DESIGN AND IMPLEMENTATION OF A GRAPH-BASED INTERFACE FOR NETWORK MODELING (GIN) USING AN OBJECT-ORIENTED APPROACH

By

CHAKRADHAR R. NANGA

Bachelor of Engineering

Bangalore University

Bangalore, India

1987

Submitted to the faculty of the
Graduate College of the
Oklahoma State University
In partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 1990

DESIGN AND IMPLEMENTATION OF A GRAPH-BASED

INTERFACE FOR NETWORK MODELING (GIN)

USING AN OBJECT-ORIENTED

APPROACH

Thesis Approved:

[illegible]

Thesis Advisor

[illegible]

Ramesh Sharda

Norman N. Durham

Dean of the Graduate College

ACKNOWLEDGMENTS

I wish to express my grateful appreciation to Dr. George E. Hedrick for his intelligent guidance and for accepting to be the major advisor and Dr. Ramesh Sharda for his interest, direction, wisdom, and counsel during the conduct of this study, writing of this thesis and throughout my graduate program. I also wish to thank my committee member Dr. David W. Miller for his guidance, inspiration, and invaluable aid.

Additionally, I wish to thank Mr. David Steiger of OXY-USA, Inc. for providing the software and the facilities for the research. Finally, I wish to express my gratitude to Dr. Keith Willett for his support by employing me as a graduate research assistant.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

General Statement of the Problem

The operations research field has made many important advances in the solution of network problems. New solution algorithms and implementation techniques have reduced the cost of solving network problems dramatically. However, very little progress has been made in the interface between these models and the model users.

Developing a user interface is very different from developing conventional software. The traditional methods, techniques, and tools that work well for software development are proving not to work so well for the development of user interface software (Hix, 1989). This demands that new approaches like object-oriented programming be used for the development of such software.

The purpose of this study is to develop a graphical user interface for a transshipment network model, using object-oriented techniques. The interface provides the capability to: 1) display the model's network structure on the CRT, 2) change the appropriate model parameter(s) corresponding to the desired 'what-if' scenario, and 3) resolve the model and graphically display the resulting optimal flows and values on the CRT.

This user interface is based on the NETFORMS(Network Formulation) concepts. NETFORMS is a modeling technique which presents mathematical problems in the form of symbolic, pictorial networks and augmented network structures (Glover, 1977).

Objective of the Study

The objective of this study is to design, implement and test an interactive graphics-based modeling system for transshipment networks using the concepts of object-oriented programming. The system supports the development, specification, modification, and use of transshipment network models. The network building phase uses interactive computer graphics to produce a pictorial representation of the network on the CRT. The system incorporates a network solver algorithm to solve the displayed network through an external solver program and then to display the results along the network on the screen.

CHAPTER II

RELATED STUDIES

User Interfaces in OR/MS

The Operations Research(OR) community, over the past few years, has seen a substantial increase in the use of microcomputers to support its traditional skills of mathematical model solutions. Important advances have been made in the solution techniques of mathematical programming models. However, significantly less progress has been made in the interfaces between these models and the model users. For many OR projects, 'the user interface has been primarily an afterthought, a necessary evil, but not given a great deal of attention' (Jones, 1988).

Yet different user interfaces for the same task produce significant, measurable differences in user performances (Card, et al, 1980). If the techniques are designed to promote maximum understanding of the complexity of the underlying problem, the user interface should be as helpful and sophisticated as the underlying algorithm (Jones, 1988).

This lack of progress in user interfaces is even more important if, as Geoffrion (1976) suggests, that the principal benefit of an OR project is 'insight, not numbers'.

Suitable representations of problems have been clearly shown to be critical to solution finding and learning. Polya (1957) suggests drawing a picture to represent mathematical problems. This is in harmony with Maria Montessori's teaching methods for children (1964). Thomas and Malhotra (1980) found that subjects given spatial representations were faster and more successful in problem-solving than subjects given an isomorphic problem with a temporal representation. Deeper understanding of relationship between problem solving and visual perception can be obtained from Arnheim (1972) and McKim (1972).

Physical, spatial, or visual representations also appear to be easier to retain and manipulate than do textual or numeric representations (Shneiderman, 1987). Werthheimer (1959) found that subjects who memorized the formula for the area of a parallelogram, A = h x b, rapidly succeeded in doing such calculations. On the other hand, subjects who were given the structural understanding of cutting off a triangle from one end and placing it on the other end could retain the knowledge more effectively and generalize it to solve related problems.

Thus the use of pictorial representations enhances the insightful analysis of mathematically modeled problems. One such pictorial representation is NETFORMS (for NETwork

FORmulationS), a modeling technique which presents mathematical programming models in the form of symbolic, pictorial networks and augmented network structures (Glover, 1977).

Hurrion (1986) and Billington (1987) report a significant improvement in a person's understanding of a semi-structured to unstructured operations research problem and the person's subsequent confidence in a solution when visual interactive modeling tools are used.

Many modeling languages have been developed to simplify the complex and involved model transformation task (Fourer, 1983). LINGO/PC (Paul, 1989) and GAMS (Bisschop, 1982) are two examples of modeling languages. Both are text-based modeling languages which require that the model be manually transformed from the modeler's form to an algebraic form. These languages then algorithmically transform the algebraic form into the optimizer form and call the solver. Tabular and summary reports are produced as outputs by both languages.

This shows that significantly little has been done as far as the graphics and user interface parts are concerned, in spite of their advantages in these kind of applications.

Creating good user interfaces for software is very difficult. Interface software is inherently difficult to write because frequently it must control many devices, each of which may be sending streams of input events asynchronously (Myers, 1989). Interface software is often large, complex, and difficult to debug and modify. An application's interface can account for a significant fraction of the code. Surveys of artificial-intelligence applications, for example, report that 40 to 50 percent of the code and runtime memory are devoted to interface aspects (Bobrow, 1986).

As interfaces become easier to use, they become harder to create. The easy-to-use, direct-manipulation interfaces popular on many modern computer systems are among the most difficult to implement. These interfaces let the user operate directly on objects that are visible on the screen, performing rapid, incremental actions (Schneiderman, 1983).

Interfaces are not only difficult to create, but there are no design strategies that guarantee that the resulting interface will be easy to learn or easy to use. The only reliable way to generate quality interfaces is to test prototypes with users and modify the design based on their comments (Swartout, 1982).

This method, called iterative design, has been used to create some of today's best interfaces. For example, a mail system's interface was tested with users and modified iteratively. In the final version, without any instruction, 76 percent of the commands that novices generated performed the expected operation, compared with 7 percent for the initial version (Good, 1984).

New technology is constantly increasing the developers' palette of interaction devices, styles, and techniques, and yet many conventional languages do not have constructs for incorporating a window, an icon, or a mouse (Hix, 1989). This suggests that new methods, techniques, and tools be used to support the revolution in user interface design.

The object oriented approach to programming is rapidly becoming accepted as a way of organizing large and complex problems so that they are modular and extensible. The programs are organized as a large collection of active objects which communicate by sending messages. Object-oriented languages are especially suitable for windowing environments.

Linton, Vlissides, and Calder (1989) state that user interfaces should be object-oriented. They further observe that objects are natural for representing the elements of a user interface and supporting their direct manipulation. Objects provide a good abstraction mechanism, encapsulating states and operations, and inheritance makes extension easy. Compared with a procedural implementation, user interfaces written in an object-oriented language are significantly easier to develop and maintain (Linton, 1989).

User Interface Design Issues

In the last decade, software has progressed rapidly from noninteractive to highly interactive programs. The direct involvement of users during the execution of these interactive programs has completely altered our view of how programs should interface to the outside world.

User interaction forces software engineers to consider many human-factors issues, such as ease of use and presentation of information. These concerns have led to interaction styles that include graphical displays, menu-based input, and mouse-based selection.

As a result, many products that prospered five years ago would meet with an early demise if they were introduced today. Even the successful interactive graphical products of today may be subject to early obsolescence if they do not evolve with the emergence of newer interaction technologies.

Good user interfaces are difficult to construct. When Wordstar's user interface was redesigned to produce Wordstar 2000, the effort was a major one, about equivalent to writing an entire new program (Dodani et. al., 1989).

