1

**PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT**

PRECISIONSM RESOLUTION TARGETS

PIONEERS IN METHYLENE BLUE TESTING SINCE 1974

1980 COUNTY ROAD 5 BURNINGWELL, WY 82007 USA
TEL 812 426 7857 FAX 812 426 7857 TLEX 810000000

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

A Graphical User Interface Server for Graph Algorithm Programs

by

Janet B. Noye, BSc(CS)

**A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of**

Master of Computer Science

School of Computer Science

Carleton University

Ottawa, Ontario

January 17, 1992

© copyright

1992, Janet B. Noye

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-75977-1

Canada

The undersigned hereby recommend to the Faculty of Graduate Studies
and Research acceptance of the thesis,

A Graphical User Interface Server for Graph Algorithm Programs

submitted by Janet B. Noye, BSc(CS)
in partial fulfillment of the requirements
for the degree of Master of Computer Science

Thesis Supervisor

Director, School of Computer Science

Carleton University

January 17, 1992

Abstract

This thesis describes a user interface facility for creating and manipulating graphs that can be used in the development of graph algorithm programs. The facility described is an adaptable graph user interface program, GraphUI, that runs on Macintosh computers. GraphUI provides a complete separation of its graphical user interface from programs which use it. GraphUI can be used by a graph program to obtain input, display output, and animate algorithms.

Programs access the user interface capabilities of GraphUI through a set of commands which may be sent to GraphUI across a variety of mediums. The commands have been designed to allow graph programmers to use graph knowledge rather than use graphical knowledge when interacting with GraphUI.

This thesis focuses on the development of an open architecture to allow graph programs from different fields, different development environments, and different platforms to make simultaneous use of a centralized interactive graph user interface.

Acknowledgements

Thank you Gordie for everything.

I would like to thank my supervisor Lou Nel for his continued support and friendship throughout my thesis. He made what could have been a very frustrating time for me much more bearable and most often a lot of fun.

Thank you Aubrey, Barb, Harold, Marlene, Debi, Nancy, Heather, Rob, and Carolyn.

Thank you to Janet for helping to get rid of lots of errors the first time through and for many good talks. I hope the world looks better when you get back from seeing it.

Thanks to Rob and Rajiv for friendship and many great cups of coffee and for humoring me by not talking about school. To Erwin for numerous good lunches.

Table of Contents

Acceptance Sheet.....	ii
Abstract.....	iii
Acknowledgements.....	iv
Table of Contents	v
List of Tables.....	vii
List of Figures.....	viii
 CHAPTER 1 : Introduction	 1
1.1 State of the Art in Graphical User Interfaces for Graph Programs.....	 3
1.1.1 Drawing Graphs	4
1.1.2 Graph Applications with Graphical User Interfaces	5
1.1.3 Adaptable Graph User Interface Tools	5
1.3 Problem Definition.....	10
1.4 Thesis Layout.....	11
 CHAPTER 2 : Design of GraphUI	 13
2.1 Requirements of GraphUI.....	13
2.2 Design Goals for GraphUI.....	14
2.3 Communication Design.....	16
2.4 User Interface Design	18
2.5 Application Interface Design.....	23
2.6 GraphUI Programming Model.....	31
 CHAPTER 3 : Implementation of GraphUI.....	 33
3.1 Introduction.....	33
3.2 Object Hierarchy.....	33
3.3 Notable Implementation Details.....	37
 CHAPTER 4 : User Interface	 46
4.1 Creating Graphs.....	46
4.2 Modifying Graphs.....	53
4.3 Displaying and Setting Attributes of Graphs	59
4.4 Printing From GraphUI.....	65
4.5 Clipboard Support.....	66
4.6 Connecting to and Using Graph Applications	69
 CHAPTER 5 : Application Interface.....	 74
5.1 GraphUI Commands.....	75
5.2 GraphUI Information Messages	101
5.3 Suggested Animation Techniques.....	108

5.4 Suggested Protocol Techniques.....	110
CHAPTER 6 : GraphApp: A Sample Program.....	113
6.1 Running GraphApp.....	113
6.2 Using GraphApp as a Programming Platform.....	123
6.3 Epilogue	144
CHAPTER 7 : Conclusions.....	146
7.1 Future Work.....	147
7.1.1 Graph Programs	147
7.1.2 Communication Platforms.....	148
7.1.3 Extensions to GraphUI.....	149
7.2 Contributions.....	150
APPENDIX A : GraphUI Detailed Message Information	152
APPENDIX B : GraphApp Implementation Details.....	155
REFERENCES	161

List of Tables

Table	Description	Page
2.1	Definition of Graph Attributes	25
2.2	Definition of Node Attributes	25
2.3	Definition of Edge Attributes	26
3.1	Object Classes With GraphUI	34
5.1	Format of the Change Graph Attributes Flag	91
5.2	Format of the Change Node Attributes Flag	93
5.3	Format of the Change Edge Attributes Flag	95

List of Figures

Figure	Description	Page
2.1	QuickDraw versus Bitmap Output	15
2.2	Logical Connections Between GraphUI and Graph Programs	17
2.3	Pictorial Representation of a Graph	19
2.4	Drawing Syntax for Multiple Edges Between Two Nodes	20
2.5	GraphUI's Drawing Representation of Arcs	20
2.6	Minimum and Maximum Node and Edge Sizes in GraphUI	21
2.7	A Graph With Node and Edge Labels	21
2.8	Model of the GraphUI Program	32
3.1	Node Extent	39
3.2	Edge Extent	39
3.3	Representation of Multiple Edges	41
3.4	How a Bent Edge is Drawn	41
3.5	A Pair of Nodes With Many Edges Connecting Them	41
3.6	Computing the Bendpoint of Bent Edges	43
3.7	Algorithm Used to Determine the Bend Point of an Edge	43
4.1	Empty GraphUI Window	47
4.2	Graph With One Node	49
4.3	What the User Sees when Creating an Edge	50
4.4	GraphUI's Representation of an Edge	50
4.5	Multiple Edges Between a Pair of Nodes	50
4.6	Completing a Graph	51
4.7	A Complete Graph on 6 Nodes	52
4.8a	The Feedback Displayed While a User Makes a Selection	55
4.8b	The Selection Set Created in Figure 4.8a	55
4.9a	Graph Before Edge 4 is Contracted	58
4.9b	Graph After Edge 4 is Contracted	58
4.10	Change Graph Attributes Window	60
4.11	Change Node Attributes Window	62
4.12	Change Edge Attributes Window	64
4.13	Page Setup for LaserWriter	65
4.14a	Graph to be Copied/Cut	67
4.14b	Portion of Graph Copied	68
4.14c	Remainder of Graph After a Cut Command	68
4.15	AppleTalk ADSP Tool Connection Settings	71
4.16	Applications' Transcript After Running Shortest Path	73
5.1	Format of All GraphUI Messages	74
6.1	Static Window Displayed by GraphApp	114
6.2	Selecting AppleTalk Connection Tool	115

6.3	Algorithms Menu Items for GraphApp	116
6.4	Input Graph to Shortest Path Algorithm	118
6.5	Results of Shortest Path Algorithm	118
6.6a	Graph Before Being Completed	120
6.6b	Graph After Being Completed By GraphUI	120
6.6c	Graph After Being Completed By GraphApp	121
6.7a	Graph Before Running Node Degrees Algorithm	122
6.7b	Graph After Running Node Degrees Algorithm	122
6.8	Graph Object Hierarchy	124
6.9	List of Algorithm Names and IDs from UAlgorithms.cp	132
6.10	Setting GraphUI's Algorithms Menu	133
6.11	The ExecuteAlgorithm Method Handles Run Algorithm Cmds	134
6.12	Algorithm Method Definitions from UGraphApp.h	134

1. Introduction

Because graphs are used to represent relationships in many domains, it is necessary for people and programs to create and manipulate graph models. To this end, work has been done on graph editing tools which users can use to create pictures of graph models. These pictures of graph models can be used to design graph data structures for application programs. Unfortunately, this is a two step process, create a picture of the graph model then use it to create the appropriate graph data structures.

To combine this two-step process into one, work has been done on graph programs which have interactive graphical user interfaces, (GUI). In such programs, the pictures of graph models created by a user are automatically converted to the correct graph data structures used by the program. Unfortunately, the GUIs of these programs have been designed to suit the needs of a specific application domain and cannot easily be reused for graph programs from a different application domain.

Since building a GUI requires considerable effort, a number of graph tools have been created which provide GUI capabilities that can be used by graph programs. The graph tools are termed adaptable graph user interface tools because the tools can be modified for new graph programs. These tools help to reduce the overall time required to build a graph program with an appropriate GUI. The manner in which graph programs access the GUI capabilities of an adaptable graph user interface tool is termed its application

interface. The application interface dictates both how a program uses the tool and the capabilities the tool provides.

The application interfaces of existing adaptable graph user interface tools are achieved by combining the tool and a graph program into a single program. To combine an adaptable graph user interface tool with a graph program requires modifying the tool's code to integrate it with the graph program. Unfortunately, this means the programmer must have knowledge of both the implementation of the tool and its development environment. Only programs which can be linked directly with the tool can make use of it. This limits the graph programs that can use the existing tools to particular development environments and platforms.

This thesis addresses the problem of how to design an application interface architecture for an adaptable graph user interface tool that does not require reprogramming the tool and does not limit the programs that use it to a specific development environment and platform. In particular we look at the problem of how to design an application interface so that it separates an adaptable graph user interface tool completely from the graph programs that use it.

In our solution to this problem, an adaptable graph user interface tool which can meet the user interface requirements of graph programs from a myriad of application domains, is designed and implemented. The application interface to our tool, which is called GraphUI, uses a graph command language that is communicated through inter-application message passing. GraphUI's graph command language uses graph entities such as nodes and edges rather than GUI entities such as circles and lines thereby isolating programmers from GUI implementation details. Message passing

allows GraphUI to provide user interface capabilities to graph programs which are running on the same computer or on different computers. This provides a centralized graph GUI that can be used simultaneously by multiple graph applications.

GraphUI has been designed to assist academic researchers in three areas: implementation, publication, and presentation. GraphUI helps the researcher implement graph algorithms by supplying the graphical user interface. To aid researchers in describing the results of an algorithm, the pictures of graphs displayed by GraphUI can be copied directly into documents. Because GraphUI can be used to animate the progress of an algorithm, researchers can show how an algorithm operates during presentations or classes.

The following section outlines current and related work in GUIs for graph programs including a number of adaptable graph user interface tools. It is followed by a section outlining the research problems addressed by this thesis.

1.1 State of the Art in Graphical User Interfaces for Graph Programs

This section describes current research in graphical user interfaces for graph programs. We look at programs that can be used to create pictures of graph models and graph applications which have graphical user interfaces that allow users to enter graph models by drawing them. We concentrate on describing existing adaptable graph user interface tools which have been designed to provide ready-made user interfaces for new graph applications.

Such tools reduce the overall development time required to build new graph programs.

1.1.1 Drawing Graphs

There are many commercial generic drawing programs such as MacDraw [12] and MacPaint [13] which can be used to create pictures of graph models. MacDraw and MacPaint are programs which allow users to draw pictures using components such as circles, squares, rectangles, lines, polygons and text. In pictures of graph models, circles or rectangles are often used to represent nodes whereas lines are often used to represent edges.

Other applications like DAG [6] have been designed specifically for drawing directed graphs. DAG automatically layouts graphs for users so that the number of edge intersections is minimized. Graph drawing programs allow users to build pictures of graph models by interactively creating nodes and edges rather than the circles and lines users of generic drawing programs must use. Thus graph drawing programs provide users with graphical abstractions for nodes and edges.

Pictures of graph models are useful both for designing input to graph programs and displaying output from graph programs. To design input, a picture of a graph model is often created and then interpreted by a user to create the correct graph data structure for a graph program. To display output the process is reversed, a graph data structure is used to draw a picture of the graph model it represents. Pictures of graph models are not necessary for creating input and conveying output, they are just make it easier for people to comprehend the models.

1.1.2 Graph Applications with Graphical User Interfaces

Graphs can be used to model many different types of relationships. Because graph models have diverse applications, there is a great number of programs that use graphs as input and output. There are so many applications which use graphs in modeling data it is impossible to list them all. Some representative example applications are: basic graph theory programs [10], network design [11], parallel processing [4], software engineering [16], and statistical modeling [18]. The referenced applications have GUIs which display pictures of graph models and allow graph models to be built pictorially. The pictures of graph models created with an application's GUI automatically generates the correct data structures for the application. The user interfaces of the applications referenced were all well suited to the needs of each application but their use was restricted to their particular application domain.

1.1.3 Adaptable Graph User Interface Tools

Adaptable graph user interface tools are graphical user interfaces that can be altered to provide new graph programs use of its GUI. This section outlines the current research in adaptable graph user interface tools which enable graph programmers to use the tools with their graph applications. Although the functionality of these user interfaces vary, they all attempt to provide user interface capabilities to programs which use graph models, regardless of the application domain of the program. The application

interface describes the way a tool is accessed by graph programs and the functionality it provides. The application interface used to adapt the following user interface tools to new graph programs varies and is the primary subject of this subsection.

Graph Browsers allow users to navigate through, observe and change graphs. The GRAB graph browser [15] can be used to browse arbitrary directed graphs and can be integrated with other applications that allow browsing of information. GRAB displays the nodes of a graph as icons and the edges as lines connecting nodes. It permits users to add and delete nodes and edges, move nodes and change node and edge labels.

One existing application of GRAB is a program browser which displays the call-graph of a program. The nodes of the graph displayed by GRAB represent the procedures of a program and the edges represent the CALLS relationship. The vi text editor has been integrated into the program browser so that GRAB invokes vi with the name of the routine to be edited whenever a node is selected by the user. The major contribution of GRAB is its ability to perform automatic layout of directed graphs and that it may be altered to invoke editors, like vi, to display information about the components of a graph displayed by GRAB.

The graph manager/browser, GMB, [8] has been designed to manage a graph abstract data type called a GMB graph and act as a graph browser to display these graphs. Application programs can use graph manager routines to create and edit GMB graphs. To help manage graphs for different applications, the graph manager can store and retrieve arbitrary information about nodes and edges on an application's behalf. This allows GMB graphs to be tailored to the needs of different applications.

The graph browser portion of GMB displays GMB graphs created by a program. It automatically lays out graphs hierarchically with a minimum number of edge crossings. Programs and users can affect the way graphs are displayed by the browser. Programs can perform the entire graph layout or issue layout directives, such as place nodes n_1 and n_2 at the same level. Layout directives allow programs to control the placement of critical nodes. Labels may be used to display information about nodes and edges using annotations. Annotations are a few lines of text which can be set by programs. Unfortunately, the arbitrary graph, node and edge information that GMB saves for programs cannot be directly displayed by the GMB browser. Associated with every GMB graph and every node and edge is a state variable. The value of a state variable can be set by programs and is used to specify display attributes. Users can associate node and edge type, colour, and visibility (ie. is this node or edge visible?), with state variable values in order to control the appearance of a graph. Programs are able to animate their execution by changing the value of state variables and effectively alter a graphs picture as the program executes.

Graph editors are tools with which graphs may be created and altered. The Graph^{Ed} [7] interactive graph editor allows graphs to be constructed and displays nodes as icons and edges as line segments. Users can create different node types by creating new node icons. Cut, copy and paste operations are supported to make it easier to edit graphs displayed by Graph^{Ed}. Like GRAB and GMB, labels can be displayed for nodes and edges. Graph^{Ed} interacts with programs through an adjacency list type of file format. The editor will display graphs written in this format on behalf of graph programs. The research does not mention if graph programs can directly affect the manner in which

graphs are displayed by GraphEd or respond to user actions through the user interface of GraphEd.

Knowledge-based graph editors are graph editors which,

"... embody a rich and diverse amount of information about how to deal with directed graphs, and because it is possible to alter or extend this information to suit a particular application". [19]

The EDGE [14] graph editor and its predecessor kb-edit [19] are knowledge-based graph editors. Both are editor kernels which can be adapted to new graph applications.

The kb-edit editor performs automatic graph layout for both users and programs. The user interface supports graph editing and graph browsing through popup menus. The user interface may be altered by replacing function calls within the editor, or by adding new functions and new menu items. Kb-edit is adapted to new applications by specifying the shape of icon to be used to display nodes, the type of line to be used to display edges, extending menus to contain new entries, and filling in pointers to routines that are to be called at specific events. User events that can cause the editor to invoke graph application procedures include adding or selecting nodes and edges, and selecting menu entries. Kb-edit is written in Knowledge Craft on Symbolics Lisp machines and relies on inheritance to reduce the amount of work required to alter it to suit the needs of new graph applications.

Because kb-edit's automatic graph layout ran too slowly and was not easily portable to new platforms, its successor, EDGE, was written in C++ with X Windows. EDGE solves the kb-edit performance problems and can be ported easily to most UNIX platforms. Adapting EDGE to suit the

requirements of a new graph application is similar to the method used to alter kb-edit. Programmers can override base functionality of the editor allowing programs to alter operations such as add and delete nodes and edges, edit, draw graph, and select action. To make it easier to alter EDGE for new application instances, a program generator has been created which reads in node and edge class definitions and creates the appropriate constructors and destructors for the new classes and creates appropriate source code for new menu entries.

A graph representation language, GRL, can be used to input graphs to EDGE. GRL permits specification of the nodes and edges of a graph and display information such as labels, node and edge styles, and layout constraints (directives).

The Networks editor [9] is a graph-based modeling system (GBMS) instance editor which allows network modelers to create new instances of graph editors, called scheme editors, to accommodate applications which use graph models. Scheme editors can be used as graphical user interfaces for applications which use a particular type of graph. To manage the node and edge information required by different graph applications, the Network editor enables users to assign variables to node and edge types called attributes. The values of attributes can be set for particular nodes and edges by users with the resulting scheme editor. New operations may be added to a scheme editor and changes may be made to existing editor functionality by linking in external functions.

1.3 Problem Definition

A user interface dictates the manner which input is obtained, output is displayed, and how a user controls a program. The graph browsers and editors described in the previous section can be adapted to suit the user interface requirements of graph applications. Each of the browsers and editors described can be used to obtain input in the form of graphs, draw graphs as output and provide varying degrees of user control over graph programs. Unfortunately, these existing tools must be modified if a programmer wishes to alter the default user control over the graph programs that use them. For example, the Networks scheme editor must have new operations linked into the editor whenever new operations are added to the editor. This results in different versions of the Networks editor for different programs. The Edge editor must have certain methods altered or overridden to allow a graph program to respond to user actions such as node or edge selection.

The goal of this thesis is to overcome the problem, that the application interfaces of existing adaptable graph user interface tools require reprogramming the tools when a programmer wishes to alter the default user control of the program. We wish to present a solution to the problem of how to design an application interface for an adaptable graph user interface tool that provides flexible user control of graph programs but does not rely upon altering the code of the tool.

We propose that an application interface which relies upon message-passing will allow us to completely separate an adaptable graph user interface tool from the programs that use it. Message passing provides a decoupling of the GUI capabilities and the graph programs which use them, allowing

them to co-exist as separate executing processes. Additionally, if the user interface is a program that is not part of the code of the applications it serves, its application interface can't rely upon changing its code. Consequently programmers will not have to change the user interface to use it with a new graph application.

To explore our application interface design, we design and implement an adaptable graph user interface tool, GraphUI, which uses our application interface. The emphasis of the GraphUI is on providing graph user interface capabilities which enable programs to obtain user input, affect the display of output to users and provide users direct access to manage the execution of graph programs using the tool.

The expected benefits of our application interface are: 1) graph application programmers will not have to work in the development environment of the adaptable graph user interface tool; 2) GUI coding issues such as graphics and graph data structures will not impact graph programs that use the tool; and 3) it will be possible to have one centralized user interface from which many graph programs can be simultaneously accessed.

1.4 Thesis Layout

The remainder of this thesis describes our proposed application interface and the adaptable graph user interface tool, GraphUI, which makes use of the interface. The first part of this thesis describes the design and implementation of GraphUI and are found in Chapters 2 and 3 respectively. Included in the description of the design are : the application interface which allows GraphUI to be separated from the applications that use it; the user

interface functionality necessary in GraphUI; the goals of the system design; the manner in which each goal was satisfied; and an overview of the programming model of GraphUI. The object hierarchy and the implementation difficulties which arose are the focus of Chapter 3. A detailed description of the user interface that GraphUI provides is found in Chapter 4. The application interface to GraphUI consists of its command set and the protocol used to interact with it and is described in Chapter 5.

A sample program called GraphApp has been created which illustrates how to use GraphUI's application interface, highlighting both low level communications and a philosophy for using GraphUI. GraphApp provides a number of data structures that manage graphs and routines which communicate GraphUI's command set. The sample program was written for a Macintosh computer and connects to GraphUI through the AppleTalk ADSP connection tool [2]. Chapter 6 describes GraphApp to illustrate how to use GraphUI's application interface. The final chapter discusses contributions and future work planned for GraphUI.

2. Design of GraphUI

The purpose of this chapter is to describe the design of our adaptable graph user interface tool, GraphUI. In particular, we look at the design of the application user interface which allows GraphUI to be separated from the programs that use it. Included is a description of the functionality required for GraphUI, the design goals that GraphUI must satisfy, and a description of the resulting design which meets the identified goals.

2.1 Requirements of GraphUI

GraphUI was created to explore an application interface to graph programs that does not require programming changes to GraphUI. The first requirement that GraphUI must satisfy is providing this application interface. The application interface relies upon message passing to communicate with programs using it. To provide a flexible interface, it is required that GraphUI communicate with programs through a variety of connections to afford programs running on different platforms access to GraphUI. This gives graph programmers the opportunity to work on computers and in development environments that suit the needs of their programs.

GraphUI must provide user interface capabilities to the programs it serves. Rather than focus on one graph operation such as automatic graph layout like similar tools do [7,8,9,14,15,19], GraphUI concentrates on providing a simple means for programs to obtain user input, display output and control

access to the programs. This provides user interface capabilities which are useful to a variety of graph applications including tools which perform automatic graph layout.

It is required that GraphUI provide an intuitive graphical user interface that can be used for graph algorithm programs from different domains. To make it easier for users to comprehend graph algorithms, it is required that programs be able to interactively affect changes to a graph during progress of their algorithms. This permits a user to observe an algorithm by watching the changes. The goal of GraphUI is to manage the user interface for graph programs so that graph programmers can use a graphical user interface but do not have to change GraphUI.

