

Graph Algorithm Visualisation Toolkit

GAVToolkit

Thomas E. Gorochowski

(0003746)

Supervisor: Dr. L. Goldberg

Year of Study: 2002-2003

Abstract

Graphs are becoming an important part of everyday life, allowing us to represent complex relationships. Many algorithms exist to analyse structures within graphs. Understanding how these algorithms work is essential for the development of new methods and teaching. Currently there is a lack of tools to allow for the visualisation of graph based algorithms.

This report looks at the steps taken in the creation of a toolkit, comprising an API (Application Programming Interface) and application that can be used to aid with the automatic visualisation of graph based algorithms. It will discuss the benefits that result and ask whether its form of visualisation is useful in understanding how algorithms work.

Keywords: Graph, Algorithm, Visualisation, Animation, Graphics, Learning

Contents

1	Introduction	4
1.1	Project Motivations	4
1.2	Project Description	6
1.3	Toolkit Structure	6
1.4	Report Structure	7
1.5	Author's Assessment of the Project	8
2	GAVToolkit API	10
2.1	Introduction	10
2.2	Analysis	11
2.2.1	Problem Analysis	11
2.2.2	Development Tools and Methods	14
2.3	Design	15
2.3.1	Overview	15
2.3.2	Data Storage	17

2.3.3	Control and Animation	26
2.3.4	Visual Output	37
3	GAVAlgorithmViewer Application	46
3.1	Introduction	46
3.2	Purpose and Features	47
3.3	Integration of the GAVToolkit API	48
3.3.1	The Basics	48
3.3.2	Advanced Features	51
4	Analysis of the GAVToolkit	53
4.1	Using the Toolkit	53
4.1.1	Algorithm Creation Overhead	54
4.2	Toolkit Output	57
4.2.1	Display of Graph Structure	57
4.2.2	Algorithm Animation and Interaction	60
4.2.3	Customisation	62
5	Conclusions	68
5.1	Conclusions	68
5.2	Personal Outcomes	70
6	Extensions	71

6.1	GAVToolkit API Extensions	72
6.2	GVAAlgorithmViewer Extensions	74
7	Further Information	76
7.1	Web Site	76
7.2	API Documentation	76
7.3	Using the Attached CD-ROM	77
A	Force-Directed Graph Layout Algorithms	79
A.1	The Spring Embedder [Eades]	80
A.2	Fruchterman and Reingold (FR)	82
A.3	Graph Embedder (GEM)	83

Chapter 1

Introduction

1.1 Project Motivations

Graphs allow for the representation of relationships between entities. They can be found everywhere and have numerous uses. A simple example is a road map, showing the relationship of roads between different locations. Many other examples exist, such as organisational charts (relationships amongst people) and 3D models (relationships between edges and vertices). As shown by these examples the diversity of application is vast. This makes them useful for many different types of situation, and is a major factor for their widespread use.

The structure of a graph is also exploited by computers. Graphs allow for the storage of data in a structured way that makes it quickly accessible. The structure can also hold additional information, for example, in a binary tree all elements in

the left sub tree are smaller than their root.

With their extensive use, many algorithms exist to analyse internal structures of graphs, leading to further useful information being found. Dijkstra's single source shortest path algorithm finds the shortest route from one node to all other reachable nodes in a graph. This information could then be used for route planning.

Understanding how these algorithms work is important as it leads to optimisations and helps with teaching. Algorithms are generally written in program code which can be difficult to understand and does not allow for a quick mental model to be built of the workings. It is generally accepted that by representing the workings of an algorithm in a graphical form, understanding is gained more easily. One reason for this is that interactions with visual objects are more closely related to our physical world.

Some projects do exist to visualise the workings of algorithms, however, they concentrate on the lower-level structures, e.g. variables and arrays. One such project is Jeliot [1]. Developed as research, it shows the interactions of variables and other simple data structures in a graphical way. Programs are submitted in a cut down version of Java and results can be transmitted across the web. It has been found to be a useful analysis tool and teaching aid.

Graphs exist at a higher level of abstraction and need to be represented in a different way. There is a severe lack in tools to visualise graph based algorithms,

this being the main motivation behind the project. The idea was to create a toolkit that would allow the workings of coded algorithms to be displayed graphically. The visualisation would be fully automatic, as well as allowing for customisation by the user.

The creation of such a toolkit would be beneficial to algorithm writers by allowing them to find problems with designs more easily and could also be used to help with teaching by allowing students to see the workings of their own, and others' creations.

1.2 Project Description

The creation of an easy to use toolkit to automatically visualise the workings of graph based algorithms. The toolkit must be flexible allowing the user to work the way they want and due to the great diversity of graphs, customisable to fit altering needs. Visualisation should be in the form of an animation that presents the algorithm in an understandable form.

1.3 Toolkit Structure

The toolkit comprises of two parts, an API (Application Programming Interface), and an application. The API performs the visualisation and is open to use by other

developers in their projects. The application acts as a wrapper¹ allowing a user to access features of the visualisation through a GUI (Graphical User Interface).

1.4 Report Structure

The report is written in such a way that the sections will generally follow on from each other. There are four main sections.

1. **GAVToolkit API** - The development of the API will be shown and its workings discussed.
2. **GAVAlgorithmViewer** - Looks at the development of an application that uses the API. It shows how the API should be used and includes information about how to customise the visualisation process.
3. **Analysis of the GAVToolkit** - Shows some actual results produced by the toolkit. The quality of the output and difficulty in writing algorithms is analysed.
4. **Conclusions and Extensions** - The outcomes of the project are detailed and further additions to the project are considered.

¹A wrapper changes the appearance to the programmer of an underlying API it uses.

1.5 Author's Assessment of the Project

The content of this report is highly varied containing theoretical components that look at the efficiency of graph data types to details about the software development. UML has been used where appropriate to communicate the workings, however, its use has been relatively small. Understanding the concepts behind how the toolkit works is more important than understanding all implementation details.

Several problems were encountered throughout the development of the toolkit. Many related to producing an animated output that would be created automatically which required the use of multi-threading aspects of Java had and Java 2D API for graphical output. The biggest problem, however, occurred with the loading of class files. These come in a compiled form and an understanding of the Java class loading mechanism was required to allow for their dynamic loading at runtime.

This report would be useful to a variety of different people. I, personally, see the greatest benefit to those interested in algorithm animation and visualisation. It looks at a possible visualisation technique and shows its implementation in Java. In addition it could also be useful to those involved in the development of open frameworks that involve graphical components.

The one weakness of the project is the lack of extensions to the toolkit due to limited time. A greater number of displays, such as 3D or text, would have helped show more of the possibilities available. Further additions could have

been created, although this would have been at the cost of the stability. A trade off had to be made and it was thought better to have a working product that included some, if limited extensions, rather than one that didn't work at all.

Even so, in context of the aims of the project it has been a success. Not only have a useable application and API been produced, but it has incorporated different aspects of computer science and shown a practical use for aspects of the subjects theory. This has lead to and interesting project that opens the doors for many developers and gives the option to customise the toolkit for various applications or to use it as an aid in development or teaching of algorithms.

Chapter 2

GAVToolkit API

2.1 Introduction

The GAVToolkit API is a framework for the automatic visualisation of graph based algorithms. Rather than a closed application that cannot be altered, the API allows for users to customise and alter aspects to fit to their varying needs. The framework performs all visualisation automatically so that a user can concentrate on the visual representation of the data, or the development of a custom application. The API allows graph algorithm visualisation can be added to any application with little effort.

The API includes several standard extensions to allow for basic 2D graphical output. Further extensions can be written in Java using the defined interfaces and abstract classes.

This chapter looks at the development of the API giving reasons for design decisions made and outlines how the visualisation works.

2.2 Analysis

Before designing the system it was important to gain a good understanding of the problem and the tools that were available to use. This section performs an indepth analysis of the problem and looks at the tools chosen for development.

2.2.1 Problem Analysis

The aim of the API is to automatically visualise a graph based algorithm. Visualisations can take many forms. The program code of an algorithm is a static and textual view that shows the exact workings. Code alone is difficult to understand as the flow of execution is not displayed and can involve jumps to different sections. Instead an alternate visualisation would be to look at the order in which lines of code are executed. This would help a user see the steps taken from an initial to final state. Although these methods may help in some situations, with graph algorithms an understanding can be gained by simply viewing the interactions an algorithm makes with an input graph. This is due to most algorithms using the structure of an input for conditional aspects of their code. By building a mental model relating the structure of an input to the interactions made, a partial

understanding of the algorithms workings can be gained.

With the structure of an input graph important to the visualisation, a way of displaying it clearly was required. Humans generally find it easier to extract information about the structure of an object if it is displayed in a graphical form. Graphs can be represented in this way by converting nodes to points and edges to lines between points. As nodes and edges can have different locations and paths, it is important that the layout of the structure can be changed to fit the particular input graph. Showing the interactions with this visual structure allows for a quicker understanding of an algorithm to be gained as the structure of the graph can be seen more easily. To help further, having the option to view multiple differing displays could lead insight not discovered with a single representation.

This argument is not shared by all. It has been shown that graphical representations can both help and hinder understanding[6]. An issue raised by many is that the output an algorithm produces hides much of the complexities that are necessary in understanding the full concepts used. This is true in many cases, however, if a user understands the steps involved in producing a solution, then code will become immediately more understandable. The user has already built a mental model of the steps taken and can use this to break down conditional aspects of an algorithms code. It is incredibly difficult to gain a full understanding without looking at actual code, however, a graphical representation can be assimilated much quicker, lets a user build a mental models of the steps involved, and

lifts language barriers so that the knowledge is accessible to everyone.

Knowing the form of visualisation, a way of producing the it automatically was required. Interactions that an algorithm makes could be analysed by the creation of a custom data type. Acting like a normal graph, the data type would still store a graph representation, however, also collect information about the changes that have occurred to its structure. These changes could then be used to show how the structure has been effected over time by the algorithm.

It was thought that interactions needed to be classified to gain an understanding of there purpose. Many algorithms such as Dijkstra's single source shortest path assign properties to nodes and edges in a graph. In Dijkstra's algorithm distances are assigned to nodes that are calculated during the running of the algorithm. Having the facility of properties in the graph data type could not only help algorithm writers, but also be used when displaying a visualisation. The properties of an node could alter the way it is displayed showing a viewer the current properties of an object.

By combining the interactions analysed by the data type and the properties of objects in a graph, a large amount of information can be collated and displayed with no interaction directly from an algorithm. This allows for a fully automatic visualisation that can be displayed in the form of an animation.

2.2.2 Development Tools and Methods

Java was chosen as the programming language for development because it enabled an object orientated programming approach, suitable for a modular design. Another key aspect of Java is that it includes support for graphics and multi-threading natively. If a language such as C or C++ had been used, additional libraries or operating system calls would have had to be made to use these features increasing development time. Java is also a well established language and highly portable with versions for all major platforms, allowing the toolkit to run virtually anywhere with little, if any, change.

To simplify development the same programming language was also chosen for algorithms. Other projects have taken a different approach. For example Jeliot[1] uses a cut down version of Java to program its algorithms. This code is converted into standard Java that can then be used by the JVM (Java Virtual Machine). Processing the algorithm code allows for an analysis of the workings to help make decisions about the visualisation process. This has the side effect of cutting out many of the features algorithm writers may want to use. Giving the programmer the full Java language makes the toolkit more flexible and allow for previously coded algorithms to be transferred across more easily. A major advantage though is that an existing tool (Java) could be utilised rather than having to create parsers and compilers for a new language.

Throughout the design of the toolkit the idea of an *open-closed* development style was employed. Used widely in the development of Java itself, it allows for features of an underlying API to be altered by the creation of extensions. The extensions are not a requirement imposed but allow for aspects of the design to be customised for the users needs. An API using this idea becomes a set of classes and interfaces that define the communication between sub-systems. These interfaces allow for extensions to be written (*open* to change), however, these do not alter the underlying API code (*closed* to change).

The UML modelling language was used to communicate the structure and workings of the toolkit. Being the standard language for developing software systems, it helped make the designs more widely understood.

2.3 Design

This section looks at the design and development of the API. It shows how the API works internally and discusses important aspects of the design.