Such an enormous effort suggests that little or no code from the previous version could be reused. This is probably due to the older user interfaces being intimately intertwined with the code that supported the word processing application. Such a total overhaul of code to introduce the latest interaction technology must clearly be avoided.

Interactive software provides a large incentive to implement the user interface as an independent component. Separation from the application has several advantages. First, one can subdivide the user interface into components that can be glued together. Second, it is possible to rapidly modify the interface for reuse in other applications that have similar interaction requirements. Third, one can alter or even replace the interface with no adverse affects on the code that defines the application. Finally, it is possible to develop the interface in an iterative manner, in which one produces successive prototypes until a design that satisfies the needs of the application and its users is found.

From these advantages, a number of principles and techniques emerge. The principles are reusability and encapsulation. The techniques are rapid prototyping and iterative development. To support the production of interactive graphical interfaces, one needs a design methodology and a development environment that supports each of these principles and techniques. This is where object-oriented design and programming come into play.

Object Oriented Programming

Object-oriented programming techniques are not new, but they are becoming more popular as programmers tackle increasingly complex projects. Object-oriented programming can help simplify the development of elaborate programs by breaking them down into logical objects that manage their own behavior and hide internal complexity. Windowing applications in particular are easier to develop and maintain if object-oriented programming techniques are used (Urlocker, 1989).

The term object-oriented programming has been used to mean different things, but one thing these languages have in common is objects. Objects are entities that combine the properties of procedures and data since they perform computations and save local state (Stefik et al., 1986). In traditional procedural languages like C or Pascal, the programmer defines data structures and writes functions and procedures to operate on the data. Although normally a correspondence exists between functions and the data on which they operate, most procedural languages offer no formal support for this correspondence; it is entirely the programmer's responsibility to manage an abstraction.

On the other hand, when a program is divided into objects, it more closely represents the logical design that is being implemented. As a result, object-oriented programs are generally easier to understand and maintain than the procedural programs (Urlocker, 1989).

Definitions of terms used with object-oriented programming follow.

Classes

Every object belongs to a class that defines its type. We say that an object is an instance of its class. The class definition comprises of two things: the specifications for data formats and the coding of the methods the instances of the class will later respond to.

Different classes are related according to a hierarchy. One ancestor class might give rise to many descendant classes, which might, in turn, be ancestors of yet other classes. The instructions and data formats are passed along to descendants.

Methods and Messages

All of the action in object-oriented programming comes from sending messages between objects. Programming in an object-oriented language involves creating objects and sending them messages or commands to do things. For example, we can create a window and then show it on the screen. In the object-oriented language Actor, this is done using the messages shown below:

```
          W := defaultNew(Window, "SAMPLE");  /*First create
it*/           show(W,1);  /* display it */
```

In this case Window is a predefined Actor class. When W receives the show message, the Actor language system looks for the matching set of instructions in its class definition, and executes them. This results, in this case, in the corresponding window being shown on the screen.

Thus, to get anything done in an object-oriented language, a message is sent to an object. The object looks to see if it knows how to do what it has been asked to, and if it does, it executes the correct function or procedure. In object-oriented languages, however, these are not called functions or procedures--they are called methods. More formally, programming in an object-oriented language is a process of sending a message to an object. The message is then matched up with a method, which is then executed.

Messages can be sent to the objects by stating the method one wants to execute, followed by the object, called the receiver, to which the message is being sent.

```
print("Hello");
```

A message looks similar to a procedure or a function call in a procedural language--on purpose. However, we are not sending a parameter to a procedure. We are sending a print message to "Hello", a String object. "Hello" looks to see if it has a method defined by the name of print, and if it does, it executes that method.

Polymorphism

The difference between methods and messages should be kept clear in mind. Often the terms are used interchangeably, but although they are related, they are two different concepts. Consider the examples below:

```
print(14);
print("Hello");
```

In both cases, we are sending a print message to an object. However, in the first case the receiver is 14, an instance of class Int. In the second case, the receiver is "Hello", an instance of class String. Although we are sending the same print message, different print methods will be executed in each case. Putting this in general terms, we say that the same message can result in the execution of different methods, depending on the class of the receiver. This quality is called polymorphism. It is a very powerful concept, because it more closely parallels the way we think.

Data Abstraction

Message sending supports an important principle in programming: data abstraction. This means that the calling programs should not make assumptions about the implementations and internal representations of data types that they use. Its purpose is to make it possible to change underlying implementations without changing the calling programs. A language supports data abstraction when it has a mechanism for bundling together all of the procedures for a data type. In object-oriented programming the class represents the data type and the values are its instance variables; the operations are methods the class responds to.

Objects have a clear division between public protocol and private implementation. For example, we might have a stack object that defines a public protocol based on the push and pop operations. The stack may be implemented as an array with variables that maintain the first and the last positions, but this representation would be considered private. By adhering to the public protocol, we can change the implementation of stacks, say, to linked lists, without having to rewrite any of our code.

Inheritance

Specialization is a technique that uses class inheritance to elide information. Inheritance enables the easy creation of objects that are almost like other objects with a few incremental changes. Inheritance reduces the need to specify redundant information and simplifies updating and modification, since information can be entered and changed in one place.

Mechanisms like inheritance are important because they make it possible to declare that certain specifications are shared by multiple parts of a program. Inheritance helps to keep programs shorter and more tightly organized.

The simplest model of inheritance is hierarchical inheritance. In a hierarchy, a class is defined in terms of a single super class. A specialized class modifies its superclass with additions and substitution. Addition allows the introduction of new variables, properties, or methods in a class, which do not appear in one of the superclasses in the hierarchy. Substitution (or overriding) is the specification of a new value of a variable or property, or a new method for a selector that already appears in some superclass. Both kinds of changes are covered by the following rule. All descriptions in a class (variables, properties, and methods) are inherited by a subclass unless overridden in the subclass.

Using inheritance one can focus on those parts of the program that are application specific. Inheritance encourages the development of small, reusable classes that become building blocks for more sophisticated classes. Inheritance also lets programmers customize existing objects for unique behavior without writing the object from scratch. This approach results in less code to maintain and test and more rapid development from prototype to final application.

Late Binding

By their nature, object-oriented languages institute late binding, the ability to match messages with the appropriate methods at run time rather than at compile time. This contrasts with an early-bound language such as C or Pascal in which the variables and functions which work on them are matched together, or bound, at compile time.

The power of late binding is evident in the case where we want to perform some general task like printing an object when we don't know what type of object we're dealing with. At runtime, the language determines the class of the object and appropriately sends the message.

It is possible to take late binding one step further by allowing one to leave open until runtime not only the class of the receiving object, but also the message name itself. That is, one can send an arbitrary message with the perform method, and, used correctly, this can be extremely powerful. The idea is to include the message name as a parameter in the perform message:

```
perform(receiver,parameter1,parameter2,.....,selector)
```

The receiver is the object which is to receive the message. The selector is a Symbol giving the name of the method that is to be executed.

When selector is a constant, perform is a variation on the normal way of sending messages:

```
perform("Hello",#print);
Hello


Sam := 16;
perform(Sam,#sqrt);
4
```

However, the real power of perform becomes evident when selector is a variable:

```
Meth := #sqrt;
Sam:= 16;
perform(Sam,Meth);
4


Meth := #print;
perform(Sam,Meth);
16


Sam := "Hello";
perform(Sam,Meth);
"Hello"
```

The power of perform can be tapped in handling the menu selection events in an application. Proper use of this mechanism eliminates much of the control structure that tends to complicate conventional code. This supports a very clean, data-driven approach to programming.

Windowing Environments

One of the major advantages of object-oriented programming languages is the ability to control windowing environments. As a matter of fact, windowing and object-oriented programming have co-evolved over the last two decades as a revolution in the programming environments.

Most windows exhibit the same fundamental behavior - they are created, opened, closed, and moved around on the screen. Although windows are simple to use, window behavior is complex to program. And with procedural languages like C, there is no concept of a class to organize and retain the complex behavior of windows. It must be reprogrammed into each window.

On the other hand, systems like Actor have predefined window classes that already behave in a standard way. A window is merely an instance of a window class. Once the class is defined, the windows just roll off the assembly line. One never reinvents the window. The advantage is that all generic capabilities, for example, of a window class, like resizing, displaying and dragging work properly without having to write or test a single line of code.

