To the Graduate Council:

I am submitting herewith a thesis written by Mei Zhang entitled "A Program Generator of Object Oriented Spatial Models." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Geography.

Bruce A. Ralston, Major Professor

We have read this thesis
and recommend its acceptance:

John B. Rehder

Thomas L Bell

Accepted for the Council:

Associate Vice Chancellor
and Dean of The Graduate School

A PROGRAM GENERATOR OF OBJECT ORIENTED SPATIAL MODELS

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Mei Zhang

August, 1993

ACKNOWLEDGEMENTS

To write an Object Oriented program generator was suggested by Dr. Bruce Ralston as my thesis project. He also started the generator by designing the program structure and defining major class type and function prototypes. He has been very helpful and patient while I was working on this project and later on writing the thesis.

I am grateful to Dr. Ralston for his guidance for past five years, started long before he became my official advisor two years ago. I want to thank him for being able to make the work fascinating, for having faith in me to be able to do whatever I want to do, and for his friendship. I would like to thank my committee members, Dr. Thomas L. Bell and Dr. John B. Rehder for their help during my study in this department.

I am indebted to my parents, Yue-e Lu and Jing-ye Zhang. I want to thank them for their work to my brain and heart all these years. I need to thank my brother Ge for always being there for me and trying to interpret everything from a physicist's point of view; and my sister Yue for making things more chaotic but pleasant than they were. Finally, I'd like to thank my friends for their care, understanding and just being around me.

ABSTRACT

This thesis involves the development of an object oriented programming (OOP) program generator for building models which integrate with GIS packages, such as ArcInfo. The OOP approach allows the user to concentrate more on the relationships between spatial entities than on program coding. The generator uses an OOP computer language, C++, to construct programs of spatial processes. These programs incorporate the OOP components of classes, procedures and function prototypes that are necessary for spatial models that require the formulation of some type of matrix. Such programs have the characteristics of being easy to change and can handle complex formulations. Three linear programming examples are used to illustrate the OOP structure of matrix based problems. The necessary classes and relationships for these problems are presented. The generator is then used to create the OOP implementation code for the LPs.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

SPATIAL MODELS AND OBJECT ORIENTED PROGRAMMING

I. Introduction:

Geographic Information Systems (GIS) are designed to work with data related to spatial or geographic features. A GIS is composed of a database system and a set of operations that is applied to those data. The National Center for Geographic Information and Analysis (NCGIA) defines a GIS as "a computerized data base management system for capture, storage, retrieval, analysis, and display of spatial (locationally defined) data."

Most GIS contain many tools of spatial analysis, such as buffering, overlay and various kind of univariate statistical methods. However, there are other forms of spatial analysis which need to be used within a GIS environment. Among these are problems that involve some kind of mathematical modeling. At present, little is available for combining more complicated mathematical models with a GIS. As Goodchild (1991) states, a GIS is "a database containing a discrete representations of geographic reality in the form of static, two-dimensional geometric objects and associated attributes, with a functionality largely limited to primitive geometric operations to create new objects or to compute relationships between objects, and to simple query and summary

descriptions."

Decision makers often encounter problems in spatial analysis which involve mathematical programming such as a warehouse routing, linear programming, and shortest path problems. However, in most commercial GIS packages, there is not enough modeling capability to adequately handle such problems. Even systems that contain modeling algorithms, such as TRANSCAD, only solve a set of problems that are specifically defined. In other words, these systems have limited flexibility.

In this thesis, I present an approach that integrates a subset of mathematical models into GIS packages by purposing an Object Oriented Programming (OOP) model that is generated by an OOP generator. That is, I develop a program (the generator) that writes Objected Oriented Programs for a class of spatial models.

II. GIS, Models, and Spatial Decision Support Systems (SDSS)

One of the main reasons that GIS lack mathematical modeling ability is that a slight change in the definition of a problem often involves serious modification in the model, and it is too complicated to construct a general formulation that could take care of all the possible cases. The relational database management model that most GIS packages use causes additional difficulty in integrating mathematical

models. The relational database is not compatible with all of the data structures that different models require. A mathematically modeled problem is often solved using special data structures.

Consider a warehouse routing problem where decision makers need to find the minimum number of truck trips necessary to ship goods from warehouses to outlying nodes and the routes of those trucks. Warehouses supply goods and the outlying nodes demand goods. For each truck assigned to a warehouse, a truck tour is built. A tour consists of a list of demand nodes a truck should visit, the order in which those nodes should be visited, and the amount that should be delivered at each node. A problem like this needs data for each warehouse, each outlying node, truck capacity and a specified network. There are special algorithms and data structures available to solve such a warehouse problem.