2.3.1 Overview

The API comprises of three main sub-systems that together produce a visualisation. The separation into sub-systems allowed for each to be developed and tested in isolation. An overview of the API can be found in Figure 2.1.

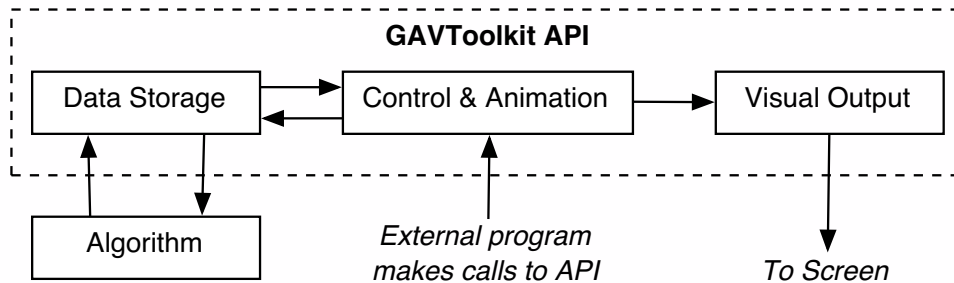


Figure 2.1: Overview of API sub-systems

Each sub-system performs a unique part of the visualisation. The separate tasks are coordinated by communication among the sub-systems which is defined by using interfaces. Each sub-system has the following general functions:

- **Data Storage** - Stores a representation of a graph that allows for properties to be stored for individual nodes and edges. Algorithms directly interact with this sub-system by making changes to its structure. These changes are signalled to the control and animation sub-system so that the display can be updated.
- **Control and Animation** - Handles all external and internal events. Users directly use this to manipulate features of the API. It can load graphs, algorithms and alter the way the visualisation is shown. Internally it controls updates to the display so that the visualisation takes the form of an animation of changes made by an algorithm.

- **Visual Output** - Produces a graphical output of the current graph structure.

It has the ability to alter the way nodes and edges are drawn by inspecting their individual properties. The output produced can be drawn to GUI components that then become automatically updated by the API.

2.3.2 Data Storage

The data storage sub-system plays a key role throughout the visualisation process, providing a data type to be used by algorithms. This is the only component of the toolkit that an algorithm directly interact with. It stores the currently loaded graph, can hold properties for individual nodes or edges, and detects when a change is made to its structure so that the control and animation sub-system can be informed to update the display.

The graph data type only stores directed graphs to simplify the design, however, this does not prove a problem for undirected graphs as methods have been included to add and delete undirected edges.

Analysis of Graph Data Types

Many different ways of representing a graph exist. One of the simplest is an adjacency matrix which can be stored as a two dimensional array. For a graph with N nodes the adjacency matrix is of size, $N \times N$. Rows in the matrix represent the starting a starting node, and columns an end node. If the value 1 is found at

a location then an edge exists between those two nodes, otherwise no such edge exists in the graph. Figure 2.2 shows the adjacency matrix representation of a directed graph.

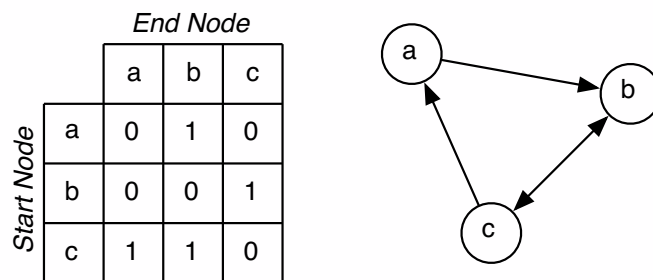


Figure 2.2: Adjacency matrix representation of a directed graph

The adjacency matrix is represented as an array in memory, making search times fast, of $O(1)$. The major drawback the design suffers from is that the required space to store a graph is large, N^2 . Most used graphs are sparse, having very few edges, however, because the adjacency matrix stores data on every possible edge the majority of entries in the matrix are for non-edges. The example in Figure 2.2 holds zero values in over 50% of the matrix.

Improvements can be made to this method if an undirected graph is represented. Due to the fact that in this type of graph an edge from $a \rightarrow b$ implies another edge from $b \rightarrow a$, the adjacency matrix becomes symmetrical along its diagonal, shown in Figure 2.3. Rather than storing both sides of the diagonal, half can be discarded with no loss of useful information.

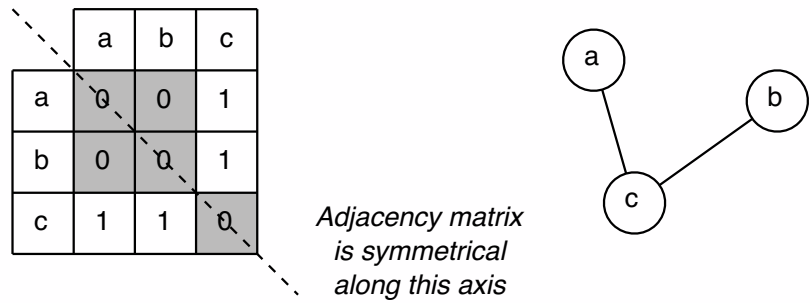


Figure 2.3: Symmetrical adjacency matrix for an undirected graph.

The biggest problem with this method is that the array used for storage is not a dynamic data structure. When nodes are added or deleted the array cannot simply be resized. Instead a new array of a larger or smaller size has to be created, and previous values copied across. The time to add a node is $O(N^2)$, where N is new number of nodes, as every value in the matrix has to be copied across and empty locations have to be set to zero. If node numbers are highly dynamic then the adjacency matrix method is not suitable.

Another structure exist to solve this problem which uses linked lists rather than arrays to store the graph data. A linked list is a dynamic data structure that can be resized quickly and efficiently. Data is stored as a sequence of elements and a pointer to the first item in the list is maintained. Each item in the list stores an element of data and a pointer to the next item. The end of a list is denoted by using a null pointer. Items can be added and deleted from the list by the manipulation pointers that is far more efficient than the resizing of an array. To search a linked

list each item is compared until the required one is found. This can make search times slow, and in the worst case for a linked list with N items, all will have to be compared giving $O(N)$ time.

A linked list only stores data in one dimension. Therefore to represent a graph multiple linked lists are used. The main list in the structure called the nodes list, holds every node that exists in the graph. Each items data in this list holds the node it represents, and a pointer to an additional list, called the edge list. Edges in the graph are represented by the existence of a node in an edge list. To find out if an edge exists the start node of the edge is found in the nodes list. Once found, the pointer to the edge list is followed and the end node of the edge is searched for. If the node exists in the list then so to does the edge, otherwise it is not present in the graph. Figure 2.4 shows a graph representation using this linked list method.

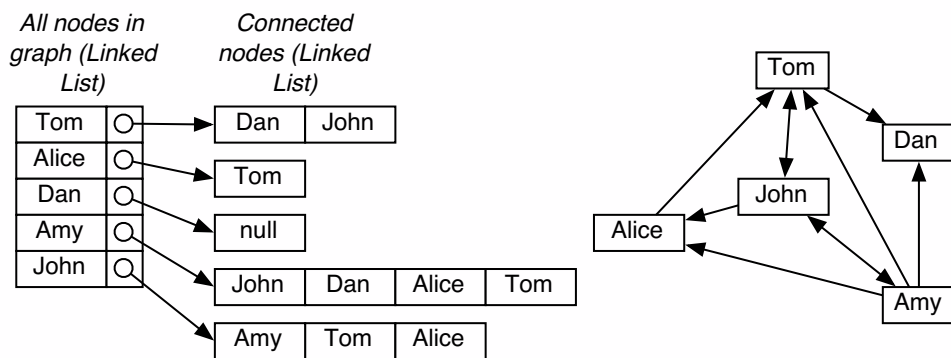


Figure 2.4: Multiple linked list representation of a graph

Space required for this method is minimised by only storing required informa-

tion, unlike the adjacency matrix . The space required for a graph with N nodes and E edges is $O(N + E)$ because an entry exists for each node and edge in the graph. In the worst case an edge would exist between every node making $E = N^2$ and giving $O(N^2)$ space, however, as most graphs are sparse $E < N^2$ for most cases.

Addition of a node takes $O(1)$ time, as a new node can be added to the start of the nodes list. Node deletion time is more costly as every reference to the node has to be found and deleted. In worst case the graph would contain every possible edge, and the node to be deleted would occur at the end of every list in the graph. This situation would give a node deletion time of $O(N^2 + N) \Rightarrow O(N^2)$, however, under general conditions node deletion time would be $< O(N^2)$.

Searching for an the existence of an edge in a graph could in worst case require searching the full node list and every edge list. This gives a total search time of $O(N^2)$. In comparison to the adjacency matrix with a search time of $O(1)$, this is much worse, however, as most graphs are sparse the size of the edge lists to be searched would be much less than N .

Under real world conditions the linked list approach gives good overall results.

Data Type Design

When designing the data type to be used by the API it was unknown what factors would be imposed on the design. Some algorithms may require a fast search speed

while others may add and delete large numbers of nodes. As shown in the previous section it is difficult to get the best of both worlds.

For most real world uses the linked list method was found to give the best overall results. The problem with this method is that search speeds were much slower than the adjacency matrix. By adapting the method to use Java's `Vector` object, the search speed could be increased while still maintaining the low space and dynamic features.

A vector in Java is a semi-dynamic data structure that stores objects in an indexed form, much like an array. Items can be added, or deleted at arbitrary positions in the list, causing following items to be shifted along. They have a capacity, which is the maximum number of items that can be stored without the need for reallocation. Once the capacity is reached, reallocation is performed, and a new capacity set. A new capacity is normally to twice the vector's previous size. This minimises the amount of reallocation that is performed, which is important as it takes significant processing. Both initial capacities and new capacity policies can be set by the user when a vector is created. The benefits a vector provides is that both searching and resizing without reallocation can be performed in $O(1)$ time. Using a vector instead of linked lists allowed for search times to be improved, while maintaining the dynamic features of the linked list.

Additional demands were also placed on the design and the analysis of the problem lead to the need for properties to be stored for individual nodes and edges.

The standard list representation of a graph could only hold information about nodes. Edges were not unique objects but instead, their existence was implied by the graphs structure. To allow for properties to be stored, nodes and edges became separate objects, shown in Figure 2.5. Nodes store a vector of out edges and a pointer to their properties, while edges hold a pointer to their start node, end node, and properties. Having the properties for nodes and edges as a pointer to an external object allows custom properties to be defined and used by altering the pointers value. Figure 2.6 shows the adapted data type that uses the node and edge objects.

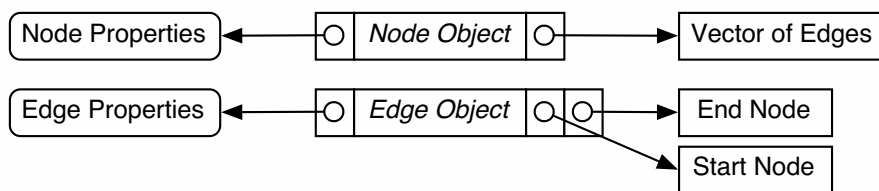


Figure 2.5: Node and Edge Objects

As well as storing a graphs structure the data type is also used by algorithms to automatically interact with the API. Interactions are in the form of changes to the graph and two types can occur:

1. **Structural Alteration** - This would result if an edge or node were added or deleted. The structure of the graph would have been altered and the display would need to be updated.

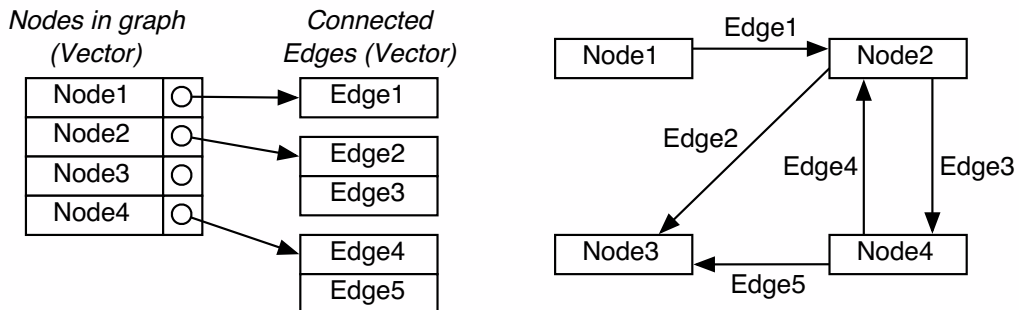


Figure 2.6: Overview of the data type used in the API

2. **Property Change** - The properties of nodes or edges in the graph have altered, however, the structure of the graph would remain the same. An update would still be required.