Summary

Specialization and message sending synergize to support program extensions that preserve important invariants (Stefik, et al., 1986). Polymorphism extends downwards in the inheritance network because subclasses inherit protocols. Instances of a new subclass follow exactly the same protocols as the parent class, until local specialized methods are defined. Splitting a class, renaming a class, or adding a new class along an inheritance path does not affect simple message sending unless a new method is introduced. Similarly, deleting a class does not affect message sending if the deleted class does not have a local method involved in the protocol. Together, message sending and specialization provide a robust framework for extending and modifying programs.

Scope of the Study

This study applied the concepts of object-oriented programming to design and develop a system, called GIN (an acronym for Graph-based Interface for Network Modeling), used for formulating, solving and analyzing minimum cost flow network models. The system is implemented in an interactive, graphics-based, microcomputer environment using the pictorial representations of NETFORMS.

The implementation of the system utilized the following tools:

* Actor - an object-oriented programming language and a sophisticated development environment, running under

Microsoft Windows.

* Microsoft Windows Software Development Kit - a set of libraries and utilities that let one create multitasking, device-independent window based applications.

CHAPTER III

USER INTERFACE IMPLEMENTATION

Message Protocols

We define eight class objects which were developed to implement the user interface for GIN. These class objects are:

(1) NetWindow class;

(2) TboxWindow class;

(3) ArcTool, SupplyTool, and DemandTool classes;

(4) SupplyNode and DemandNode classes;

(5) SupEditNodeWin and DemEditNodeWin classes;

(6) ArcDialog class;

(7) NetDataBase class; and

(8) CostFlowDialog class.

Together these classes are placed into the instance hierarchy shown in Figure 1. This figure also includes an additional abstract class object, Object, which serves as a starting point, or root, for the class tree. This class defines protocols common to all objects in the class tree. The classes Window, Edit and Dialog are all formal classes defined by Actor, and the rest of the classes shown in the class tree inherit class and instance methods from these,

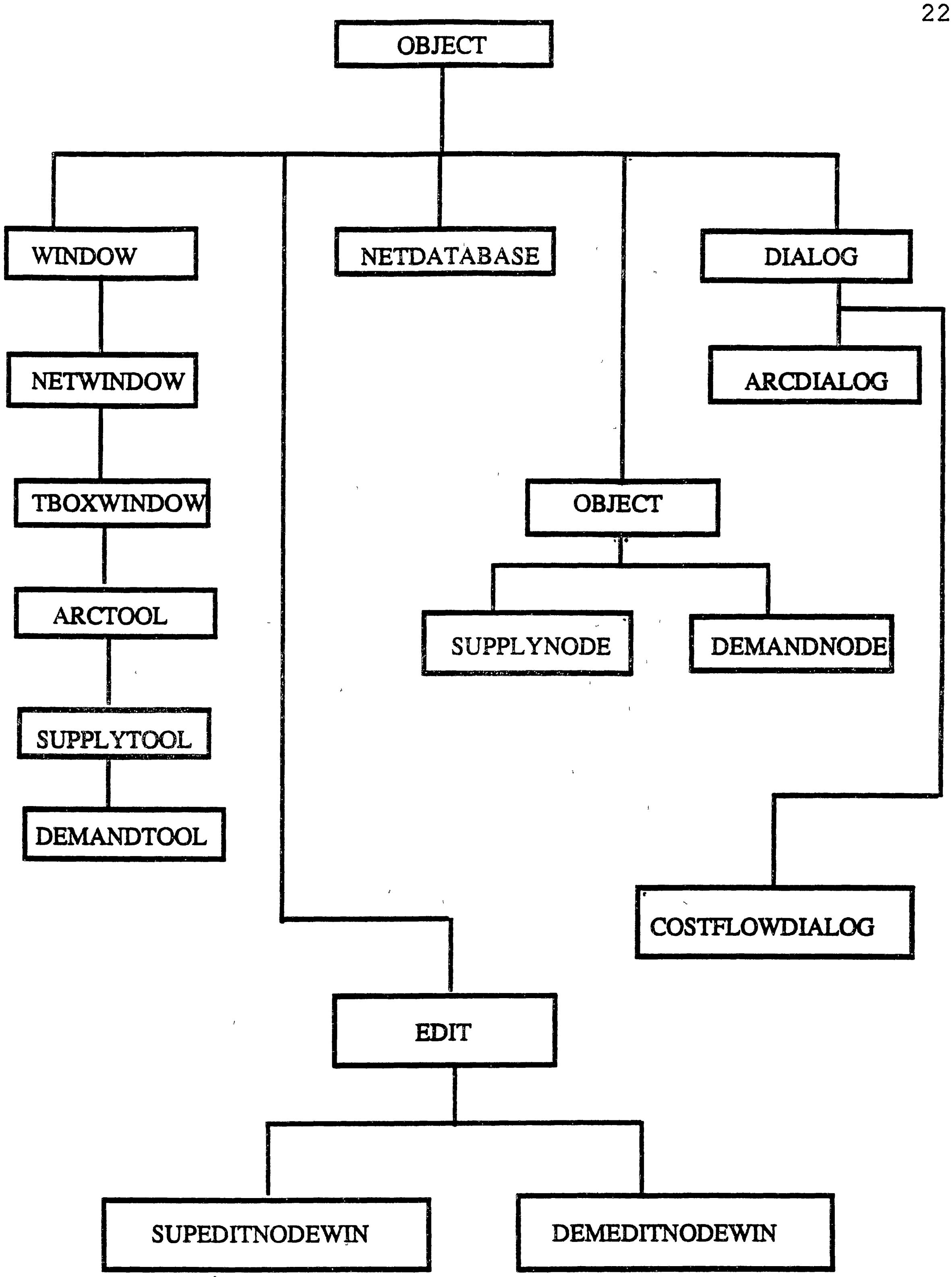

Figure 1. Instance Class Hierarchy

depending on their hierarchical placement in the class tree.

In the following pages I describe the classes developed for the GIN interface, along with information for each class, such as which object it inherits from, which objects inherit from it and the details of the methods defined for that class.

NetWindow Class

The NetWindow class maintains a picture of a network in a window. The NetWindow is responsible for the drawing of the network. NetWindow descends from the class Window and inherits all of its instance variables and methods. In addition, more instance variables and methods are added to this class to suit network creation and manipulation. This class contains the main command method which matches the command messages sent by Actor in response to menu events and edit control changes. This method in turn sends additional messages to other objects, depending on the wParam and lParam values it receives as a part of the message. For example, this method sends an updateName message to a SupplyNode object whenever the name of a supplynode changes.

Inherits From:	Window
Inherited By:	TboxWindow

Instance Message Protocols

init

The new method which is used to create an object sends out the init message, which is responsible for initializing the newly created object's instance variables. This method in turn sends out messages to other objects and to itself which results in the setting of the node lists, calculation of the textmetrics, loading of the appropriate instance variables with the values of handles to the supply and demand node icons, setting of the default arc, cost and flow colors and finally setting the menus.

setMenus

Creates a new dictionary and loads menu resources. Also, sets up a menu action table each entry of which consists of a constant ID and a message that will be performed.

command

Handles menu events. If the contents of the edit control change, messages are sent to the corresponding objects indicating the change. Looks up the ID in the actions table and performs the appropriate method, if found.

drag

Responds to mouse drags and keeps updating the point object endpt to the corresponding mouse position.

beginDrag

Sets the input focus to the window and the mouse position is saved in the instance variables. Also sets the instance variable icontodraw to a value corresponding to the currently chosen icon in the toolbox.

endDrag

Draws the supplynode or demandnode icon at the specified point in the client rectangle. If the connect option has been chosen from the toolbox the end point is noted down and an arc is drawn from begpt to endpt.

initNodeLists

Initializes the supply, demand, frompoint, topoint lists. Each of them are initialized with OrderedCollection objects of size ten to start with. These individual lists will grow as objects are added to them.

updateArcs
Updates the beginning and end points of the arcs connecting two nodes by modifying the fromPointList and toPointList respectively.

drawConnections
Connects all the nodes, by drawing arcs, after all the supply and demand nodes are painted.

getNodeName
Creates and runs a modal dialog box so that the user can enter the name of the supply and demand node created.