For example, in TRAILMAN (Ralston et al. 1992), a warehouse routing problem is solved using a four step algorithm. First, each demand node is assigned to the "nearest" warehouse. Then, for each warehouse, the distances between the nodes which are assigned to it are calculated. The third step combines the demand nodes subject to the constraints associated with the network and a truck. Finally those tours are assigned to the links on the network. Each algorithm requires a specific data structure representing these data. For example, finding paths between nodes requires

a forward star data structure while combining tours is accomplished with a heap data structure.

This problem could be varied on the objective and/or constraints. For example, the decision maker may want to minimize the total cost, total time used, total distance traveled, or maximize the amount shipped, instead of minimizing the number of trucks. He might vary the problem by changing the constraints on trucks, warehouse locations, demands, and network structure. To accommodate these variations, the above algorithm and data management may need to be restructured.

In some cases, the problems are well defined and specialized GIS are developed. These systems are called spatial decision support systems (SDSS) because they support decision making. Stand alone GIS packages BTMS (Ralston and Liu, 1990) and TRAILMAN (Ralston et al., 1992) or commercial packages like TRANSCAD fall into this category. SDSS are designed to solve specific problems with mathematical models and often try to make use of GIS data management and display facilities. In general, these kind of systems "provide faster response times, are frugal in code, and can be produced inexpensively compared to the large scale GIS (Church, 1990)." However, these special purpose application packages are stand alone systems that have limited potential for growth and flexibility beyond a specific application or set of models.

There is a practical need to develop SDSS or specialized

GIS systems that are flexible, and able to address complex, ill-defined problems. Such a system should be able to be:

- applied to a variety of problems;
- used with a variety of operating systems; and
- adapted to new problems as they arise.

The OOP generator developed here offers a way to address some of these issues, particularly the third point.

III. Matrix Formulation Problems

One way for SDSS or specialized GIS to be flexible is to make use of similarity of different problems' model structure. Many models require formulating a matrix of some type. Linear programming (LP) problems and contiguity checking problems are two examples.

LP problems are one kind of mathematical problem that are encountered frequently by GIS users. LP includes minimization and maximization problems such as minimum cost and maximum flow problems.

What follows is a typical minimum cost problem. Suppose there is a region with interior nodes and port nodes (Figure 1.1). External goods coming into this region are stored at the ports. The objective is to transport goods into interior demand nodes in a minimum time and/or at a minimum cost, subject to constraints associated with transport and storage facilities. This is illustrative of a project currently being