It is not the job of the data type to update the displays. Instead this task is delegated to the control and animation sub-system, which is signalled about the type of change.

Data Type Implementation

The data type is implemented using five classes, shown in Figure 2.7.

The `GAVGraphData` class is the central entity a stores all nodes in the graph. It also contains methods to manipulate the graphs structure by adding or deleting nodes and edges. All graphs are stored in a directed manner, however, this class includes methods relating to undirected edges.

`GAVNode` and `GAVEdge` classes hold a representation of a node and an edge.

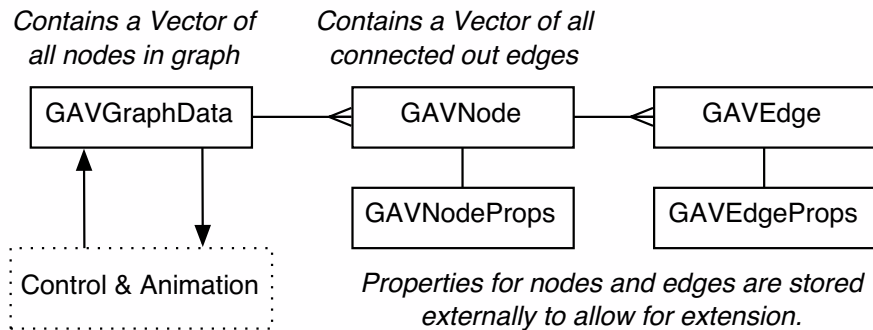


Figure 2.7: Class diagram of the API data type

For nodes methods exist to get a vector of out edges and properties. Edges have methods to retrieve start node, edge node, and properties.

Properties for nodes and edges are contained within the `GAVNodeProps` and `GAVEdgeProps` classes. These classes have a number of standard members such as, label, colour, selected, and visited. Properties cannot be accessed directly but instead have to be set through method calls. The reason for this is so that the method can signal to the API that a property change has occurred. Allowing direct access would not let the changes be monitored.

Only the bare minimum functions were included in the data type. This helped make the sub-system more easily understandable and maintainable. So that users could still have access to more complex functions, a utility class was created called `GAVUtils`, which can be found in the `GAVToolkit.utils` package. Some of the methods it includes can find opposite edges, adjacent nodes, and can search

for cycles. A full listing of methods can be found in the documentation. All methods in the class are defined static so that they can be called without the need to create an instance of the class.

2.3.3 Control and Animation

The control and animation sub system handles all interactions with the user, processes internal events, and performs the animation of algorithms. Figure 2.8 shows the internal design. From this it can be seen that the sub-system is based around the `GAVGraph` class. The advantage of this design is that users wanting to integrate the API into their application only require the understanding of a single class.

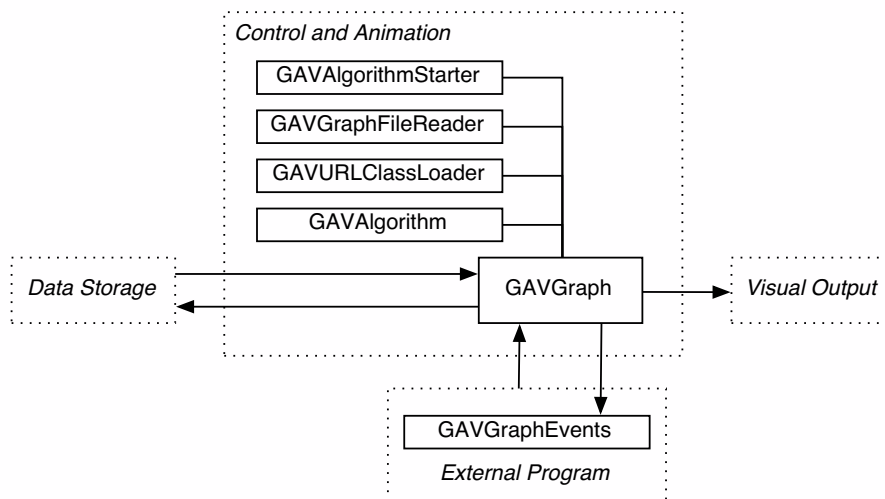


Figure 2.8: Control and Animation Sub-system Overview

Many functions are performed by the sub-system and integrating them all into one class would have complicated the implementation. To help simplify this the `GAVGraph` class was designed to perform coordination between the other sub-systems but use utility classes to carry out many of its own functions.

Algorithms (`GAVAlgorithm`)

One of the most important tasks for the sub-system is to handle the loading and animation of algorithms. All algorithms written for the toolkit extend the `GAVAlgorithm` class. The class acts as an interface so that every algorithm can be used in the same way.

Several methods exist in the `GAVAlgorithm` class that should be implemented by an algorithm.

- **`run ()`** - This contains the code for the algorithm. It is called by the API when the algorithm is played.
- **`cleanUp ()`** - If resources are allocated during the running of the algorithm they should be freed by this method. When the algorithm ends or is reset, the method is invoked.
- **`getName ()`** - Returns a `String` containing the name of the algorithm. This information can be used by applications using the toolkit.

- **getDescription()** - Returns a `String` containing a description of the algorithm. Again this can be used by applications using the toolkit.

In addition to acting as an interface for algorithms, the `GAVAlgorithm` class is also used by the API to help with the resetting of an animation. This will be discussed in a later section.

The code of an algorithm is not directly used by the API. Only compiled class files can be loaded. Classes contained within a compiled file cannot directly be used by a Java application. Therefore, it was necessary for the API to include a dynamic class loading mechanism that could be used at runtime. This would allow the class definitions contained within the compiled files to be extracted.

Dynamic Class Loading (`GAVURLClassLoader`)

All class loading in Java is carried out by the JVM (Java Virtual Machine). The JVM is used to run all Java programs but is generally hidden from the programmer. Dynamic class loading is possible through the use of standard Java API objects that communicate directly with the JVM.

The JVM loads classes using what are known as, class loaders. These encapsulate the class loading aspects of the JVM. During the creation of Java, it was unknown the form in which classes may need to be loaded. By default a, boot strap loader, is integrated into the JVM that will load a class from an array of

bytes. It searches for files contained within the `CLASSPATH`¹ environment variable and if the a corresponding class file is not found then the loading fails and an exception is raised.

Classes that need to be loaded may not be located within the standard path or may be present over network links. These situations are catered for by using additional class loaders. Multiple class loaders are possible in a JVM and they work together through a delegation technique. When a new class loader is created it is given a parent. Generally this will be the boot strap loader, however, a chain of class loaders can be built. When a request is made to load a class, the message is first sent to the the class loaders parent, before attempting to load the class itself. This checks to make sure that the class cannot be found in a standard location.

The Java API includes a `ClassLoader` abstract class. This can be used to write additional class loaders. It could not be assumed that the location of every algorithm loaded into the toolkit would be contained within the `CLASSPATH`. Therefore, an additional class loader was created, the `GAVURLClassLoader`. This class loader extends Java's `URLClassLoader`, a subclass of the `ClassLoader`. It allows for algorithms to be loaded from any URL (Universal Resource Location), hides the complexities of the loading process, and creates a `Class` object from the file. A URL can specify files on a local disk, across a local network, or

¹This variable is present on both Windows and UNIX implementations of Java. However, other implementations may represent the standard path for class files in a different way.

even on the internet.

Algorithm Animation

Once an algorithm has been loaded into the API it must then be animated. An animation is created by showing a sequence of images (frames) in succession with a small delay between each. Every frame in an animation should show a small change from a previous frame. When animating an algorithm the frames become static images of the data type at a point in time. As an algorithm runs the data type will be manipulated causing change. By creating a frame for every change that is made, an animation of the manipulations an algorithm makes through time can be seen. This is how animation in the API is created.

Manipulations to the data type cause the data storage sub-system to signal that a change has occurred. These signals are used to set the point at which a new frame is created. So that the animation is visible to the naked eye a delay is added so that the updates are not too fast. The speed of the animation can be altered by increasing or decreasing this delay.

A multi-threaded design was used to allow for this type of animation. Two main threads exist in the system, the algorithm and `GAVGraph` object. The algorithm thread executes the code of the current algorithm and the `GAVGraph` thread acts as a clock, continually cycling. When an update is required the algorithm stops executing and waits until the `GAVGraph` thread reaches the end of its

cycle. Once both threads are in synch the algorithm continues.

Synchronisation of the threads is performed using Java's methodology of notifiable objects. These are initiated by calling the `wait()`, `notify()`, and `notifyAll()` methods of an `Object`. When the `wait()` method is invoked the calling thread stops running and waits until a `notify()` or `notifyAll()` call is made by another thread to the same object. At the point where a notify call is made both threads become synchronised. This synchronisation technique can be seen in Figure 2.9.

The toolkit implements a `Waiter` class which includes the `wait` and `notify` calls, `waitOn()` and `signal()`, to perform all synchronisation.

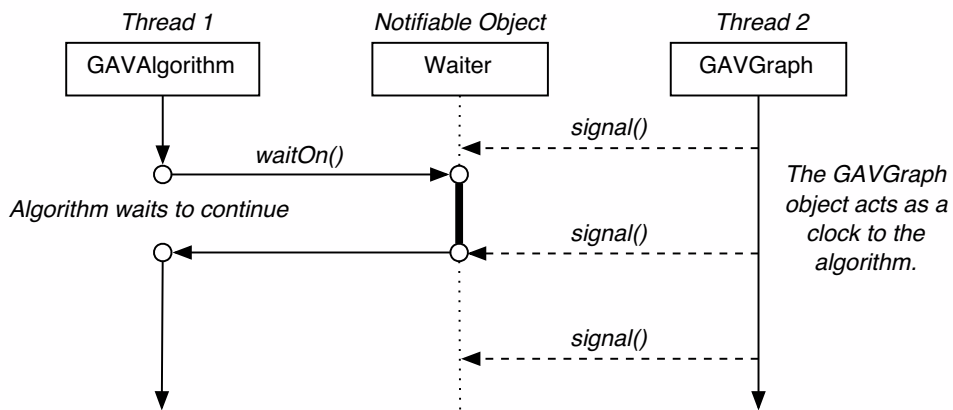


Figure 2.9: Use of Notifiable Objects for Animation

When an algorithm is in a paused state execution of the algorithm thread stops. Rather than letting the `GAVGraph` thread continually cycle, wasting processing

resources, it is stopped using a `Waiter` object.

Running an Algorithm (`GAVALgorithmStarter`)

The previous section has shown that all algorithms run in a separate thread. To create a new thread for an algorithm the `GAVALgorithmStarter` class is used. Given an algorithm it will create a new `Thread` object and assign the code from the algorithm to it.

This is possible due to the `GAVALgorithm` abstract class implementing Java's `Runnable` interface. The interface allows for classes that do not extend `Thread` to be run concurrently. It lets the code from a `Runnable` class be assigned to an existing `Thread` object. Running this thread will then cause the algorithm to execute concurrently.

Resetting an Algorithm

Whilst running, the API gives the option to reset the algorithm back to its start. For an algorithm that has ended this is achieved by running its thread again. A problem arises when a partially complete algorithm needs to be reset. A thread cannot be restarted until it has safely finished running. Under normal conditions this would involve waiting for the algorithm to fully end. As some algorithms may run for long periods of time or not end at all, this was not acceptable.

It was known that an algorithm would make interactions with the graph data

type throughout its running. This was used to raise an `Exception` when the algorithm was required to be reset. An `Exception` is a special type of object in Java that is normally used to signal an error. It can, however, be used to stop the current sequence of execution and move it to an exception handler. In the case of resetting an algorithm the exception handler unallocates any resources used by the algorithm and exits the thread safely.

This is implemented in the API using a standard `NullPointerException`. When a reset is signalled a flag is set in the `GAVGraph` object. As updates occur to the graph data type, checks are made see if the reset flag has been set. If so, a `NullPointerException` is raised. The exception is handled by the `GAVAlgorithm` object. This calls the `cleanUp()` method for the algorithm to unallocate any resources and then ends the thread safely. Program 2.3.1 shows pseudo code for the exception handler.