invertToolBox
Inverts the appropriate child windows in the toolbox depending on which drawing tool has been chosen.

findNodeType
Finds and returns the node type (supply or demand) depending on the ID parameter passed.

getFromNodeName
Returns the name of the fromNode, corresponding to its relative screen display position.

getToNodeName
Returns the name of the toNode, corresponding to its relative screen display position.

dist
Returns the distance between two points on the screen. The returned value is in the form of a point object.

fileSave
Responds to the menu choice to save the file. If the file is not yet named, prompts the user for a name.

fileSaveIt
Saves the networks data into a .NET file on disk. The file conforms to the NETFLO program's FORTRAN data file format.

savePicFile
Saves information regarding the point object lists into a .PIC file on disk which is essential for the reconstruction of the graphical network image

on the screen.

fileOpen
Pops up a FileDialog box which is used to read the name of the network file to be opened for processing.

fileOpenIt
Opens the named file to read the network information from disk. Copies the file into a stream so that it can be read faster. Data is read from the streams corresponding to PIC and RES files and messages are sent to supply and demand nodes which result in the corresponding network, with associated results being displayed on the screen.

setUpFromPointList
Reads the lines corresponding to frompoint locations from the stream and adds them in sequence to the fromPointList object.

setUpToPointList
Reads the lines corresponding to the topoint locations from the stream and adds them in sequence to the toPOintList object.

checkError
Checks if there is a file error and displays an error box. Returns the error number or zero for no error.

getFromNodeNum
Returns the number of the fromnode, corresponding to its relative screen display position.

getToNodeNum
Returns the number of the tonode, corresponding to its relative screen display position.

getNodeReq
Reads the node requirement of a particular node.

readResults
Reads the results from a .RES file and initializes the cost/flow list objects.

initTextMetrics
Initializes the instance variables tmWidth and tmHeight to the current width and height (in terms of one hundredths of an inch) of characters.

arcColor
    Unchecks the previously selected arc color and places a check mark in front of the newly chosen one. Also, sets the instance variable arccolor to a constant which reflects the currently chosen color.

findIndex
    Traverses through the from and to node orderedCollections and checks for the existence of the specified node. Returns a value of -1 if the node is not found.

chngSupNodeName
    Runs a dialog box to read the name of the old supply node and its new name. If the supplyNode exists then its name is changed to the new name and messages are sent to the appropriate objects indicating this change. An error message is flashed if the node does not exist.

chngDemNodeName
    Runs a dialog box to read the name of the old demand node and its new name. If the demandNode exists then its name is changed to the new name and messages are sent to the appropriate classes indicating this change. An error message is flashed if the node does not exist.

paint
    Sends additional paint messages to the supply, demand node objects which results in the screen being redrawn.

deleteNode
    Runs a RDNDNAME_DIALOG object to read the name of the node to be deleted. If the node exists then messages are sent to the corresponding object resulting in its deletion from the screen as well as from the internally maintained database. An error message is flashed when an attempt is made to delete a node which does not exist.

deleteArc
    Runs a CHNDNM_DIALOG to object to read the names of the nodes connected by the arc. If the arc exists then messages are sent to the corresponding object resulting in its deletion from the screen as well as the internally maintained database. An error message is flashed when an attempt is made to delete a node which does not exist.

TboxWindow Class

This is the class associated with the "TOOL_BOX" window. When the application starts up, a toolbox window is created as a child window of the main NetWindow. Also, this window contains three child windows -- one each for the supply, demand, and arc tools.

Inherits From:	NetWindow
Inherited By:	None

Instance Message Protocols

create
    Creates a child window with a border and a caption TOOLBOX. This is the toolbox window.

init
    Initializes the symbol 'inverted' to nil indicating that none of the drawing tools have been chosen from the toolbox currently.

createChildren
    Sends 'new' messages to the supplyTool, demandTool and arcTool classes resulting in the creation of three child windows, one for each of the drawing tools.

paint
    Sends 'show' messages to each of the newly created child windows, which results in their display on the screen.

ArcTool, SupplyTool, and DemandTool Classes

Arc, Supply and DemandTool classes are responsible for keeping track of which one of them has been currently chosen by the user. For example, if the user presses the mouse button in the supplyTool window, a message is sent to this object indicating that a mouse click event has occurred. Now, this object sends messages to itself to invert its client rectangle, indicating that it has been chosen. Similarly, the other two objects are responsible for keeping track of whether they have been chosen or not. These two objects also invert their client areas in response to messages they receive if any mouse activity takes place in their respective windows' client rectangles.

Inherits From:	NetWindow
Inherited By:	None

Instance Message Protocols

create
Creates an arc(supply, demand)Tool child window with border to display the connect(supply, demand)node icon.

endDrag
Sets the symbol 'icontodraw' to LINE(ICON1, ICON2) so that any further mouse activity results in the corresponding icon being displayed. In addition, the invertToolBox message is sent to invert the corresponding child window in the toolbox based on the currently selected tool.

paint
Redraws the CONNECT(supplynode, demandnode) icons in the toolbox window.

invertArc(Sup, Dem)Box
Inverts the Arc(Sup, Dem) child window in the parent toolbox window.

SupplyNode and DemandNode Classes

SupplyNode and DemandNode classes descend directly from the NetNode class. The SupplyNode and DemandNode objects keep track of their name, type and the relative display position on the screen. The relative position of these nodes have to be saved to avoid recomputing these coordinate values each time the NetWindow has to redraw itself. In fact, the SupplyNodeList and DemandNodeList maintained by the NetNode class are lists of instances of the objects of the respective SupplyNode or DemandNode classes.

Inherits From:	Object
Inherited By:	None

Instance Message Protocols

initNupdateSupply(Demand)
Creates a supply(demand) node and initializes its instance variables with the node's relative display position on the screen, the type of the node and its name.

setUpSup(Dem)List
Reads the lines corresponding to the supply(demand) nodes and adds them to the supplyNodeList.

addToSupply(Demand)List
Adds the supply(demand) node object to the corresponding supply(demand)NodeList.

paintSupply(Demand)Nodes
This method draws the supply(demand) node icons on the screen at their relative display positions. In addition, it displays the node's name next to it.

SupEditNodeWin and DemEditNodeWin

Whenever a supply or a demand node is created a name must be provided. Appropriate editing facilities are provided for this. The classes SupEditNodeWin and DemEditNodeWin, which descend from the Edit class, serve this purpose. Whenever the name of a supply or demand node is changed messages are sent by these controls to the appropriate objects indicating the change.

Inherits From:	Edit
Inherited By:	None

Instance Message Protocols

init
  Initializes the newly created edit control to SUPPLY(DEMAND).

updateSup(Dem)NodeName
  Modifies value of the 'name' instance variable of the corresponding supply(demand) node object, if the contents of the edit control has changed. Also, sends messages to the dataBase class indicating the changes that have taken place.

ArcDialog

The ArcDialog class descends from the formal Dialog class. Instances of this class are used to initialize and edit the costs associated with each existing arc.

Inherits From: Dialog

Inherited By: <u>None</u>

Instance Message Protocols

initDialog
This takes care of the initializations of the dialog box before it is actually displayed. It sets the caption and some default values for the edit controls.

command
This method ends the dialog and returns a value indicating which of the OK or CANCEL buttons have been pressed.

<u>NetDataBase</u>

The NetDataBase class descends directly from the Object class. This database object is responsible for keeping track of node interconnections and arc attributes. Messages are sent to this object by supply and demand nodes if a name changes. Also, any changes to the attributes associated with an arc are sent as messages to this object. Additions or deletions of nodes/arcs to the network are also notified to this class through appropriate messages. The Database object sends messages to self to update any changes that have occurred.

Inherits From: <u>Object</u>

Inherited By: <u>None</u>

Instance Message Protocols

init
    Initializes the instance variables representing the costs associated with an arc to zero.

updtDbVals
    Updates the instance variables of the database object to reflect the changes in the attribute values associated with arc objects.

updtDbNmFlds
    Searches the database for the name, and, if found, replaces it with the new name of the supply or demand node object.

printDbase
    Prints the contents of the network database at that particular time.

createNinitDbEntry
    Creates a new instance of the database class. Sends a setDataBaseEntry message to itself.

setDataBaseEntry
    Initializes the instance variables of the newly created database object with the appropriate values

addToDataBase
    Adds a database object to the database OrderedCollection.