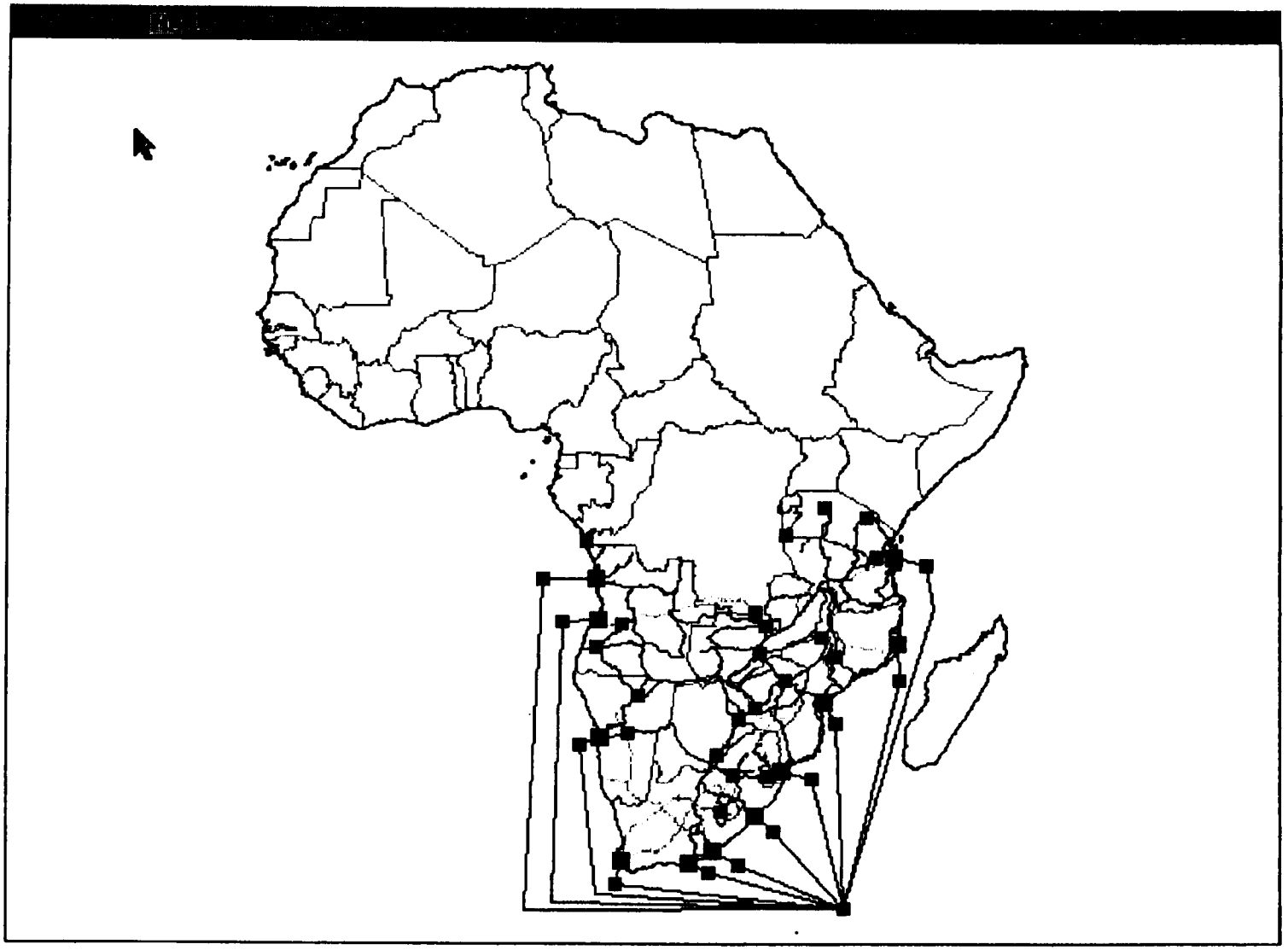


Figure 1.1 Application of TRAILMAN in Southern Africa

undertaken in southern Africa(TRAILMAN). The goods coming into the ports are food aid and the demand nodes represent distribution centers, towns, and feeding camps.

The standard LP formulation involves finding the optimal value, the maximum or the minimum, of a linear function of variables, subject to a set of linear constraints associated with these variables. The mathematical formulation is

$$\text{MIN} \quad \Sigma cx$$

Subject to

$$Ax <= b$$

where c, b are (n X 1) vectors, x is the (n X 1) decision variable vector and A is the (m X n) coefficient matrix represents relationships among decision variables. In geographical problems, every column corresponds to a variable that is associated with a spatial entity and every row corresponds to a constraint that is associated with a spatial entity(Figure 1.2).

	A0	A1	A2	A3	A4	.	.	.		
Obj	c0	c1	c2	c3	c4	.	.	.		
N0	a00	a01	.	.	.	.	.	.	>=	d0
N1	a10	.	.	.	.	.	.	.	>=	d1
N2	.								>=	d2
N3	.								>=	d3
.	.								>=	.
.	.								>=	.
.	.								>=	.

Figure 1.2

Sample LP Matrix

Checking the contiguity of a zone also has a matrix formulation. Suppose a region is divided into a set of zones. A typical problem is to find out the zones that are adjacent to a particular zone. In a contiguity matrix, the columns and the rows represent the zones (Figure 1.3). If two zones have a common border then a 1 is entered into their row column intersection, otherwise a 0 is entered.