The final requirement of this thesis is to provide an example graph program that uses GraphUI for its user interface. The purpose of the example program is to show graph programmers how to use GraphUI's application interface, and to provide a program that graph algorithms can be added directly to that uses GraphUI.

2.2 Design Goals for GraphUI

The application interface is to be designed such that the programs that use GraphUI do not have to be written in the same development environment or even on the same computer. Thus the choice of computer and development environment of GraphUI will not greatly impact its use by other programs. We choose to implement GraphUI on Macintosh computers to take advantage of their graphical user interface capabilities.

Macintosh computers are renowned for their ease of use. This is partially due to the Macintosh's graphical user interface and moreover to the consistency between applications. A program that follows the Macintosh User Interface Guidelines [1] operates in a manner that is similar to other programs adhering to the guidelines. One design goal is to ensure that GraphUI, as all good Macintosh programs, follows the Macintosh User Interface Guidelines. In following the guidelines, GraphUI ensures that undo of commands is provided, cut, copy, and paste into the clipboard are supported, standard file conventions for creating, opening, printing, and saving documents are supported, area and point selection mechanisms are provided, and printing uses QuickDraw routines for high quality printing. In short, support the things which have made the Macintosh user interface popular. Figure 2.2 shows the difference between a printed bitmap and a picture printed using QuickDraw routines.

Figure 2.1 : QuickDraw versus Bitmap Output

Another design goal of GraphUI is to establish a reasonable and well defined representation for drawing pictures of graphs, similar to those of

existing adaptable graph user interface tools. To provide a clear user interface, the pictures must make sense to the observer.

The final design goal of GraphUI is to make the application interface use a graph command language rather than use a graphical command language. To a person writing a program using the application interface, a graph should be a data structure rather than a picture and this must be reflected in the command language. For example, the application interface should add a node to a graph by sending the command "add node" not "draw circle at (x, y) with radius w". Thus the programmer does not need a thorough understanding of user interface programming and its terminology.

In summary the design goals for GraphUI are as follows:

1. Adhere to the Macintosh User Interface Guidelines;
2. Establish a reasonable and well defined representation for drawing pictures of graphs; and
3. Ensure the application interface uses a graph language that is easily understood by graph programmers.

The remainder of this chapter discusses the manner in which GraphUI has been designed to satisfy the necessary requirements and to meet each of the design goals.

2.3 Communication Design

To satisfy the requirement of providing an application interface which has the ability to communicate with programs through a variety of connections, GraphUI was designed to communicate using the Macintosh

Communications Toolbox Connection Manager [2]. The communications toolbox allows programmers to implement basic connections without being concerned with the underlying connection protocols. Connection tools handle the underlying protocols.

The Connection Manager provides a program with channels that carry data between the program and another computer process running on the same computer or any other computer. GraphUI uses channels to exchange messages by sending and receiving blocks of data. Figure 2.2 shows the logical connection of GraphUI to graph programs through channels.

Figure 2.2 : Logical Connections Between GraphUI and Graph Programs

By allowing the user of GraphUI to interactively create connection channels that use different connection tools, the application interface of GraphUI has the potential to communicate with programs running on a variety of computers without altering GraphUI. If the connection tools provided with the Communications Toolbox do not handle the underlying protocol required to connect to a particular computer, a connection tool may be written to handle the protocol without changing the code of GraphUI.

2.4 User Interface Design

The following section describes the design of GraphUI with respect to the user interface. It concentrates on the representation used in drawing graphs and the operations that users are able to perform using GraphUI. Drawing and manipulating graphs is a necessary component in any graph user interface tool. The operations which we permit users are straightforward and designed to make it easy for users to create graph input models for graph programs and to access programs using the application interface transparently.

2.4.1 Drawing Conventions

Graphs are drawn in windows. A separate window is provided for each graph thereby allowing multiple graphs to be displayed simultaneously. As far as the user is concerned, a graph is a window containing a picture of the graph, see Figure 2.3. The physical entity that a graph models can be a map of highways connecting cities, a telephone network, a person's family

history, or many other things. Rather than attempt to accommodate all different graphical representations of nodes and edges, GraphUI uses a fixed representation.

Figure 2.3 : Pictorial Representation of a Graph

All nodes are represented by circles, all edges by lines. It is possible to have more than one edge between a pair of nodes. To ensure that all edges are visible, multiple edges between a pair of nodes are drawn by bending the additional edges between a pair of nodes, see Figure 2.4.

Many graphs distinguish between arcs and edges, arcs have direction whereas edges do not. To illustrate that an arc is directed and thus not an

edge, an arrow is drawn on the arc, pointing to its head node. Figure 2.5 illustrates the representation of arcs.

Figure 2.4 : Drawing Syntax for Multiple Edges Between Two Nodes

Figure 2.5 : GraphUI's Drawing Representation of Arcs

When explaining an algorithm using a picture of a graph, it is often necessary to distinguish particular nodes or edges. Examples of this are marking the edges that form a cutset isolating two nodes and distinguishing the source node of a search operation. The representation used to indicate that a node or edge is marked, or selected, is to draw the node or edge in gray rather than black.

It is likely that two states, marked and unmarked, are not enough to convey detailed information about nodes and edges. For this reason, the sizes of nodes and edges are variable, allowing nodes to have a radius from 0 to 20 and edges to have a thickness of 1 to 10. Figure 2.6 shows two graph components, the left component has edges with thickness 1 and nodes of radius 0, the right component has edges with thickness 10 and nodes of radius 20.

Figure 2.6 : Minimum and Maximum Node and Edge Sizes in GraphUI

The final means of distinguishing nodes and edges drawn with GraphUI are labels. There is a mechanism with which labels can be drawn for any node or edge. Any alphanumeric value is a valid label. Figure 2.7 shows a graph in which nodes are labeled with the city they represent and arcs are labeled with the distance between the cities.

Figure 2.7 : A Graph With Node and Edge Labels

2.4.2 User Interface Operations

The set of operations a user can perform on a graph has been designed to allow users to create, edit, save, and print graphs. The manner in which a user performs the functions is left to the user interface description in Chapter 4.

To create a graph the user adds nodes and edges. GraphUI requires that an edge have a distinct tail and head node when it is added. Once created, the user can modify a graph. Two different types of graph modifications can be made using GraphUI.

The first type of graph modification changes the topology, or structure, of a graph. The most obvious way to change the structure is to add additional nodes and edges. The structure of a graph can also be modified by deleting nodes and edges. Since an edge cannot exist without both its head and tail nodes, deleting a node also deletes all edges connected to it. Adding and deleting nodes and edges are the only two fundamental ways to alter the structure of a graph. There is no structural alteration that cannot be simplified into some combination of adds and deletes.

The second type of graph modification is to change the graph's appearance but not its structure. One method of altering a graph's appearance is to reposition its nodes. Since an edge is a connection between two nodes, a graph is repositioned by moving nodes; the edges attached to a node always follow it. To further alter the appearance of a graph, users are allowed to set the size of nodes and edges, change labels, mark, or select, edges and nodes, and show or hide the direction of edges.

Once created, graphs are stored in application documents which the user can save or print. Since GraphUI is a Macintosh application, users can copy graphs to the clipboard and paste them directly into word processing or drawing package documents.

GraphUI includes commands to contract edges and to complete a graph. These two graph operations were included in GraphUI because they are often used to construct graphs and were viewed as user interface short cuts.

2.5 Application Interface Design

In order to create our application interface we must know the functionality that GraphUI provides applications. In this section we discuss the user interface capabilities of GraphUI and in chapters 5 and 6 we discuss exactly how our application interface affords programs these capabilities.

GraphUI was required to provide a graphical user interface which can be used by a variety of graph programs. A user interface is the method in which a user interacts with a program. The responsibilities of a user interface are: to manage the way input is entered by the user, to manage access to the functions of the program, and to manage the display of output presented to the user.

To provide a user interface, there are four things that GraphUI does for a graph algorithm program: obtain input, display output, animate algorithms, provide users access to the graph program's algorithms. Each of these is accomplished with a command language.

To ensure that GraphUI can accommodate graph programs from different domains of graph theory, it has been designed to manage only the user interface. User interface tools such as GMB [8] or EDGE [14,19] could make use of GraphUI to provide the user interface to their programs which perform automatic graph layout. GraphUI does not include graph algorithms or enforce a specific graph data structure because these depend upon the requirements of individual graph algorithms.

2.5.1 Graph Information Managed by GraphUI

Graph programs are able to obtain their input from GraphUI. Most graph algorithms require a graph as input but the necessary graph information differs from algorithm to algorithm. Because GraphUI is designed to work with graph algorithms from different domains, it is not possible to accurately predict all possible input requirements.

To accommodate differing input requirements, GraphUI manages graph information in the following manner. All graphs, nodes, and edges have a set of static attributes maintained by GraphUI. The attributes are composed of information commonly used in graph algorithms. In addition to the static attributes, are four general purpose attributes: two integers and two real numbers are available. These predefined general purpose attributes are used to avoid altering GraphUI to include new attributes required by a specific graph algorithm. They allow users to set and observe values of attributes that programs require unlike the attributes implementation in GMB which only give programs access to the attributes [8]. Finally if the static and general purpose attributes do not provide all of the information required by an

algorithm, a means of obtaining integers and real numbers directly from the user is provided.

The attributes which describe graphs, nodes and edges are as follows in Tables 2.1, 2.2, and 2.3.

Attribute	Definition
Generic Integer 1	a general purpose integer
Generic Integer 2	a general purpose integer
Generic Float 1	a general purpose real number
Generic Float 2	a general purpose real number

Table 2.1 : Definition of Graph Attributes

Attribute	Definition
probability	the probability of the node
label	which of the attributes of the node should it be labeled with
name	text string representing the name of the node
radius	radius of the circle used to represent the node when it is displayed
cost	a real number cost of the node
capacity	a real number capacity of the node for use with flow problems
weight	a real number that is generally associated with the value of a particular node
visited	has this node been visited?
source	is this node a source node?
target	is this node a target node?
Generic Integer 1	a general purpose integer
Generic Integer 2	a general purpose integer
Generic Float 1	a general purpose real number
Generic Float 2	a general purpose real number

Table 2.2 : Definition of Node Attributes

Attribute	Definition
probability	the probability of the edge
label	which of the attributes of the node should it be labeled with
name	text string representing the name of the edge
thickness	the thickness of the line used to represent the edge when it is displayed
head node	a reference to the head node of the edge
tail node	a reference to the tail node of the edge
distance	a real number normally used to represent the cost of travelling the edge or the length of the edge. An edge's distance does not have an affect on the length of line used to represent the edge.
flow	a real number capacity of the node for use with flow problems
cost	a real number cost of the edge
capacity	a real number capacity of the edge for use with flow problems
is directed	is this edge an arc? If it is it will be drawn with an arrow pointing to its head node
Generic Integer 1	a general purpose integer
Generic Integer 2	a general purpose integer
Generic Float 1	a general purpose real number
Generic Float 2	a general purpose real number

Table 2.3 : Definition of Edge Attributes

2.5.2 Design of GraphUI's Command Language

GraphUI has been designed to communicate with graph programs using bi-directional channels through which blocks of data may be exchanged. These channels are used to communicate two different types of information between graph programs and GraphUI. The first type of information that is communicated are requests sent from graph programs to GraphUI. These requests are called commands and are used by graph programs to control the

operation of GraphUI. Commands allow programs to control the display of information to users and to manage the input which is required from users. It is the command set which determines the capabilities GraphUI affords programs and as a result is fundamental to the application interface.

The second type of data that is communicated are information messages and are sent from GraphUI to graph programs. Information messages are included so programs can monitor the effect users and programs have on graphs managed by GraphUI. These messages are intended for those programs wanting to maintain up-to-date copies of the graphs open in GraphUI and those programs wishing to react to user actions or changes in the graphs displayed by GraphUI. It is possible to react to changes which are made by programs as well as those made by users thus allowing different applications to be integrated. For example it is possible for an automatic graph layout program to react to the addition of a new node from a project management program by redrawing the graph according to layout constraints.

Information messages provide programs with control similar to the control provided in EDGE and kb-edit which allow routines to be invoked when particular use actions occur [14,19]. These routines however must be added directly to the editors in order to invoke them.

It was previously noted that GraphUI provides programs with a means of obtaining input, displaying output, animating algorithms, and making a program's algorithms accessible to users. The following section discusses the types of commands included in GraphUI which enable it to manage user interface requirements of graph programs. The actual command language is found in the application interface description in Chapter 5.

Obtaining Input:

Input to a graph algorithm is very often a graph. Unfortunately the structure of the graph (i.e. the manner in which nodes are connected by edges) is often not sufficient information for an algorithm to run. To allow programs to obtain the input they require, GraphUI has a command that allows a program to obtain a description of a graph, including the attributes of each component of the graph. Also included in GraphUI are commands which allow a program to directly prompt the user to enter specific information that is not contained in the attributes. The adaptable user interface tools discussed in section 1.1.3 allowed for graphs as input but made no note of providing a means for programs to obtain information that was not part of their graph data structures.

The way in which a program interprets the values of the attributes does not matter to GraphUI. Programs must be documented as to what values they expect attributes to have before and after an algorithm has run to enable users to set the attributes correctly and to interpret the output of the algorithm. GraphUI allows both users and programs to view and set the attributes values.

Messages provide a different means of obtaining input. Messages inform a program when a graph has been altered. This includes any additions, deletions, node movements, selection of edges and nodes, and changes in attributes of nodes, edges, and graphs. A program can monitor these messages and keep an accurate account of the graphs displayed by GraphUI and to react appropriately to changes.

Displaying Output:

Output from a graph program is often a graph data structure. To make it easier for a user to interpret output, GraphUI allows programs to display output as the pictorial representation of a graph. The user interface graph manipulation commands that were discussed in section 2.5.2 are available to programs. GraphUI's command language contains commands for creating graphs and modifying both the structure and appearance of graphs. A picture of a graph may not be sufficient to represent all the results of an algorithm. To compensate for this, GraphUI allows a program to display text to the user. A text output window provides a place for algorithms to display additional information in textual format.

Altering Graphs from Programs:

One of the requirements of GraphUI is to allow a graph program to interactively alter a graph displayed by GraphUI while an algorithm is running. GMB allows programs to do this through state variables which can be set to particular display attributes [8]. We do this using graph editing commands of GraphUI which allow both a graph's structure and its appearance to be changed. Structure editing commands let programs add and delete edges, nodes, and graphs, and contract edges which are echoed to the user through the picture of the graph. These commands can be used to show how an algorithm is affecting a graph's structure. They may also be used to build a new graph to reflect what the algorithm is doing.

The physical attributes of nodes and edges such as size, position, and colour, black and gray, can be changed by a program through a number of commands that alter the appearance of graphs. Labels of nodes and edges can

be changed to new values through commands to relay information when a program runs. For example a shortest path algorithm can display the cost of reaching each node in the node's label as it is calculated. This shows not only the cost, but the order in which each node is reached. Programs can choose to label nodes and edge with any alphanumeric string or they can label nodes and edges with any attribute of the node or edge. Allowing programs to label nodes and edges with their attributes makes it easy for programs to draw users attentions to attributes which are important to the program. Programs can also move nodes to new positions in a number of steps thereby making the movement more apparent to the user and control the placement of nodes on the display.

By supplying commands that alter the user's picture of a graph, GraphUI provides graph programmers basic animation capabilities. Algorithm animation in itself is an expansive area of research [5]. There is no one agreed upon best way to animate an algorithm so that it is clear to a user what the algorithm is doing but it is agreed that animation aids in user comprehension.

Accessing Algorithms From GraphUI:

GraphUI provides users access to a graph program's algorithms. To do this GraphUI provides a menu in which graph programs can post the names of their algorithms. The menu is built dynamically, permitting programs to add and remove algorithm names from the menu as they execute. When a user selects an algorithm from the menu, a message is sent to the corresponding program requesting the algorithm be run for the user.

Only some algorithms need a graph as input. To help manage availability of algorithms, a program is able to specify whether an algorithm requires a graph as input. If a graph is required, the algorithm is only selectable if there is a graph displayed by GraphUI.

2.6 GraphUI Programming Model

In the previous sections of this chapter the manner in which GraphUI was designed to meet the design goals and to satisfy the functional requirements was described. This chapter closes with a description of the programming model used to construct GraphUI. GraphUI is logically divided into four components, a User Interface Interpreter, a GraphUI Command Interpreter, a Graph and Display Manager, and a Communication Manager, see Figure 2.8. These components together form GraphUI.

The Communication Manager is required to manage low-level communications between GraphUI and graph programs. It is responsible for receiving commands, sending replies and sending information messages. When commands are received it is the job of the Command Interpreter to process the command and perform the requested operation. The Command Interpreter was needed to translate program requests for the Graph and Display Manager.

The Graph and Display Manager controls all drawing of graphs for users and programs alike. This component is necessary to manage the appearance of the graph and its internal data structure. The User Interface Interpreter translates user interactions with the computer, such as mouse

movements, mouse clicks, and key presses, into display commands processed by the Graph and Display Manager.

Figure 2.8 : Model of the GraphUI Program

3. Implementation of GraphUI

3.1 Introduction

To make it easier to adhere to the Macintosh user interface guidelines, GraphUI was written in C++ using the MacApp® application framework. MacApp is a framework of Object Pascal code that helps to manage Macintosh programs. Since MacApp provides a frame for constructing applications, including basic window management, mouse handling, and menu management, it is difficult to describe GraphUI's implementation without elaborating on intricacies of MacApp. A thorough discussion of MacApp is beyond the scope of this thesis but several books are available on the subject [3,21]. Included in this chapter is a description of the object hierarchy used in implementing GraphUI and a discussion of some of the portions of code that were particularly difficult to implement. This chapter provides insight into the way GraphUI has been constructed without concentrating on Macintosh specific issues.

3.2 Object Hierarchy

Some of the objects used to build GraphUI were MacApp objects, others were subclasses of MacApp objects created to handle the specific needs of this program. Twenty five objects with a total of over three hundred methods

were created in the development of GraphUI. Of these classes there are a number which warrant explanation. Table 3.1 contains all of the objects in GraphUI and their superclasses. Superclasses in plain type are MacApp superclasses; those in bold are GraphUI superclasses. The objects that have an asterisk beside their entry in Table 3.1 are described in this section.

Object Class :	Superclass :
TAlgorithm*	TObject
TGraphUIIcon	TIcon
TGraphUIPalette	TControl
TCompleteGraphCommand	TCommand
TConnection*	TObject
TContractEdgeCommand	TCommand
TDeleteCommand	TCommand
TDragNodeCommand	TCommand
TEdgeAttrDialog	TDialog
TEdgeView*	TGraphComponentView
TGraphAttrDialog	TDialog
TGraphComponentView*	TView
TGraphUIApplication*	TApplication
TGraphUIDocument*	TDocument
TGraphUIWindow	TWindow
TGraphView*	TView
TLabel	TStaticText
TMakeEdgeCommand	TCommand
TMakeNodeCommand	TCommand
TNodeAttrDialog	TDialog
TNodeView*	TGraphComponentView
TRealText*	TEditText
TSelectAllCommand	TCommand
TSelectorCommand	TCommand
TSet*	TList

Table 3.1 : Object Classes Within GraphUI

* - Discussed in this section

Bold - GraphUI Superclasses

TGraphUIApplication : its purpose is to manage anything that the application should be able to do regardless of whether it has any documents (graphs) opened. It handles opening documents, creating documents, managing open and close of connections with other programs, acting on GraphUI commands received from connected programs, sending messages to all connected programs, and giving connections and the connection manager time to manage themselves.

TGraphUIDocument : its purpose is to manage the data of a document. This management includes reading and writing documents to disk, creating the necessary views required to display a graph when it is opened, printed, or copied to the clipboard, and managing user access to commands graph editing commands.

TGraphView : its purpose is to manage the display of graphs within GraphUI's windows. Management of the display includes creation of nodes and edges, selection of nodes and edges by point and area selection, and handling the user feedback when nodes are dragged. The graph view is a part of the window that contains a graph. TGraphView contains any number of subviews which are the components of the graph. Currently there are only two components, nodes and edges, implemented respectively as TNodeView and TEdgeView.

TGraphComponentView : is an abstract class which is the superclass of all components of a graph. A graph component is any part of the graph that is displayed, not including labels. Currently there are two components, nodes

and edges. If different classes of nodes and edges are implemented they should be subclasses of TGraphComponentView. TGraphComponentView handles common features of nodes and edges including determining if a node or edge has been selected by the user and what happens if a component is dragged by a user.

TNodeView : its purpose is to manage a node that is displayed on the screen. It takes care of deleting and undeleting itself, managing its memory, and displaying itself as specified by a user or program.

TEdgeView : its purpose is to manage an edge that is displayed on the screen. It takes care of deleting and undeleting itself, contracting and uncontracting itself, managing its memory, and displaying itself as specified by a user or program.

TSet : is an extension of the MacApp TList object. Sets and lists are used to store objects that are related. A set differs from a list in that duplicate objects are not permitted in a set. Provided with TSet are the common set operations membership, union, intersection, and difference. A useful method of TSet, called MakeClassSubSet is used to create a new set containing all of its elements of a particular object class.

TRealText : MacApp provides object classes to help obtain and edit user input. There is a TEditText object used to obtain text from a user. The TNumberEdit object is used to obtain an integer from a user. TRealText was created as an extension to TEditText to allow users to enter real numbers as text.

TAlgorithm : its purpose is to hold information about an algorithm appearing in GraphUI's algorithm menu. The algorithm information is the name of algorithm to appear in the menu, the connection to the program which owns the algorithm, the ID of the algorithm, a flag determining if there must be a graph to run the algorithm, and a flag indicating if the algorithm name has been added to the algorithms menu. All of the algorithms displayed by GraphUI are placed into a set managed by TGraphUIApplication.

TConnection : is an object for managing connections created with the Communications Toolbox. This object handles the low-level sending and receiving of data for GraphUI on behalf of TGraphUIApplication.

3.3 Notable Implementation Details

In this section some of the more interesting aspects of the code are discussed. It is not meant to document the entire system; rather to discuss some of the novel parts of the code.

3.3.1 Communications

Low-level communications are implemented with TConnection objects. To understand the workings of connection objects we must look at the physical means used by GraphUI to communicate with other programs.