Program 2.3.1 Pseudo Code for `GAVAlgorithm` Exception Handler

```
Thread starts running
  try{
    call run() method of algorithm
  }
  catch(NullPointerException){
    call cleanUp() method of algorithm
  }
Thread ends safely
```

All algorithm resetting is transparent to the algorithm writer and included into all algorithms through the `GAVAlgorithm` abstract class.

Updating Displays

When a change is made to the data type the displays will require updating. The API allows for multiple displays so that a problem can be viewed from several angles. These displays are stored in a `Vector` in the `GAVGraph` class. When an update is required each display in turn is signalled to redraw. Once all are updated an algorithm can continue running.

Loading Native Graph Files (`GAVGraphFileReader`)

The API allows for graphs to be stored in a basic text file. These files can be loaded and then used as input for an algorithm. The file format allows for nodes and edges to be defined, edge labels, and comments. An example of the file format can be seen in Figure 2.10.

A graph file is made up of a number of objects that are contained by the `<` and `>` control characters. These objects can define the nodes and edges in the graph. The statements `<{NODES}>` and `<{EDGES}>` signals the start of a nodes or edges section.

Graph files are loaded using the `GAVGraphFileReader` class. This contains a simple parser that builds a graph data type representation from a file. It

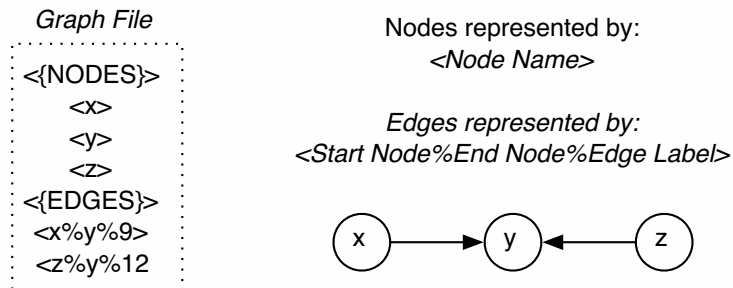


Figure 2.10: Graph file format

works by searching for control characters to find objects in the file. These objects are then converted into calls to add nodes and edges to the data type. Given the File of a graph a GAVGraphData object is returned.

Graphs are loaded into the API by using the `openGraphFromFile(aFile)` call on a GAVGraph object. This method uses the GAVGraphFileReader to perform the request.

External Communications

Another important role of the control and animation sub-system is to handle external communication with the user. A user can talk to the toolkit through the GAVGraph class. This contains a number of methods that a user can call to alter the API set up or effect the visualisation. A full listing of methods can be found by reading the javadoc documentation, however, the most important methods are:

- **play ()** - Plays the current algorithm.

- **step ()** - Step one action forward in the animation.
- **pause ()** - Pauses the current algorithm.
- **reset ()** - Resets the algorithm to the start.
- **addDisplay ()** - Adds a given display to the visualisation
- **redrawDisplay ()** - Updates all displays (no restructuring)
- **restructureDisplay ()** - Restructures (creates new layout) and then redraws all displays.
- **setSpeed (aSpeed)** - Sets the current speed of the animation.
- **openGraphFromFile (aFile)** - Opens a graph from a given file. This causes any loaded algorithm to be reset.
- **openAlgorithmFromFile (aFile)** - Opens an algorithm from a given file. Causes any loaded or running algorithm to be destroyed and then reset.

Communication to External Programs (GAVGraphEvents)

Some of the methods described in the previous section alter the running state of the API. For example, if the `play ()` method is called then the API will move to a running state. There are four states that the API can be in.

1. **Unplayable** - This would occur if no graph or algorithm was loaded.

2. **Paused** - When an algorithm has been running but it paused.
3. **Playing** - An algorithm is currently running.
4. **At End** - The algorithm has finished running and is at the end.

The current state of the API is useful to an application because many will provide playback controls. The state of these controls is important as it may be inappropriate to allow a play button to be visible if an algorithm is running.

To allow external applications to be informed when a state change occurs the `GAVGraphEvents` interface was created. The interface contains a single method, `setMode(aMode)` where `aMode` is the new state of the API. Applications implementing this interface can then be informed of updates by calling the `setEventHandler(aHandler)` method in the `GAVGraph` class, where `aHandler` is a reference to the application.

2.3.4 Visual Output

The visual output sub-system creates static visualisations of the data type that act as frames in the animation of an algorithm. It became clear from the problem analysis that the form of visualisation was vital to understanding the workings of an algorithm, making it important for the structure of the input to be clear.

Even though many different visualisation may be produced, the steps involved remain the same.

A layout is produced to show the structure of the graph, setting locations for nodes and edges. Using this information nodes and edges are drawn with their appearance set by their properties to show the interactions an algorithm is having.

Both of these steps are not always required to produce an output as previous results can sometimes be used. When the sub-system is signaled to create a visualisation the type of change that has occurred is also sent. Two types of change exist, structural and property. In both cases the graph has to be drawn, however, a new layout is only required after a structural change. Figure 2.11 shows a flow diagram of the steps involved to create a visualisation.

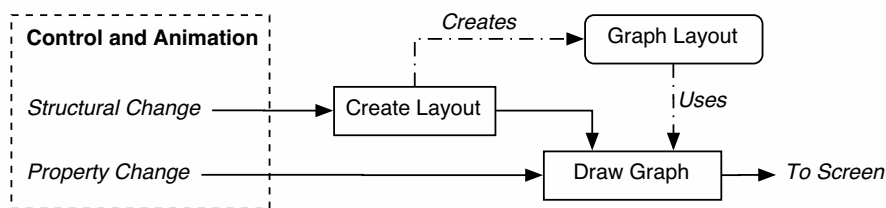


Figure 2.11: Flow diagram of the update procedure

Minimising the number of layouts that are produced is important for two reasons. First, a layout takes a high level of computation to complete as no efficient² methods exist to produce good layouts. Secondly, a newly created layout may differ from a previous one causing any knowledge discovered about the structure to be lost.

²Efficiency in this case is defined as not being able to be performed in polynomial time.

It was important when designing this sub-system to allow a user to fully customise all aspects. This is due to the diversity of graphs and algorithms that the API may be asked to display. To make customisation easier the sub-system was split into two layers, one defining the steps involved to create a visualisation, and the other to implement each of the steps. This way by using a different implementation of a step would allow an alternate visualisation to be produced.

Layer 1: Process Definition

This layer defines the steps involved to create a visualisation. Individual tasks were split into separate classes and interfaces were used to define the communication between them. All classes in this layer have no implementation of the task that they represent, instead they simply exist to coordinate the steps of the visualisation. Figure 2.12 shows the classes that make up this layer.

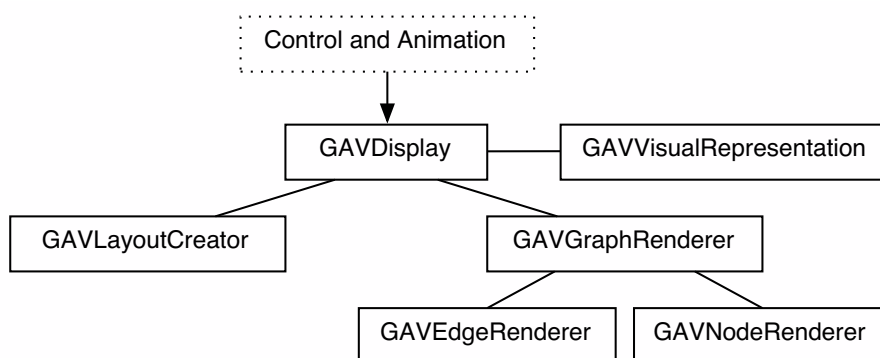


Figure 2.12: Layer 1 - Process Definition - Class Diagram

The `GAVDisplay` class represents a display in the system and is required to handle calls from the the control and animation sub-system. Layout creation is the task of a `GAVLayoutCreator` which will, given a `GAVGraphData` object produce a layout in the form of a `GAVVisualRepresentation` object. This layout is stored by the `GAVDisplay`. When the graph needs to be redrawn, the current layout and a `Graphics` object (to draw on), is sent to the `GAVGraphRenderer`. This in turn uses the `GAVNodeRenderer` and `GAVEdgeRenderer` classes to draw individual nodes and edges to the given `Graphics` object.

Layer 2: Visualisation Implementation

This layer contains specific implementations to produce an actual visualisation. Classes in layer 1 are extended so that the communication aspects of the design get carried through. Figure 2.13 shows the classes implemented in this layer with their location corresponding to the classes they extend in Figure 2.12.

The `GeneralDisplay` class is implemented as an extended `JPanel`. This allows the display to have access to a `Graphics` object to draw to the screen, but also allows the display to act as a standard GUI component.

Layouts are implemented in the form of a `GeneralVisualRepresentation` which stores the location of nodes and edges in two lists. Each list contains tuples of the form $(Node/Edge, Location/Path)$ that can be searched to find the

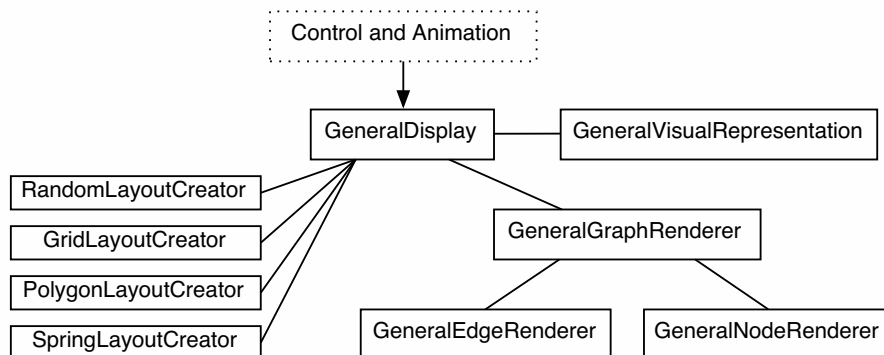


Figure 2.13: Layer 2 - Visualisation Implementation - Class Diagram

location or path of a particular node or edge.

The most complex part of this layer is the creation of a layout and the drawing of this to the screen. Both had to be shown in a clear way to give the greatest chance of understanding.

Creating a Graph Layout (**GAVLayoutCreator**)

The graph data type in the API only contains information about relations among nodes and properties nodes and edges contain. None of these give any indication to the locations that nodes should be placed or the paths that edges should take. It is the job of layout creators implementing the `GAVLayoutCreator` interface to produce locations and paths of nodes and edges for a given graph structure.

Graphs come in many different forms and all contain internal structures such as cycles. It is important that a layout of a graph shows this because it will make

the structure more understandable and pleasing to the eye. For small graphs it seem simple to produce a good layout, however, as the number of nodes and edges increases it becomes increasingly difficult to detect internal structures, however, this is not always necessary to create a pleasing layout.

Many aesthetic properties contribute to good layout. Figure 2.14 shows some examples. The crossing of edges can make relationships confusing and so minimising them is important. Node density can also make a graph difficult to understand because a high density will cause edges to become indistinguishable. Spreading them equally helps and makes the layout more readable. Also, by only using edges of similar length makes relationships appear unbiased.

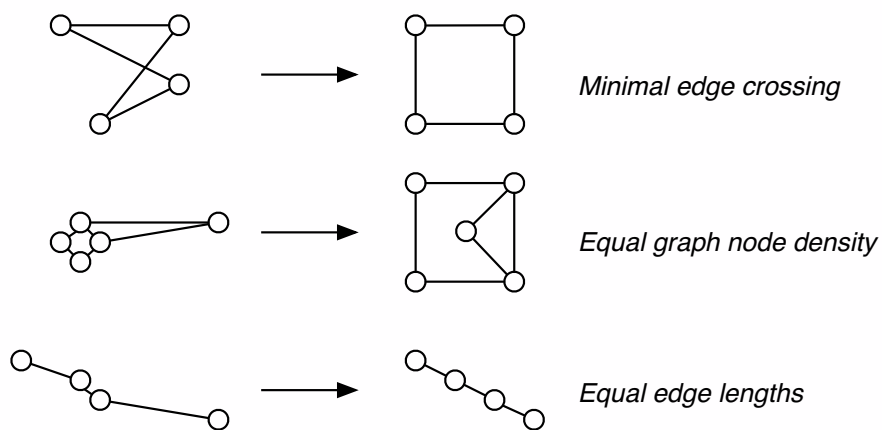


Figure 2.14: Desirable aesthetics

Graph layout aesthetics is a very large area of research and the limited time for this project did not allow for the development of new layout methods. Instead,

several existing methods were studied to allow for their implementation.

