify” utility to convert the tabs to spaces.

4. Create the implementation file in the $ANIMHOME/src directory using
/**
 * Classname: AlgImpl
 * Version information:
 * Date: 9/4/2002
 * Copyright notice:
 */

/**
 * This algorithm was implemented as an example algorithm for the Algorithm Animator program. It can be used
 * as a shell for users wanting to implement a new algorithm to be animated using the Algorithm Animator
 * program.
 * @author <author>
 */

public class AlgImpl extends GraphicAlg {

/**
 * All classes that extend GraphicAlg should have a no-argument constructor that first makes a call to
 * super(), does any initialization necessary and makes a call to <code>GraphicAlg.setUserPanel()</code>
 * with the user-defined I/O panel for this algorithm. If there is no initialization to be done, and no
 * specialized UserPanel to be created, then the implementer can skip the no argument constructor and just
 * rely on the default one.
 */

public AlgImpl() {
    super();
    setUserPanel(AlgImplPanel.getAlgImplPanel());
    // Add custom code here
} // AlgImpl()

/// *** See Figure 15 for the rest of the code that goes here***
}
} // end AlgImpl

Figure 14: AlgImpl.java.hide – Page 1

the file AlgImpl.java.hide as the shell. Figures 14 and 15 show the AlgImpl.java.hide file.

- Give it the same filename as specified in the .properties file by
  the filename (but not the same extension!) of the alg.implfilename
  entry. Instead of the .class extension that may or may not have
  been specified in the .properties file, this file must have a .java
  extension (e.g., BubbleSort.java).

- Must extend the GraphicAlg class (which is an abstract class)
  and implement the run() method.

- Mandatory no-argument constructor must contain a call to super()
  (this call must be on the first line of the constructor) followed by a
/**
 * Implements the <Algorithm> algorithm
 */
public void run() {
    warmUp(); // for i := 1 to n
    try {
        // This is where the main part of the animation goes. In here you can make calls to the following
        // methods within the framework:
        //
        // pause() - causes the algorithm to pause for the default number of milliseconds
        //
        // pause(float p) - causes the algorithm to pause for p times the default number of milliseconds.
        //
        // nextLine() - informs the framework that the next line of text should be highlighted
        //
        // nextLine(int s) - informs the framework that the next s - 1 lines of text should be
        // skipped in terms of highlighting and that the s-th next line should be
        // highlighted (this is useful where the algorithm has conditional statements)
        //
        // nextLoop() - informs the framework that this iteration of the current loop is now finished and
        // that the first line of the loop in the algorithm should be highlighted in order
        // to start the next iteration
        //
        // endLoop() - informs the framework that the currently executing loop has ended and that the line
        // of text immediately following the current loop should be highlighted
        //
        // repaint() - as needed to update the graphical display when a change is made.
        //
    } catch (StopRequestedException e) { }
    finally {
        coolDown();
    }
} // run()

/**
 * Paints the algorithm's animation onto the graphics object for the animation portion of the AnimPanel.
 * @param g The graphics object onto which this algorithm is animated
 */
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    // Add code to animate the algorithm here
} // paintComponent(Graphics g)

/**
 * Creates a String object to represent the contents of this
 * object for ease of output.
 * @return String object representing this <Algorithm> object
 */
public String toString() {
    StringBuffer sb = new StringBuffer(super.toString());
    return (sb.toString());
} // toString()
call to set the `myUserPanel` variable to the appropriate user panel (if one exists for this algorithm).

- The `run()` method must call the `warmUp()` method on the first line.

- The animation code should be put within a `try/catch/finally` clause in order for the Framework to be able to stop the algorithm’s animation when requested.

- The `run()` method must call the `coolDown()` method on last line (within the `finally` clause).

- The `run()` method can call the `pause()` method with or without a `double` parameter to have the graphic panel spend more time on a particular line of the algorithm.

- The `run()` method must call the `nextLine()` method when the next line of the text algorithm should be highlighted.

- The `run()` method must call the `nextLoop()` method at the end of the loop (before the loop’s closing brace).

- The `run()` method must call the `endLoop()` method after the loop (immediately after the loop’s closing brace).

5. Optionally, create a `<algorithm>Panel.java` which will be displayed in the bottom right corner of the screen and can display algorithm-specific output or obtain algorithm-specific input.

- Must inherit from the `UserPanel` class.

- Its methods can be called from the algorithm implementation classes in order to send output to the screen or retrieve input from the end user.
Conclusion

The Algorithm Animator program is a tool that can be used by computer science students hoping to gain better insights into the workings of basic algorithms. It is important because in addition to being a tool that can be used to help a student learn about particular algorithms, it is a framework to which other algorithms can easily be added. It differs from other programs that animate algorithms in that it is the only one that coordinates the algorithm’s animation with the highlighting of the appropriate line of the algorithm’s text. Its design and implementation have been presented in this paper.

Future work on the project could include:

1. Program additional algorithms to be implemented.
2. Improve upon the functionality of current algorithms.
3. Create an online help facility.
4. Time the algorithms – show how the time taken to run the algorithms corresponds with the timing expected by calculation with the big-Oh notation.
5. Add the ability to handle recursion.
6. Add the ability to “step” forwards or backwards through the algorithm.
7. Make it into an applet so that it can be used directly from the World Wide Web.

References