The Macintosh Communication Toolbox Connection Manager [2] allows programs to be written independent of how computers are connected. Communication is through abstract connections called channels which provide bi-directional data streams. Programs manage reading from and writing to channels and the connection manager uses connection tools to transmit data between the two ends of a connection. The Communications Toolbox provides three standard connection tools: a modem connection tool, a serial connection tool, and an AppleTalk ADSP connection tool. Facilities are provided to help write customized tools to work with the Communications Toolbox. GraphUI maintains one connection for each program with which it communicates. Each connection is managed by a separate TConnection object. All of the open connections are stored in a set of active connections stored as an instance variable of the application. When a connection is closed by either GraphUI or the program on the other end, it is removed from the set of active connections. New connections are always added to the set. Each time through GraphUI's main event loop it tests for newly opened and newly closed connections. Processing time is provided for each connection allowing it to manage itself.

Reading from and writing to connections is done asynchronously by the TConnection object. This object sends replies and messages for GraphUI and receives commands from connected graph programs. When it receives a command, the command is passed to a TGraphUIApplication method, called ExecuteCommand, for processing.

3.3.2 Point Object Selection

Point selection is done by moving the cursor over a graph component and clicking. To make it easier to determine if a mouse click in the graph view is on a graph component, `TGraphComponentView` uses a method called `IsPointOnView`. This method is passed a mouse position and returns true if the mouse is in the view.

The default `IsPointOnView` method returns true if the mouse point is within the extent of the view. This default is used for nodes because nodes occupy most of their extent, see Figure 3.1. Edges must use a different technique because simply being within the extent is not accurate enough, see Figure 3.2. The method that is used is taken from "C++ Programming With MacApp" [21].

The `IsPointOnView` method of an edge, draws the edge in off screen memory to determine if the current mouse position is on the edge. If the pixel corresponding to the mouse position is set in memory, the edge has been selected. To make it easier to hit thin edges with the mouse, edges with thickness less than 4 are drawn with thickness 4.

Figure 3.1 : Node Extent

Figure 3.2 : Edge Extent

3.3.3 Dragging Nodes

The method GraphUI uses to drag nodes deviates from the way which is recommended by the user interface guidelines. The user interface guidelines suggest that an object be dragged by leaving the object where it is and moving around a gray rectangle of the area the object occupies. This was not seen as satisfactory because when a node moves, the edges attached to the node must move. Nodes are usually moved to reposition the graph and make it look better. To make it easier to reposition the graph, a node and its edges are dragged, not an outline of the affected area.

3.3.4 Drawing Edges

Most of the drawing in GraphUI is straightforward; circles are used to represent nodes and edges are represented as lines. The challenge comes when drawing the bent lines that are used to represent multiple edges connecting a single pair of nodes.

The ideal way to draw the bent lines used to represent multiple edges between a pair of nodes, such as those in Figure 3.3 is as follows. Compute the midpoint of the straight line connecting the head and tail nodes of the edge. Compute a point (x,y) from which a straight line connecting (x,y) and the midpoint is perpendicular to the straight line and has a fixed length l , see Figure 3.4. To draw the bent edge, draw a line from the head node to (x,y) and a line from (x,y) to the tail node. When drawing more than two edges between the same pair of nodes, l is made positive and negative to bend the

lines on either side of the edge, and l is increased by a fixed amount, see Figure 3.5.

Figure 3.3 : Representation of Multiple Edges

Figure 3.4 : How a Bent Edge is Drawn

Figure 3.5 : A Pair of Nodes With Many Edges Connecting Them

There was a practical problem with using this method. In order to compute (x,y) , the bendpoint, the following equation had to be solved:

$$l = ((x \text{ coord of midpoint} - x)^2 + (y \text{ coord of midpoint} - y)^2)^{0.5}$$

Since l is known, y or x can easily be computed by fixing the other. When a node is dragged by the user, the edges connected to the node are redrawn many times. Unfortunately the square root calculation is slow. When a node is dragged the recalculations of the bendpoints slows the feedback of the dragging so much that the redrawing cannot keep up with the user.

As a practical solution to the problem of drawing bent edges, the following approach is taken in GraphUI. The calculation of (x,y) is approximated. The first step computes the slope and the midpoint of the straight line from the head to the tail node. If the line has a slope of between -0.5 and 0.5, the bend point is computed by adding a constant amount to the y coordinate of the midpoint, thus raising the bendpoint directly above the line, see Figure 3.6. If the slope of the line is greater than 2.0 or less than -2.0, the bend point is computed by subtracting a constant amount from the x coordinate of the midpoint, thus moving the bendpoint directly to the left, see Figure 3.6. If the slope of the line is between 0.5 and 2.0, the bendpoint is computed by subtracting a constant amount from the x coordinate of the midpoint and adding a constant amount to the y coordinate of the midpoint, see Figure 3.6. This moves the bendpoint counter clockwise from the midpoint, above and to the left of the line. Finally, if the slope of the line is between -2.0 and -0.5, the bendpoint is computed by adding a constant amount to the x coordinate of the midpoint and adding a constant amount from the y coordinate of the midpoint, see Figure 3.6. This moves the bendpoint counter clockwise from the midpoint, above and to the right of the line. See also Figures 3.7.

Figure 3.6 : Computing the Bendpoint of Bent Edges

```

if  $-5 < \text{slope} < 5$  then the bend point is
     $\text{midPoint.h} + 0, \text{midPoint.v} + \text{Amount to bend by}$ 

if  $5 < \text{slope} < 2$  then the bend point is
     $\text{midPoint.h} - \text{Amount to bend by}, \text{midPoint.v} + \text{Amount to bend b}$ 

if  $2 < \text{slope} \vee \text{slope} < -2$  then the bend point is
     $\text{midPoint.h} - \text{Amount to bend by}, \text{midPoint.v} + 0$ 

if  $-2 < \text{slope} < -5$  then the bend point is
     $\text{midPoint.h} + \text{Amount to bend by}, \text{midPoint.v} + \text{Amount to bend b}$ 

```

Figure 3.7 : Algorithm Used to Determine the Bend Point of an Edge

The constant amount that an edge is bent is altered from a positive to a negative number to make the edges bend on either side of the straight line connecting the head node and the tail node and is increased to move additional edges farther and farther from the midpoint, see figure 3.5.

3.3.5 Building the Algorithms Menu

The algorithms menu is altered dynamically when programs want algorithms added or removed from the menu. The menu is rebuilt in the `DoSetupMenus` method of `TGraphUIApplication`. In `DoSetupMenus`, the algorithms menu is reset, removing all of the menu items from the menu. In resetting the menu, each algorithm in the set of registered algorithms has its `fAddedToMenu` flag assigned false to indicate that it has not been added to the menu. Next each algorithm in the set of registered algorithms is added to the menu. The menu item number which is the number passed to `DoMenuCommand` when the user selects the algorithm, is computed. Each menu item is enabled or disabled depending on its need for a graph and if there is one active.

The interesting calculation is that of the menu item number for each of the menu entries. A menu item number is the menu number multiplied by 256 plus the position at which the item was placed in the menu. The menu is set up each time an algorithm is added or deleted by a program. What follows is the section of code for building the menu from the set of registered menus.

```
// if there are any algorithm names to be displayed, enable and setup the menu
if (fAlgorithms->GetSize() > 0) {
    aMenuHandle = MACGetMenu(mAlgorithms);    // get reference to the algorithms menu
    EnableItem(aMenuHandle, THE_MENU);         // enable the menu
}
```



```

// for each algorithm name to be displayed, compute its menu item number
for (int i = 1; i <= fAlgorithms->GetSize(); i++) {
    anAlgorithm = (TAlgorithm *)fAlgorithms->At(i);

    if (!anAlgorithm->fAddedToMenu) { // has the algorithm been added yet?
        anAlgorithm->fAddedToMenu = TRUE; // if it has not, add it.
        anAlgorithm->Lock(TRUE);
        pstrcpy(aName, anAlgorithm->fName);
        anAlgorithm->Lock(FALSE);
        AppendMenu(aMenuHandle, aName); // adds the name to the menu
        anAlgorithm->fAlgorithmNumber = -(mAlgorithms*256 + i); // ** menu item number
    } // if (!anAlgorithm->fAddedToMenu)

    if (anAlgorithm->fNeedsGraph) {
        if (the active window is a graph)
            EnableItem(aMenuHandle, i); // enable the menu item - there is a graph
        }

        else { // there are no graphs, don't enable the algorithm
            DisableItem(aMenuHandle, i); // disable the menu item - there is no graph
        }
    }

    else { // does not need the graph so enable the item
        EnableItem(aMenuHandle, i);
    }

} // for (int i = 1; i <= fAlgorithms->GetSize(); i++)

} // if (fAlgorithms->GetSize() > 0)

```

4. User Interface

GraphUI can be used by both people and programs. Although both can use GraphUI to perform similar tasks, the interaction with GraphUI is different. Programs use our application interface to interact with GraphUI while people use the keyboard and mouse.

This chapter explores how GraphUI is used by a user. It is assumed that the reader is familiar with using Macintosh programs and a number of Macintosh conventions such as dragging and resizing windows are not discussed. The chapter is divided into sections describing how to use GraphUI with respect to building and modifying graphs, printing and copying to other types of documents, and accessing graph programs from GraphUI.

Although the user interface is not part of the research problem addressed by this thesis, we feel it is necessary to include a description of the user interface to provide readers with a clear understanding of the interaction between GraphUI and application programs by showing how a user uses GraphUI to access other programs.

4.1 Creating Graphs

When GraphUI is launched a window similar to the one in Figure 4.1 is opened. The *palette* that appears down the left side of the window is used to control the action taken when the mouse is pressed in the graph view.

There are three icons which from top to bottom are: pointer, node, and edge. The icon which is active is drawn in white with a black background - the others appear in black with a white background. There can only be one icon active at a time. To make an icon active, press the mouse over the corresponding icon.

Figure 4.1 : Empty GraphUI Window

When the pointer icon is active, nodes and edges can be selected. When the node icon is active, nodes can be created, and when the edge icon is

active edges can be created. When the pointer, node, or edge icon is active, GraphUI is respectively in *pointer mode*, *node creation mode*, or *edge creation mode*.

The area that appears to the right of the palette is the *graph view* in which graphs are drawn. This is referred to as simply the graph. The scroll bars that are along the right and bottom of this window are used to position the graph. The window is similar to many Macintosh windows with a document title, close box, zoom box and resize box.

Creating Nodes

To create a node in the graph, enter node creation mode by selecting the node icon from the palette. Position the cursor at the point in the graph to place the node and press the mouse button. This will create a node centered at the mouse position. Figure 4.2 shows a graph with one node. The number appearing just below the node is its unique ID. This label may be altered to show different information about the node. Node labels are described in detail in section 4.3. The create node operation can be undone by selecting *Undo* from the Edit Menu before another *undoable* command is issued. Once undone, the node can be brought back by selecting *Redo* from the Edit Menu.

Figure 4.2 : Graph With One Node .

Creating Edges

Edges are created while in edge creation mode. An edge is created by positioning the mouse over the tail node of the edge and pressing the mouse button. With the mouse pressed, move the mouse so that it is positioned over the desired head node and release the mouse button. Figure 4.3 shows an edge as it is being created; Figure 4.4 shows the finished edge whose tail node has ID = 1 and head node ID = 2. The number which appears just below the middle of the edge is its label displaying the ID of the edge. This label can be changed to show different information about the edge and is described in detail in section 4.3. Multiple edges between a pair of nodes are "bent" so that all edges are visible, see Figure 4.5. The create edge operation is undoable from the Edit Menu.

Figure 4.3 : What the User Sees when Creating an Edge

Figure 4.4 : GraphUI's Representation of an Edge

Figure 4.5 : Multiple Edges Between a Pair of Nodes

Completing Graphs

To make it easier to create graphs, a method for completing a graph has been provided. A complete graph is a graph in which there is an edge between every pair of nodes. To complete the graph shown in Figure 4.6, select the Complete Graph command from the Edit menu of GraphUI. Figure 4.7 shows the resulting complete graph. Please note that this command completes a graph as if it were undirected; if the graph is considered directed, additional edges must be added manually.

Figure 4.6 : Completing a Graph

Figure 4.7 : A Complete Graph on 6 Nodes

Saving and Closing Graphs

Once a graph has been created it can be saved in a GraphUI document for later use. To save a graph, use the Save or Save As command in the File Menu. These commands open a Macintosh Standard File Save Dialog for saving the Document. Notice that once a graph has been saved, the name in the title bar changes to the name of the document the graph was stored in. When a graph is closed or GraphUI is terminated, the user able to save any graph which has been altered since it was last saved. A graph is closed by clicking on the close box or selecting the Close command from the File Menu. GraphUI is terminated by selecting Quit from the File Menu.

Opening Graphs

Opening a previously saved graph can be done in two ways. A graph will be opened and GraphUI launched if a GraphUI document is double clicked with the mouse from the Finder or the MultiFinder. To open a graph once GraphUI is running, select the Open command from the File Menu. This brings up a Macintosh Standard File Open Dialog box. An empty graph such as the one in Figure 4.1 is opened when GraphUI is launched without a document and when New is selected from the File Menu.

4.2 Modifying Graphs

Once a graph has been created, it is often necessary to edit it. The structure of a graph can be changed by adding nodes, deleting nodes, adding edges, and deleting edges. Nodes can be repositioned in the graph, not affecting the graph's structure, only its appearance. This section describes the ways that GraphUI allows a user to modify the structure of a graph. It also details how to alter the graph's appearance by moving nodes and edges and how to select nodes and edges of a graph.

Selecting nodes and edges change their appearance and indicate which nodes and/or edges commands are to be operated on. The standard cut, copy and paste commands can also be used to modify a graph. These commands are included in the description of clipboard support found in section 4.5.

Selecting Nodes and Edges

The nodes and edges of a graph must be *selected* to indicate which components a particular operation is to be performed on. Any number of nodes and edges may be selected at the same time. To show that a particular node or edge is selected, it is drawn in gray rather than black and its label is written in italics. The set of selected nodes and edges is called the *selection set*. There are two methods provided in GraphUI for selecting nodes and edges, range selection and point selection. Both selection methods require GraphUI be in pointer mode.

Point selection is the simplest method of selection: a node or edge is selected by clicking on it. Clicking on the label of a node or edge does not select it. Range selection is done by selecting any node or edge completely enclosed in an area specified with the mouse. To specify the area, click the mouse at one corner of the area and drag the mouse to the opposite corner of the area. The enclosing area is shown by a rectangle that appears as the mouse is dragged. Any node or edge that is completely within the rectangle is selected. Figure 4.8 shows an example of range selection.

The selection set may be altered to add or remove nodes and edges. Clicking on a node or edge with the shift key pressed reverses its selection state, selecting it if it was not selected and unselecting it if it was selected. Using the shift key in conjunction with range selection toggles the selection state of each node or edge within the selected area.

Figure 4.8a : The Feedback Displayed While a User Makes a Selection

Figure 4.8b : The Selection Set Created in Figure 4.8a

Repositioning Nodes

GraphUI must be in pointer mode to reposition nodes. To move a single node, click the mouse on the node and with the mouse button down drag the node to its new position and release the mouse button. As the node is being dragged, all of the edges attached to the node are moved. Neither node nor edge labels are shown while the node is dragged but reappear once the move is complete.

If nodes or edges were selected and the node being dragged is not selected, the selected items are unselected before the drag operation begins. If the node that is being dragged is selected and there are other selected nodes, all of the selected nodes are moved. The nodes that are selected move relative to the node being dragged, so that they remain the same distance from the dragged node. If the shift key is pressed at the start of the drag operation, its action is applied before the drag begins. This means that if the node was selected, it is unselected first and then dragged, resulting in it being the only node moved. An unselected node is selected first and then it and any other selected nodes are moved.

If the repositioning was not satisfactory, the operation can be undone by selecting Undo from the edit menu.

Deleting Nodes and Edges

Deleting a node removes not only the node but all edges connected to the node because an edge cannot exist without both its head and tail nodes. Deleting an edge only removes the edge. To delete both nodes and edges from a graph, select the nodes and edges to be deleted and press the delete or backspace key or select Clear from the Edit Menu. To delete only selected

nodes, issue the Delete Node(s) command from the Edit Menu and to delete only selected edges, issue the Delete Edge(s) command from the Edit Menu.

Contracting Edges

The Contract Edge(s) command in the Edit Menu is used to contract selected edges. If one edge is selected, it is contracted by dragging the tail node so that it is on top of the head node in six steps to animate the drag, deleting all edges that connect the head node to the tail node, moving all of the tail node's remaining edges to the head node, and deleting the tail node. When the contract is undone, the uncontract operation is done in the reverse order of the contract. If multiple edges are selected, each edge is contracted in the order they were selected starting with the first selected edge. If one of the selected edges is deleted as a result of a previous contraction, it is ignored by the contract operation. Undoing a contract with several edges, uncontracts the edges in the reverse order they were contracted in. Figure 4.9a and 4.9b shows a graph before and after edge 4 is contracted.

Figure 4.9a : Graph Before Edge 4 is Contracted

Figure 4.9b : Graph After Edge 4 is Contracted

4.3 Displaying and Setting Attributes of Graphs

The structure of a graph is described by the way nodes are connected by edges. Algorithms which require graphs as input often require more than structural information. Information specific to a graph, a node or an edge is managed by GraphUI through sets of attributes. This information can be accessed from programs using GraphUI.

The following describes how information can be viewed and modified by a user, enabling a user to enter input and view output from algorithms. A complete discussion of the information managed by GraphUI for graphs, nodes and edges is found in Chapter 2. The information is saved when a graph is saved and does not get reset when graphs are opened unless the user specifically changes the values.

Graph Attributes

The information about a specific graph is called the graph's *attributes*. The attributes of a graph are initialized from a set of default graph attributes when a new graph is created. The default values for new graphs can be set by selecting the Default Graph Values command from the Info Menu. This command opens a *change graph attributes* window as in Figure 4. 10.

From a change graph attributes window any of the attributes can be altered by selecting the attribute and entering the new value. The four graph attributes are general purpose variables. To provide a more accurate description of what the attribute is used for, the labels of these four attributes can be changed by selecting the text of the label and typing a new name. Once

the attributes have been set to the correct values, clicking OK sets the default graph attributes to the values of the attributes in the window, clicking CANCEL leaves the default graph attributes as they were before the window was opened.

The image shows a dialog box titled "Information For New Graphs". It contains four input fields for graph attributes. The first two are integers, both set to 0. The next two are floats, both set to 0.0000. At the bottom, there are two buttons: "OK" and "Cancel".

Attribute	Value
Integer 1	0
Integer 2	0
Float 1	0.0000
Float 2	0.0000

Figure 4.10 : Change Graph Attributes Window

To set the attributes of a specific graph, make the window containing the graph the active window, ensure that there are no selected edges or nodes in the graph, and select Get Info from the Info Menu. This opens a change graph attributes window which contains the values of the attributes of the active graph. The attribute values are set as they were for default attributes.

To set the value of some of the attributes for every open graph to the same value, the Set Value For All Graphs menu item can be used. With this command if the window is closed with the OK button, only those attributes and labels whose values were changed between opening and closing the window are affected.

Node Attributes

A *change node attributes* window is used to alter the attributes of particular nodes and default node values. Figure 4.11 shows a change node attributes window. Attributes are changed if the window is closed with the OK button and not changed if the window is closed with the Cancel button.

The majority of attributes are changed by selecting the current value and typing in a new value. The exceptions to this are the visited, source, target and label attributes. The visited, source and target attributes are changed by clicking in the box beside the name of the attribute or directly on the name. This toggles the value between TRUE, indicated with an X in the box, and FALSE, indicated by an empty box.

The label attribute is used to determine what, if any, attribute to label the node with. This attribute is set by pressing the mouse on the word label, positioning the mouse so that the desired label value is highlighted and releasing the mouse. The Integer 1 and 2 and the Float 1 and 2 attributes may have their labels changed to reflect the value they represent.

Default Information For New Nodes			
Probability	<input type="text" value="1.0000"/>	Name	<input type="text"/>
Cost	<input type="text" value="0.0000"/>	Capacity	<input type="text" value="0.0000"/>
Weight	<input type="text" value="0.0000"/>	Radius	<input type="text" value="6"/>
Integer 1	<input type="text" value="0"/>	Label	<input type="text" value="ID"/>
Integer 2	<input type="text" value="0"/>	Float 1	<input type="text" value="0.0000"/>
☐ Visited		Float 2	<input type="text" value="0.0000"/>
☐ Source		<input type="button" value="OK"/> <input type="button" value="Cancel"/>	
☐ Target			

Figure 4.11 : Change Node Attributes Window

To set the attributes of an individual node or a number of nodes, select the nodes to be affected, and select Get Info from the Info Menu. This presents, beginning with the first selected node, a change node attributes window for each of the selected nodes. The ID of the node appears in the title of the window to help distinguish which node the window represents.

To change some of the attributes for every node in a graph to the same value, select the Set Value For All Nodes command from the Info Menu. This command sets only those attributes whose values are different after the window is closed and it sets those changed values for every node in the active graph.

Each graph has a set of default attributes that nodes are assigned when they are created. To alter the default node attributes for a particular graph, make the graph the active window, select Default Node Values from the Info Menu and set the attributes to their desired values. The default node attributes are saved with the graph.

Edge Attributes

A *change edge attributes* window is used to alter the attributes of edges and default edge values. Figure 4.12 shows a change edge attributes window. Attributes are changed if the window is closed with the OK button and not changed if the window is closed with the Cancel button. Most of the attributes are changed by selecting the current value and typing in a new value. The exceptions to this are the isDirected and Label attributes.

The isDirected attribute is changed by clicking in the box beside the word Directed? or directly on the word Directed?. This toggles the value of isDirected between TRUE, indicated with an X in the box, and FALSE, indicated by an empty box. The Label attribute is used to determine what, if any, attribute to label the edge with. This attribute is set by pressing the mouse on the word Label and positioning the mouse so the desired label value is highlighted and releasing the mouse. The Integer 1 and 2 and the Float 1 and 2 attributes can have their labels changed to reflect the value they represent.