The main area considered was force-directed methods which use the analogy that a graph can be considered as a physical system. Forces are assigned to aspects of the graph, such as edges, and the physical system is left to find a state of equilibrium or minimum potential energy.

Eades spring embedder layout treats nodes as rings that are connected to other nodes by springs. Edges translate to springs of unit length and extra springs of infinite unit length are added to nodes with no edge between them. The physically built system is then released until equilibrium³ is found. The state of the final system gives the locations for nodes and paths of edges in the layout.

The major problem with this approach is that the point of equilibrium found may not give the minimum potential energy for the whole system. This would occur when opposing forces counteract each other and create a stable situation. Even though this problem can exist in many of the layouts produced, and no proof exists to back up the quality of the layouts, for small graphs the final results are generally good.

Many other force-directed methods exist with some using additional forces and cooling schedules to help converge to a layout quicker. Further information about force-directed layout methods can be found in Appendix A.

³This may not be the case for all implementations as some may use other conditions to halt on. See Appendix A for more information.

To give users the option of viewing the structure of a graph in several different ways, four layout creators were implemented.

- **Random** - Nodes are randomly positioned within a (500,500) square.
- **Grid** - Nodes are placed in the largest square grid possible and extra nodes are added to bottom of grid.
- **Polygon** - For n nodes a regular polygon with n sides is created and nodes are placed at the points of the polygon.
- **Spring** - Based on Eades spring embedder creates a layout by assuming the graph as a physical system.

Each of these were written by using the `GAVLayoutCreator` interface and return a `GeneralVisualRepresentation` of their final layout.

Drawing the Graph

Once a layout for a graph has been created it is possible to draw the graph to the screen. All drawing is performed by the `GeneralGraphRenderer` class. This uses a `Graphics` object as a canvas and draws using Java's 2D API[4]. Nodes and edges are drawn separately by the `GeneralNodeRenderer` and `GeneralEdgeRenderer` classes, respectively. This allows the appearance of nodes and edges to be altered separately.

When the draw request is sent, the `GeneralGraphRenderer` extracts positions for each node and edge from the `GeneralVisualRepresentation`. This information is then sent to the `GeneralNodeRenderer` and `GeneralEdgeRenderer` classes so that nodes and edges are drawn at the correct locations.

Both the `GeneralNodeRenderer` and `GeneralEdgeRenderer` classes use Java's 2D API for all their drawing needs. Java 2D is part of the standard JDK and includes classes to perform graphical operations on a `Graphics` object. It works using a context rendering model. All Java 2D primitive, such as lines or ovals, can have their context set by method calls, this can include attributes like colour, width and shape. Once the context of an object is set correctly it can be drawn to the `Graphics` object and will have the features specified by its attributes.

This idea fitted well into the idea of node and edge properties effecting their appearance. The context of of the graphics object to represent the node or edge is set using the node or edge properties, then when the node is drawn it is displayed in the correct manner.

Chapter 3

GVAAlgorithmViewer Application

3.1 Introduction

The GVAAlgorithmViewer application is a GUI (Graphical User Interface) to view the visualisation of graph algorithms in simple to use environment. All visualisation tasks are performed by the GAVToolkit API so that the application to concentrate on handling user interactions and requests.

Using the GVAAlgorithmViewer as an example, this chapter shows the way in which the GAVToolkit API can be integrated and provides information about customising the visualisation process. Sample code is included making an understanding of Java useful but not essential.

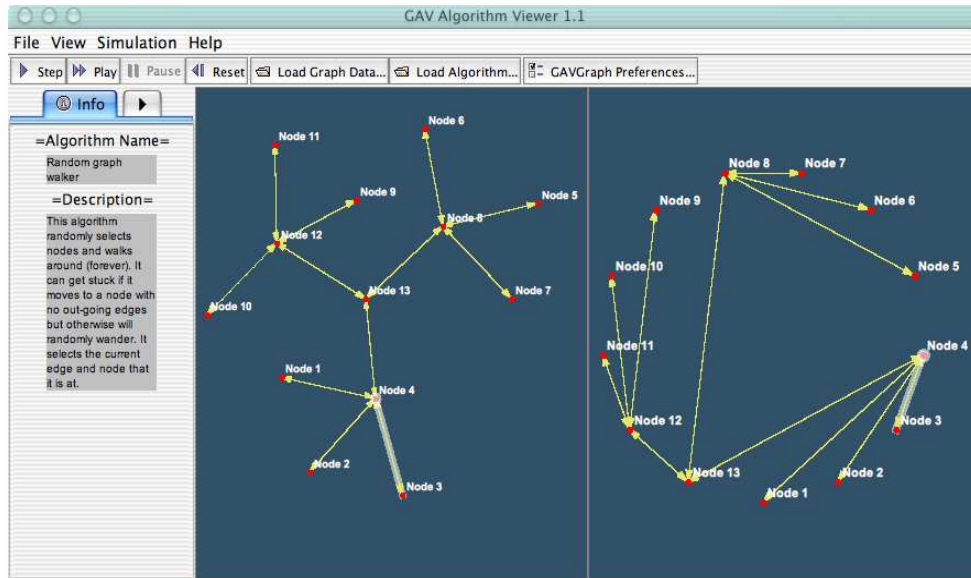


Figure 3.1: GAVAlgorithmViewer Application in Action

3.2 Purpose and Features

The main purpose of the application is to act as a front-end to the API giving access to all its features. Being easy to use it allowed for quicker testing of new visualisations and verified that integration was possible.

The application has been designed to be easy to use, stable, and robust so that users can feel safe exploring the system. Most aspects of the user interface are self explanatory through the use of standard dialogues and icons, minimising the need for large amounts of documentation.

All features provided by the API can be used within the application, including multiple displays and customisations. The main viewing window can be split into

1, 2, or 4 displays with each being fully independent of the others. These can be customised using different layout creators, node renderers, or edge renderers with changes taking effect immediately.

3.3 Integration of the GAVToolkit API

This section will look at the integration of the API into a standard Java application. The basic design principles will be discussed and example source code will be provided to show possible implementations.

3.3.1 The Basics

When writing an application to use the GAVToolkit API all interactions are performed through a `GAVGraph` object. This is the only object required when using the toolkit and gives access to all API functions.

The `GAVAlgorithmViewer` application contains a single `GAVGraph` object. All interactions with the user interface result in calls to this object which in turn updates any visualisations.

Configuring the API

Once a `GAVGraph` object has been created it must first be configured before being used. This task involves setting displays, loading a graph, and loading an

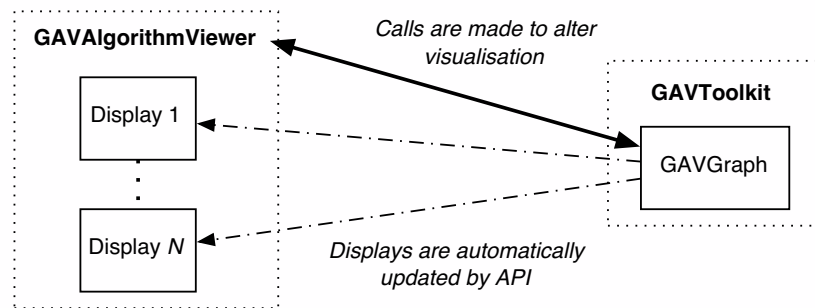


Figure 3.2: GAVAlgorithmViewer's Use of GAVToolkit API

algorithm.

All visualisation is shown through displays and a standard one is included when a `GAVGraph` object is created. All displays in the API are in the form of a `GeneralDisplay` object, which will automatically handle mouse events to both move and zoom a graph. The `GeneralDisplay` class is a sub class of Java's `JPanel`, allowing any display to be treated as a standard Java GUI component. Once a display has been added to the user interface, visualisation will be shown immediately.

To retrieve the current displays from the API the `getDisplays()` method is used, which returns a `Vector` object of every display being updated. If multiple windows are required extra displays can be created using the `addDisplay(newDisplay)` method.

The configuration of the API is completed when both an algorithm and graph are also loaded. This is performed using the `openGraphFromFile(aFile`

) and `openAlgorithmFromFile(aFile)` methods where `aFile` is the algorithm or graph file to be used. An example of the possible steps in a full setup are shown in Program 3.3.1.

Program 3.3.1 Configuration of the API

```
GAVGraph api = new GAVGraph();
JFrame gui1 = new JFrame();
JFrame gui2 = new JFrame();

GeneralDisplay d1 =
    (GeneralDisplay) (api.getDisplays().get(0));
gui1.getContentPane().add((JPanel)d);

api.openGraphFromFile(theGraph);
api.openAlgorithmFromFile(theAlgorithm);

GeneralDisplay d2 = new GeneralDisplay();
api.addDisplay((GAVDisplay)d2);
gui2.getContentPane().add((JPanel)d);
```

Starting Visualisation

After configuration is complete the API is ready for visualisation. All playback methods are self explanatory and should be made to the `GAVGraph` object in use.

- **play()** - Starts the algorithm running.
- **step()** - Moves to the algorithms next action.
- **pause()** - Pauses the algorithm.

- **reset ()** - Resets the algorithm to its beginning.
- **setSpeed (newSpeed)** - Sets the speed of the animation. A higher value for the speed will cause a slower animation.

These few methods will provide for most programmers all the functionality required and show the simple nature in which the toolkit can be used fully.

3.3.2 Advanced Features

The more advanced feature of the toolkit revolve around its ability to be customised to fit a specific need. All alterations to the API can be performed during visualisation, which will cause updates to be instantaneous.

There are three aspects of the visualisation that may want to be changed by a user — the layout creator, node renderer, or edge renderer.

Layout Creator - The layout creator can be changed for an individual display by using the `setLayoutCreator (newLayout)` on the `GeneralDisplay` object representing the display.

Node and Edge Renderers - Node and edge renderers are part of the `GeneralGraphRenderer` object that is contained within each `GeneralDisplay`. The graph renderer can be retrieved using the `getGraphRenderer ()` method on a `GeneralDisplay` object. Then by using the `GeneralGraphRenderer`

returned both the node and edge renderers can be set using the `setNodeRenderer (newRenderer)` and `setEdgeRenderer (newRenderer)` methods. Changes to the renderers will cause an immediate effect on the display.

Program 3.3.2 shows an example of altering a display to use a custom layout creator, node renderer, and edge renderer. The variable `api` in the example is a reference to a `GAVGraph` object.

Program 3.3.2 Customising the Visualisation Process

```
MyLayoutCreator lc = new MyLayoutCreator();
MyEdgeRenderer er = new MyEdgeRenderer();
MyNodeRenderer nr = new MyNodeRenderer();

GeneralDisplay d =
    (GeneralDisplay) (api.getDisplays().get(0));

d.setLayoutCreator((GAVLayoutCreator)lc);

GeneralGraphRenderer gr = d.getGraphRenderer();
gr.setNodeRenderer((GAVNodeRenderer)nr);
gr.setEdgeRenderer((GAVEdgeRenderer)er);
```

Chapter 4

Analysis of the GAVToolkit

This chapter will look at the toolkit in use and analyse the requirements it imposes on a programmer and the quality of the output produced.

4.1 Using the Toolkit

There are several types of user that may consider using the toolkit. One type would be application programmers that want the toolkit to provide the visualisation features they do not want to implement themselves. They would be concerned about the additional work of integration and the ease of use. The previous chapter showed that due to the toolkit being based around a single class both integration and ease of use were improved. An application can produce visualisations in as little as 5 lines of code (see Program 4.1.1).