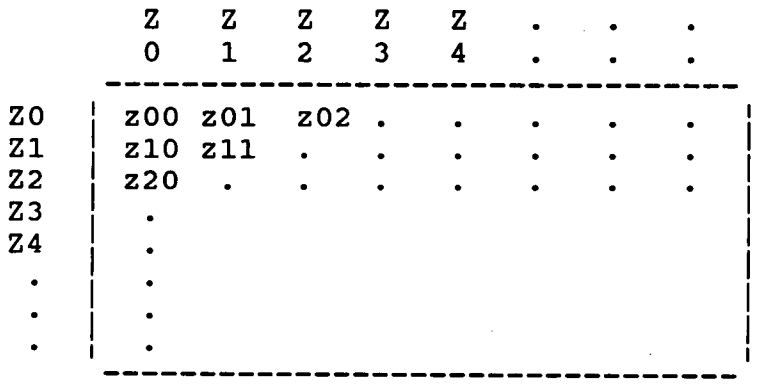


Figure 1.3

Sample Contiguity Matrix

The question is how to efficiently formulate, solve and report results of problems like those listed above in a GIS environment. Furthermore, if the problem is changed slightly, the system must adjust. For example, if new constraints need to be added to roads, or certain types of nodes need to be excluded from some path of an LP problem, the system must be able to incorporate these modifications.

A GIS system or a SDSS needs tools that can facilitate

the formulation and use of various spatial models. These tools need to be "sufficiently general to be of use in a wide variety of problems, and flexible enough to be tailored to specific problems (Ralston, forthcoming)." With these tools users are allowed to concentrate on the characteristics of and relationships between spatial objects, instead of on how the objects are processed by some computer code. Further, these tools should make the association of model results with the GIS database easier and faster. The OOP approach best suits these requirements.

IX. Object Oriented Programming:

OOP languages are more modular, flexible, portable and reusable than non-OOP languages. These features are interrelated with each other, and they are represented in the forms of data structures and logic design of OOP languages. OOP allows programmers to create new data types, and these user-defined data types tend to partition a program into stable pieces. For example, a real world object such as a road is treated as an OBJECT in OOP. An OBJECT encapsulates an allowable set of values for data and an allowable set of operations that can be performed on the data type. An OBJECT also holds inheritance from related data types. INHERITANCE was developed to express relationship between types. A base type has all the characteristics and behaviors that are shared

among the types derived from it, and it represents the core of ideas about the objects in the system. It is a generic and abstract definition and derived types tend to be more specific and real valued. The spatial entities of a GIS, the lines, points and polygons can be well encapsulated as "objects".

The process of defining data types encapsulates the information about objects and the relationships among them. "Once you've modeled your problems as a set of types, writing the code becomes remarkably simple; it's mostly a matter of creating variables and sending messages to them. The variables take care of the details and ensure their own integrity (Entsminger, p.5)."

There are GIS packages, such as SYSTEM 9 (Berrill and Moon, 1992), that claim to be object-oriented in their software structure, or the data management structure is different from the systems that came before. However, many of them only got to the stage that moved away " from the CAD concepts of tiles (sheets), layers and graphical features to center one's attentions on real world objects, their attributes, relationships and behaviors (Newell, 1992)."

Newell categorized three levels of object oriented implementation in a GIS.

- object centered data models: Object oriented data management that makes use of generic class structures.
- encapsulation of behavior: Programs written are

independent of the detailed internal implementation of an object. Functions and operators can be defined and used on different datatypes. External system databases can be encapsulated within an agent object so that programs written within the GIS are protected from unexpected modifications.

- inheritance between classes: Code reuse is maximized. Changes made to a parent class are automatically transferred to its inherited classes. It is easier to maintain consistency in both the internals of the system and in the user interface.

My models fall in the second and third levels. They not only employ object-oriented data management, but also are designed to make good use of the encapsulation of behavior in objects. They rely heavily on inheritance between classes and polymorphism in the interfacing process of data types. Encapsulation, inheritance, polymorphism and some other OOP features are discussed more fully in the next chapter.

CHAPTER 2

CHARACTERISTICS OF OOP LANGUAGE C++

C++ (Stevens, 1990, Stroustrup, 1987) is an object oriented programming language used in this thesis. C++ is called a hybrid OOP language because it was created from the existing language C keeping all of that language's capabilities. "The new language adds extensions to C that improve its syntax, expand its application, and bring to it some of the features of object-oriented programming (Stevens, p.XV)." In this chapter, some of the C++ features that are employed in the OOP models and the generator are presented.

I. Class and Encapsulation

C++ allows programmers to create new data types called CLASSES. CLASSES are used to encapsulate data and operations. Data are the characteristics of an object and operations, such as functions and procedures, are its behaviors. Encapsulation refers to the action of combining data with the operations to manipulate that data into a single package.