Default Information For New Edges			
Probability	<input type="text" value="1.0000"/>	Name	<input type="text"/>
Flow	<input type="text" value="0.0000"/>	Head Node ID	<input type="text" value="100"/>
Distance	<input type="text" value="0.0000"/>	Tail Node ID	<input type="text" value="4"/>
☐ Directed?		Thickness	<input type="text" value="1"/>
Integer 1	<input type="text" value="0"/>	Float 1	<input type="text" value="0.0000"/>
Integer 2	<input type="text" value="0"/>	Float 2	<input type="text" value="0.0000"/>
Cost	<input type="text" value="0.0000"/>	Capacity	<input type="text" value="0.0000"/>
Label	<input type="text" value="ID"/>	<input type="button" value="OK"/> <input type="button" value="Cancel"/>	

Figure 4.12 : Change Edge Attributes Window

To set the attributes of an individual edge or a number of edges, select the edges and choose Get Info from the Info Menu. This presents, beginning with the first selected edge, a change edge attributes window for each of the selected edges. The ID of the edge appears in the title of the window to help distinguish which edge the attributes window represents.

To change some of the attributes for every edge in a graph to the same value, select Set Value For All Edges from the Info Menu. This command sets only those attributes whose values are different after the window is closed, and it sets those changed values for every edge in the active graph .

Each graph has a set of default attribute values that edges are assigned when they are created. To alter the default edge attributes for a particular

graph, make the graph the active window, select Default Edge Values from the Info Menu, and set the attributes to their desired values. The default edge attributes are saved with a graph.

4.4 Printing From GraphUI

A graph created with GraphUI can be printed from the Finder and from within GraphUI. To print a graph from finder, select the document containing the graph and choose Print from the Finder's File Menu. This prints the graph to the default printer.

To set up for different types of printing from within GraphUI use the Page Setup in the File Menu. The values that can be set with this command are dependent upon the default printer selected with Chooser. Generally, the quality of print and the orientation of the page can be specified and any reduction/enlargements permitted by the printer. Figure 4.13 shows the page set up when using a LaserWriter printer.

Figure 4.13 : Page Setup for LaserWriter

There are two methods of printing a graph from within GraphUI. Both methods require the window containing the graph be the active window. The first method is to select Print One from the File Menu. This prints the graph according to the options specified in Page Setup. The second method is to use the Print ... command from the File Menu which manages a number of preferences depending upon the printer (eg. the number of copies and the portion of the graph to be printed).

A graph larger than a single page is printed on multiple pages. The pages of a graph are printed so that they can be pasted together to form a rectangle containing the entire graph.

The Applications' Transcript window which is described in section 4.6 can also be printed by making it the active window and selecting the Print One or the Print ... command.

4.5 Clipboard Support

Full clipboard support of cut, copy and paste are included in GraphUI. The copy command duplicates each selected node and edge, and removes all extra space around this copied subgraph. In order for an edge to exist, it must have a head and tail node; thus if the head or tail node of a selected edge is not selected, it is still copied to the clipboard. Figure 4.14a and 4.14b show a graph with a selection and the resulting subgraph copied to the clipboard.

The cut command is similar to the copy command except every selected node and edge is deleted from the graph once it has been copied. Figure 4.14c shows what is left of the graph in Figure 4.14a after a Cut

command. If the head or tail node of a selected edge is not selected, the node remains in the graph. All edges of a selected node are deleted when the node is cut from the graph.

The paste command pastes any subgraph in the clipboard into the active graph. The top left corner of the subgraph is pasted into the center of the active graph. The entire subgraph is selected so it is easy to reposition the nodes and edges pasted into the graph.

When a graph is pasted into a drawing package the individual components of the graph (nodes, edges and labels) are editable. Nodes are simply filled ovals, edges are lines and labels are text. It is easier to edit the location of nodes and their edges using GraphUI. A drawing package is useful for adding extra text to the graph and repositioning node and edge labels. Once a graph has been edited in a drawing package, it cannot be pasted into a GraphUI graph.

Figure 4.14a : Graph to be Copied/Cut

Figure 4.14b : Portion of Graph Copied

Figure 4.14c : Remainder of Graph After a Cut Command

Entire graphs can be copied to the clipboard with Copy Graph so they can be pasted directly into a document such as a report. Unlike the Copy and Cut commands, Copy Graph copies the entire graph, not just the selected portion. Nodes and edges that were selected when the graph was copied, remain gray. The Copy Graph command allows selection information that may have been used in animating an algorithm to be preserved when copied into documents. To copy the active graph, select Copy Graph from the Edit Menu. This gets rid of extra white space around the graph making it fit into

as small a rectangle as possible. Graphs which have been copied into the clipboard with the Copy Graph command can be pasted directly into a graph created with GraphUI.

The Applications' Transcript, described in the following section, can also be copied to the clipboard and pasted into other documents as text. This is done by making the transcript active, selecting the desired text, and issuing the Copy command from the Edit Menu. The Show Clipboard command in the Edit Menu displays the contents of the clipboard.

4.6 Connecting to and Using Graph Applications

GraphUI alone provides a useful graph editor for drawing graphs. Connecting GraphUI to applications that perform algorithms on graphs, allows it to become the user interface to the graph program. Chapter 5 describes in detail the application interface used to gain access to GraphUI from a program. This section describes how a user gains access to an application using GraphUI as its user interface. The instructions for running an application's algorithms will differ and must be provided with the documentation of the application.

Creating A Connection

The name supplied by a graph algorithms program and the connection tool required to connect to the program must be known to connect GraphUI to the graph algorithms program. This information should be provided in the instructions for running the program. For now, consider only that there is at

least one graph program which uses GraphUI running somewhere that can be accessed by a Communications Toolbox connection and that GraphUI is running.

Before establishing a connection between GraphUI and a graph application, the Communications Toolbox Connection Manager must know how to connect to the application. The Connection Settings command in the Comm Menu is used to set the type of connection being made. The default connection type is the AppleTalk ADSP tool which can connect GraphUI to another application on the same computer or a different computer on the same AppleTalk network.

Clicking on the Method field in the connection settings window provides a list of all possible connection types. Select the type which is appropriate for the application and make certain the connection parameters in the window are set correctly. Figure 4.15 shows an AppleTalk connection being set with the local name of the connection set to GraphUI. The values for each of the different connection tools are described in the Macintosh® Communications Toolbox Manual [2].

Each connection created between an application and GraphUI has two ends. One of the ends *waits for a connection* and the other *opens* to a waiting connection. If the application is waiting to be connected to, set up the connection and issue an Open Connection command from the Comm Menu. If the application wants to open a connection to a waiting GraphUI, set up the connection settings for the type of connection required and select Wait for Connection from the Comm Menu. Wait for Connection will wait until another program does an open to it or the connection is closed.

Once one connection has been established, additional connections can be created in the same way.

The screenshot shows a 'Connection Settings' dialog box. At the top left is the title 'Connection Settings'. To the right are 'OK' and 'Cancel' buttons. Below the title, the 'Method:' field is set to 'AppleTalk ADSP Tool'. A horizontal line separates this from the main settings area. In the center, 'Connection Type:' is set to 'ADSP', flanked by two computer icons. Below this, there are two large empty rectangular areas with scrollbars. The left area is labeled 'Zone: *' and the right area is labeled 'Name: Remote User'. A small computer icon labeled 'Remote' is positioned between these two areas. At the bottom, there are two more fields: 'Local Zone: *' and 'Local Name: GraphUI'.

Figure 4.15 : AppleTalk ADSP Tool Connection Settings

Closing a Connection

Connections are closed in the order they were opened. The first connection opened is the first connection closed. A connection can be closed from either end. To close the oldest connection from GraphUI, select the Close Connection command from the Comm Menu. Connections closed from an application are disposed of by GraphUI.

Executing an Algorithm of a Graph Program

Applications connected to GraphUI can add menu items to GraphUI's Algorithms Menu. If an application permits execution of any of its algorithms from GraphUI directly, the names of the available algorithms will appear in the Algorithms Menu. Algorithms which do not require a graph as input are always available, algorithms that require a graph as input are not available unless the active window contains a graph. What happens once an algorithm has been selected from the Algorithms Menu is determined by the application program and the specific algorithm.

Connected applications can request information from a user through GraphUI. They can request an integer value or a real number. These requests are presented in a modal dialog box. The request will have a message describing the data to be provided. Simply enter the requested value and press OK. If the value is not acceptable, an error message will be displayed. The window will not close until an acceptable value has been entered.

The Applications' Transcript Window

The Applications' Transcript window is a window into which text may be written on behalf of a connected application. Each time an application writes to the window, the window is opened if it is not already open. The window can be opened at any time by selecting Open Transcript from the File Menu. The window can be closed with the close box of the window, the Close Transcript command from the File Menu, and, if the transcript is the active window, with the Close command.

Text may be entered and edited using standard text editing techniques and cut, copy and paste to and from the clipboard. Regardless of the cursor

position, text from an application is added to the end of the text in the window.

This window can be printed with the Print One and Print command but the contents cannot be saved; thus text to be kept should be printed or copied to the clipboard and pasted into a word processing document. Figure 4.16 shows the contents of the transcript window after executing the shortest path algorithm described in Chapter 6.

Starting Shortest Path Algorithm ---
Finished Shortest Path Algorithm ---

Figure 4.16 : Applications' Transcript After Running Shortest Path

5. Application Interface

Message passing is the basis of our application interface to GraphUI. It is used to send information between GraphUI and graph programs. A message is a sequence of bytes that can be sent and received. Message passing enables GraphUI to be used from a separate running program. The message set describes the application interface to GraphUI.

For now let us consider the format of messages. Each message has a start of message character, a length byte, an opcode and additional information (if any) required for the command contained in the message, Figure 5.1. The start of message character is a single byte with value 0X02. The length field, an unsigned long (8 bytes), is the number of bytes in the message not including the start of message character or the length field. These are followed by a 4 byte opcode describing the contents of the message.

There are three different types of messages: commands, replies, and information messages. Commands are sent to GraphUI to be acted upon, replies are sent by GraphUI in response to a command and information messages are sent from GraphUI to inform programs that a graph has been altered.

Figure 5.1 : Format of All GraphUI Messages

Many of the messages refer to graph, node and edge IDs. IDs are integer values that distinguish individual graphs, and nodes and edges within a graph. All graphs have a unique ID. Every edge in a graph has a different ID from every other edge and every node has an ID which is different from every other node in the graph. To identify a node, both the node ID and the graph ID are required. To identify an edge, both the edge ID and the graph ID are required.

The next two sections in this chapter describe the command set and the set of information messages. They assume that the start of message and length fields are included in the message and describe the opcode and the additional information in the message. There is not one fixed method for using commands and messages. Programs can choose to act on or ignore any information sent to them by GraphUI. Some suggestions are made at the end of the chapter to help with algorithm animation and methods of using information sent by GraphUI.

5.1 GraphUI Commands

Messages sent to GraphUI are called commands. Each command has a reply which is a message sent from GraphUI to the program that sent the command. Commands enable a program to use GraphUI to perform tasks similar to those a user can do directly from GraphUI, such as create and modify graphs. Each reply has an opcode and an error code.

The value of the opcode of each reply is the same as the opcode of the command it responds to. The error code tells the program if GraphUI was able to execute its last command. The values of the opcodes, error codes, and the data types used to describe the format of commands, replies and information messages are included in Appendix A.

A number of errorCodes may be returned in response to any command. BAD_OPCODE is returned if the opcode specified in the command is not a valid command opcode. BAD_COMMAND_SIZE is returned if the size of the command received does not match the size expected for the command. The NO_COMMAND error code is returned if the command's length field is zero. Finally if the command was received and executed without problem, NO_ERROR is returned as the errorCode. Other error codes returned for specific commands are documented for each command.

5.1.1 Creation Commands

Creation commands are used to build new graphs and to add nodes and edges to a graph. When a new graph, node or edge is created, the reply to the command includes the ID to be used to refer to the graph, node or edge in subsequent commands. This ID must be used to reference a graph, node or edge in subsequent commands.

Creating New Graphs

To create a new graph which, in effect, opens a new untitled window like the one in Figure 4.1, the add graph command is used. This command

creates a new graph which receives the default graph attributes as set in GraphUI. The add graph command and its reply have the following formats:

```
typedef struct {
    OpType          opcode;
    } AddGraphCmd;

typedef struct {
    OpType          opcode;
    ErrorCodeType   errorCode;
    GraphIDType     graphID;
    } AddGraphReply;
```

Where:

opcode - *ADD_GRAPH*

graphID - *the numeric ID of the graph that was created.*

Possible errorCodes:

NACK - *if a window could not be opened.*

Creating New Nodes

A new node is added to a graph with the add node command. A node cannot be added to a graph that does not exist. When a new node is created, its attributes are set to the default node attributes of the graph it is added to. The following structures describe the format of the add node command and its reply:

```
typedef struct {
    OpType          opcode;
    GraphIDType     graphID;
    NodeLocationType location;
    } AddNodeCmd;

typedef struct {
    OpType          opcode;
    ErrorCodeType   errorCode;
    NodeIDType      nodeID;
    } AddNodeReply;
```

Where :

opcode - **ADD_NODE**

graphID - the ID of the graph to add the node to

location - the X, Y coordinate in the graph view to position the center of the node at. If this value is (0,0), the node is placed at a random location within the visible portion of the graph view.

nodeID - the numeric ID of the node that was created

Possible error codes :

BAD_GRAPH_ID - the graphID specified did not match one of the open graphs.

NACK - an unexpected error occurred.

Creating New Edges

The add edge command is used to add an edge to a graph. An edge cannot be added to a graph that does not exist. The edge must have a head and tail node in the graph. The following structures describe the format of the add edge command and its reply:

```
typedef struct {
    OpType          opcode;
    GraphIDType     graphID;
    NodeIDType      headNode;
    NodeIDType      tailNode;
} AddEdgeCmd;

typedef struct {
    OpType          opcode;
    ErrorCodeType   errorCode;
    EdgeIDType      edgeID;
} AddEdgeReply;
```

Where :

opcode - **ADD_EDGE**

graphID - the ID of the graph to add the edge to

headNode - the ID of the head node of the edge

tailNode - the ID of the tail node of the edge

edgeID - the numeric ID of the edge that was created

Possible errorCodes are:

BAD_GRAPH_ID - the graphID specified did not match one of the open graphs.

BAD_NODE_ID - the ID specified for the head or tail node did not exist.

NACK - an unexpected error occurred.

5.1.2 Modification Commands

Modification commands are used to alter the appearance and/or the structure of graphs.

Deleting / Closing Graphs

To close an open graph in GraphUI, the delete graph command is provided. The delete graph command closes a graph and lets the user save the graph if changes have been made since it was last saved. The following is the format of the delete graph command and its reply.

```
typedef struct {  
    OpType      opcode;  
    GraphIDType graphID;  
}DeleteGraphCmd;  
  
typedef struct {  
    OpType      opcode;  
    ErrorCodeType errorCode;  
}DeleteGraphReply;
```

Where:

opcode - *DELETE_GRAPH*

graphID - *the ID of the graph to be deleted*

Possible errorCodes are:

BAD_GRAPH_ID - *the graphID specified did not match one of the open graphs.*

NACK - *an unexpected error occurred.*

Moving Nodes

A node is moved to a new location within a graph with the move node command. This command animates the move if the number of steps to move the node is greater than 1. The location specified is the position to

place the center of the node in the graph. When a node is moved its edges are adjusted to the new position. The format of the command and its reply are as follows:

```
typedef struct {
    OpType          opcode;
    GraphIDType     graphID;
    NodeIDType      nodeID;
    NodeLocationType location;
    StepsType       steps;
} MoveNodeCmd;

typedef struct {
    OpType          opcode;
    ErrorCodeType   errorCode;
} MoveNodeReply;
```

Where :

opcode - *MOVE_NODE*

graphID - *the ID of the graph containing the node to be moved.*

nodeID - *the ID of the node to be moved.*

location - *the new X, Y coordinate for the center of the node*

steps - *the number of steps to perform the move in. For each step the node is moved 1/steps of the way to the new position so it appears the node is being dragged to the new location.*

Possible errorCodes are:

BAD_GRAPH_ID - *the graphID specified did not match one of the open graphs.*

BAD_NODE_ID - *the node ID specified was not in the graph.*

NACK - *an unexpected error occurred.*

Deleting Nodes

A node is deleted from a graph with the delete node command. Deleting a node also deletes any edge connected to it. Once a node and its edges have been deleted from a graph, their IDs are no longer valid.

```
typedef struct {
    OpType          opcode;
    GraphIDType     graphID;
    NodeIDType      nodeID;
} DeleteNodeCmd;
```

```
typedef struct {
    OpType          opcode;
    ErrorCodeType   errorCode;
}DeleteNodeReply;
```

Where :

opcode - *DELETE_NODE*

graphID - *the ID of the graph containing the node to be deleted.*

nodeID - *the ID of the node to be deleted.*

Possible errorCodes are :

BAD_GRAPH_ID - the graphID specified did not match one of the open graphs.

BAD_NODE_ID - the node ID specified was not in the specified graph.

NACK - an unexpected error occurred.

Contracting Edges

An edge is contracted in two steps. First all edges which connect the edge's head node and tail node including the edge being contracted are deleted. Next either all of the edges which were connected to the tail node are moved to the head node and the tail node is deleted, or all of the edges attached to the head node are moved to the tail node and the head node is deleted.

The contract edge command lets an edge be contracted to either its head or tail node. When the contract edge command is used, GraphUI animates the move of the head node to the tail node or vice versa in six steps. The following is the format of the contract edge command and its reply.

```
typedef struct {
    OpType          opcode;
    GraphIDType     graphID;
    EdgeIDType      edgeID;
    Boolean          headToTail;
}ContractEdgeCmd;
```

```
typedef struct {
    OpType          opcode;
    ErrorCodeType   errorCode;
}ContractEdgeReply;
```

Where :

opcode - **CONTRACT_EDGE**

graphID - the ID of the graph containing the edge to be contracted

edgeID - the ID of the edge to be contracted

headToTail - flag that dictates which way to contract the edge. If the value is **TRUE**, the head node's edges are moved to the tail node and the head node is deleted; if **FALSE**, the tail node's edges are moved to the head node and the tail node is deleted.

Possible errorCodes are:

BAD_GRAPH_ID - the graphID specified did not match one of the open graphs.

BAD_EDGE_ID - the edge ID specified was not in the graph.

NACK - an unexpected error occurred.

Deleting Edges

An edge is deleted from a graph with the delete edge command. Once an edge has been deleted by GraphUI it may no longer be referenced. The format of the delete edge command is as follows:

```
typedef struct {
    OpType          opcode;
    GraphIDType     graphID;
    EdgeIDType      edgeID;
} DeleteEdgeCmd;

typedef struct {
    OpType          opcode;
    ErrorCodeType   errorCode;
} DeleteEdgeReply;
```

Where :

opcode - **DELETE_EDGE**

graphID - the ID of the graph to containing the edge to be deleted

edgeID - the ID of the specific edge to be deleted

Possible errorCodes are:

BAD_GRAPH_ID - the graphID specified did not match one of the open graphs.

BAD_EDGE_ID - the edge ID specified was not in the graph

NACK - an unexpected error occurred.

Selection Commands

Selecting nodes and edges of a graph is a method of changing the appearance of the graph. Every selected node or edge is drawn in gray rather than black. There are four commands that affect which nodes and edges are selected: get selection, add to selection, remove from selection, and clear selection.

Get Selection

The get selection command returns the nodes and edges that are currently selected in the specified graph. This set of selected components of a graph is a useful form of input to a graph program as indicated in section 4.4. The format of the get selection command is as follows:

```
typedef struct {
    OpType          opcode;
    GraphIDType     graphID;
} GetSelectionCmd;

typedef struct {
    OpType          opcode;
    ErrorCodeType   errorCode;
    CounterType     numberOfNodes;
    CounterType     numberOfEdges;
    NodeIDType      selectedNodes[numberOfNodes];
    EdgeIDType      selectedEdges[numberOfEdges];
} GetSelectionReply;
```

Where :

opcode - *GET_SELECTION*

graphID - *the ID of the graph whose selection is to be returned*

numberOfNodes - *the number of nodes that are selected in the graph*

numberOfEdges - *the number of edges that are selected in the graph*

selectedNodes - *a list of the IDs of each node in the graph that is selected*

selectedEdges - *a list of the IDs of each edge in the graph that is selected*

Possible errorCodes are :

BAD_GRAPH_ID - the *graphID* specified did not match one of the open graphs.

NACK - an unexpected error occurred.

Add To Selection Command

This command selects the nodes and edges specified in the command if they are not already selected. Nodes or edges selected prior to receipt of this command remain selected. Selection changes the colour of the node or edge from black to gray. The nodes and edges added to the selection are considered part of the selection set and are used with user interface commands that require selected nodes and/or edges. The format of the add to selection command is :

```
typedef struct {
    OpType          opcode;
    GraphIDType     graphID;
    CounterType     numberOfNodes;
    CounterType     numberOfEdges;
    NodeIDType      nodesToSelect[numberOfNodes];
    EdgeIDType      edgesToSelect[numberOfEdges];
} AddToSelectionCmd;
```

```
typedef struct {
    OpType          opcode;
    ErrorCodeType   errorCode;
} AddToSelectionReply;
```

Where :

opcode - **ADD_TO_SELECTION**

graphID - the ID of the graph whose selection is to be added to

numberOfNodes - the number of nodes to be selected

numberOfEdges - the number of edges to be selected

nodesToSelect - a list of the IDs of each node to be selected

edgesToSelect - a list of the IDs of each edge to be selected

Possible errorCodes are :

BAD_GRAPH_ID - the *graphID* specified did not match one of the open graphs.

BAD_NODE_ID - one or more of the nodes specified does not exist

BAD_EDGE_ID - one or more of the edges specified does not exist.

NACK - an unexpected error occurred.

Remove From Selection Command

This command unselects the nodes and edges specified if they are selected. If any specified node or edge was not selected, it remains unselected. Other selected nodes or edges not specified in this command remain selected. This changes the colour of the node or edge from gray to black. The nodes and edges removed from the selection are no longer in the graph's selection set. The format of the remove from selection command is :

```
typedef struct {
    OpType          opcode;
    GraphIDType     graphID;
    CounterType     numberOfNodes;
    CounterType     numberOfEdges;
    NodeIDType      nodesToUnselect(numberOfNodes);
    EdgeIDType      edgesToUnselect(numberOfEdges);
} RemoveFromSelectionCmd;

typedef struct {
    OpType          opcode;
    ErrorCodeType   errorCode;
} RemoveFromSelectionReply;
```

Where :

opcode - REMOVE_FROM_SELECTION

graphID - the ID of the graph whose selection is to have items removed from it.

numberOfNodes - the number of nodes that are to be removed from the graph's selection (ie. unselected) by this command

numberOfEdges - the number of edges that are to be removed from the graph's selection (ie. unselected) by this command

nodesToUnselect - a list of the IDs of each node in the graph that is to be removed from the graph's selection

edgesToUnselect - a list of the IDs of each edge in the graph that is to be removed from the graph's selection

Possible errorCodes are :

BAD_GRAPH_ID - the graphID specified did not match one of the open graphs.