Program 4.1.1 Java Code Required to Integrate a Visualisation

```
GAVGraph gav = new GAVGraph();
a_GUI_Component.add(
    (GeneralDisplay) (gav.getDisplays().get(0)) );
gav.loadGraphFromFile( aGraph );
gav.loadAlgorithmFromFile( anAlgorithm );
gav.play();
```

Another important type of user is the programmer wanting to write algorithms. They would see the toolkit as a way to improve their development speed and understanding, but would be wary of the overhead imposed when writing compatible algorithms. They would want it to be as small as possible so that little extra work is required.

4.1.1 Algorithm Creation Overhead

Even though a visualisation can be created from any algorithm written for the toolkit it was feared that extra calls to enhance the display would be required to produce an understandable output. This would increase the size of algorithm code and defy the point of making the process automatic.

To judge the additional effort required to create an algorithm, several were implemented¹. This included BFS (Breadth First Search), DFS (Depth First Search),

¹Source code for all the algorithms is available on the provided CD-ROM or from the project website.

Minimum Weight Spanning Tree (Greedy), Minimum Weight Spanning Tree (Kruskal), and Single Source Shortest Path (Dijkstra).

Initially it was thought that a direct comparison could be made with an algorithm not written for visualisation. By analysing the extra amount of code in the toolkit's implementation, an idea of the overhead could be gained. This was not possible though as general implementations would not have access to graph properties or utility functions, available in the toolkit's data type. These extra features could biased any comparisons leading to inaccurate results.

Without being able to make direct comparisons a different approach was taken. Rather than comparing an algorithm with others, each was broken down into its own attributes which could then be used for analysis. By looking at the types of statement contained within an algorithm, estimates of the overhead could be made.

Three attribute were used to analyse the algorithms — total lines of code, number of property calls (updates to node or edge properties), and number of API calls (to directly perform visualisation). Both property and API calls were also considered as a percentage of the total algorithm to better show the amount they accounted for. Results from this analysis are shown by the table in Figure 4.1.

From the results it was evident that only a small amount of calls were directly to the API (3%-6%). These calls would have no purpose in an algorithms implementation and are directly related to additional code.

Algorithm	Total lines	Property calls	API calls
Breadth First Search (BFS)	46	17 (37%)	2 (4%)
Depth First Search (DFS)	65	28 (43%)	2 (3%)
Min Weight Spanning Tree (Greedy)	34	13 (38%)	1 (3%)
Min Weight Spanning Tree (Kruskal)	35	10 (28%)	2 (6%)
Shortest Path (Dijkstra)	59	18 (30%)	3 (5%)

Figure 4.1: GAVToolkit Algorithm Attributes

Property calls are in contrast relatively high accounting for 28%-43% of an algorithm. It is more difficult to class the extent at which these add an overhead due to them possibly being used usefully to store intermediate results. It was necessary to look at the algorithms code for these calls in order to gauge an estimate of the overhead. From this it was obvious that only a small amount (35% max) of these calls were an overhead.

Overall the total overhead for the implemented algorithms ranged from around 10% to 20%. Considering that most algorithms are short and no longer than 100 lines the extra effort to create a compatible algorithm is 10-20 lines. Having a compatible algorithm allows for easier debugging and gives the option of use in a wider area such as teaching or analysis. Taking into account the benefits the toolkit provides the overhead was thought to be acceptable.

4.2 Toolkit Output

In addition to creating output that is pleasing to the eye, the toolkit requires that both the structure of the graph and the interactions made by algorithms are clear. The toolkit takes the approach of allowing understanding to be gained from relating the structure of an input to the interactions of an algorithm. Having both these in a clear manner helps the learning process.

4.2.1 Display of Graph Structure

Graphs were known to come in many different forms making it difficult to show every type in a clear manner. The structure of a graph is portrayed by its layout and four layout creators were implemented for the toolkit. It was hoped that if a graph could not be displayed well in one, another could be used to show it in a more suitable way.

To analysis the effectiveness of the layouts created, several input graphs with differing attributes were used. This would show if particular types of graph were suited to a specific layout. Results produced are shown in Figure 4.2.

The results showed a number of interesting points. In all cases output from the random layout was always unclear and of no use but to show the benefits a good layout can give.

The grid layout maintains a near constant node density and produced clearer

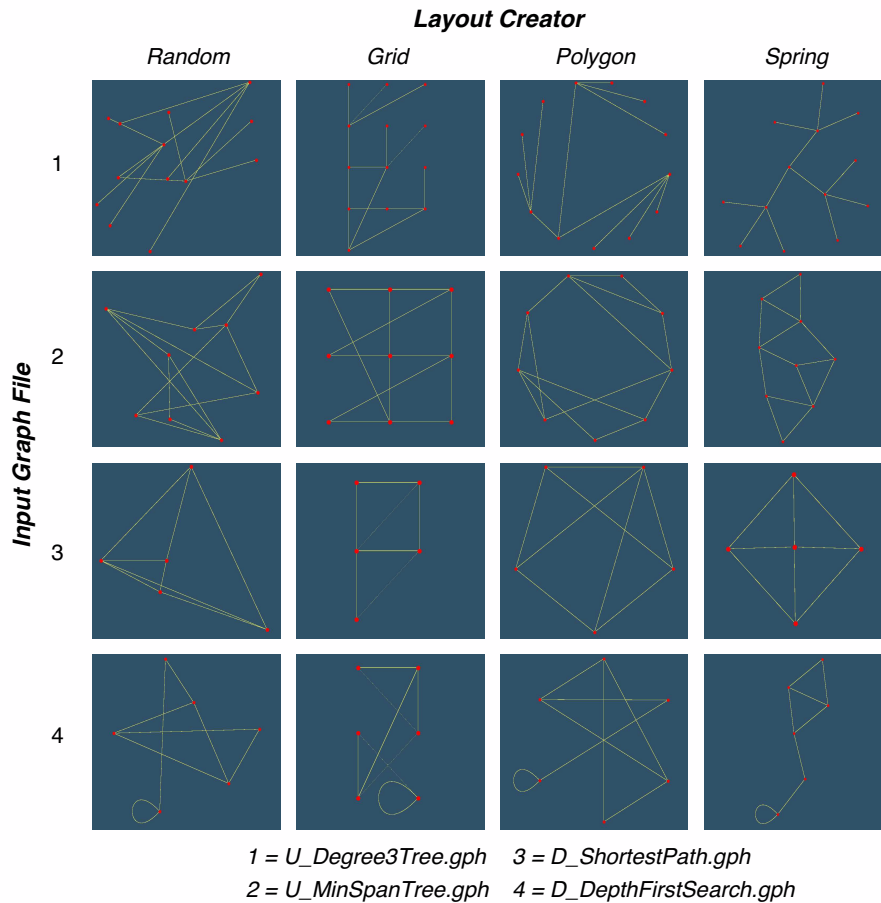


Figure 4.2: Layout Results

results than the random method. Its biggest problem was having misleading start and end locations for edges. A situation in some layouts occurred where an edge between two nodes passes through an additional node between them. This made it look as though the single edge was in fact two smaller edges. As the number of edges increases so to does the chance of this occurring, making the layout more suitable for graphs with a small ratio of edges to nodes.

Some of the clearest results were produced from the polygon layout. Its output did include a large amount of edge crossings, however due of the placement of nodes, edges were easier to distinguish as they did not pass through additional nodes. A problem does, however, arise with graphs that have an even number of nodes. Due to nodes being placed on a regular polygon, even numbers can cause multiple edges to cross at a central point. This can give the impression that an extra node exists at the center of the graph, making the layout confusing. Graphs containing odd numbers of nodes do not suffer from this problem. The major advantage of this layout is that the same graph will always produce the same layout, making output more predictable and allowing for the recreation of results.

The clearest results produced were from the spring layout. In all cases it created planar output showing the graphs full structure and making it easy to understand. This is most evident in graphs 1 and 2, where the structure is much clearer than in all other types of layout. Even though the results produced are good the spring layout has a number of problems. The processing required is significantly larger than all the other methods, however, improvements could be made by using a different implementation such as FR or GEM. Also due to nodes being given random start locations every layout produced is unique making it impossible to recreate exact results on demand.

The methods included with the toolkit give a good breadth of differing types

of layout, allowing for the creation of a clear and understandable layout in most situations and for most types of graph.

4.2.2 Algorithm Animation and Interaction

One of the main objectives of the toolkit was to show the interactions of an algorithm through animation. The frames in the animation should relate directly to the steps taken by an algorithm. It was also hoped that this process of seeing the interactions would help a viewer understand, at a high level, how the algorithm works or its purpose.

To produce an animated visualisation for analysis, two algorithms, DFS and BFS, implemented for a previous section were used. These were chosen because the way they search a graph is distinctly different and should be evident in the output. Figures 4.3 and 4.4 show snapshots of the animation produced for each algorithm.

In both algorithms the interactions being made are highly visible. They are shown by three main attributes — size, colour, and label. The size of a node or width of edge is used to show when an object has been visited. This gives an indication to both the extent and form of exploration made by an algorithm.

Colour is used generally to classify a type, for example white may represent *not found*, grey — *found*, and red — *finished*. Using colour for this purpose

connects well to our real world where it is common to use different colours to to group similar information.

Labels allow textual information to be stored on an edge or node. This is the most difficult information to understand because unlike colour and size it requires larger processing to be understood. They are required though, as some information cannot be shown in any other suitable way. The labels in a graph are used normally to show distances or names and if used correctly can help give pointers to and algorithms ultimate goal.

The use of these attributes can be seen in both algorithms, however, each uses them in a specific way to give an indication of their purpose.

The DFS output shows the type of exploration it performs by the selection of the current path. It can be seen to search deep into the graph first with no branching. Throughout this searching the colour of edges change, giving the indication that the algorithm is classifying them in some way. Although it is not known exactly how the algorithm is doing this, an abstract understanding (searching deep with no branching) of the method is understood.

BFS has a different purpose, and uses different attributes. It to selects edges showing the exploration being performed, however, it can be seen to slowly work its way through the graph branching to all closest nodes first. Labels are updated when a node is selected for the first time, with numbers that increase from 0 at the start. Again like in DFS although it is not known exactly how the distances

are being calculated, a high-level understanding of the method (large amount of branching) can be gained.

In both cases a light understanding of the purpose and method can be seen from the visualisation alone, showing the usefulness of the visualisation technique. It is dubious the extent to which the workings of the algorithms are understood, however, useful information has been gained that may help when looking at the actual algorithms code. It would have been interesting to have performed further research into this area and gain a greater understanding of the processes involved when interpreting the visualisations, however, time did not permit.

These results showed the toolkit displayed interactions in a clear understandable way, helping to give indications to an algorithms purpose and method.

4.2.3 Customisation

Due to unknown uses of the toolkit in external applications, it was designed to be highly customisable so that specific aspects could be changed. Making a system customisable can lead to instabilities as extensions may cause bottle necks or crashes. Attempts have been made to minimise the chance of this in the toolkit by making sure that every aspect that can be customised performs a single well defined task. This stops the integration of extra functionality that is not required. Also the use of interfaces helps implementers understand how their extensions

should behave.

To test the customisation features, extensions to the toolkit were written. These included an animated spring layout, node renderers, and edge renderers. These could then be used to alter every part of a visualisation. It was not possible to show the animated spring layout in this report, however, it can be used within the `GVAAlgorithmViewer` application. Results from the toolkit are shown in Figure 4.5.

Originally it was thought that customisations would only be of use to make the displays more aesthetically pleasing, and not directly help with understanding. This was shown to be incorrect after realising that they could be used to display properties in a more suitable way.

Research into graph algorithms led to the discovery of network flow based methods. These use a graph to represent the structure of a network, and the weights of edges to represent flows. Displaying such an algorithm using the standard displays would cause the flows to be shown as numbers. This is not ideal and makes it difficult to quickly see changes that may occur. By instead showing flow graphically as the width of a line, the information can be understood more quickly. As all aspects to a visualisation can be changed a new edge renderer was written to graphically show the flows and can be seen in Figure 4.5 labelled the *large line* edge renderer. To fully show the benefits this provides, an example algorithm `RandomFlows` was included with the toolkit.

The customisation features of the toolkit allow it to be used in specific areas where conventional fixed output does not produce adequate results. They can make output more appropriate for a type of algorithm and allow application developers to alter visualisations to make them more unique, yet still give access to underlying features of the toolkit.