Before the name C++, the language was called **C with classes**. Classes are the way to build new data types and they lead the language toward object-oriented programming. A CLASS declaration includes data members, constructor functions, destructor functions and member functions. For instance, the

following class **shape** contains data on center locations, number of edges, perimeter and is also able to draw itself.

```
class shape{
    float center_x;
    float center_y;
    int   sides;
    float perimeter;
    shape(float x, float y, int s, float p);
    ~shape(){};
    virtual drawing();
}
shape first, second;
```

In this class, center_x, center_y, sides, perimeter are the data members. shape() is the constructor function, which always takes the same name as the class name. It allocates memory for every instance of the class. ~shape() is the destructor function. When a class object goes out of scope, the destructor function is called. It could be defined to release the memory that is occupied by an instance of this class. Virtual function drawing() is designed to draw the shape on a screen according to the data members' values. The data members store the attributes of a shape and the constructor initializes the value of an instance of the class.

A class definition only defines a data type, no instance of class exists until one is declared. The class **shape** has instances **first** and **second** here. An instance of a class is

also called an "object." In the example above, **first** and **second** are objects of type **shape.**

II. Inheritance

Suppose now it is required to define data types for a rectangle and a circle which are shapes but, which have different attributes and thus need different drawing functions. In addition to all the data members of a **shape**, a rectangle needs width and length as new data members in its description. OOP allows the programmer to reuse the code in shape by letting **rectangle** inherit data and functions from **shape.** That is, **rectangle** is a child class of **shape.** For class **circle**, there is no new data member, but it needs a special drawing function, as does **rectangle.**

```
class rectangle : public shape{
    float width;
    float length;
    rectangle(float w, float q, float x, float y);
    ~rectangle(){};
    virtual drawing();
}

class circle : public shape{
    circle(float x, float y, int s, float p);
```

```
    ~circle(){};
    virtual drawing();
}
```

Similarly, **square** and **hexagon** can be defined as child classes of **shape**. These child classes are called derived classes and the parent class, **shape**, is called the base class. All the child classes need specific drawing functions since to draw a circle on a screen is different from drawing a square. This is the reason that they are defined virtual. By defining drawing as virtual, the drawing function in a child class overrides that of its parent class. This is a method for a child class to modify its parent's member functions. When a virtual function is called, the version which corresponds to the caller's class level is used. If the function is not defined at this level, the function of its parent is used. In an OOP program, an object should not be treated as the specific type that it is but as a member of its base type whenever it is possible. In other words, one always tries to work on generic classes. This is possible because of polymorphism.

III. Polymorphism

A circle, a rectangle, a square, and a hexagon can be drawn because they are shapes, although the drawing function

each specific type is different. When a function can have many forms, a program is said to use polymorphism.

Polymorphism is important, first, because a programmer doesn't have to re-implement everything each time he creates a new type of shape. The programmer just builds on the existing objects. An object-oriented program that draws shapes doesn't have to be rewritten just because it needs to handle a new type of shape. The program always knows that any shape can respond to the drawing message and act accordingly. Second, using polymorphism to create an extensible program is crucial since a programmer can't know everything about a program or the problem he needs to solve while developing the program. Programs need to change in response to new information.

"The needs to change and evolve have been a nemesis of software development because change is expensive and the inability to change often leads to obsolescence (Entsminger, p.33)." Change has to be considered an integral part in program design. New information might come from understanding a system differently or as the problem changes in the real world. Polymorphism reflects this need for change by letting the user create extensible programs which are capable of being extended.

Polymorphism is made possible by the option of LATE or DYNAMIC BINDING that OOP possesses. In early or static binding the compiler resolves all references to functions by

the time the program is loaded. However, with polymorphism, when a message is sent to an object, it is required that the object figures out which method to use. Thus, the compiler needs to resolve some references at run time. This is called LATE or DYNAMIC binding. This option is important "because it lets you defer decisions and connections until run time, thus making the system more flexible and easier to extend. It also means I can give my objects general requests, and they can determine how to respond. We tell the objects what to do, and they figure out how to do it (Entsminger, p.36)." C++ has this option by defining a function as 'virtual'. There is a table of possible functions for a single call. These functions are declared as virtual in the hierarchy of data types.

IV. Data Hiding

There is another OOP feature in class definition, the protection of data and function members. For example, here is the definition of **HighWay**:

```
class HighWay{
private:
     char Name[9];
     float Cost;
     float Capacity;
```

```
        HighWay(char *n, float co, float cap);
        HighWay(char *n, float co);
        ~HighWay();
    public:
        float FlowAmount(float total);
    }
    HighWay highway#1, highway#2;
```

The class HighWay has three private data members, Name, Cost and Capacity, two constructor functions HighWay(), destructor function ~HighWay() and public member function FlowAmount().