BAD_NODE_ID - at least one of the nodes specified does not exist

BAD_EDGE_ID - at least one of the edges specified does not exist.

NACK - an unexpected error occurred.

Clear Selection Command

This command unselects every selected node and edge in a graph. The format of the command and its reply is as follows:

```
typedef struct (  
    OpType          opcode;  
    GraphIDType     graphID;  
) ClearSelectionCmd;  
  
typedef struct (  
    OpType          opcode;  
    ErrorCodeType   errorCode;  
) ClearSelectionReply;
```

Where :

opcode - *CLEAR_SELECTION*

graphID - *the ID of the graph whose selection set is to be cleared*

Possible errorCodes are :

BAD_GRAPH_ID - *the graphID specified did not match one of the open graphs.*

NACK - *an unexpected error occurred.*

5.1.3 Commands that Manage Information with GraphUI

There are a number of different commands that get information from GraphUI for a graph program. These commands provide graph programs with a means of obtaining user input through GraphUI.

Specific information about a graph, node or edge is called its attributes. Attributes provide extra information other than structural about a graph, node or edge. Graph attributes, the information maintained by GraphUI about each graph, are fixed, as are node and edge attributes. Attributes are used to provide information that is common among many graph domains.

Unfortunately, the set of attributes may not contain sufficient information for all programs using GraphUI. If information cannot be

2

of/de

2

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

PHOTOLITH IN METHYLENE BLUE TESTING SINCE 1974

MICRO
CORPORATION

1000 COUNTY ROAD 5, SUITE 100, WILSON, NJ 07097 USA

TEL: 609-426-7800 FAX: 609-426-7801 TOLL FREE: 800-541-0000

supplied by attributes there are two commands for prompting the user to supply information directly from GraphUI's user interface. What follows is a description of the commands used to manage graph information.

Get Graph Command

The get graph command is probably the most useful of all of the commands used with GraphUI. Given the ID of a graph, the get graph command returns a description of the graph and its nodes and edges. The description of the graph includes the attributes of the graph and the ID and attributes of each node and edge in the graph. The following is the format of the get graph command:

```
typedef struct {
    OpType          opcode;
    GraphIDType     graphID;
} GetGraphCmd;

typedef struct {
    OpType          opcode;
    ErrorCodeType   errorCode;
    CounterType     numberOfNodes;
    CounterType     numberOfEdges;
    GraphAttributesType attr;
    NodeInfoType    nodeInfo(numberOfNodes);
    EdgeInfoType    edgeInfo(numberOfEdges);
} GetGraphReply;
```

Where :

opcode - GET_GRAPH

graphID - the ID of the graph to be returned

numberOfNodes - the number of nodes in the graph

numberOfEdges - the number of edges in the graph

attr - the attributes of the graph. The actual contents of the attributes are described in the attributes section in Chapter 2 and the graph attributes structure type in Appendix A.

nodeInfo - a list of the node ID followed by the attributes of each node in the graph. The actual contents of the node attributes are described in the attributes section in Chapter 2 and the node attributes structure type in Appendix A.

edgeInfo - a list of the edge ID followed by the attributes of each edge in the graph. The actual contents of the edge attributes are described in the attributes section in Chapter 2 and the edge attributes structure type in Appendix A.

Possible errorCodes are:

BAD_GRAPH_ID - the graphID specified did not match one of the open graphs.
NACK - an unexpected error occurred.

Get Graph Attributes Command

The get graph attributes command returns the attributes of the graph specified. The format of the command and reply are as follows:

```
typedef struct {
    OpType          opcode;
    GraphIDType     graphID;
} GetGraphAttrCmd;

typedef struct {
    OpType          opcode;
    ErrorCodeType   errorCode;
    GraphAttributesType attributes;
} GetGraphAttrReply;
```

Where:

opcode - *GET_GRAPH_ATTRIBUTES*
graphID - the ID of the graph whose attributes are requested
attributes - the attributes of the graph as defined in Chapter 2

Possible errorCodes are:

BAD_GRAPH_ID - the graphID specified did not match one of the open graphs.
NACK - an unexpected error occurred.

Get Node Attributes Command

The get node attributes command returns the attributes of the node specified in the command. The format of the command and reply are as follows:

```
typedef struct {
    OpType          opcode;
    GraphIDType     graphID;
```

```

NodeIDType      nodeID;
} GetNodeAttrCmd;

typedef struct {
    OpType      opcode;
    ErrorCodeType  errorCode;
    NodeAttributesType  attributes;
} GetNodeAttrReply;

```

Where:

opcode - *GET_NODE_ATTRIBUTES*
graphID - *the ID of the graph containing the node*
nodeID - *the ID of the node whose attributes are to be obtained*
attributes - *the attributes of the node as defined in appendix A*

Possible errorCodes are:

BAD_GRAPH_ID - *the graphID specified did not match one of the open graphs.*
BAD_NODE_ID - *the node specified in the command was not in the graph*
NACK - *an unexpected error occurred.*

Get Edge Attributes Command

The get edge attributes command returns the attributes of the edge specified in the command. The format of the command and reply are as follows:

```

typedef struct {
    OpType      opcode;
    GraphIDType  graphID;
    EdgeIDType  edgeID;
} GetEdgeAttrCmd;

typedef struct {
    OpType      opcode;
    ErrorCodeType  errorCode;
    EdgeAttributesType  attributes;
} GetEdgeAttrReply;

```

Where:

opcode - *GET_EDGE_ATTRIBUTES*
graphID - *the ID of the graph containing the edge*
nodeID - *the ID of the edge whose attributes are to be obtained*
attributes - *the attributes of the node as defined in Chapter 2 and Appendix A*

Possible errorCodes are:

BAD_GRAPH_ID - the graphID specified did not match one of the open graphs.
BAD_EDGE_ID - the edge ID specified in the command was not in the graph
NACK - an unexpected error occurred.

Set Graph Attributes Command

The set graph attributes command is used to set the values of a graph's attributes, the default graph attributes, and the attributes for all open graphs. The attributes that are changed depend upon the changed attributes flag in the command. This enables a program to change the attributes it is using, leaving other attributes unaffected.

The graphID field of the command is used to determine the action of the command. If the graphID is set to 0, the default attributes are set; if the graphID is set to 0XFFFF, the attributes for all open graphs are set; otherwise the graphID is the ID of the graph whose attributes are set. The following is the structure of the command and its response:

```
typedef struct {
    OpType          opcode;
    GraphIDType     graphID;
    AttrFlagType    changedAttributes;
    GraphAttributesType attributes;
} SetGraphAttrCmd;

typedef struct {
    OpType          opcode;
    ErrorCTYPE     errorCode;
} SetGraphAttrReply;
```

Where :

opcode - SET_GRAPH_ATTRIBUTES

graphID - 0 - set the default graph attributes

0XFFFF - set attributes of all open graphs

otherwise - the ID of the graph whose attributes are to be set

changedAttributes - a 64 bit flag; used to determine which attributes are to be set. Only those attributes whose bit is set to 1 in this flag are set. Table 5.1 shows the bit positions in the flag for each graph attribute that can be set, other bit positions are reserved for future use.

attributes - the values of the attributes to be set. Only the values of the attributes set to be changed with the **changedAttributes** flag are used.

Possible errorCodes are:

BAD_GRAPH_ID - the *graphID* specified did not match one of the open graphs.

NACK - an unexpected error occurred in GraphUI while executing the command

Attribute	Bit Position*
generic integer 1	1
generic integer 2	2
generic float 1	3
generic float 2	4
generic integer 1 label	5
generic integer 2 label	6
generic float 1 label	7
generic float 2 label	8

Table 5.1 : Format of the Change Graph Attributes Flag

*(bit positions are from least significant to most significant)

Set Node Attributes Command

The set node attributes command is used to set the values of a node's attributes, the default node attributes for a particular graph, and the attributes for all nodes in a graph. The attributes that are changed depend upon the changed attributes flag in the command. This enables a program to change the attributes it is using, leaving other attributes unaffected.

The **nodeID** field of the command is used to determine the action of the command. If the **nodeID** is set to 0, the default node attributes are set for the graph specified in **graphID**; if the **nodeID** is set to 0xFFFF, the attributes

for all nodes in the graph specified in graphID are set; otherwise the nodeID is the ID of the node in the specified graph whose attributes are set. The following is the structure of the command and its response:

```
typedef struct {
    OpType          opcode;
    GraphIDType     graphID;
    NodeIDType      nodeID;
    AttrFlagType    changedAttributes;
    NodeAttributesType attributes;
} SetNodeAttrCmd;

typedef struct {
    OpType          opcode;
    ErrorCodeType   errorCode;
} SetNodeAttrReply;
```

Where :

opcode - SET_NODE_ATTRIBUTES

graphID - the ID of the graph containing the node(s) or the default node attributes that are to be set.

nodeID - 0 - set the default node attributes of the graph
 0XFFFF - set attributes of all nodes in the graph
 otherwise - the ID of the node whose attributes are to be set

changedAttributes - a 64 bit flag used to determine which attributes are to be set. Only those attributes whose bit is set to 1 in this flag are set. Table 5.2 shows the bit positions in the flag for each node attribute that can be set, other bit positions are reserved for future use.

attributes - the values of the attributes to be set. Only the values of the attributes set to be changed with the changedAttributes flag are used.

Possible errorCodes are:

BAD_GRAPH_ID - the graphID specified did not match one of the open graphs.

BAD_NODE_ID - the nodeID specified was not in the graph.

NACK - an unexpected error occurred in GraphUI while executing the command

Attribute	Bit Position*
generic integer 1	1
generic integer 2	2
generic float 1	3
generic float 2	4
generic integer 1 label	5
generic integer 2 label	6
generic float 1 label	7
generic float 2 label	8
probability	9
name	10
label	11
radius	12
cost	13
capacity	14
weight	15
visited	16
source	21
target	22

Table 5.2 : Format of the Change Node Attributes Flag

*(bit positions are from least significant to most significant)

Set Edge Attributes Command

The set edge attributes command is used to set the values of an edge's attributes, the default edge attributes for a particular graph, and the attributes for all edges in a graph. The attributes that are changed depends upon the changed attributes flag in the command. This enables a program to change the attributes it is using, leaving other attributes unaffected.

The edgeID field of the command is used to determine the action of the command. If the edgeID is set to 0, the default edge attributes are set for the

graph specified in graphID; if the edgeID is set to 0XFFFF, the attributes for all edges in the graph specified in graphID are set; otherwise the edgeID is the ID of the edge in the specified graph whose attributes are set. The following is the structure of the command and its response:

```
typedef struct {
    OpType          opcode;
    GraphIDType     graphID;
    EdgeIDType      edgeID;
    AttrFlagType    changedAttributes;
    EdgeAttributesType attributes;
} SetEdgeAttrCmd;

typedef struct {
    OpType          opcode;
    ErrorCodeType   errorCode;
} SetEdgeAttrReply;
```

Where :

opcode - SET_EDGE_ATTRIBUTES

graphID - the ID of the graph containing the edge(s) or the default edge attributes that are to be set.

edgeID - 0 - set the default edge attributes of the graph
 0XFFFF - set attributes of all edges in the graph
 otherwise - the ID of the edge whose attributes are to be set

changedAttributes - a 64 bit flag used to determine which attributes are to be set. Only those attributes whose bit is set to 1 in this flag are set. Table 5.3 shows the bit positions in the flag for each edge attribute that can be set, other bit positions are reserved for future use.

attributes - the values of the attributes to be set. Only the values of the attributes set to be changed with the changedAttributes flag are used.

Possible errorCodes are:

BAD_GRAPH_ID - the graphID specified did not match one of the open graphs.

BAD_EDGE_ID - the edgeID specified was not in the graph.

NACK - an unexpected error occurred in GraphUI while executing the command

Attribute	Bit Position*
generic integer 1	1
generic integer 2	2
generic float 1	3
generic float 2	4
generic integer 1 label	5
generic integer 2 label	6
generic float 1 label	7
generic float 2 label	8
probability	9
name	10
label	11
cost	13
capacity	14
thickness	17
is directed?	18
distance	19

Table 5.3 : Format of the Change Edge Attributes Flag

*(bit positions are from least significant to most significant)

Get Integer Command

The get integer command is used to obtain an integer value from the GraphUI user. This command opens a dialog box with a prompt explaining the desired value and a place to input the value. When the user closes the dialog box, a reply containing the number the user entered is returned in response to this command. The amount of time before the reply to this command is received is dependent upon how long the user takes before closing the dialog box. The format of this command and its reply are as follows:

```
typedef struct {
```

```

OpType      opcode;
PromptType  prompt;
} GetIntegerCmd;

typedef struct {
OpType      opcode;
ErrorCTYPE  errorCode;
int         number;
} GetIntegerReply;

```

Where:

opcode - *GET_INTEGER*

prompt - *the Pascal type string to be displayed in the dialog to prompt the user to enter the desired value.*

number - *the number that the user entered in the dialog*

Possible errorCodes are:

NACK - *an unexpected error occurred in GraphUI while executing the command*

Get Real Number Command

The get real number command is used to obtain a real number from the GraphUI user. The command opens a dialog box with a prompt explaining the desired value and a place to input the number. When the user closes the dialog box, a reply containing the number the user entered is returned in response to this command. The amount of time before the reply to this command is received is dependent upon how long the user takes before closing the dialog box. The format of this command and its reply are as follows:

```

typedef struct {
OpType      opcode;
PromptType  prompt;
} GetRealCmd;

typedef struct {
OpType      opcode;
ErrorCTYPE  errorCode;
float       number;
} GetRealReply;

```

Where:

opcode - *GET_REAL*

prompt - *the Pascal type string to be displayed in the dialog to prompt the user to enter the real number.*

number - *the number that the user entered in the dialog.*

Possible errorCodes are:

NACK - *an unexpected error occurred in GraphUI while executing the command*

5.1.4 Algorithms

Algorithms in a program can be made available to GraphUI users through GraphUI's Algorithms menu. This menu is built dynamically by connected programs with menu items added and removed with the add algorithm and remove algorithm commands. Making a program's algorithm available through GraphUI's user interface means the program does not have to manage algorithms from its own user interface.

The set and remove algorithm commands are provided to help manage the size of the menu. The algorithms menu is linear and can grow to a maximum of 255 menu items. Large menus are cumbersome for a user and it is recommended that menu items be added and removed based upon the user's needs.

When a user selects an algorithm that was added to GraphUI, GraphUI sends a message to the associated program requesting the corresponding algorithm be run for the user. The execution of the algorithm is the responsibility of the program providing the algorithm.

Add Algorithm Command

The add algorithm command is used to add the name of an algorithm to GraphUI's algorithm menu. The name of the algorithm is not guaranteed to be unique. If other connected programs have algorithms with the same name, the name simply appears twice in the menu. The set algorithm command supplies GraphUI with an integer ID to use as a reference to the algorithm. If a program uses the same integer ID for two or more algorithms, the program will not be able to distinguish which algorithm was requested. Duplicate algorithm IDs between applications using GraphUI do not cause a discrepancy. The following is the format of the add algorithm command and its reply.

```
typedef struct {
    OpType          opcode;
    NameType        algorithmName;
    AlgorithmIDType algorithmID;
    Boolean         needsGraph;
} SetAlgorithmCmd;

typedef struct {
    OpType          opcode;
    ErrorCodeType   errorCode;
} SetAlgorithmReply;
```

Where :

opcode - *SET_ALGORITHM*

algorithmName - *the name for the algorithm to be place in GraphUI's algorithm menu.*

algorithmID - *the ID to be used by GraphUI to refer to the algorithm when the user requests that the algorithm be executed.*

needsGraph - *is a flag which if set to TRUE means the algorithm cannot be run unless the active window is a graph window (the ID of the active graph is sent with the request to run the command). If the flag is set to FALSE, the algorithm is not dependent on having a graph active and the algorithm should be available for execution at all times. Algorithms that are not available to be run because there is no active graph, appear in the algorithms menu but can not be selected.*

Possible errorCodes are:

NACK - *an unexpected error occurred in GraphUI while executing the command*

Delete Algorithm Command

The delete algorithm command is used to remove the name of an algorithm displayed in GraphUI's algorithm menu. Any algorithm names added by the program with algorithm ID matching the one specified in the command are removed from the menu.

A program does not have to use this command before it closes as GraphUI always removes the algorithms of a program when the connection to it is closed. The following is the format of the command and its reply.

```
typedef struct {
    OpType          opcode;
    AlgorithmIDType algorithmID;
} DeleteAlgorithmCmd;

typedef struct {
    OpType          opcode;
    ErrorCodeType   errorCode;
} DeleteAlgorithmReply;
```

Where :

opcode - *DELETE_ALGORITHM*

algorithmID - *the ID of the algorithm to be removed from the menu.*

Possible errorCodes are:

NACK - *an unexpected error occurred in GraphUI while executing the command*

Run Algorithm Message

The run algorithm message is not a command sent to GraphUI; rather it is an unsolicited message that is sent from GraphUI when a program's algorithm has been selected by the user. The format of this command is described in the next section of this chapter but the message is mentioned here to remind programmers that if algorithms are added to GraphUI's menu

with the set algorithm message, the program must be prepared to watch for execute algorithm messages and handle the messages accordingly.

Display Text Command

GraphUI has a window, called the Applications' Transcript window where programs and users can write text. In this window algorithms can write results that cannot be conveyed by looking at a graph. Text that is written to the Applications' Transcript is appended to the end of the window regardless of the position of the cursor in the window. Carriage returns are not added after each display text command and should be included as necessary in the text (carriage return is 0X0D). The format of the display text command is as follows:

```
typedef struct {
    OpType          opcode;
    PromptType      text;
} DisplayTextCmd;

typedef struct {
    OpType          opcode;
    ErrorCodeType   errorCode;
} DisplayTextReply;
```

Where:

opcode - *DISPLAY_TEXT*

text - *the text to be written to the Applications' Transcript window*

Possible errorCodes are:

NACK - *an unexpected error occurred in GraphUI while executing the command*

Set Messaging Command

The Set Messaging command is used to allow a program to select whether it wishes to have information messages sent to it. When a program

first connects to GraphUI the default is not to send the messages. The format of the set messaging command is as follows:

```
typedef struct {
    OpType          opcode;
    Boolean          wantsMessages;
} DisplayTextCmd;

typedef struct {
    OpType          opcode;
    ErrorCodeType   errorCode;
} DisplayTextReply;
```

Where:

opcode - *SET_MESSAGING*

wantsMessages - *if TRUE then send information messages; if FALSE do not send information messages*

Possible errorCodes are:

NACK - *an unexpected error occurred in GraphUI while executing the command*

5.2 GraphUI Information Messages

Information messages are messages that are sent from GraphUI to connected programs when a graph changes. These can be used by a program to keep accurate information about open graphs in GraphUI. A program can select whether or not it wants to receive information messages with the Set Messaging command.

Run Algorithm Message

The run algorithm message is sent to a program whenever a user selects an algorithm from GraphUI's menu that was added by the program. The run algorithm message is the only message sent by GraphUI that is not sent as information only. It is sent even if the program is not receiving

information messages. This message must be handled by a program if it makes algorithms available through GraphUI's algorithm menu. The remaining messages can be used or ignored at a program's discretion. It is recommended that a program with algorithms added to GraphUI's menu look for run algorithm messages at least once through its main event loop. The format of a run algorithm message is as follows.

```
typedef struct {
    OpType          opcode;
    AlgorithmIDType algorithmID;
    GraphIDType     graphID;
} RunAlgorithmMsg;
```

Where:

opcode - *RUN_ALGORITHM*

algorithmID - *the ID of the algorithm that the user asked to run*

graphID - *the ID of the active graph if the algorithm needs a graph; 0 otherwise.*

Graph Added Message

The purpose of the graph added message is to inform programs that a new graph has been added or opened. It is sent whenever a new graph is created or an old graph opened. The format of the message is as follows:

```
typedef struct {
    OpType          opcode;
    GraphIDType     graphID;
} GraphAddedMsg;
```

Where:

opcode - *GRAPH_ADDED*

graphID - *the numeric ID of the graph that was added (opened)*

Graph Deleted (Closed) Message

The graph deleted message is sent each time a graph is closed to inform all programs using the graph that it is no longer available. The format of the graph deleted message is as follows:

```
typedef struct {  
    OpType          opcode;  
    GraphIDType     graphID;  
} GraphDeletedMsg;
```

Where:

opcode - *GRAPH_DELETED*

graphID - *the numeric ID of the graph that was deleted (closed)*

Node Added Message

The node added message is sent to programs to inform of the addition of a new node in an existing graph. The message includes the location of the node for those programs maintaining graph appearances. The format of the node added message is as follows:

```
typedef struct {  
    OpType          opcode;  
    GraphIDType     graphID;  
    NodeIDType      nodeID;  
    NodeLocationType location;  
} NodeAddedMsg;
```

Where:

opcode - *NODE_ADDED*

graphID - *the numeric ID of the graph into which the node was added*

nodeID - *the numeric ID of the node that was added to the graph*

location - *the (X,Y) position at which the new node is centered*

Node Deleted Message

The node deleted message is sent to all programs whenever a node of a graph is deleted. The format of the node deleted message is as follows:

```

typedef struct {
    OpType          opcode;
    GraphIDType     graphID;
    NodeIDType      nodeID;
} NodeDeletedMsg;

```

Where:

opcode - *NODE_DELETED*

graphID - *the numeric ID of the graph containing the node that was deleted*

nodeID - *the numeric ID of the node that was removed from the graph*

Edge Added Message

The edge added message is sent to programs to inform them of the addition of a new edge to an existing graph. The message includes the head and tail node of the edge to help programs manage graph structure. The format of the edge added message is as follows:

```

typedef struct {
    OpType          opcode;
    GraphIDType     graphID;
    EdgeIDType      edgeID;
    NodeIDType      headNode;
    NodeIDType      tailNode;
} EdgeAddedMsg;

```

Where:

opcode - *EDGE_ADDED*

graphID - *the numeric ID of the graph into which the edge was added*

edgeID - *the numeric ID of the edge that was added to the graph*

headNode - *the ID of the head node of the new edge*

tailNode - *the ID of the tail node of the new edge*

Edge Deleted Message

The edge deleted message is sent to programs whenever an edge of a graph is deleted. The format of the edge deleted message is as follows:

```

typedef struct {
    OpType          opcode;
    GraphIDType     graphID;
    EdgeIDType      edgeID;
} EdgeDeletedMsg;

```

Where:

opcode - *EDGE_DELETED*

graphID - *the numeric ID of the graph containing the edge that was deleted*

edgeID - *the numeric ID of the edge that was removed from the graph*

Node Moved Message

The node moved message informs programs whenever a node in a graph moves to a new location in the graph. It is intended for use by programs managing the physical appearance of graphs. The new location is the new center point of the node. The format of the node moved message is as follows:

```
typedef struct {  
    OpType          opcode;  
    GraphIDType     graphID;  
    NodeIDType      nodeID;  
    NodeLocationType location;  
} NodeMovedMsg;
```

Where:

opcode - *NODE_MOVED*

graphID - *the numeric ID of the graph which contains the node that was moved*

nodeID - *the numeric ID of the node that was moved*

location - *the new (X,Y) position of the center of the node*

Graph Attributes Changed Message

The graph attributes changed message is sent to inform programs whenever an individual graph's attributes or the default graph attributes are changed. The format of the graph attributes changed message is as follows:

```
typedef struct {  
    OpType          opcode;  
    GraphIDType     graphID;  
    GraphAttributesType attributes;  
} GraphAttrChangedMsg;
```

Where :

opcode - *GRAPH_ATTR_CHANGED*

graphID - 0 - default graph attributes were changed
 otherwise - ID of the graph whose attributes were changed
 attributes - the values of the attributes due to the change.