CostFlowDialog Class

This class descends from the Dialog class. Instances of this class are used to read and alter the color settings of costs and flows when they are displayed. The default colors for the display of costs and flows are red and green, respectively.

Inherits From:	Dialog
Inherited By:	None

Instance Message Protocols

initDialog

This sets the dialog box captions and initializes instance variables (which reflect the state of the cost/flow display colors) to their default values of red/green.

command

Sets the instance variables 'cstclr' and 'floclr' to the ID codes of the currently set radio buttons. Sends the flipFormat message to itself if the radio button settings are changed. Notes down the final color selection if the OK button is pressed.

getColor

Returns the RGB value of a color, depending on the color ID from the corresponding radio button in the groupbox of the displayed dialogbox.

flipFormat

Sets the currently chosen radio button after resetting the previously chosen ones.

Network Creation, Modification, and Solution

This chapter explains the various parts of the GIN window. Networks will be created, modified, solved and saved. Some of the tools and their purposes will be discussed. In addition, a description of network creation and its solution is explained.

The basic network window appears on the screen, as shown in Figure 2, when GIN is launched. This screen contains everything one needs to create, modify, and save a network. The window consists of a Menu Bar, Tool Box, Close Box, Size Box, Scroll Bars and a Zoom Box.

The GIN menu bar features five pull-down menus : File, ArcColors, Results, Change and Delete, in addition to the

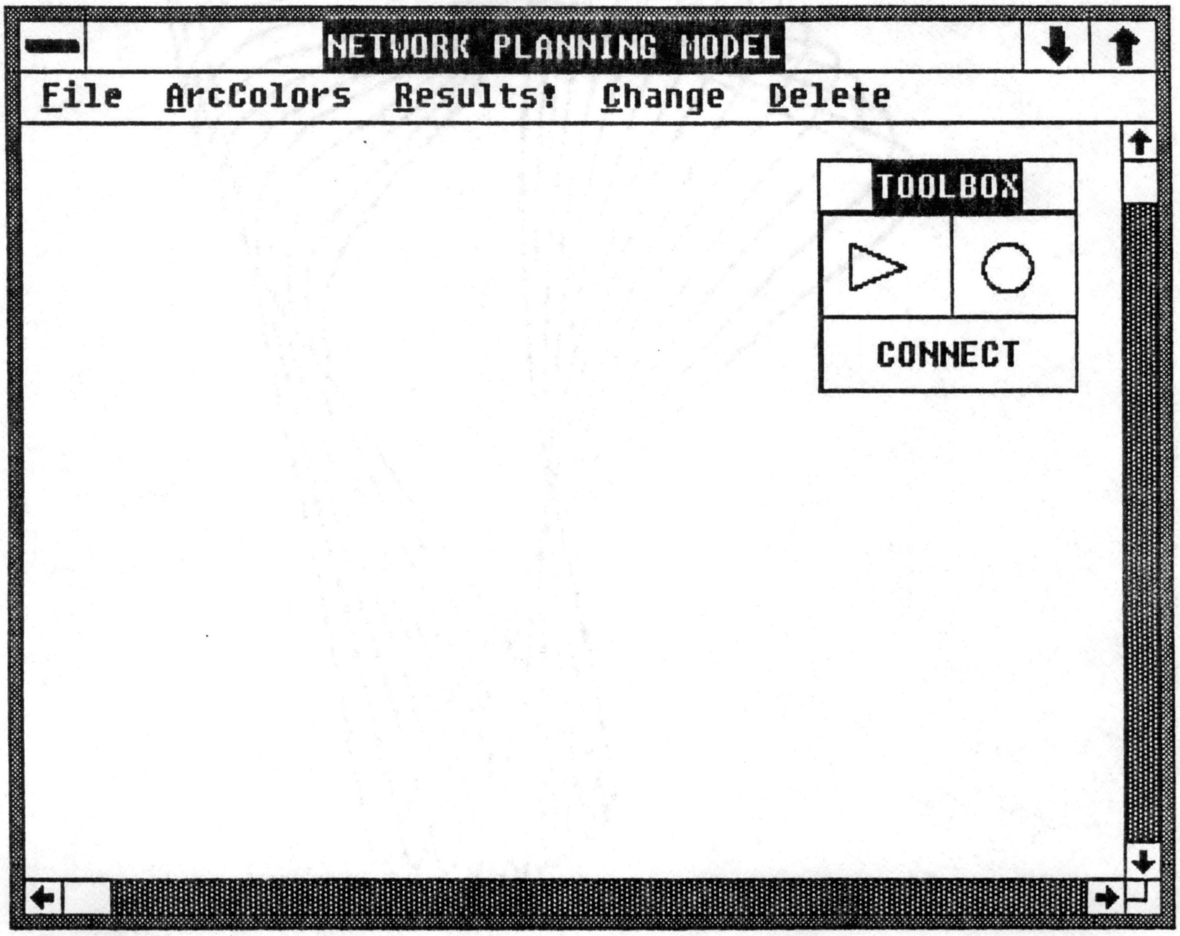

Figure 2. Basic Network Window

control menu of Microsoft Windows. Some commands will produce sub-menus or dialog boxes. When one chooses a command with a ... to its right, a dialog box will appear.

Dialog boxes prompt one to specify various settings and conditions, or choose specific options. When a dialog box is displayed, one needs to respond by clicking the appropriate button box. Some buttons in dialog boxes are outlined with a thick black line. This indicates the default action, which is accepted if the return key is pressed instead of clicking a button. A check mark preceding a menu item indicates that the item is in effect for the currently selected network.

The following is a list of the menus and the different types of commands found on each one:

FILE	Commands used to save the current network image and to open a previously saved image from a file.
ArcColors	To set the color in which the subsequent arcs will be drawn.
Results	To select the colors of the costs/flows on the result network image.
Change	Commands used to change the names of the nodes and edit arc attributes.
Delete	Commands used to delete an existing node or an arc.

The box in the right hand corner of the GIN window contains the tools one needs to create a new network. There are three different tools in the tool box : the Supply Tool, Demand Tool and Connect Tool. The tool box is a child window, which in turn contains three more child windows, one each for the supply, demand and connect tools. The tool box, being a child window, can be dragged to any part of the GIN window by clicking on its title bar and dragging to the desired location and releasing the mouse button.

A user of GIN would create the network by choosing the appropriate tools from the tool box. The supply tool is used to create a supply node and the demand tool is used to create a demand node. By choosing the connect tool an arc can be drawn between two existing nodes.

After a supply or demand node icon is chosen from the toolbox, the user has to click the mouse button in the client area of the window to create the corresponding node. As each node is created, a dialog box pops up to read the name of the newly created node. This name can be changed by choosing the appropriate option under the Change menu.

Arcs can be drawn between any two existing nodes by choosing the connect option from the toolbox. After the connect option is chosen, the user has to drag the mouse, keeping the right button depressed, from the fromnode to the tonode and release it at the tonode. This draws an arc between the two nodes named fromnode and tonode. The arc class has been designed in such a way that an arc can only be drawn between two existing nodes. An error message is flashed and no arc is drawn when an attempt is made to draw an arc which does not connect two nodes. The color of the arc, by default, is set to blue. This can be changed to a different color by selecting the desired color from the arccolors pulldown menu. A child window control is established along with each of the newly created arcs. Clicking in this child window control pops up an "attribute editor" which can be used to input new attributes or edit the previously given attributes. The attribute editor contains three edit window controls, one for each of the three attributes.

A database class keeps track of details such as which node is connected to which other nodes and what their attribute values are. Once an arc is drawn between two existing nodes and its associated attributes are filled in, an instance of the database class is created and the appropriate instance variables are initialized.

It is possible to delete an arc after it is drawn by choosing the deleteArc option under the Delete menu. A dialog box is run in response to this choice, requesting the names of the two nodes connected by the arc. After the two names are entered and the OK button is chosen, messages are sent to the arc which results in a value being returned indicating whether an arc exists connecting the two nodes. If so, the arc is deleted and the screen is updated to reflect this change. A message is also sent to the data base class indicating the deletion of an arc from the network, requesting the database be updated. Attempt to delete an arc which is non-existent results in an error.