The ability to hide some of its members from the rest of the program is a distinction of classes and C structures. The data members and the member functions can be declared as 'private', 'protected' or 'public'. A function can be a 'friend' to a class's private data members. Private class members can be accessed only by the class' member functions and specially declared friend functions of the class. Protected class members are private except to member functions of derived classes. Finally, a public class member grants access to all functions within the scope of the object of the class. These features control the degrees of visibility of data members and functions of a class in the program. This control is very useful when some data values need only to be accessed, but not accidently changed, by the programmer. For

example, we may not want any changes to be made to the physical characteristics of a highway, such as its capacity or its length.

V. Overloading

Function and operator OVERLOADING also distinguishes C++ from non-OOP languages. Function overloading is a way to make multiple versions of the same function name available to the whole program. It allows programmers to define different functions with same name. The functions are distinct only in the parameters that are passed to them. Overloading allows an object to react to other objects differently depending on what they are. In the definition of class HighWay, there are two overloaded constructor functions. One passes the parameter capacity and one doesn't:

```
HighWay::HighWay( char *n, float co, float cap)
{
     strcpy(Name, n);
     Cost = co;
     Capacity = cap;
}
HighWay::HighWay( char *n, float co)
{
     strcpy(Name, n);
```

```
        Cost = co;
        Capacity = UNLIMITED;
    }
```

These two constructor functions build different objects. A function is overloaded because it performs a generic task, but there are different permutations of what it does.

Operator overloading allows operators such as "=", "+" and the casting operator to have different meanings depending on the arguments on either side of the operator. For example, "=" can be redefined for classes and used as:

```
highway#1 = highway#2;
```

and it will copy everything in highway#2 to highway#1.

VI. Message Passing

User-defined object types contain user-defined data (characteristics) and operations (behaviors). These operations are called 'methods'. To call any of them, a general request is made to the object. This process is known as 'sending a message' to the object.

A protocol is the set of messages to which an object can respond. Generally, a programmer should try to design objects that share a common base type or are compatible with each other to have the same protocol. If the objects are designed with a consistent, generic protocol, these objects are likely to be reusable. Inheritance and polymorphism allow a base

type to establish a common interface to a group of types, which are created by inheritance from the base type. This interface determines which message an object can receive and what the protocol is.

The "system understands and uses the interface to the base without knowing the particulars of the derived types, it is easily extended (Entsminger, p.34)." If a new type is added to an existing system by deriving it from a base type, the system only uses the base interface and it immediately knows what to do with the new type.

In matrix formed problems such as LP and contiguity checking problems, a matrix is formed by columns and rows. Every row or column is an object, an instance of a class. And the coefficients in the matrix represent the relationships between the corresponding row and column. Each row interacts with each column, and if there is relationship between them, some meaningful coefficient is generated. The process of interacting is message passing from an object to another. The message is about what kind of a column is sent to every row, and the row reacts based on the message received.

"A simple analogy would be to consider a market full of buyers and sellers. A seller might receive a message that a particular buyer has just entered the market. The seller would then send that buyer a message of just what kind of product he or she is selling. If it is not something the buyer wants, he will ignore the message. Otherwise, the buyer

and seller will then engage in a negotiation which might lead to a sale (Ralston)."

With OOP, if a programmer carefully designs the new data types and creating objects, it will make modification much easier compared to non-OOP. OOP tries to develop systems in the computer to be a direct mapping of the system in the real world. This implies that if the system in the real world changes, it's easy to change in the computer. This simplicity makes software design, development, and maintenance faster and cheaper.

With classes and encapsulation, inheritance, polymorphism, data hiding, function and operator overloading, and the message passing strategy of C++, the OOP models are designed not only to solve current problems but also to leave space for further development.

CHAPTER 3

OOP MODELS FOR GEOGRAPHICAL LP PROBLEMS

The OOP models constructed by the generator developed for this thesis are specially constructed to solve geographical problems within a GIS environment. They deal with spatial entities such as polygons, lines and points. Information about these objects is obtained from attribute files of a GIS package. The generator gives the models an object oriented logical design by defining the class hierarchy, function prototypes and core code. Before introducing the generator, this chapter first describes how the OOP models are used to approach three geographical LP problems.