Node Attributes Changed Message

The node attributes changed message is sent to inform programs whenever an individual node's attributes or the default node attributes of a graph are changed. The format of the node attributes changed message is as follows:

```
typedef struct (
    OpType          opcode;
    GraphIDType     graphID;
    NodeIDType      nodeID;
    NodeAttributesType attributes;
) NodeAttrChangedMsg;
```

Where :

opcode - *NODE_ATTR_CHANGED*
 graphID - the ID of the graph containing the node(s) or the default node attributes that were changed.
 nodeID - 0 - default node attributes of the graph were changed
 otherwise - ID of the node whose attributes were changed
 attributes - the values of the attributes or default attributes due to the change.

Edge Attributes Changed Message

The edge attributes changed message is sent to inform programs whenever an individual edge's attributes or the default edge attributes of a graph are changed. The format of the edge attributes changed message is as follows:

```
typedef struct (
    OpType          opcode;
    GraphIDType     graphID;
    EdgeIDType      edgeID;
    EdgeAttributesType attributes;
) EdgeAttrChangedMsg;
```

Where :

opcode - *EDGE_ATTR_CHANGED*

graphID - *the ID of the graph containing the edge(s) or the default edge attributes that were changed.*

edgeID - 0 - *default edge attributes of the graph were changed*

otherwise - *ID of the edge whose attributes were changed*

attributes - *the attributes values of the changed attributes. Only the values of the attributes set to have changed with the changedAttributes flag are guaranteed to be correct.*

Added To Selection Message

The added to selection message is sent to inform programs that nodes and edges have been selected in a graph. The format of the added to selection message is as follows:

```
typedef struct {
    OpType          opcode;
    GraphIDType     graphID;
    CounterType     numberOfNodes;
    CounterType     numberOfEdges;
    NodeIDType      nodesSelected[numberOfNodes];
    EdgeIDType      edgesSelected[numberOfEdges];
} AddedToSelectionMsg;
```

Where :

opcode - *ADDED_TO_SELECTION*

graphID - *the ID of the graph in which the nodes and edges were selected*

numberOfNodes - *the number of nodes that were selected in the graph*

numberOfEdges - *the number of edges that were selected in the graph*

nodesSelected - *a list of the IDs of each node that was selected in the graph*

edgesSelected - *a list of the IDs of each edge that was selected in the graph*

Removed From Selection Message

The removed from selection message is sent to inform programs that nodes and edges have been removed from a graph's selection. The format of the remove from selection message is as follows:

```
typedef struct {
    OpType          opcode;
    GraphIDType     graphID;
    CounterType     numberOfNodes;
```



```

CounterType      numberOfEdges;
NodeIDType       removedNodes[numberOfNodes];
EdgeIDType       removedEdges[numberOfEdges];
} RemovedFromSelectionMsg;

```

Where :

opcode - *REMOVED_FROM_SELECTION*

graphID - *the ID of the graph containing the nodes and edges that were removed from the selection*

numberOfNodes - *the number of nodes that were removed from the selection*

numberOfEdges - *the number of edges that were removed from the selection*

removedNodes - *a list of the IDs of each node that was removed from the graph's selection*

removedEdges - *a list of the IDs of each edge that was removed from the graph's selection*

Selection Cleared Message

The selection cleared message is sent to inform programs that there are no longer any selected nodes or edges in a graph. The format of the selection cleared message is as follows:

```

typedef struct {
    OpType      opcode;
    GraphIDType graphID;
} SelectionClearedMsg;

```

Where :

opcode - *SELECTION_CLEARED*

graphID - *the ID of the graph whose selection set was cleared*

5.3 Suggested Animation Techniques

This section describes a few of the techniques that can be used to animate algorithms when they are being run with GraphUI. The techniques are founded on two ideas, graph creation and graph modification.

Creation of Graphs

Some algorithms create graphs based upon a graph provided as input to the algorithm, such as an algorithm which produces a random spanning tree of a given graph. Other algorithms simply produce graphs as part of their output, such as an algorithm that creates data flow diagrams. Programs that create graphs can use GraphUI to build new graphs. This has the advantage that it permits the user to watch as a graph is built and to save the results of an algorithm as a picture.

Manipulation of Graphs.

Graphs that are used as input or created as output of an algorithm can be manipulated by programs to give the user a better understanding of what the algorithm is doing. Having GraphUI visually manipulate a graph as an algorithm manipulates it, can effectively animate the operation of the algorithm. Here are a few suggestions of how to visually manipulate a graph when programming with GraphUI.

Send the appropriate commands to GraphUI to echo any operations in the algorithm that alter the structure of the graph. In echoing additions or deletions of nodes and edges, the user is made aware of how the algorithm is affecting the input graph.

Use the selection set of the graph to single out particular nodes and edges from the rest of the graph by clearing the selection and selecting the nodes and edges of interest. These selected nodes and edges appear in gray and are easily distinguished from their unselected black counterparts.

Change the radius of nodes to make them larger or smaller. This is a change that is readily apparent to a user, as is changing the thickness of edges.

Moving nodes is also useful for illustration to the user. The contract edge command provided with GraphUI is much easier to understand when the operation is animated by moving the tail node of an edge to the head node. One can determine which edges should be attached to the head node as the operation is being done.

Label nodes and edges with attributes that are being changed by the algorithm. This way a user may watch the new values as they are set by the algorithm.

Animation can be very useful in the early stages of algorithm development to help in debugging the algorithm. Although, animation requires extra work on the part of the programmer to send the appropriate commands to GraphUI, the extra work will result in a user interface for graph programs that is both powerful and easy to use.

5.4 Suggested Protocol Techniques

Interaction between GraphUI and a program can be done in a variety of ways. The design of GraphUI is such that it can be used from a number of different programs running on different computers connected in different ways. There is more than one way to use GraphUI once a program has connected to it. Programs can use the information messages sent from GraphUI and manage all of the graphs open in GraphUI. Programs can ignore all information messages except run algorithm messages and get the information they require once they have been asked to execute an algorithm. Programs can send commands synchronously, waiting for each reply before

continuing, or asynchronously continuing and getting replies later when they have arrived.

Experiments with protocol techniques for using GraphUI with applications that run on Macintosh computers were done and the following observations were made. If a single computer is used to run both GraphUI and the graph program, the application must not request synchronous reads and writes when using the connection manager. Synchronous reads and writes do not give other running applications processing time and deadlock results. If a read is issued before the reply is sent by GraphUI, the machine locks.

One solution to the problem of not being able to specify that a read is synchronous is to never wait for replies. This method works but it precludes the use of any GraphUI command that returns information, greatly reducing the use of GraphUI.

A second solution is to wait for GraphUI to send its reply before attempting to read it. This is achieved by a polling loop that continually polls the connection manager waiting for that data that is to be read. For polling to work when both GraphUI and the graph program are running on the same computer, the `WaitNextEvent` system call must be invoked from within the polling loop. This call relinquishes the processor to give other programs processor time. This method allows the entire command set of GraphUI to be used.

A third alternative combines the first and second approaches by using a polling loop so that a program can receive replies but only waiting for replies that provide information about the graph. For example wait for the reply to a `get graph` command but do not wait for the reply to a `delete node` command.

This approach minimizes the amount of time a graph program sends waiting while providing the complete command set. This does ignore error codes and should only be used with connections tools that handle error correction such as the AppleTalk ADSP connection tool.

6. GraphApp: A Sample Program

A sample program, GraphApp, was written to illustrate one method of writing a program that uses GraphUI's application interface. GraphApp is designed to be used as a framework for creating Macintosh graph programs that interface to GraphUI. New graph algorithms can be added directly to GraphApp.

The sample program performs three uncomplicated graph operations: Dijkstra's shortest path algorithm which computes the lengths of the shortest paths from a source node to all other nodes in a graph; an algorithm to compute the degree of each selected node; and an operation to complete a directed graph by adding edges. The purpose of GraphApp is twofold; show how to use GraphUI's application interface; and illustrate some of the different ways that an algorithm can be animated using GraphUI.

This chapter describes the expected use of GraphUI, by both a user and a programmer. This is accomplished by describing how GraphApp's algorithms are accessed through GraphUI's application interface and how GraphApp has been designed to help the programmer to easily include new graph algorithms.

6.1 Running GraphApp

Alone, GraphApp has a very limited user interface. The only operation it supports is the quit command which terminates the program.

The remainder of user interaction with GraphApp is done through GraphUI. This section describes the user's interaction with GraphApp via GraphUI.

When GraphApp is launched a static window is opened, see Figure 6.1. This window does not change or accept input and serves only to remind a user that GraphApp is running. There are no documents associated with this application and the File Menu contains only one item, Quit.

Figure 6.1 : Static Window Displayed by GraphApp

GraphApp is run by launching it and GraphUI in either order. From GraphUI select Connection Settings from the Comm menu. A dialog box describing the configuration of the connection is displayed. If the method is not set to AppleTalk ADSP Tool, select the Method field and select "AppleTalk ADSP Tool" from the pop up menu presented, see Figure 6.2. In the scrollable area under the Name field, the word GraphApp should appear. If it does not appear, make certain GraphApp is running. Finish setting the connection by typing GraphApp in the name field or select GraphApp from the list of waiting connections, which automatically sets the Name field, and click the OK button. This operation tells the connection manager which application to connect to GraphUI. Finally, select Open Connection from the Comm menu to open the connection to GraphApp.

Figure 6.2 : Selecting AppleTalk Connection Tool

To confirm the connection is made, pull down GraphUI's Comm Menu and Close Connection should now be an available menu option. If Close Connection is not available, make sure AppleTalk is active. Once the connection is opened, GraphApp sends the commands to register its the names of its algorithms in GraphUI's Algorithms menu. "Shortest Path", "Complete Graph - Directed", and "Node Degrees" are all registered in the Algorithms menu, see Figure 6.3. These three algorithms are now available to the user.

Figure 6.3 : Algorithms Menu Items for GraphApp

6.1.1 Shortest Path Algorithm

The shortest path algorithm computes the cost of the shortest paths from a source node to all other nodes in a graph. A path is a series of edges which connect one node to another node. An edge is a path from its tail node to its head node. The cost of travelling a particular edge is stored in the edge's distance attribute. For each node the shortest distance from it to the source node is computed and stored in the node's cost attribute. The cost of a node

which is unreachable from the source node is infinity, represented by 9999999.0000 in this algorithm.

To run the shortest path algorithm, create the desired input graph, see Figure 6.4, set the distance of each edge to its appropriate value, and make note of the ID of the source node. Ensure the graph is the active window and select Shortest Path from the Algorithms menu.

The algorithm begins by writing a start of algorithm message in the Applications' Transcript. The user is then prompted with a dialog box to enter the ID of the source node. If the ID entered does not correspond to the ID of a node in the graph, an error message is written in the Applications' Transcript and the algorithm terminates. Next, the shortest path algorithm ensures that the graph is directed by making any edges arcs. Shortest path converts an edge to an arc by setting the isDirected attribute. GraphUI draws the arc with an arrow pointing to its head node. It then labels the arcs with their corresponding distances, similar to Figure 6.4. Node labels are set to the node's cost attribute and each node is initially set to a cost of infinity, implying the node has not been reached. The source node is assigned a cost of zero. As the cost of reaching each node is computed, the node's label is updated with the cost and the node is selected.

When the algorithm has finished an end of algorithm message is written in the Applications' Transcript and the selection is cleared. Figure 6.5 shows the result of running shortest path on the graph in Figure 6.4 using Boston as the source node.

Figure 6.4 : Input Graph to Shortest Path Algorithm

Figure 6.5 : Results of Shortest Path Algorithm

6.1.2 Complete Graph Algorithm

GraphUI has an editing short cut command to complete an undirected graph. When GraphUI completes a graph, if there is no edge connecting two nodes it creates an edge to connect them. A command to complete a directed graph has been included in GraphApp to illustrate that algorithms can be animated by echoing any changes an algorithm makes to a graph in GraphUI. The complete graph algorithm of GraphApp completes a graph as if it were directed.

For each pair of nodes A and B, if there is no arc connecting node A to node B, an arc connects its tail node to its head node, it adds an arc connecting A to B. Likewise if there is no arc connecting B to A, it adds an arc connecting B to A. All arcs added by complete graph are added to both the GraphApp data structure and GraphUI's displayed graph.

To use GraphApp's complete graph command, make the graph to be completed the active window and select Complete Graph - Directed from the Algorithms menu. The arcs required to complete the graph are added one at a time. The algorithm writes a start and end of algorithm message in the Applications' Transcript. Figures 6.6a, 6.6b, 6.6c show a graph before it is completed, after it has been completed by GraphUI and after it has been completed by GraphApp.

Figure 6.6a : Graph Before Being Completed

Figure 6.6b : Graph After Being Completed By GraphUI

Figure 6.6c : Graph After Being Completed By GraphApp

6.1.3 Node Degrees Algorithm

The node degrees algorithm computes the degree of nodes. The degree of a node is the number of edges that are incident to it. To use the node degrees algorithm, select the nodes for which the degree is to be computed and choose Node Degrees from the Algorithms menu.

The algorithm begins by writing a start of algorithm message in the Applications' Transcript. If no nodes are selected, a warning is written to the Applications' Transcript and the algorithm ends. The label of each selected node is set to the integer 1 field which represents the degree of the node. The integer 1 field of each selected node is set to the value of the node's degree as the degree is computed. The algorithm terminates by writing an end of algorithm message in the Applications' Transcript. Figures 6.7a and 6.7b show a graph before and after the node degrees algorithm has run.

Figure 6.7a : Graph Before Running Node Degrees Algorithm

Figure 6.7b : Graph After Running Node Degrees Algorithm

6.1.4 Quitting GraphApp

When finished with GraphApp the connection to GraphUI must be closed with GraphUI's Close Connection command or by quitting either GraphApp or GraphUI. To use GraphApp again from GraphUI open a new connection to GraphApp as described previously. Once GraphApp use of

GraphApp is no longer required, use the Quit command in GraphApp's File menu to terminate it.

6.2 Using GraphApp as a Programming Platform

To use GraphApp to build a new graph application it is necessary to understand: the graph data structures used in GraphApp, how GraphApp communicates with GraphUI, and what steps must be followed to add a new graph algorithm to GraphApp. The purpose of this section is to provide this information.

6.2.1 Graph Data Structures

GraphApp has been implemented using a hierarchy of C++ objects which serve as graph data structures, see Figure 6.8. Objects which appear in bold in Figure 6.8 are MacApp objects. The TObject class is the superclass of all MacApp objects. The TGraphModel and the TGraphComponent classes are abstract classes used to model graphs and their component parts. Neither class is directly instantiated.

In addition to providing a set of data structures, the graph objects provide programmers with methods that communicate with GraphUI. It should not be assumed that the graph classes provide all methods required to manipulate graphs for algorithms. These methods will have to be written as new algorithms require them. The few methods that are not used to communicate with GraphUI were necessary for the three graph algorithms contained in GraphApp.

Figure 6.8 : Graph Object Hierarchy
Bold - MacApp Objects

TSet - a descendent of the MacApp **TList** object. **TSet** is a container object into which groups of related objects may be placed. It is modeled after a mathematical set. It differs from **TList** in that an object may only be added to a set once; duplicate members are not permissible.

Instance Variables :

None

Methods :

Union(set, resultingSet)

Fills resultingSet with the union of set and itself

Intersection(set, resultingSet)

Fills resultingSet with the intersection of set and itself
Difference(set, resultingSet)
Fills resultingSet with any objects in it that are not in set.
Insert(theElement)
Adds theElement to the set if it is not already in the set.
Delete(theElement)
Removes theElement from the set.
IsMember(theElement)
Returns TRUE if theElement is in the set
CountClassMembers(classID)
Returns the number of objects in it whose class ID = classID
MakeClassSubSet(classID, subSet)
Fills subSet with all elements in it whose class ID = classID.

TNodeSet - a descendent of the TSet object which can only contain TNode Objects.

Instance Variables :

None

Methods :

(See TSet)

TEdgeSet - a descendent of the TSet object which can only contain TEdge Objects.

Instance Variables :

None

Methods :

(See TSet)

TGraphComponent : an abstract data class used to group common features of any component of a graph.

Instance Variables :

fID - the numeric ID of the component used to refer to its graphical representation in GraphUI.
fGraph - is the TGraph object that this component is in.

Methods :

GetID()

Returns the fID instance variable
SetID(theID)
Sets the fID instance variable
GetGraph()
Returns the graph object this component is in
GetGraphID()
returns the numeric ID of its graph which is used to refer to the graphs graphical representation in GraphUI.
Select()
Sends a message to GraphUI to select the component
Unselect()
Sends a message to GraphUI to unselect the component

TNode : is a descendent of TGraphComponent and is the object used to represent nodes in a graph.

Instance Variables:

fEdgeList - the TEdgeSet containing all of the edges incident to this node
fProb - the node's probability attribute
fLabel - the node's label attribute
fName - the node's name attribute
fRadius - the node's radius attribute
fCost - the node's cost attribute
fCapacity - the node's capacity attribute
fWeight - the node's weight attribute
fVisited - the node's visited attribute
fSource - the node's source attribute
fTarget - the node's target attribute
fInt1 - the node's generic integer 1 attribute
fInt2 - the node's generic integer 2 attribute
fReal1 - the node's generic real 1 attribute
fReal2 - the node's generic real 2 attribute
fInt1Label - the node's generic integer 1 label attribute
fInt2Label - the node's generic integer 2 label attribute
fReal1Label - the node's generic real 1 label attribute
fReal2Label - the node's generic real 2 label attribute

Methods :

MoveNodeTo(theLocation)
Moves this node to the new location in GraphUI.
AddEdge(theEdge)
Adds the edge to the edge set of the node
RemoveEdge(theEdge)
Removes the edge from the edge set of the node
Degree()

Returns the degree of the node (used in Node Degrees algorithm)
SetAttributes(theAttributes, echo)
Sets all of the node's attributes and tell GraphUI to set them if echo = TRUE
EchoAttributes(void)
Sends all of the nodes attributes to GraphUI (faster than setting one at a time)
GetAttributes(theAttributes)
Puts the attributes of this node into a NodeAttributesType structure.

The remaining attributes of TNode are Setxxx and Getxxx methods. Setxxx(attribute, echo) methods set the value of the corresponding attribute and has GraphUI set the value if echo = TRUE. For example SetProbability(prob, echo), sets the node's probability attribute. Getxxx methods return the value of a given attribute. These commands get the value as stored by GraphApp, they do not go to GraphUI for the value. A complete list of the Setxxx and Getxxx methods is provided in Appendix B.

TEdge :

Instance Variables :

fHead - the TNode object that is the head node of this edge
 fTail - the TNode object that is the tail node of this edge
 fProb - the edge's probability attribute
 fLabel - the edge's label attribute
 fName - the edge's name attribute
 fThickness - the edge's thickness attribute
 fIsDirected - the edge's isDirected attribute
 fFlow - the edge's flow attribute
 fDistance - the edge's distance attribute
 fCost - the edge's cost attribute
 fCapacity - the edge's capacity attribute
 fInt1 - the edge's generic integer 1 attribute
 fInt2 - the edge's generic integer 2 attribute
 fReal1 - the edge's generic real 1 attribute
 fReal2 - the edge's generic real 2 attribute
 fInt1Label - the edge's generic integer 1 label attribute
 fInt2Label - the edge's generic integer 2 label attribute
 fReal1Label - the edge's generic real 1 label attribute
 fReal2Label - the edge's generic real 2 label attribute

Methods :

GetHead()

Returns the TNode object that is the head node of this edge

GetTail()

Returns the TNode object that is the tail node of this edge

SetHead(theHead)

Sets the head node of this edge to theHead

SetTail(theTail)

Sets the tail node of this edge to theTail

SetAttributes(theAttributes, echo)

Sets all of the edge's attributes and tell GraphUI to set them if echo = TRUE

EchoAttributes(void)

Sends all of the edge's attributes to GraphUI (faster than setting one at a time)

GetAttributes(theAttributes)

Puts the attributes of this edge into a EdgeAttributesType structure.

The remaining attributes of TEdge are Setxxx and Getxxx methods. Setxxx(attribute, echo) methods set the value of the corresponding attribute and has GraphUI set the value if echo = TRUE. For example SetThickness(thickness, echo), sets the edge's thickness attribute. Getxxx methods return the value of a given attribute. These commands get the value as stored by GraphApp, they do not go to GraphUI for the value. A complete list of the Setxxx and Getxxx methods is provided in Appendix B.

TGraphModel: is an abstract data class used to hide graph interaction with GraphUI from the programmer.

Instance Variables :

fID - the numeric ID of the corresponding graph open in GraphUI.

fInt1 - the graph's generic integer 1 attribute

fInt2 - the graph's generic integer 2 attribute

fReal1 - the graph's generic real 1 attribute

fReal2 - the graph's generic real 2 attribute

fInt1Label - the graph's generic integer 1 label attribute

fInt2Label - the graph's generic integer 2 label attribute

fReal1Label - the graph's generic real 1 label attribute

fReal2Label - the graph's generic real 2 label attribute

Methods :

GetID()

Returns the fID instance variable

AddNode()

Adds a node to the corresponding GraphUI graph and returns the ID of the new node.