Similarly, an existing supply or a demand node can also be deleted by choosing the deleteSupply or deleteDemand node options, respectively, from the Delete menu. A dialog box is run in response to either of these choices requesting the name of the node to be deleted. After the name is entered and the OK button is chosen messages are sent to the supply, demand node classes which results in a value being returned indicating the existence of the node. The node is deleted from the screen, if it exists. The database class is notified through a message of this change. An attempt to delete a non-existent node results in an error.

The system has been designed in such a way that any changes made to the graphical representation of the network on the screen is automatically reflected in the internal database. This is achieved by passing messages to the database class, in response to messages received from Windows indicating the occurrence of some changes on the screen.

Thus, at any given instant the contents of the database reflect the state of the graphical network displayed on the screen.

A sample network created using this procedure is as shown in Figure 3.

The network created can be saved in a file for later retrieval. The Save command under the File menu has to chosen for this purpose. This command saves two different files :

NET File	-	This file has information about the graphical network created on screen in a format suitable for the solver program.
PIC File	-	This file has data which is essential for the reconstruction of the graphical network image on screen.

The file formats and their contents are given in APPENDIX A.

After the network is solved, the user can choose the Open option from the FILE menu to display the resulting costs and flows alongside each arc. By default, the costs are displayed in red and the flows in green. These can be altered by changing the appropriate radio-button settings from the corresponding cost/flow group boxes in the dialogbox, which is displayed in response to the selection of the Results menu option. Figure 4 shows the network on the screen after its solution.

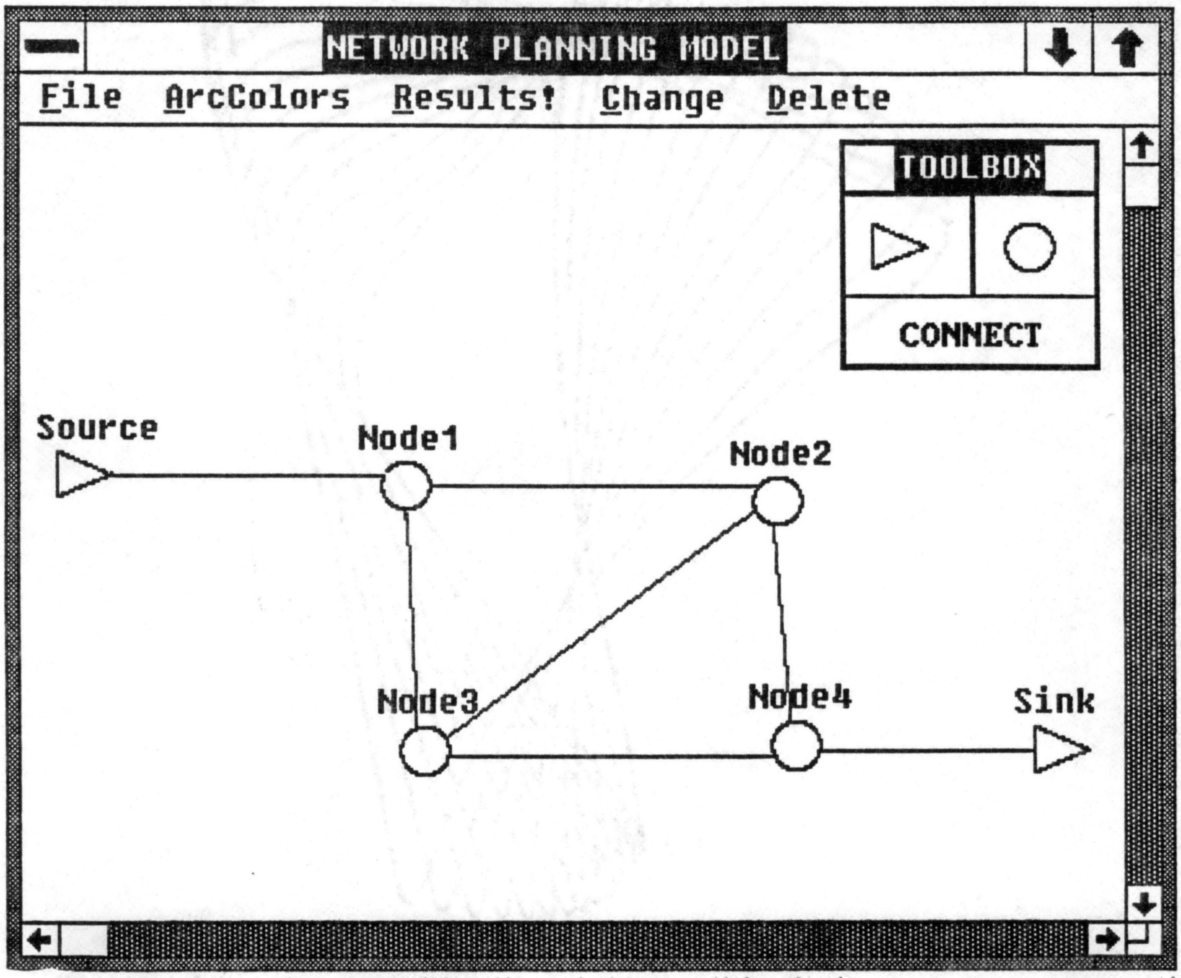

Figure 3. Window after Network Creation

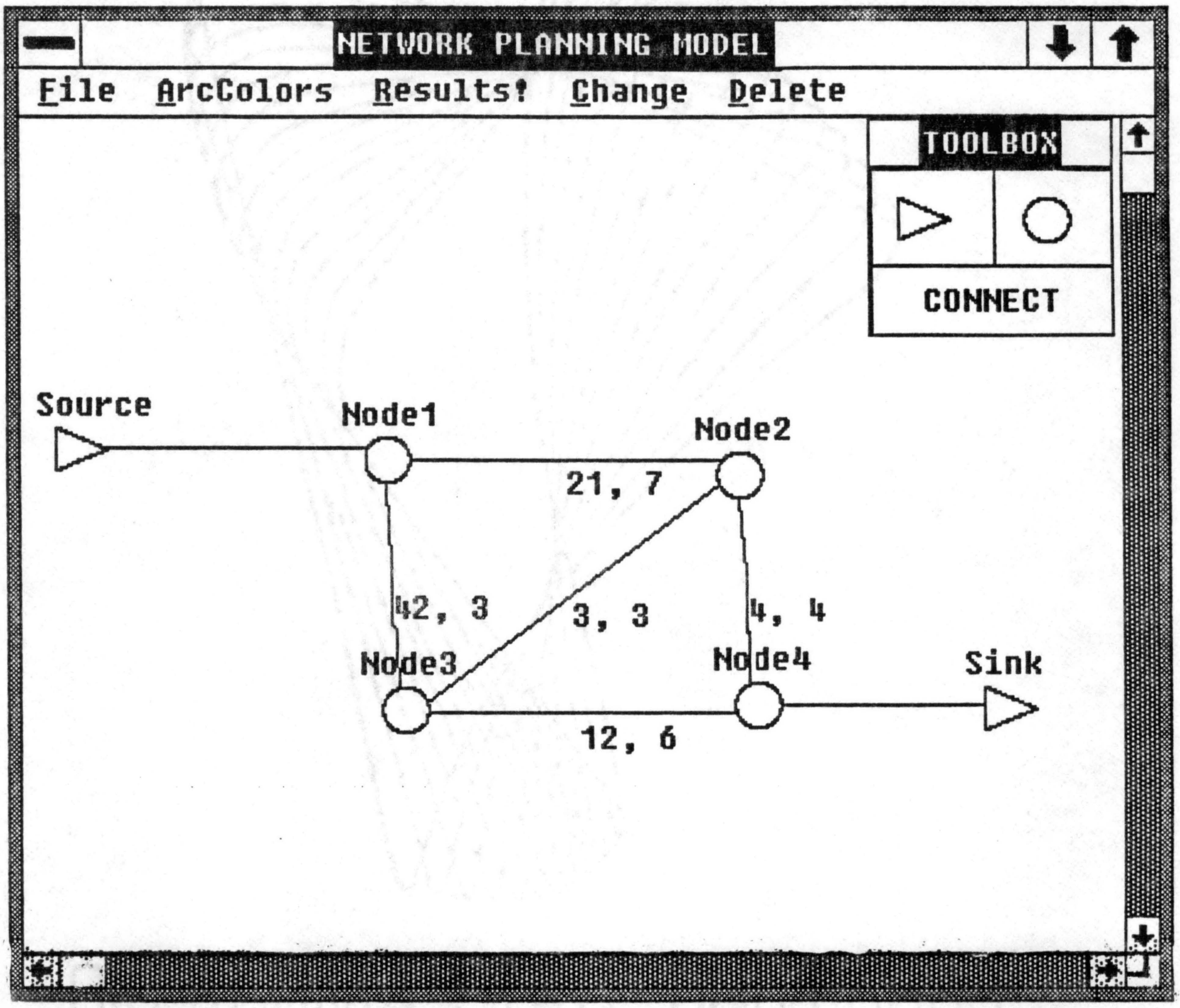

Figure 4. Window after Network Solution

Support Tools

Microsoft Windows

Microsoft Windows is an operating environment that runs under MS-DOS version 2.0 or later, generally on IBM personal computers or compatible machines.