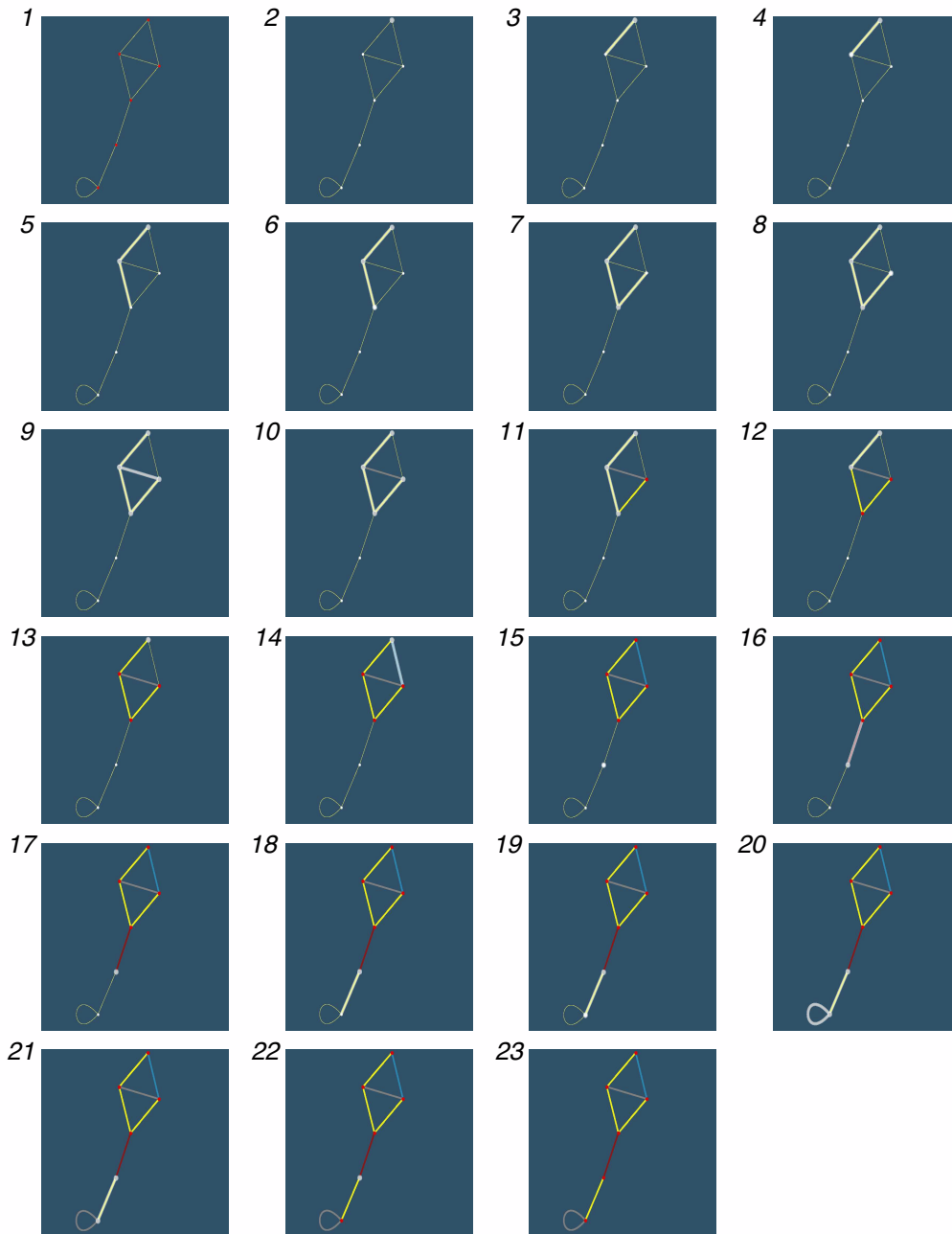


Figure 4.3: Animated Visualisation of the Depth First Search (DFS) Algorithm (Graph used: D_DepthFirstSearch.gph)

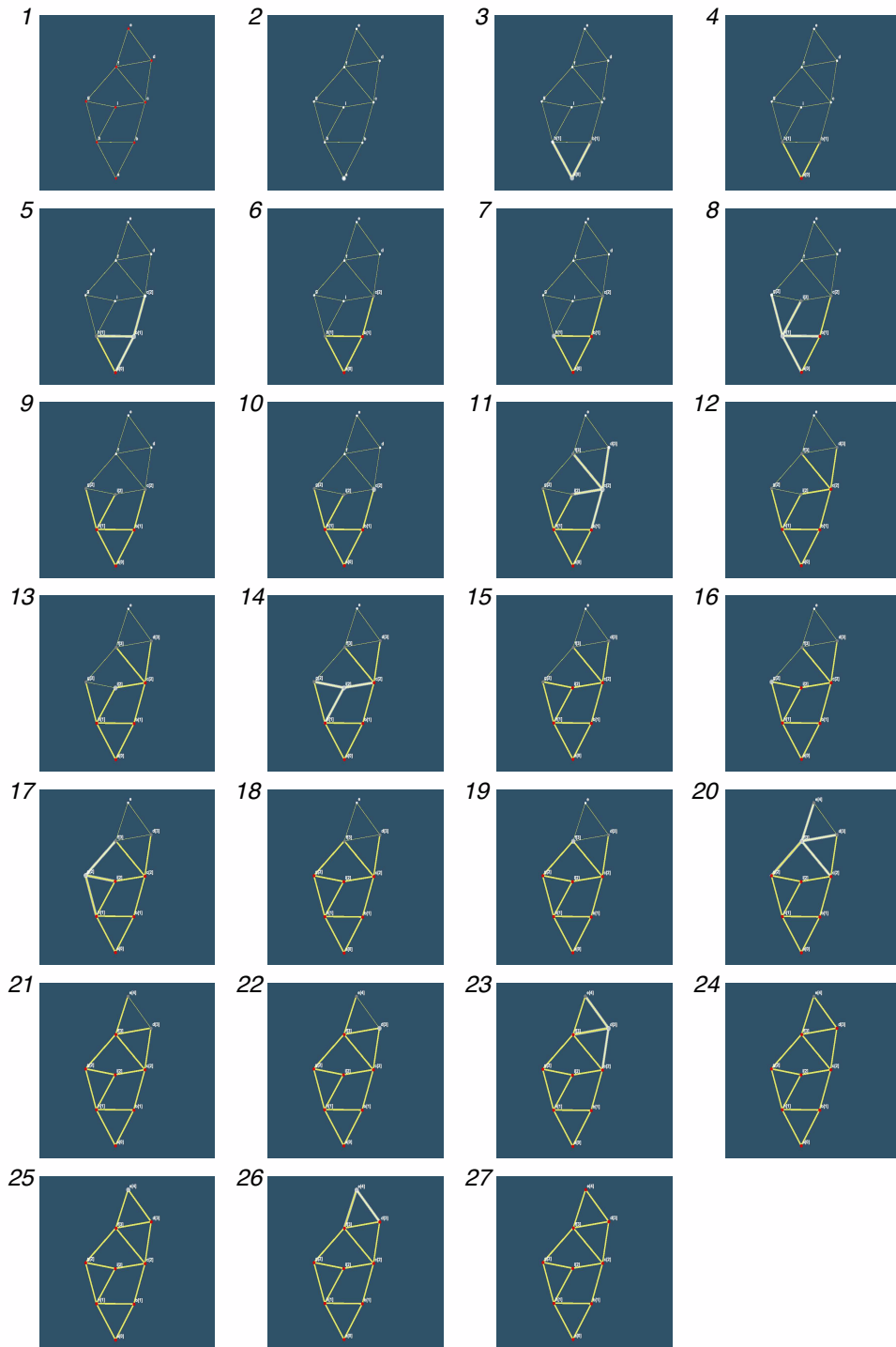


Figure 4.4: Animated Visualisation of the Breadth First Search (BFS) Algorithm
 (Graph used: U_MinSpanTree.gph)

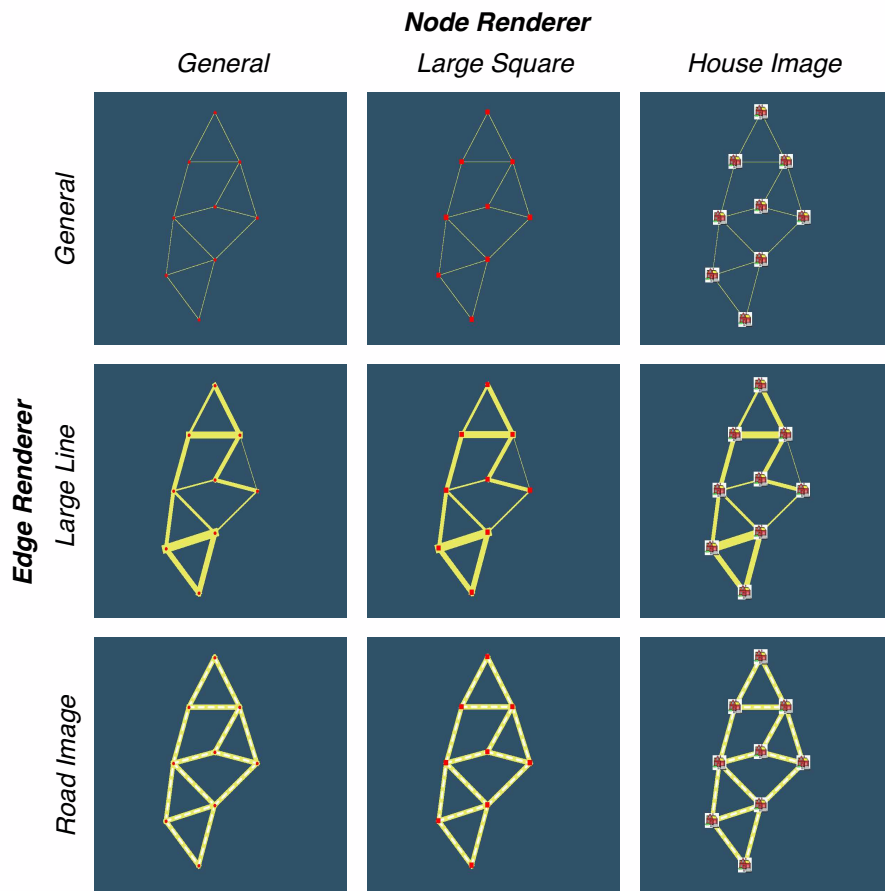


Figure 4.5: Customisation Results

Chapter 5

Conclusions

5.1 Conclusions

This project has led to the successful creation of an easy to use toolkit, comprising an API and application, for the automatic visualisation of graph based algorithms. Visualisations produced by the toolkit have been shown to both be aesthetically pleasing and helpful when trying to understand how an algorithm works.

An important aim of the toolkit was to provide visualisations that would help with understanding. The toolkit took the approach of letting a viewer relate the structure of an input graph to the interactions made by an algorithm. It was shown that these interactions give a basic abstract representation of the decisions made by an algorithm, however, the extent to which this could aid understanding was

questioned. They were thought in many ways to be too abstract, losing much of the useful information, however, results did show that even though an algorithm could not be fully understood, a good indication to its purpose and the method could be seen.

The production of visualisations led to the study of layout methods, concentrating on the force-directed techniques of — the spring embedder, FR, and GEM. Several layouts were implemented for the toolkit including, random, grid, polygon, and spring. The quality of the output produced by each of these methods was also analysed and it was shown that clearest results were created by the polygon and spring layouts.

A large aspect of the project related to the creation of a hybrid graph data type to be used by algorithms. Existing graph storage methods were analysed, leading to the alteration of a linked list method using of Java's `Vector` object to speed up search times. The data type also allowed for the storage of properties on individual nodes and edges, vital for algorithm interactions.

With algorithms in the form of compiled class files, a custom class loader was created to extract the algorithms. This required understanding of the JVM and its class loading mechanism. The created class loader allowed for an algorithm object to be extracted from a class file at any given location (URL).

All visualisation has been shown using the `GVAAlgorithmViewer` application. Developed to help with testing it has also shown the ease at which the toolkit

can be integrated into any application. It is a good example to other developers considering the toolkits use.

Much of the success of the project is due to its management which has improved throughout. This led to the creation of well defined objectives, deadlines for completion, regular meetings with the project supervisor, and a web site to communicate current progress.