AddEdge(head, tail)

Adds an edge to the corresponding GraphUI graph and returns the ID to refer to the edge as

DeleteNode(theNode)

Deletes the node from the corresponding GraphUI graph

DeleteEdge(theEdge)

Deletes the edge from the corresponding GraphUI graph

Select(theSelection)

Selects every graph component : theSelection set in the corresponding GraphUI graph

Unselect(theSelection)

Unselects every graph component in theSelection set from the corresponding GraphUI graph

GetSelection()

Get the GraphUI selection set for this graph and returns a set of every selected graph component

ClearSelection()

Clears the GraphUI selection set for this graph

SetAttributes(theAttributes, echo)

Sets all of the graph's attributes and tell GraphUI to set them if echo = TRUE

EchoAttributes(void)

Sends all of the graph's attributes to GraphUI (faster than setting one at a time)

GetAttributes(theAttributes)

Puts the attributes of this graph into a GraphAttributesType structure.

The remaining attributes of TGraphModel are Setxxx and Getxxx methods. Setxxx(attribute, echo) methods set the value of the corresponding attribute and has GraphUI set the value if echo = TRUE. For example SetInt1(int1, echo), sets the graph's probability generic integer 1 attribute. Getxxx methods return the value of a given attribute. These commands get the value as stored by GraphApp, they do not go to GraphUI for the value. A complete list of the Setxxx and Getxxx methods is provided in Appendix B.

TGraph: is the basic graph class. This class type does not make any assumptions as to whether or not the graph is directed and can thus accommodate graphs obtained directly from GraphUI.

Instance Variables :

fNodeList - a TNodeSet object containing all of the TNode objects in this graph.

fEdgeList- a TEdgeSet object containing all of the TEdge objects in this graph.

Methods :

NewNode()

Creates a new node for this graph, sends correct commands to add the new node in the corresponding GraphUI graph, and returns the newly create TNode object.

NewEdge(head, tail)

Creates a new edge which has head node, head, and tail node, tail, and sends the command to add the new edge in the corresponding GraphUI graph. It returns the newly create TEdge object.

RemoveNode(theNode)

Removes the node from the node set and deletes the node from the GraphUI graph.

RemoveEdge(theEdge)

Removes the edge from the edge set and deletes the edge from the GraphUI graph.

GetEdgesBetween(node1, node2)

Returns an edge set containing each edge that connects node1 and node2.

GetNodeFromID(nodeID)

Returns the TNode object whose GraphUI ID is equal to nodeID.

GetEdgeFromID(edgeID)

Returns the TEdge object whose GraphUI ID is equal to edgeID.

Adjacent(node1, node2)

Returns TRUE if there is an edge in which node1 is the tail node and node2 is the head node. Otherwise it returns FALSE. (This is used by the complete graph algorithm)

MakeAdjacent(node1, node2)

Creates an edge in which node1 is the tail node and node2 is the head node if one does not exist. The edge is added to the corresponding GraphUI graph.

TDiGraph: is the directed graph class which is descendent of TGraph.

Instance Variables :

(See TGraph)

Methods :

NewEdge(head, tail)

Creates a new edge which has head node, head, and tail node, tail, for this graph, and sends command to add the new edge in the corresponding GraphUI graph. It ensures the edge is directed, and returns the newly create TEdge object.

InitializeFromGraph(graph)

This method initializes the DiGraph from a TGraph object passed to it. It clones each of the nodes and edges in graph and adds them to the node list and edge list of this directed graph. The GraphUI graph that corresponds is not duplicated; rather each edge is made directed and the isDirected default edge attribute is set. The node and edge IDs of this directed graph correspond to the same GraphUI graph as the original TGraph did.

GetAdjacentEdge(node1, node2)

Returns the TEdge in which node1 is the tail node and node2 is the head node or NULL if there is no such edge.

TUndirectedGraph: is the undirected graph class which is descendent of TGraph.

Instance Variables :

(See TGraph)

Methods :

NewEdge(head, tail)

Creates a new edge which has head node, head, and tail node, tail, for this graph, sends correct commands to add the new edge in the corresponding GraphUI graph, ensuring that the edge is not set to directed, and returns the newly create TEdge object.

Adjacent(node1, node2)

Returns TRUE if there is an edge in which node1 is connected to node2 regardless of which is the head and which is the tail. Otherwise it returns FALSE.

MakeAdjacent(node1, node2)

Creates an edge in which node1 is the tail node and node2 is the head node if node1 and node2 are not adjacent. The edge is added to the corresponding GraphUI graph.

6.2.2 Communicating With GraphUI

Programs that use GraphUI's application interface do so through message passing. Each time GraphApp wants GraphUI to do some action, for example, change a node's label, it must send a command to GraphUI. To make it easier to access GraphUI, a set of routines were written that create, format and send all of the GraphUI commands discussed in Chapter 5. Each GraphUI command has been implemented as a method of the TGraphAppApplication object and is described in detail in Appendix B.

Each command whose reply provides information, such as the GetSelection or GetGraph commands, return the replied information in an appropriate format. Functions that obtain information do not return to its caller until a reply has been received from GraphUI. Functions that do not obtain information do not wait for a reply.

Information messages that can be received from GraphUI to inform a program when a graph has been altered are not used by GraphApp. The only unsolicited message that GraphApp waits for is an ExecuteAlgorithm message.

6.2.3 Adding Algorithms to GraphApp

Two steps must be followed to add a new algorithm to GraphApp: the algorithm must be written and the algorithm must be made available to the user. GraphApp has been divided into a number of different source code files to make it easier to add algorithms. Only the UAlgorithms.cp and UGraphApp.h files must be changed to add a new algorithm. To show how an algorithm is added to GraphApp, the method used to add Dijkstra's shortest path algorithm to GraphApp is outlined.

The first step was to assign the algorithm a name that would appear in GraphUI's Algorithms menu and an ID that was unique among the algorithms in GraphApp. The name and ID of the algorithms that are in GraphApp are found in the defines at the start of the UAlgorithms.cp file. The algorithm name was set to "Shortest Path" and the ID of the algorithm to 3, Figure 6.9. The name of the algorithm could have been any string but the algorithm ID had to be different from the other two algorithm IDs in GraphApp and not equal to zero, (algorithm ID = 0 is reserved).

```
// Algorithm IDs Currently in GraphApp. Add new algorithms here
// NOTE: algorithm ID of 0 is reserved to indicate no algorithm to run

#define COMPLETE_GRAPH 1
#define NODE_DEGREES 2
#define SHORTEST_PATH 3

// Names of algorithms that appear in the GraphUI algorithms menu
#define COMPLETE_GRAPH_ALG_NM "\pComplete Graph - Directed"
#define NODE_DEGREES_ALG_NM "\pNode Degrees"
#define SHORTEST_PATH_ALG_NM "\pShortest Path"
```

Figure 6.9 : List of Algorithm Names and IDs from UAlgorithms.cp

Once an algorithm name and ID have been assigned a command must be sent to GraphUI to add the name of the algorithm to GraphUI's Algorithms menu. The `DoSetupAlgorithmsMenu` method registers the names of GraphApp's algorithms when GraphApp connects to GraphUI. An `AddAlgorithm` call must be added to the `DoSetupAlgorithmsMenu` to make Shortest Path appear in the menu. The syntax for the add algorithm call is `AddAlgorithm(algorithm name, algorithm ID, does the algorithm need a graph)`. Figure 6.10 shows the `AddAlgorithm` calls for the algorithms in GraphApp after Shortest Path was added. The `NEEDS_GRAPH` flag tells GraphUI that the shortest path algorithm must be run on a graph.

```
// Add each of the algorithms to GraphUI's menu
AddAlgorithm(COMPLETE_GRAPH_ALG_NM, COMPLETE_GRAPH, NEEDS_GRAPH);
AddAlgorithm(NODE_DEGREES_ALG_NM, NODE_DEGREES, NEEDS_GRAPH);
AddAlgorithm(SHORTEST_PATH_ALG_NM, SHORTEST_PATH, NEEDS_GRAPH);
```

Figure 6.10 : Setting GraphUI's Algorithms Menu

GraphUI sends a `RUN_ALGORITHM` message when the user selects one of GraphApp's algorithm names from its algorithms menu. Upon receipt of a `RUN_ALGORITHM` message GraphApp calls `ExecuteAlgorithm`, a method found in `UAlgorithms.cp`. `ExecuteAlgorithm` looks at the ID of the algorithm to be run and calls the appropriate routine to perform the algorithm, see Figure 6.11. A new case statement had to be added to include the shortest path algorithm.

```

// Execute the algorithm for the user - Add additional algorithms here
based upon algorithm ID

switch (id) { // What algorithm do we execute?

case COMPLETE_GRAPH:
    this->CompleteGraph(graphID);
    break;
case NODE_DEGREES:
    this->NodeDegrees(graphID);
    break;
case SHORTEST_PATH:
    this->ShortestPath(graphID);
    break;

default:
    DisplayText("\p-- Invalid Algorithm ID --");
    break;
}

```

Figure 6.11 : The ExecuteAlgorithm Method Handles Run Algorithm Cmds

The final step to add a new algorithm to GraphApp is to write the algorithm. The algorithm can be a function or a method of the application object. Shortest Path has been implemented as a method of the application object. A description of the syntax of the method was added to UGraphApp.h. This must be done for any new method. Figure 6.12 shows the definitions of the methods that handle the algorithms presently in GraphApp. Place other new methods' descriptions just below the existing definitions.

```

// Methods which implement the graph algorithms in GraphApp
virtual pascal void CompleteGraph(GraphIDType graphID);
virtual pascal void NodeDegrees(GraphIDType graphID);
virtual pascal void ShortestPath(GraphIDType graphID);

```

Figure 6.12 : Algorithm Method Definitions from UGraphApp.h

An algorithm can do anything using the methods provided to send commands to GraphUI for managing the user interface and using the graph data structures. Shortest Path begins by obtaining the graph to run the algorithm on. Once it has a TGraph object, it changes the graph to a directed graph, because the shortest path algorithm operates on a directed graph.

The next step the shortest path algorithm takes is to determine which node is the source node. There are a number of different ways this could have been done. The source node could have been selected, or its source attribute set, or either of the two graph generic integer attributes could have been set to the ID of the source node. The way the source node is obtained in shortest path is by prompting the user for the ID of the source node. If the ID supplied does not correspond to a node of the graph the algorithm issues a warning and terminates.

Once the algorithm has the source node, it sets the source node's source attribute to true and labels the node with the source attributes so that the user has visual confirmation of the entered source node. The edges of the graph are labeled with their distances so the user is aware of the distance of each edge. The final preparation before starting Dijkstra's algorithm is to set each node to not visited and the cost attribute to the distance of the edge from the source node to the node. If there is no edge connecting the source node to it, the cost is set to infinity. The source node is set to visited, it is selected, and its cost is set to zero (there is no cost to reach the source node).

The algorithm runs by finding the node with the smallest cost from the unvisited nodes, marking it as visited, and selecting it to indicate that its shortest path has been found. It then updates the costs of the unvisited nodes and continues with the algorithm until the cost of the shortest path has been

computed for each node or the smallest cost is infinity which means the remaining nodes are not reachable from the source node.

By implementing the graph data structures so they hide most of the communication with GraphUI, the resulting shortest path program, is coded similarly to a shortest path program that does not use GraphUI. What follows is source code of Dijkstra's shortest path algorithm of a program not using GraphUI and the shortest path algorithm as taken from GraphApp. The two versions of the shortest path algorithm are included to show that GraphApp's version looks like graph code not user interface code, satisfying a criteria of the design. Finally the GraphApp's Complete Directed Graph and Node Degrees algorithms are included as two additional examples of programming with GraphUI.

Shortest Path Algorithm Without GraphUI :

```
pascal void TGraphAppApplication::ShortestPath(GraphIDType graphID) (  
  
    float          graph[MAX][MAX];          // the current graph  
    Boolean        visited[MAX];             // visited[i] - was node i visited?  
    float          cost[MAX];                // cost[i] - cost of reaching node i  
    int            sourceNode;               // number of the source node  
    int            nodeWithSmallestDist;     // node with the smallest cost value  
    int            numberOfNodes;           // number of nodes in the graph  
    int            i,j;                     // loop counters  
  
    // Tell the user we are starting the algorithm  
    printf("Starting Shortest Path Algorithm — \n");  
  
    // Get the graph from the user in the form of a distance adjacency matrix, this call  
    // returns the number of nodes in the graph  
    numberOfNodes = GetGraphFromUser(&graph);  
  
    // Get the source node to run the algorithm from  
    printf("Enter number of source node\n");  
    scanf("%i", &sourceNode);  
  
    // Make certain the node the user specified exists  
    if ((sourceNode < 0) || (sourceNode >= numberOfNodes)) {  
        printf("Source node specified does not exist, algorithm terminating\n");  
        return;  
    }  
  
    // Set up for the algorithm, we use the visited array to indicate that the  
    // shortest path to a node has been found, and the cost array to save the  
    // costs of reaching each node.  
    for (i = 0; i < numberOfNodes; i++) {  
        visited[i] = FALSE;  
        cost[i] = INFINITY;  
        // Determine if this node has an edge from the source node to this one  
        if (graph[sourceNode,i] != INFINITY) {  
            cost[i] = graph[sourceNode,i];  
        }  
    }  
  
    // Set the source node to visited and cost to zero since we are here  
    visited[sourceNode] = TRUE;  
    cost[sourceNode] = 0;  
  
    // Find the node with the smallest cost and add that node to the set of visited nodes, then  
    // update the costs of the remaining unvisited nodes  
    for (i = 0; i < numberOfNodes; i++) {  
        // find the node with the smallest distance which has not yet been visited  
        float smallestDist = INFINITY;  
        nodeWithSmallestDist = MAX;  
    }  
}
```

```

for (j = 0; j < numberOfNodes; j++) {
    if (!(visited[j]) && (cost[j] < smallestDist) ) {
        smallestDist = cost[j];
        nodeWithSmallestDist = j;
    }
}

// if the nodeWithSmallestDist is not set, then we are through, set i and get out of main
// loop
if (nodeWithSmallestDist == MAX) {
    i = numberOfNodes;
}

else {    // else keep going

    // Got the node with the smallest distance not yet visited, mark that node as visited
    visited[nodeWithSmallestDist] = TRUE;
    printf("Cost of reaching node %i is %f\n", nodeWithSmallestDist, smallestDist);

    // Update all of the distances
    for (j = 0; j < numberOfNodes; j++) {

        if ( !(visited[j]) ) {
            if (graph[nodeWithSmallestDist, aNode] + cost[nodeWithSmallestDist] < cost[j]) {
                cost[j] = graph[nodeWithSmallestDist, aNode] + cost[nodeWithSmallestDist] ;
            }
        }
    }
}

// Tell the user we have completed the algorithm
printf("Finished Shortest Path Algorithm -- \n");
}

```

Shortest Path Algorithm Using GraphUI :

```
pascal void TGraphAppApplication::ShortestPath(GraphIDType graphID) {

    TGraph          *graph;           // the current graph
    TDiGraph         *directedGraph;   // the current graph made directed
    NodeIDType       sourceID;         // ID of the source node
    TNode            *sourceNode;      // node to search from
    TNode            *nodeWithSmallestDist; // node to be visited next
    TNode            *aNode;           // node used in loops
    TEdge            *anEdge;          // edge used in loops
    EdgeAttributesType edgeAttr;        // attributes for all edges
    NodeAttributesType nodeAttr;        // attributes for all nodes
    AttrFlagType      changeAttr;       // which attributes should we set
    CostType          cost;             // cost of getting to a node
    int               i,j;             // loop counters

    // Tell the user we are starting the algorithm
    DisplayText("\pStarting Shortest Path Algorithm -- ");

    // Get the graph that the user is looking at
    graph = this->GetGraph(graphID);

    // Make the graph directed for this algorithm
    directedGraph = new TDiGraph;
    FailNIL(directedGraph);
    directedGraph->InitializeFromGraph(graph);
    graph->Free();    // get rid of the graph

    // Get the source node to run the algorithm from
    sourceID = (NodeIDType)GetInteger("\pEnter ID of Source Node");
    sourceNode = directedGraph->GetNodeFromID(sourceID);

    // Make certain the node the user specified exists
    if (sourceNode == NULL) {
        DisplayText("\pSource node specified does not exist, algorithm terminating");
        return;
    }

    // show the user which node is the source node
    sourceNode->SetSource(TRUE, !ECHO);
    sourceNode->SetLabel(kNodeSourceLabel, !ECHO);
    sourceNode->EchoAttributes();

    // Label all of the edges with their distances
    edgeAttr.label = kEdgeDistanceLabel;
    changeAttr = LABEL_FLAG;
    SetEdgeAttr(graphID, ALL_EDGES, edgeAttr, changeAttr);

    // Label all of the nodes with their costs so we can observe as they are computed
```



```

nodeAttr.label = kNodeCostLabel;
nodeAttr.cost = -1;
changeAttr = LABEL_FLAG | COST_FLAG;
SetNodeAttr(graphID, ALL_EDGES, nodeAttr, changeAttr);

// Set up for the algorithm, We use the visited attribute to indicate that the
// shortest path to a node has been found, and cost to save the
// cost of reaching a node. These these two variable changes are not echoed
// to GraphUI and are used as intermediate variables for this algorithm
// Cost is only echoed after we have determined that it is the cheapest way

for (i = 1; i <= directedGraph->fNodeList->GetSize(); i++) {
    aNode = (TNode *)directedGraph->fNodeList->At(i);
    aNode->SetVisited(FALSE, !ECHO);
    cost = INFINITY;
    aNode->SetCost(&cost, !ECHO);
    // Determine if this node has an edge from the source node to this one
    anEdge = directedGraph->GetAdjacentEdge(sourceNode, aNode);
    if (anEdge != NULL) {
        cost = (CostType)anEdge->GetDistance();
        aNode->SetCost(&cost, !ECHO);
    }
}

// Set the source node to visited and cost is zero since we are here
sourceNode->SetVisited(TRUE, !ECHO);
sourceNode->SetCost(0, ECHO);

for (i = 2; i <= directedGraph->fNodeList->GetSize(); i++) {
    // find the node with the smallest distance which has not yet been visited
    float smallestDist = INFINITY;
    nodeWithSmallestDist = NULL;
    for (j = 1; j <= directedGraph->fNodeList->GetSize(); j++) {
        aNode = (TNode *)directedGraph->fNodeList->At(j);
        if (!(aNode->GetVisited()) && (aNode->GetCost() < smallestDist) ) {
            smallestDist = aNode->GetCost();
            nodeWithSmallestDist = aNode;
        }
    }

    // if the nodeWithSmallestDist is not set, then we are through, set i and get out of main
    // loop
    if (nodeWithSmallestDist == NULL) {
        i = (int) (directedGraph->fNodeList->GetSize() + 1);
    }

    else { // else keep going

        // Got the node with the smallest distance not yet visited , Mark that node as visited
        nodeWithSmallestDist->SetVisited(TRUE, !ECHO);
        cost = nodeWithSmallestDist->GetCost();
    }
}

```

```

nodeWithSmallestDist->SetCost(&cost, ECHO);

// Update all of the distances
for (j = 1; j <= directedGraph->fNodeList->GetSize(); j++) {
    aNode = (TNode *)directedGraph->fNodeList->At(j);

    if (!(aNode->GetVisited())) {
        anEdge = directedGraph->GetAdjacentEdge(nodeWithSmallestDist, aNode);
        if (anEdge != NULL){
            if ( (anEdge->GetDistance() + nodeWithSmallestDist->GetCost()) <
                aNode->GetCost()) {
                cost = (anEdge->GetDistance() + nodeWithSmallestDist->GetCost());
                aNode->SetCost(&cost, !ECHO);
            }
        }
    }
}

// All done with the graph so free it
directedGraph->Free();

// Tell the user we have completed the algorithm
DisplayText("\pFinished Shortest Path Algorithm -- ");
}

```

Complete Graph Algorithm :

```
pascal void TGraphAppApplication::CompleteGraph(GraphIDType graphID) {  
    // The purpose of this command is to complete a directed graph  
  
    TGraph          *graphToComplete;  
    TDiGraph        *directedGraph;  
  
    // Tell user we are starting the algorithm  
    DisplayText("\pStarting Complete Graph Algorithm ---");  
  
    // Get the graph from GraphUI  
    graphToComplete = this->GetGraph(graphID);  
  
    // Make the graph into a directed graph  
    directedGraph = new TDiGraph;  
    FailNIL(directedGraph);  
    directedGraph->InitializeFromGraph(graphToComplete);  
    graphToComplete->Free();  
  
    // For each node in the graph see if it is connected to ever other node  
    // in the graph. If they are not connected, connect them.  
  
    for (int i = 1; i < directedGraph->fNodeList->GetSize(); i++) {  
        TNode *aNode = (TNode *)directedGraph->fNodeList->At(i);  
        for (int j = i+1; j <= directedGraph->fNodeList->GetSize(); j++) {  
            TNode *anotherNode = (TNode *)directedGraph->fNodeList->At(j);  
            directedGraph->MakeAdjacent(aNode,anotherNode);  
            directedGraph->MakeAdjacent(anotherNode, aNode);  
        }  
    }  
  
    // We no longer need the graph so free it  
    directedGraph->Free();  
  
    DisplayText("\pFinished Complete Graph Algorithm ");  
}
```

Node Degrees Algorithm :

```
pascal void TGraphAppApplication::NodeDegrees(GraphIDType graphID) {

    // Compute and display the node degrees

    TGraph          *graph;           // Graph to operate on
    TNode           *node;            // a node of the graph
    NodeAttributesType nodeAttributes; // attributes for setting node attributes
    AttrFlagType     changeLabel;      // change the attributes label
    TSet             *selection;        // all selected nodes and edges
    TNodeSet         *selectedNodes;   // Nodes selected by the user

    // Tell user we are starting the algorithm
    DisplayText("\pStarting Node Degree Algorithm -- ");

    // Get the graph to do the algorithm on
    graph = GetGraph(graphID);

    // Get the selection set of the Graph
    selection = GetSelection(graph);

    // Get a subset of containing only selected nodes
    node = new TNode;
    FailNIL(node);
    selectedNodes = NewNodeSet();
    selection->MakeClassSubSet(node->GetClass(),(TSet *)selectedNodes);

    // Calculate the degree of each selected node
    if (selectedNodes->GetSize() > 0) {

        // label each node with genericInt1, which contains the degree
        changeLabel = GENERIC_INT1_FLAG | LABEL_FLAG;
        nodeAttributes.label=kNodeInt1Label;

        // For each selected node get its degree and set its int1 variable to the degree
        for (int i = 1; i <= selectedNodes->GetSize(); i++) {
            node = (TNode *)selectedNodes->At(i);
            nodeAttributes.genericInt1 = node->Degree();
            SetNodeAttr(graphID, node->GetID(), nodeAttributes, changeLabel);
        }
    }
    else { // else there were no selected nodes
        DisplayText("\p*** Only computes degrees of selected nodes");
    }

    // Tell user we are finished the algorithm
    DisplayText("\pCompleted Node Degree Algorithm -- ");

} //Node Degrees
```

6.3 Epilogue

In developing GraphApp, experiments were done to determine the overhead caused by communicating with GraphUI, taking note of the speed of the animation of algorithms. GraphApp's communication was initially programmed using asynchronous message passing, not waiting for a reply from any command. While this method made an algorithm and its animation run very quickly, full use of GraphUI was not possible. Any command that required a reply could not be used and special states had to be created in GraphApp to obtain the active graph before running an algorithm.