Windows provides a multitasking graphical windowing environment that runs programs especially written for Windows and some current programs written for MS-DOS. Programs especially written for Windows have a consistent appearance and command structure and are thus often easier to learn and use than conventional MS-DOS programs. Users can easily switch between different programs running under Windows and exchange data between them. Windows programs run identically on a variety of hardware configurations.

For the future, Windows is also an integral part of the new protected mode operating system developed by Microsoft and IBM, Operating System/2. Under this system, Windows is called the Presentation Manager. The OS/2 Presentation Manager is seen as the principal environment for new OS/2 application programs (Petzold, 1988).

Windows, like object-oriented languages, operates on a message-passing paradigm. Objects receive messages that cause them to change(that is, perform operations on) themselves. Windows has many objects, but one type of object ranks first in importance - the object called the "window".

From the user's perspective, a window is the rectangular area that a particular program occupies on the screen. Although a single program can create many windows, usually the program has a single main, or top-level window. To the user this is the program's work space on the screen.

From the programmer's perspective, a window is an object that receives and processes messages. Messages inform the window of all events that affect the window.

When a program creates one or more windows, each window has certain characteristics - a style, a size, a position on the screen, a character string in its caption bar, a menu. But a window is defined by more than merely its appearance. A window is defined also by how it responds to a particular message.

Windows is an event-driven system, meaning that programs respond to events that the user or other programs initiate. These events correspond to actions like pressing a key, clicking the mouse, or selecting a menu item. Whenever an event occurs, Windows sends a message to notify the program.

More specifically, when the user presses a key, for example, Windows sends a WM_KEYDOWN message with the virtual key code of the key that was pressed. The "WM" is the mnemonic for "Windows Message". Other windows messages are WM_COMMAND (indicating the user selected a menu command), WM_LBUTTONDOWN (the user clicked the left mouse button), WM_VSCROLL (the user clicked in the vertical scroll bar), and WM_PAINT (Windows wants the window to redraw itself).

Windows messages are always sent with two parameters to convey additional information. These are known as the word parameter, or wParam, and the long parameter, or lParam. The wParam contains a 16-bit word value; the lParam sometimes contains a 32-bit long pointer to other data.

If an application has multiple windows, the windows messages are sent to the appropriate window. For example if the F1 key is pressed when one window has the input focus, a message is sent to that window; the main window is not informed. Making windows responsible for their own events simplifies application development.

Actor

Several object-oriented programming languages exist for personal computers (e. g., Smalltalk, C++, Objective-C and Actor among others). We selected the Actor object-oriented programming language to develop our graphical user interface for the modeling system.

Actor is an object-oriented programming language which allows the creation of standalone Microsoft Windows applications. In fact, Actor is a complete programming environment. One can type the Actor language statements in the workspace window and get immediate feedback. Windows, menus, and dialogs can be created and modified directly in the development environment. Code is written in a browser, a special editor that lets a user create new classes and compiles methods as they are written. The compiler translates Actor statements into a low-level format used at run time. The browser also shows the hierarchy of the classes in the system. Everything in Actor is an object and all operations are performed by sending messages to objects.

The relationship between Actor objects and Windows entities is similar to the relationship between file variables and files in most high-level languages. For example, in Pascal, one can declare a variable of type File. To use the variable, however, one must assign it the name of an actual disk file. The file variable is an abstraction of the physical file on the disk. In the same way, Actor objects belonging to classes like Window and Dialog are abstractions of underlying areas of memory managed by Windows.

Although both Actor and Windows send messages, the messages are processed separately. Unlike Windows messages, Actor messages are not queued at all and are therefore very efficient. The main function in a Windows application called, WinMain, normally includes a very short loop that translates and dispatches Windows messages. The application must also define a WndProc function that processes the messages.

From an object-oriented perspective, it is the window itself that responds to the messages. After all, a window is not just a data structure, it is both the data and the functionality. Thus Actor manages the WinMain and WndProc functions, the Windows message queue, and other low-level details.

Actor translates Windows messages into equivalent Actor messages, enabling one to process all the messages in the same way. The Actor classes Window and WindowsObject define many high-level messages that hide the generic details of Windows programming and allow the programmer to concentrate on application specific behavior.

The WM_PAINT method defined in the Actor class Window, automatically locks down an area of memory known as a display context used for redrawing. It then calls the Windows function BeginPaint, sends an Actor paint message, calls the EndPaint function, and lastly frees the memory used for drawing.

The WM_PAINT methods defined for class Window is shown in Figure 5. Since this method is inherited by all descendants of class Window, they only need to define a higher-level paint method that knows how to redraw the contents of the window. The Actor paint message will be sent whenever Windows sends a WM_PAINT message.

In addition to Windows sending messages, like the WM_PAINT message described above, window objects can send messages to other windows or even to themselves.

```
/*
Trap MS-Windows message to paint self, a Window.
Sends a paint(self) message with the dispaly
context.

Self is a Window or ancestor of class Window. The
arguments, wParam and lParam are ignored.

hDC, hPS, lpPS are local variables.
     hDC  :    handle to the display context
     hPS  :    handle to the paint structure
     lpPS :    long pointer to locked down paint
               struct
*/

Def WM_PAINT(self, wParam, lParam | hDC, hPS, lpPS)
{
hPS  :=   asHandle(paintStruct);
lpPS :=   globalLock(hPS);
hDC  :=   Call BeginPaint(hWnd,lpPS);
paint(self,hDC);
Call EndPaint(hWnd,lpPS);
globalUnlock(hPS);
^0;
}!!
```

Figure 5. WM_PAINT Method for Window Class

Current Status of the System

At present, the external solver cannot be invoked through the selection of a menu command due to DOSs 640K limit on applications and the size of the solver (370K).

Hence, the user has to temporarily leave GIN and invoke the solver at DOS command prompt by typing "NETFLO", and inputting the appropriate file names as requested. This process produces the .RES file. Now, GIN has to be invoked and the OPEN command under the FILE menu has to be chosen for the results to be displayed graphically on the screen.

CHAPTER IV

SUMMARY AND CONCLUSIONS

We proposed a set of class objects arranged in an inheritance hierarchy and having corresponding message protocols. These protocols provide the object-oriented GIN user the power and flexibility to interact with the graphical representation of the transshipment network model on the screen.

Since GIN represents the problem being modeled as a graphic display, the manager/decision maker can directly use its mouse-oriented graphical interface to formulate, modify and solve the problem. In addition, the manager can understand the visual model more easily than the mathematical model and can thus effectively contribute to its development and monitor its validity.

Finally, we implemented GIN, in a personal computing environment, using our proposed class objects and message protocols. This implementation attests to the feasibility of an O-O graphical user interface for such a modeling environment and its ease of use for the formulation, modification and solution of network problems.

CHAPTER V

RECOMMENDATIONS FOR FURTHER STUDY

The present study shows that object-oriented GIN is a viable possibility. Regardless, there are several directions which future research endeavors might pursue.

First, must extend GIN via both interface and algorithms to embrace additional network models, especially generalized networks, networks with single side constraints and LPs with embedded networks.