5.2 Personal Outcomes

The project has also led to many personal achievements. Having not completed a project of this size previously, good time management and planning skills were vital to meet the tight deadlines. It has given me an understanding of the methods to tackle similar problems, and has led to furthered understanding in a wide area of subjects from graph layout creation, to JVM internals.

Chapter 6

Extensions

When designing the toolkit it was anticipated that the main use would be for debugging and analysis purposes. During development further research was carried out and it became obvious that computer aided learning is highly sort after. Although the toolkit can, in it's current form, be used for learning, extensions to increase its teaching potential would be beneficial.

Learning is a complex idea, however, it has been shown that user interaction with a problem can help. By extending the toolkit to allow for this, a user could directly influence the visualisation process. This could be used to test the understanding a user has, for example, a program could ask them periodically to pick the next step that an algorithm they have studied should take. Teachers could use this to help with learning or as a means of testing to see how well an algorithm has been understood.

A problem that the toolkit as a whole faces is its speed. The size of the graphs used during this project have been relatively small, having no more than 20 nodes. It would be possible to view graphs with hundreds of nodes, however, the speed of the toolkit would decrease rapidly. Part of this problem is due to Java being used as the programming language. It was chosen because it would allow for rapid development, but this was at the expense of execution speed. Java is compiled to byte code form, that is then interpreted, however, even with improvements to interpretation using JIT (Just In Time) compilers the speed divide with languages that are directly compiled for a system is still large. Now that the design has been shown to work well the toolkit could be rewritten in a lower level language, such as C or C++ to give a quicker running speed, better options for optimisation, and the ability to use larger graph data types for visualisation.

6.1 GAVToolkit API Extensions

The main role of the API has been to act as a data type and visualise the running of an algorithm. As the visualisation can be customised many extensions lead to further add-ons being created. Extra displays could be written, such as 3D or text, to help users look at problems from a different perspective. This could be useful for specific fields where the data is collected from a known domain and where displaying the data in a related way helps interpretation.

In a similar way a greater number of layout creators could be produced. The structure of the graph is conveyed by the layout and being able to choose from a larger range could make the internal structures of an input graph easier to decode.

Output from the toolkit is currently very limited with the only option being an animation to the screen. This requires the use of the toolkit and does not make distributing specific output easy. Having the feature to export an animation to a video file in QuickTime¹ or AVI² format would allow users to easily save a specific output they have produced that can then be viewed by those who do not have access to the toolkit.

The input of graph data is also limited with only a native format that only allows for nodes, edges, and edge weights to be stored. With the current interest into XML, the format could be rewritten to take advantage and allow for node and edge properties to be stored. Alternatively, an import module could be written to accept standard formats, such as graph6³. Having this option would allow existing graphs to be used aiding users that have large data sets that are not easily converted.

Another limitation that the API currently has is that only one algorithm can be visualised at a time. Allowing multiple algorithms to be shown simultaneously

¹A video format created by Apple (see <http://www.apple.com/quicktime>).

²Audio Video Interleaved, is a video format developed by Microsoft for the Windows operating system.

³Standard graph format for undirected graphs (see <http://cs.anu.edu.au/~bdm/data/formats.html>).

could help a user see differences between methods and allow them to understand why ones runs quicker than another. Two different minimum cost spanning tree algorithms were included with the toolkit, greedy and Kruskal. Looking at both separately can be misleading and it may seem that both work in the same way, however, if both could be viewed at the same time the differing steps would become obvious.

Finally, one of the most useful extensions would be to produce automatic analysis of an algorithm. The API already processes the interactions an algorithm has with the graph data type and by analysing this information results could be calculated about the efficiency and interaction statistics. The results could be used to see how an algorithm is effected by differing graph types, or could be used to compare the running characteristics. It may give pointers to areas where a given method works well or signal problems that may occur.

6.2 GAVAlgorithmViewer Extensions

The current purpose of the GAVAlgorithmViewer application is to give the user access to all features included in the API. Other than controlling GUI aspects of the application, it has very little functionality of its own. Extensions could be created to give the application a more unique purpose and to perform tasks separate, yet related to the algorithm visualisation process.

One direction that could be taken is to integrate the algorithm writing process. Presently algorithms written for the toolkit are created and compiled separately before being visualised. These separate steps could be brought together by adding code editing features and automatic compilation to the application. This could help development speed as less switching between applications would be required.

These editing features could also be used during visualisation to show the current line of execution, helping with problem debugging and learning.

With these extensions the application would become more than a simple wrapper to the API, but would facilitate the development, analysis, and learning of graph based algorithms in a single package.

Chapter 7

Further Information

7.1 Web Site

The web site for the toolkit includes general information, news, documents produced for the project, links to related resources, documentation for the API, and a downloadable version of the toolkit for use. The web site can be found at the following address.

`http://www.dcs.warwick.ac.uk/~ csvbt/`

7.2 API Documentation

For those interested in developing applications using the toolkit it is advisable to review the documentation that can be found on the web site and included in the

package containing the developer edition of the toolkit.

All documentation has been created using `javadoc`¹ and includes listings of all classes and methods in the toolkit. It may also be useful to read Chapters 2 and 3 of this report to gain an understanding of how the toolkit works internally and how to customise its features for use in an application.

7.3 Using the Attached CD-ROM

The attached CD-ROM at the back of the report holds the current developer version of the GAVToolkit. The toolkit can be found in a directory labelled `GAVToolkit`. For information about the release and how to use it a file named `README` can be found. The distribution includes both source code and compiled code for the GAVToolkit API and GAVAlgorithmViewer application. Some example graphs, algorithm, and documentation for the API are also included.

In addition to the toolkit a directory labelled `Website` contains the current web site and the `Documents` directory has all documents created for the project, including the specification, and progress report.

¹`javadoc` is a documentation tool distributed with Java. Related information can be found at <http://java.sun.com/javadoc>

Bibliography

- [1] <http://www.cs.helsinki.fi/research/aaps/Jeliot/>

- [2] **Introduction to Algorithms** - MIT Press 1999 (ISBN 0-262-53091-0) -
Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

- [3] **Algorithms For The Visualisation Of Graphs** - (ISBN 0-13-301615-3) -
Giuseppe Di Battista, Peter Eades, Roberto Tamassia, Ioannis G. Tollis

- [4] **Java 2D API Graphics**
Prentice Hall (ISBN 0-13-014266-2) - *Vincent J. Hardy*

- [5] **Java Structures - Data Structures in Java for the Principled Programmer** - McGraw Hill (ISBN 0-07-116091-4) - *Duane A. Bailey*

- [6] **Do Algorithm Animations Aid Learning?** - 1996 - *M. D. Byrne, R. Catrambone, J. T. Stasko*

- [7] **Empirically Assessing Algorithm Animations as Learning Aids** - MIT Press 1998 - *J. Stasko, J. Domingue, M. H. Brown, B. A. Price*

Appendix A

Force-Directed Graph Layout

Algorithms

There are many different variations of force-directed techniques for creating graph layouts. All algorithms of this type, however, share common characteristics. They all assign forces to aspects of the graph representation and then find a point of equilibrium or minimum potential energy.

Most methods are iterative which has the advantage of allowing changes to the layout to be shown in small understandable steps. This allows users to use previous mental models of the graph during the creation of the new layout.

Very few actual results exist to recommend the quality of the graphs produced, but for small instances the methods tend to work well. Disadvantages include difficulty in extending outside straight line drawings and slow running times.

A.1 The Spring Embedder [Eades]

This algorithm is arguably the most well known of all force directed layout methods. It uses the idea that a graph can be treated as a physical system of rings connected by springs. A layout is produced by finding a point of equilibrium in the system. Nodes are replaced by rings and edges by springs of unit length. A spring exerts an outward force on its connections when compressed and an inward force when stretched. The size of unit length gives a general scaling to the final layout, i.e. larger unit length leads to a larger scale. Extra springs of infinite unit length are added to all pairs of nodes that do not have an edge between them.

The purpose of these infinite edges is to make sure that each node tries to spread away from each other helping remove overlapping layouts. An elementary example is a graph with just 3 nodes and 2 edges, where all nodes are connected by a path. The final layout will always contain the 3 nodes with the edges of unit length, however, the leaf nodes could overlap if the additional infinite length edges were not added. Having these edges causes the unconnected nodes to repel each other removing the overlapping case.

The force that acts on a node is defined by the following equation.

$$F(v) = \sum_{(u,v) \in E} f_a(d_{uv}) \times \left(\frac{u-v}{d_{uv}}\right) - \sum_{(u,v) \in V^2} f_r(d_{uv}) \times \left(\frac{u-v}{d_{uv}}\right) \quad (\text{A.1})$$

The repulsive and attractive factors (f_a and f_r) are defined as follows.

$$f_a(d_{uv}) = S_{uv}(d_{uv} - l_{uv}) \quad (\text{A.2})$$

$$f_r(d_{uv}) = \frac{R_{uv}}{d_{uv}^2} \quad (\text{A.3})$$

where S_{uv} is the stiffness of spring between u and v , R_{uv} is the repulsive force between u and v , d_{uv} is the distance between u and v , and l_{uv} is the unit spring length between u and v .

When implementing this method it is sometimes found that oscillations can occur, hindering the chance of a clear layout being produced. To help minimise the possibility of oscillations, friction is considered with every movement. Program A.1.1 shows pseudo code for the spring embedder, where T is the number of iterations performed.

Program A.1.1 The Spring Embedder - Pseudo Code

```
Place each node at random location
Repeat T times
  For each node, N in the graph
    Calculate force on N
  Move all nodes, Friction * force on node
Layout complete
```

A.2 Fruchterman and Reingold (FR)

This layout algorithm extends the general spring embedder by adding additional attractive and repulsive factors. To calculate the force acting on node v the same equation (A.1) is used. The attractive and repulsive factors are, however, different and defined as.

$$f_a(d_{uv}) = \frac{d_{uv}^2}{k} \quad (\text{A.4})$$

$$f_r(d_{uv}) = \frac{-k^2}{d_{uv}} \quad (\text{A.5})$$

Different factors (A.4, A.5) are used because they produce results that are similar to A.2 and A.3, however, are more efficient to calculate.

To help converge to a final layout quicker a cooling schedule is used. This is a function that given an iteration number will return a temperature which is the maximum distance that a node can be moved. Generally it is assumed that the most significant movements occur at earlier iterations suggesting that a cooling schedule should start at an initial value and decrease in an inverse linear fashion. It should be noted, however, that a better cooling schedule can dramatically increase the speed at which a layout is found.

The FR algorithm normally terminates after a set number of iterations. This can cause the algorithm to end prematurely or continue to run after a final layout is produced. Some implementations calculate the total energy for the system and

terminate when a set energy level is met.

Program A.2.1 shows the pseudo code for the FR algorithm, where T is total number of iterations.

Program A.2.1 Fruchterman and Reingold (FR) - Pseudo Code

```
Place each node at random location
Repeat T times
    TEMP = Calculate temperature for iteration T
    For each node, N in the graph
        Calculate force on N
    Move all nodes, force on node (force <= TEMP)
Layout complete
```

A.3 Graph Embedder (GEM)

The Graph Embedder (GEM) layout algorithm is also an extension of the general spring embedder. It does differ in some important ways though. Unlike both the spring embedder and FR it does not move all nodes at once but instead uses a random permutation of the nodes and moves each individually.

Like FR, GEM includes a cooling schedule to help converge to a final layout quicker. The cooling schedule does not take the current iteration number, but uses the previous movements of a node. The probability that a node is oscillating, rotating, or moving towards its final location is calculated and the most probable

is chosen. If the node is found to be moving towards its final location then the temperature is increase, else the temperature is decreased. Using this method helps to minimise both oscillations and rotations during the creation of a layout.

GEM runs for a maximum number of iterations or ends if the global temperature of the system falls below a given value. Program A.3.1 shows the pseudo code for the GEM layout algorithm where `TEMP_MIN` is the minimum temperature, and `T` is the maximum number of iterations.

Program A.3.1 Graph Embedder (GEM) - Pseudo Code

```
Place each node at random location
Repeat T times or until current temperature < TEMP_MIN
  For each node N, from random permutation of nodes
    TEMP = calculate temperature of N
    Calculate force on N
    Move node N, limited by TEMP
Layout complete
```