The next method of communication tried was synchronous message passing. GraphApp waited for the reply to each command before continuing. This second method made all of GraphUI's commands available but the animation ran much slower.

The method of communication finally used in GraphApp was a combination of the first two methods. When sending commands whose reply only provides an error message, GraphApp does not wait for the reply and ignores it when it does arrive. When sending commands that provide information in their reply such as AddNode, GetGraph or GetInteger, GraphApp waits until the reply is received before continuing. By combining both synchronous and asynchronous message passing, speed versus functionality is optimized.

It is important to note that these experiments performed to determine the most efficient way to communicate with GraphUI required no changes to GraphUI. This shows that different types of interface experiments can be

performed but do not require the experimenter to have intimate knowledge of GraphUI.

7. Conclusions

We have demonstrated through the application interface of GraphUI that we can provide an application interface to an adaptable graph user interface tool that does not require altering the tool. Message passing with a command set and information messages allows our adaptable graph user interface tool, GraphUI, to be completely separated from the programs that use it. Programs are able to use GraphUI's application interface and run not only on the same computer as GraphUI but on other computers which can physically connect to GraphUI. The command language and information messages permit applications to control the manner in which input is obtained from users, to control how graphs and other textual information are displayed to users, and to respond appropriately to user or other program actions.

GraphUI is a completed Macintosh graph user interface server program which affords graph programs a centralized graph user interface. It was important that GraphUI adhered to the Macintosh user interface guidelines so the user interface it provides behaves in a consistent manner. Adhering to the guidelines also ensured that our program could be integrated with word processors and drawing packages for reporting results of algorithms using GraphUI's application interface.

It was discovered through experimenting with the application interface that communicating with GraphUI using a combination of asynchronous and synchronous message passing optimizes the performance of the application interface.

7.1 Future Work

With GraphUI complete there are a number of new areas that may be pursued. Most of the future work outlined for GraphUI is in specific graph theory domains and in particular in developing graph algorithm programs that use GraphUI's application interface. The following section outlines areas for additional research using GraphUI.

7.1.1 Graph Programs

The full benefits of GraphUI's application interface will not be realized until a number of different domain specific graph programs have been written to use it. There is a need for a set of fundamental graph theory algorithms to use with GraphUI. From a base of fundamental graph algorithms, domain specific applications can be constructed. Presently, plans are being made to begin work on a suite of network reliability tools to work with GraphUI. Network reliability was the original motivation for building our adaptable user interface tool.

Another area of program development which should not be overlooked is existing code. Taking an existing graph program which does not

have a satisfactory user interface and modifying it to use GraphUI or to take an existing library of graph routines and change it to animate its algorithms through GraphUI would be worthwhile.

Work can also be done with GraphUI on the animation of algorithms, in particular to animate a number of algorithms used in courses to make them easier for students to comprehend.

GraphUI has a command language that is designed to be flexible and to allow GraphUI to be used in diversified domains. Only through programming with GraphUI can it be determined if there are any features that are lacking or not well handled in the interface.

7.1.2 Communication Platforms

Currently the programs that have been written for GraphUI have been developed for Macintosh computers. The sample program included in this thesis provides a starting point for writing graph programs which run on Macintosh computers. Similar sample programs must be developed for other computers. To be included in each sample program is code that manages a physical connection to GraphUI, code that automates the sending of commands, and code that manages information messages from GraphUI.

A good starting point is to develop a sample program which communicates serially on a Unix workstation such as the Sun or the HP. No changes to GraphUI would be necessary. There is a connection manager serial tool so no programming would be necessary for the Macintosh end of the communications. The Unix sample program would be responsible for handling communications through the serial port of the computer.

It will take considerable work to implement the communications for each new platform, but once the initial program has been written, others for the same platform will be much easier, especially if the first program is designed as a platform for constructing new graph programs.

7.1.3 Extensions to GraphUI

The representation used by GraphUI to display nodes and edges is fixed. To provide a more accurate picture for a user, it is worthwhile to provide a means of changing what nodes and edges look like. This is particularly important for displaying nodes. Nodes of a graph can represent almost any form of physical data, a circle may not be sufficient to represent the data. One representation that would be worthwhile to network reliability programmers is to display nodes as subgraphs. Factoring algorithms often build graphs of subgraph components, a representation for subgraphs would therefore be useful.

The design of GraphUI focused on the application interface and the commands and information messages which it is comprised of. As programs are developed which use GraphUI, commands may need to be changed or added. One such change that is foreseen is an improved means of managing the algorithms menu. The linear algorithms menu is not well suited for programs that contain many different algorithms. A hierarchical menu system for the algorithms would help to rectify this drawback.

As program development begins, it may become apparent that certain attributes are missing from the sets of attributes and others are not being used. The attribute set should be adjusted according to the findings.

GraphUI was designed to allow graph programs to potentially run on computers other than Macintoshes by forcing the separation of user interface from graph algorithms. For Macintosh graph algorithm applications, it is possible to incorporate their algorithms directly into GraphUI, resulting in a single application. This can be done using the already defined application interface to GraphUI but forgoing the sending of messages.

7.2 Contributions

In closing we would like to summarize the contributions that were made by the work described in this thesis.

1. We have proposed and implemented a solution to the problem that existing adaptable graph user interface tools require modifying the tool's code when graph programs use their GUI capabilities and alter the default user control over graph programs. Our solution presents an adaptable graph user interface tool, GraphUI, which uses message passing to provide graph programs GUI capabilities. Programmers do not have to use the same development environment used to implement GraphUI because graph programs and GraphUI are not combined into a single program.
2. Our application interface provides an open architecture which permits several graph programs running on different computers to simultaneously access GraphUI's user interface capabilities. The open architecture of GraphUI allows users to access the functionality of different programs, possibly

running on different computers, transparently from one standardized user interface. For example, an application which specializes in graph layout can be run together with another graph application.

Appendix A : GraphUI Detailed Message Information

..... Message Error Codes

NO_ERROR	= 0
BAD_COMMAND_SIZE	= -1
INVALID_OPCODE	= -2
BAD_GRAPH_ID	= -3
BAD_NODE_ID	= -4
BAD_EDGE_ID	= -5
NO_COMMAND	= -6
NACK	= -9999

..... Message Opcodes

ADD_GRAPH	= 0x0000
DELETE_GRAPH	= 0x0001
ADD_NODE	= 0x0002
DELETE_NODE	= 0x0003
ADD_EDGE	= 0x0004
DELETE_EDGE	= 0x0005
CONTRACT_EDGE	= 0x0006
MOVE_NODE	= 0x0007
GET_GRAPH	= 0x0008
SET_GRAPH_ATTRIBUTES	= 0x0009
GET_GRAPH_ATTRIBUTES	= 0x000A
SET_NODE_ATTRIBUTES	= 0x000B
GET_NODE_ATTRIBUTES	= 0x000C
SET_EDGE_ATTRIBUTES	= 0x000D
GET_EDGE_ATTRIBUTES	= 0x000E
GET_REAL	= 0x000F
GET_INTEGER	= 0x0010
SET_ALGORITHM	= 0x0011
DISPLAY_TEXT	= 0x0013
ADD_TO_SELECTION	= 0x0014
REMOVE_FROM_SELECTION	= 0x0016
CLEAR_SELECTION	= 0x0018
GET_SELECTION	= 0x0019
DELETE_ALGORITHM	= 0x001A
SET_MESSAGING	= 0x001B

// The following are the opcodes of information messages

RUN_ALGORITHM	= 0x0012
GRAPH_ADDED	= 0x8000
GRAPH_DELETED	= 0x8001
NODE_ADDED	= 0x8002

NODE_DELETED	= 0x8003
EDGE_ADDED	= 0x8004
EDGE_DELETED	= 0x8005
NODE_MOVED	= 0x8007
GRAPH_ATTR_CHANGED	= 0x8009
NODE_ATTR_CHANGED	= 0x800B
EDGE_ATTR_CHANGED	= 0x800D
ADDED_TO_SELECTION	= 0x8014
REMOVED_FROM_SELECTION	= 0x8016
SELECTION_CLEARED	= 0x8018

***** Variable types in messages

typedef	short	OpType
typedef	OpType	*OpPtr
typedef	short	GraphIDType
typedef	short	NodeIDType
typedef	short	EdgeIDType
typedef	short	AlgorithmIDType
typedef	short	ErrorCodeType
typedef	short	StepsType
typedef	Point	NodeLocationType
typedef	float	ProbabilityType
typedef	float	CostType
typedef	float	CapacityType
typedef	float	WeightType
typedef	int	NodeRadiusType
typedef	short	EdgeThicknessType
typedef	float	FlowType
typedef	float	DistanceType
typedef	short	LabelType
typedef	Str255	NameType
typedef	Str255	PromptType
typedef	int	CounterType
typedef	long	AttrFlagType

//***** Attributes of Graphs, Nodes, and Edges

```
typedef struct {
    int          genericInt1;
    int          genericInt2;
    float        genericFloat1;
    float        genericFloat2;
    NameType     genericInt1Label;
    NameType     genericInt2Label;
    NameType     genericFloat1Label;
    NameType     genericFloat2Label;
} GraphAttributesType;

typedef struct {
    ProbabilityType  prob;
```

LabelType	label;
NameType	name;
NodeRadiusType	radius;
CostType	cost;
CapacityType	capacity;
WeightType	weight;
Boolean	visited;
Boolean	source;
Boolean	target;
int	genericInt1;
int	genericInt2;
float	genericFloat1;
float	genericFloat2;
NameType	genericInt1Label;
NameType	genericInt2Label;
NameType	genericFloat1Label;
NameType	genericFloat2Label;
} NodeAttributesType;	

typedef struct {

ProbabilityType	prob;
LabelType	label;
NameType	name;
EdgeThicknessType	thickness;
Boolean	isDirected;
NodeIDType	headNode;
NodeIDType	tailNode;
FlowType	flow;
DistanceType	distance;
CostType	cost;
CapacityType	capacity;
int	genericInt1;
int	genericInt2;
float	genericFloat1;
float	genericFloat2;
NameType	genericInt1Label;
NameType	genericInt2Label;
NameType	genericFloat1Label;
NameType	genericFloat2Label;
} EdgeAttributesType;	

Appendix B : GraphApp Implementation Details

Setxxx and Getxxx Methods of TNode

SetProbability(prob, echo)
GetProbability()
SetLabel(label, echo)
GetLabel()
SetName(name, echo)
GetName()
SetRadius(radius, echo)
GetRadius()
SetCost(cost, echo)
GetCost()
SetCapacity(capacity, echo)
GetCapacity()
SetWeight(weight, echo)
GetWeight()
SetVisited(visited, echo)
GetVisited()
SetSource(source, echo)
GetSource()
SetTarget(target, echo)
GetTarget()
SetInt1(int1, echo)
GetInt1()
SetInt2(int2, echo)
GetInt2()
SetReal1(real1, echo)
GetReal1()
SetReal2(real2, echo)
GetReal2()
SetInt1Label(label, echo)
GetInt1Label()
SetInt2Label(label, echo)
GetInt2Label()
SetReal1Label(label, echo)
GetReal1Label()
SetReal2Label(label, echo)
GetReal2Label()

Setxxx and Getxxx Methods of TEdge

SetProbability(prob, echo)
GetProbability()
SetLabel(label, echo)
GetLabel()
SetName(name, echo)
NameType GetName(void)
SetThickness(radius, echo)
GetThickness()
SetDistance(distance, echo)
GetDistance()
SetFlow (flow, echo)
GetFlow()
SetCapacity(capacity, echo)
GetCapacity()
SetCost(cost, echo)
GetCost()
SetIsDirected(isDirected, echo)
GetIsDirected()
SetInt1(int1, echo)
GetInt1()
SetInt2(int2, echo)
GetInt2()
SetReal1(real1, echo)
GetReal1()
SetReal2(real2, echo)
GetReal2()
SetInt1Label(label, echo)
GetInt1Label()
SetInt2Label(label, echo)
GetInt2Label()
SetReal1Label(label, echo)
GetReal1Label()
SetReal2Label(label, echo)
GetReal2Label()

Setxxx and Getxxx Methods of TGraphModel

SetInt1(int1, echo)
GetInt1()
SetInt2(int2, echo)
GetInt2()
SetReal1(real1, echo)
GetReal1()
SetReal2(real2, echo)
GetReal2()
SetInt1Label(label, echo)
GetInt1Label()
SetInt2Label(label, echo)
GetInt2Label()
SetReal1Label(label, echo)
GetReal1Label()
SetReal2Label(label, echo)
GetReal2Label()

Communicating with GraphUI

The following are TGraphAppApplication methods which are used to communicate with GraphUI

AddGraph()

- *returns : ID of the newly created graph*

AddNode(theGraph)

- *theGraph is the ID of the graph to add the node to*
- *returns : ID of the newly created node*

AddEdge(theGraph, head, tail)

- *theGraph is the ID of the graph to add the edge to*
- *head is the ID of the edge's head node*
- *tail is the ID of the edge's tail node*
- *returns : ID of the newly created edge*

RemoveGraph(theGraph)

- *theGraph is the ID of the graph to be closed*
- *returns : nothing*

RemoveNode(theGraph, theNode)

- *theGraph is the ID of the graph containing the node to be deleted*
- *theNode is the ID of the node to be deleted*
- *returns : nothing*

RemoveEdge(theGraph, theEdge)

- *theGraph is the ID of the graph containing the edge to be deleted*
- *theEdge is the ID of the edge to be deleted*
- *returns : nothing*

DisplayText(theText)

- *a Pascal string to be appended to the applications' transcript*
- *returns : nothing*

AddAlgorithm(theAlgorithm, ID, needsGraph)

- *theAlgorithm is the name of the algorithm for the algorithm menu*
- *ID is the numeric ID to specify when asking to run the algorithm*
- *needsGraph if TRUE must execute on a graph, FALSE no graph*
- *returns : nothing*

DeleteAlgorithm(algorithmID)

- *algorithmID is the ID of the algorithm to remove from the menu*
- *returns : nothing*

GetGraph(graphID)

- *graphID* is the ID of the graph to return
- *returns* : a TGraph object which is the graph that was obtained

GetInteger(thePrompt)

- *thePrompt* is the Pascal string to prompt the user with
- *returns* : the integer the user entered

GetReal(thePrompt)

- *thePrompt* is the Pascal string to prompt the user with
- *returns* : the real number the user entered

MoveNode(theGraph, theNode, newLoc)

- *theGraph* is the ID of the graph containing the node to be moved
- *theNode* is the ID of the node to be moved
- *newLoc* is the (X,Y) window coordinate to move the node to
- *returns* : nothing

ContractEdge(theGraph, theEdge)

- *theGraph* is the ID of the graph containing the edge to be contracted
- *theEdge* is the ID of the edge to be contracted
- *returns* : nothing

SetGraphAttr(theGraph, attr, whichAttrs)

- *theGraph* is the ID used to determine which attributes are to be set (see GraphUI's SET_GRAPH_ATTRIBUTES command)
- *attr* are the attribute values
- *whichAttrs* is a flag indicating which attributes to set
- *returns* : nothing

SetNodeAttr(theGraph, nodeID, attr, whichAttrs)

- *theGraph* is the ID of the graph containing the node
- *theNode* is the ID used to determine which attributes are to be set (see GraphUI's SET_NODE_ATTRIBUTES command)
- *attr* are the attribute values
- *whichAttrs* is a flag indicating which attributes to set
- *returns* : nothing

SetEdgeAttr(theGraph, edgeID, attr, whichAttrs)

- *theGraph* is the ID of the graph containing the edge
- *theEdge* is the ID used to determine which attributes are to be set (see GraphUI's SET_EDGE_ATTRIBUTES command)
- *attr* are the attribute values
- *whichAttrs* is a flag indicating which attributes to set
- *returns* : nothing

GetGraphAttr(theGraph, attr)

- *theGraph* is the ID used to determine which attributes are to be gotten (see GraphUI's GET_GRAPH_ATTRIBUTES command)
- *attr* are the returned attribute values

- *returns : nothing*

GetNodeAttr(theGraph, nodeID, attr)

- *theGraph is the ID of the graph containing the node*
- *theNode is the ID used to determine which attributes are to be gotten (see GraphUI's GET_NODE_ATTRIBUTES command)*
- *attr are the returned attribute values*
- *returns : nothing*

GetEdgeAttr(theGraph, edgeID, attr)

- *theGraph is the ID of the graph containing the edge*
- *theEdge is the ID used to determine which attributes are to be gotten (see GraphUI's GET_EDGE_ATTRIBUTES command)*
- *attr are the returned attribute values*
- *returns : nothing*

ClearSelection(theGraph)

- *theGraph is the ID of the graph whose selection is to be cleared*
- *returns : nothing*

AddToSel(graphID, numNodes, numEdges, theNodes, theEdges)

- *theGraph is the ID of the graph whose selection is to be added to*
- *numNodes is the number of nodes to be selected*
- *numEdges is the number of edges to be selected*
- *theNodes is a handle to a list of IDs of nodes that are to be selected*
- *theEdges is a handle to a list of IDs edges that are to be selected*
- *returns : nothing*

RemoveFromSel(graphID, numNodes, numEdges, theNodes, theEdges)

- *theGraph is the ID of the graph whose selection is to be affected*
- *numNodes is the number of nodes to be unselected*
- *numEdges is the number of edges to be unselected*
- *theNodes is a handle to a list of IDs of nodes that are to be unselected*
- *theEdges is a handle to a list of IDs edges that are to be unselected*
- *returns : nothing*

GetSelection(graph)

- *graph is the graph object whose selection set is requested*
- *returns : a set containing the node and edge objects of graph that are selected*

References

- [1] Apple Computer Inc., "Human Interface Guidelines: The Apple Desktop Interface", Addison-Wesley Publishing Company, (1987).
- [2] Apple Computer Inc., "Macintosh Communications Toolbox Reference", Apple Computer Inc., (1990).
- [3] Apple Computer Inc., "MacApp Versions 2.0B9, 2.0, 2.0.1", Apple Computer Inc., (1989 - 1990).
- [4] Duane Bailey, Janice E. Cuny, Craig P. Loomis, "ParaGraph: Graph Editor Support for Parallel Programming Environments", International Journal of Parallel Programming, Vol. 19, No. 2, pg 75-110, (1990).
- [5] Marc H. Brown, "Algorithm Animation", MIT Press, (1988).
- [6] Gasner, North, Vo, "dag - A program that draws graphs", Software - Practice and Experience", Vol 18, pg 1047-1062, (November 1987).
- [7] Michael Himsolt, "GraphEd: An Interactive Graph Editor", STACS '89, pg 532-533.
- [8] David Jablonowski, Vincent Guarna, "GMB: A Tool for Manipulating and Animating Graph Data Structures", Software - Practice and Experience, Vol. 19(3), pg 283-301 (March 1989).
- [9] Christopher V. Jones, "An Introduction to Graph-Based Modeling Systems, Part I : Overview", ORSA Journal on Computing, Vol. 2, No. 2, pg 136-151, (Spring 1990).
- [10] Kocay, William, "Groups & Graphs, a Macintosh application for graph theory", Journal of Combinatorial Mathematics and Combinatorial Computing, Vol. 3, pg. 195-206 (1988).
- [11] R. Legel, "CADCON --- computer aided design of computer networks", Systems Analysis Modeling Simulation, Vol 5(3), pg. 263-272, (1988).
- [12] "MacDraw User's Guide", Claris Computer Corporation, (1988).
- [13] "MacPaint User's Guide", Claris Computer Corporation, (1988).

- [14] Frances Newbery Paulisch, Walter F. Tichy, "EDGE: An Extendible Graph Editor", *Software - Practice and Experience*, Vol. 20(S1), S63-S88, (June 1990).
- [15] Lawrence Rowe, Michael Davis, Eli Messinger, Carl Meyer, Charles Spirakis, Allen Tuan, "A Browser for Directed Graphs", *Software - Practice and Experience*, Vol. 17(1), pg 61-67, (1987).
- [16] A. Schurr, "IPSEN-environment: an integrated and incremental project support environment", *Graph Grammers and Their Application to Computer Science*, 4th Internation Workshop Proceedings, pg. 66, (1991).
- [17] Gursharan S. Sidhu, Richard F. Andrews, Alan B. Oppenheimer, "Inside AppleTalk", Addison-Wesley Publishing Company, (1989).
- [18] R.A. Stine, C.V. Jones, "Statistical modelling using a graph-based interface", *Computer Science and Statistics: Proceedings of the 19th Symposium on the Interface*, pg. 157-62, (1987).
- [19] Walter F. Tichy, Frances J. Newbery, "Knowledge-based Editors for Directed Graphs", *Proc. 1st European Software Engineering Conference, ESEC'87, Lecture Notes in Computer Science No. 289*, pg 101-109, Springer Verlag, (1987).
- [20] Egon Wanke, "Plexus: A System for Implementing Hierarchical Graph Algorithms", *STACS '88*, pg. 403-404, (1988).
- [21] David A. Wilson, Larry S. Rosenstein, Dan Shafer, "C++ Programming With MacApp", Addison-Wesley Publishing Company, (1990).

END

2/3·12·92

FIN