Second, we must investigate hierarchical decomposition and its effect on the comprehension of the model by the decision maker; i.e., when and how to aggregate groups of arcs and nodes into a representation with reduced detail and yet retain as much comprehension as possible.

Third, we must investigate ways to incorporate sensitivity analysis factors visually into the NETFORM models. Such factors could help the decision maker determine which what-if case should be specified next and which model parameter should be investigated further.

Fourth, we must investigate ways, (perhaps using Artificial Intelligence (AI)/Expert System(ES)) to create, automatically, a visual graphic network directly from the n-

BIBLIOGRAPHY

Arnheim, R. (1972), "Visual Thinking," University of California Press, Berkeley, CA.

Bhaskar, K. S. (October 1983), "How Object-Oriented is your System ?" SIGPLAN Notices (ACM), Vol. 18, pp. 8-11.

Billington, J.N. (1987), "Visual Interactive Modelling and Manpower Planning," European Journal of Operational Research, Vol. 30, pp. 77-84.

Bisschop, J. and Meerus, A. (1982), "On the Development of a General Algebraic Modeling System in a Strategic Planning Environment," Mathematical Programming Study 20, North-Holland Publishing Company, pp. 1-29.

Bobrow, D.G., Mittal, S., and Stefik, M. J. (1986), "Expert Systems: Perils and Promise," Communications of ACM, Vol. 29, pp. 880-894.

Booch, G. (February 1986), "Object-Oriented Development," IEEE Transactions on Software Engineering, Vol. SE-12, No. 2, pp. 211-221.

Card, S.K., Moran, T. P., and A. Newell (1980), "Computer Text Editing: An Information-Processing Analysis of a Routine Cognitive Skill," Cognitive Psychology, Vol. 12, pp. 32-74.

Carroll, J.M., Thomas, J. C., and Malhotra, A. (1980), "Presentation and Representation in Design Problem Solving," British Journal of Psychology, Vol. 71, pp. 143-153.

Cox, B. J. (1986), "Object-oriented Programming: An Evolutionary Approach," Addison-Wesley Publishing Company, Reading, MA.

Dodani, M. H., Hughes, C. E., and Moshell, J. M. (March 1989), "Separation of Powers," BYTE, pp. 255-262.

Duncan, R. (1986), "Advanced MS-DOS Programming," Microsoft Press, Redmond, WA.

Foley, J.D., and Van Dam, A. (1984), "Fundamentals of Interactive Computer Graphics," Addison-Wesley Publishing Company, Reading, MA.

Fourer, R. (1983), "Modeling Languages versus Matrix Generators," ACM Transactions on Mathematical Software, Vol. 9, pp. 143-183.

Geoffrin, A. M. (1976), "The purpose of Mathematical Programming is Insight, Not Numbers," Interfaces, Vol. 7, No. 1, pp. 81-92.

Glover, F., Klingman, D., and McMillan, C. (1977), "The NETFORM Concept," Proceedings of ACM 1977 Annual Conference, pp. 283-289.

Good, M. D. (1984), "Building a User-Derived Interface," Communications of the ACM, Vol. 27, pp. 1032-1043.

Hix, D. (1989), "User Interfaces: Opening a New Window on the Computer," IEEE Software, Vol. 6, pp. 8-10.

Hurrion, R. D. (1986), "Visual Interactive Modelling," European Journal of Operational Research, Vol. 23, pp. 281-287.

Jones, C. (1988), "User Interfaces," Unpublished Manuscript, The Wharton School, University of Pennsylvania.

Kennington, J. L. and Helgason, R. V. (1980), "Algoritms for Network Programming," John-Wiley and Sons, Inc., New York.

Linton, M. A., Vlissides, J. M., and Calder, P. R. (1989), "Composing User-Interfaces with InterViews," Computer, Vol. 22, pp. 8-22.

McKim, R. H. (1972), "Experiences in visual Thinking," Brooks/Cole Publishing Company, Monterey, CA.

Microsoft Windows User Manual, Microsoft Corporation.

Montessori, M. (1964), "The Montessori Method," Schoken, New York.

Myers, A.B. (1989), "User Interface Tools : An Introduction and Summary," IEEE Software, Vol. 6, pp. 15-23.

Myers, B. and Doner, C. (1988), "Graphics Programming Under Windows," SYBEX, Alameda, CA.

Newman, W. M., and Sproull, R. F. (1979), "Principles of Interactive Computer Graphics," McGraw-Hill, New York.

Ornstein, R. E. (1973), "The Nature of Human Concsiousness," Freeman Press.

Paul, J. P. (1989), "LINGO/PC: Modeling Language for Linear and Integer Programming," OR/MS Today, Vol. 16, No. 2.

Petzold, C. (1988), "Programming Windows," Microsoft Press, Redmond, WA.

Polya, G. (1957), "How to Solve It," Doubleday, New York.

Schneiderman, B. (1983), "Direct Manipulation: A Step Beyond Programming Languages," Computer, Vol. 16, pp. 57-69.

Schneiderman, B. (1987), "Designing the User Interface Strategies for Effective Human-Computer Interaction," Addison-Wesley Publishing Company.

Stefik, M. and Bobrow, D. G. (1986), "Object Oriented Programming: Themes and Variations," The AI Magazine, Vol. 6, No. 4, pp. 40-62.

Swartout, W. and Balzer, R. (1982), "The Inevitable Interwining of Specification and Implementation," Communications of ACM, Vol. 25, pp. 438-440.

The Whitewater Group, ACTOR, 600 Davis Street, Evanston, IL 60201.

Urlocker, Z. (1989), "Whitewater's Actor : An Introduction to Object-Oriented Concepts," Microsoft Systems Journal, pp. 33-44.

Werthheimer, M. (1959), "Productive Thinking," Harper and Row, New York.

APPENDIXES

APPENDIX A

NET FILE FORMAT

NET FILE FORMAT

```
/* NUMBER OF NODES */
I10


/* NUMBER OF NODE        REQUIREMENT */
         I10             I10


/* ONE BLANK LINE */


/* NUMBER OF ARCS TO NODE 1, NUMBER OF ARCS TO NODE 2, .....
*/
I10  I10  I10  I10  I10  ........


/* ONE BLANK LINE */


/* NAME    FROM    TO    UNIT_COST    UPPER_BOUND    LOWER_BOUND
*/
   A10     I10    I10      I10           I10              I10


/* ONE BLANK LINE */
```

APPENDIX B

PIC FILE FORMAT

PIC FILE FORMAT

```
/* NUMBER OF SUPPLY NODES */
NS

/* CARTESIAN COORDINATES OF EACH OF THE NS SUPPLY NODES */
(X1, Y1)
(X2, Y2)
     .
     .
     .
(XNS, YNS)

/* NUMBER OF DEMAND NODES */
ND

/* CARTESIAN COORDINATES OF EACH OF THE ND DEMAND NODES */
(X1, Y1)
(X2, Y2)
     .
     .
     .
(XND, YND)

/* NUMBER OF ARCS */
NA

/* CARTESIAN COORDINATES OF EACH FROM POINT OF THE ARC */
(X1, Y1)
(X2, Y2)
     .
     .
     .
(XNA, YNA)

/* NUMBER OF ARCS */
NA

/* CARTESIAN COORDINATES OF EACH TO POINT OF THE ARC */
(X1, Y1)
(X2, Y2)
     .
     .
(XNA, YNA)
```

VITA 2

Chakradhar R. Nanga

Candidate for the Degree of

Master of Science

Thesis: DESIGN AND IMPLEMENTATION OF A GRAPH-BASED INTERFACE FOR NETWORK MODELING (GIN) USING AN OBJECT-ORIENTED APPROACH

Major Field: Computing and Information Sciences

Biographical:

Personal Data: Born in Pakala, Andhra Pradesh, India, February 5, 1965, the son of Dr. N.C.R.Reddy and Eswari.

Education: Graduated from Sri Sathya Sai Institute of Higher Learning, Puttaparthi, India, 1982; received Bachelor of Engineering degree in Computer Science from Bangalore Institute of Technology, Bangalore, India in May 1987; completed requirements for the Master of Science degree at Oklahoma State University in May, 1990.

Professional Experience: Research Assistant, Department of Economics, Oklahoma State University, August, 1988, to February, 1990; Software Engineer, Kirloskar Computer Services Limited, June, 1986, to December, 1987.