I. Example I: The Transportation Problem

The first example problem is built on a single mode network (N, A) (Figure 3.1), which contains seven nodes and eight arcs. Each arc is assumed to have infinite capacity. A demand is associated with each node (negative value implies a supply node) and a transport cost corresponds to each arc. The objective of the problem is to satisfy the demand of every node in the network at a minimum transportation cost. This is called a Transportation Problem which is typically solved by LP. A formal statement of the problems is

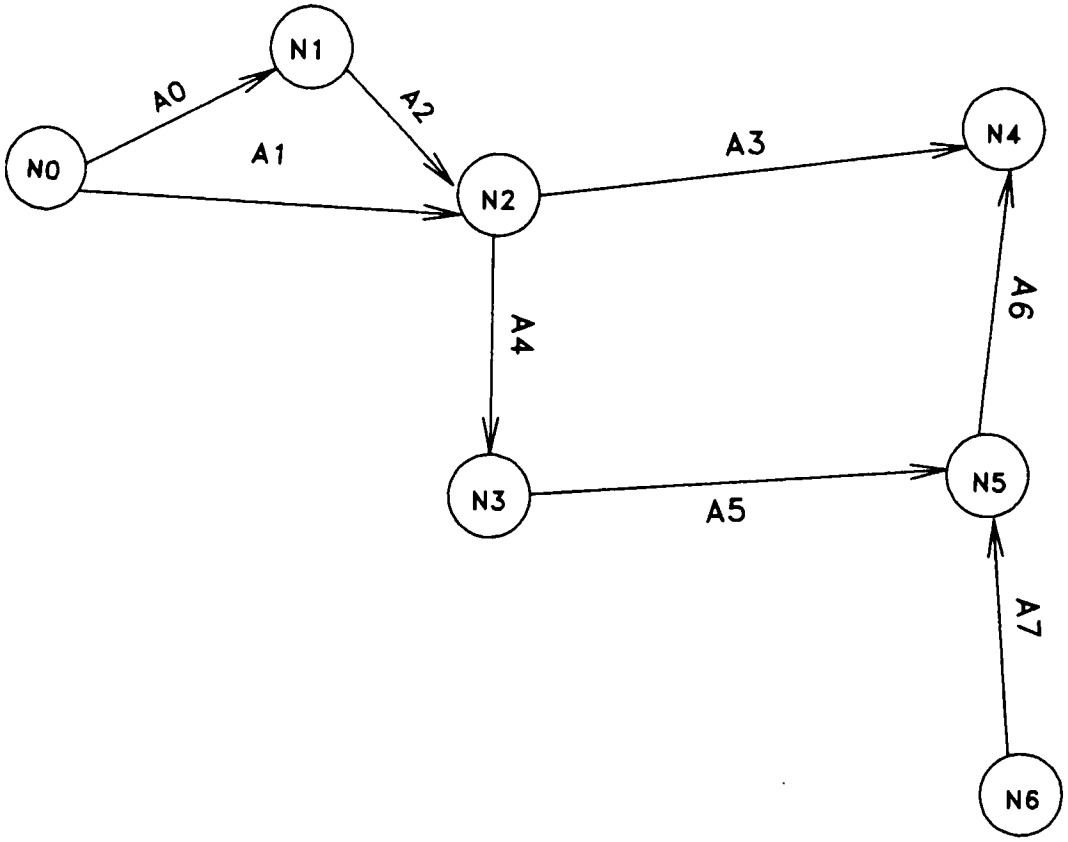


Figure 3.1

Network of Example I

$$\text{Min} \sum_{i \epsilon A} C(i)A(i)$$

$$\text{s.t.} \sum_{i \epsilon A} A(i)X(ij) \geq D(j) \quad \text{for all } j \epsilon N \qquad (1)$$

In the objective function, C is a vector of costs associated with each arc, and A(i) is the amount to be transported along arc i. The A(i) are the decision variables of the problem. Since the demand D(j) at each node j has to be satisfied, a constraint is associated with each node. X(ij) is a variable which equals 1 if arc i enters node j , -1 if arc i leaves node j, and 0 otherwise.

The following LP matrix (Figure 3.2) is another way of presenting the formulation of (1). It represents the nodes and arcs with columns and rows.

	A 0	A 1	A 2	A 3	A 4	A 5	A 6	A 7		
Obj	10	12	6	7	10	9	8	11		
N0	-1	-1							>=	-100.0
N1	1		-1						>=	0.0
N2		1	1	-1	-1				>=	300.0
N3					1	-1			>=	-1200.0
N4				1			1		>=	500.0
N5						1	-1	1	>=	-800.0
N6								-1	>=	0.0

Figure (3.2)

LP matrix of Example I

This matrix contains 8 columns and 8 rows. The 8 columns in the dashed box represent 8 arcs in the network and starting from the second row, each row is associated with a node. In

each column, there are 3 non-zero values. The value in the first row is the cost associated with each arc, and the pair of -1 and 1 values in two of the next 7 rows indicates where each arc starts and where it ends, respectively. For instance, the -1 in the intersection of row N2 and column A3 indicates arc A3 leaves node 2, and the 1 in row N4 shows that this arc enters node 4. The cost function of the LP is stored in the first row of the matrix. The following 7 rows represent the 7 demand constraints that are associated with each node. For example, the row N2 associated with node 2 stands for the inequality constraint:

$$A1 + A2 - A3 - A4 >= 300.0$$

The directions of all inequalities are in the next column to the right and the right hand side of the inequalities are in the last column. Most commercial LP solvers can read this matrix (in MPS format) and solve the problem. Thus, the key to solving an LP problem is to formulate the matrix.

II. Example II : Arc Capacities

Now, suppose some of the arcs in the network shown in example I (Figure 3.1) have finite capacity. This is the second example, which is simply the first example with capacitated arcs. Besides the demand constraints, another set of constraints is required to ensure none of the capacities are violated.

$$\sum_{j \epsilon A} A(i)Y(ij) <= \text{Capacity}(i) \quad \text{for all } i \epsilon A$$

where Y(ij) is a binary variable equal to 1 if i equals j and 0 otherwise. The LP matrix of example II (Figure 3.3) is:

	A 0	A 1	A 2	A 3	A 4	A 5	A 6	A 7		
Obj	10	12	6	7	10	9	8	11		
N0	-1	-1							>=	-100.0
N1	1		-1						>=	0.0
N2		1	1	-1	-1				>=	300.0
N3					1	-1			>=	-1200.0
N4				1			1		>=	500.0
N5						1	-1	1	>=	-800.0
N6								-1	>=	0.0
AC0	1								<=	200.0
AC1		1							<=	150.0
AC2			1						<=	200.0
AC5						1			<=	500.0
AC6							1		<=	500.0

Figure 3.3

LP matrix of Example II

The upper part of the matrix is exactly the same as example I. The added arc capacity constraints are depicted in rows from AC0 to AC6. There are no capacity constraints corresponding to arc A3, A4 and A7 since they are arcs with infinite capacity. The row AC5 represents the following constraint:

A5 <= 500.0

III. Example III : Multiple Modes

For the previous network, a second transportation mode between nodes is introduced(Figure 3.4). The solid line arcs are of mode A, and the dashed lines are of mode B. To model this multimodal problem, not only are constraints on the new arcs needed, but logical arcs also are required in the network. In example III, a node is modeled with 2 logical nodes and logical arcs between them (Figure 3.5). The logical arcs are transfer arcs used to transfer goods from one mode to another. For simplicity, it is assumed that the B mode arcs and transfer arcs all have infinite capacity. The mathematical formulation of example III is:

$$\text{Min} \sum_{i \epsilon A} [CA(i)A(i) + CB(i)B(i)] + \sum_{i \epsilon N} [CAB(i)AB(i) + CBA(i)BA(i)]$$

s.t.

$$\sum_{i \epsilon A} A(i)X(ij) + \sum_{i \epsilon A} B(i)X(ij) \geq D(j) \quad \text{for all } j \epsilon N$$

$$\sum_{i \epsilon A} A(i)X(ij) + \sum_{i \epsilon N} [AB(i)Z(ij) + BA(i)Z(ij)] \geq \min(D(j),0.0) \quad \text{for all } j \epsilon N$$

$$\sum_{i \epsilon A} B(i)X(ij) + \sum_{i \epsilon N} [AB(i)Z(ij) + BA(i)Z(ij)] \geq \min(D(j),0.0) \quad \text{for all } j \epsilon N$$

$$\sum_{i \epsilon A} A(i)Y(ij) \leq \text{Capacity} \quad \text{for all } j \epsilon N$$

where

A(i): flow amount on A mode arc i.

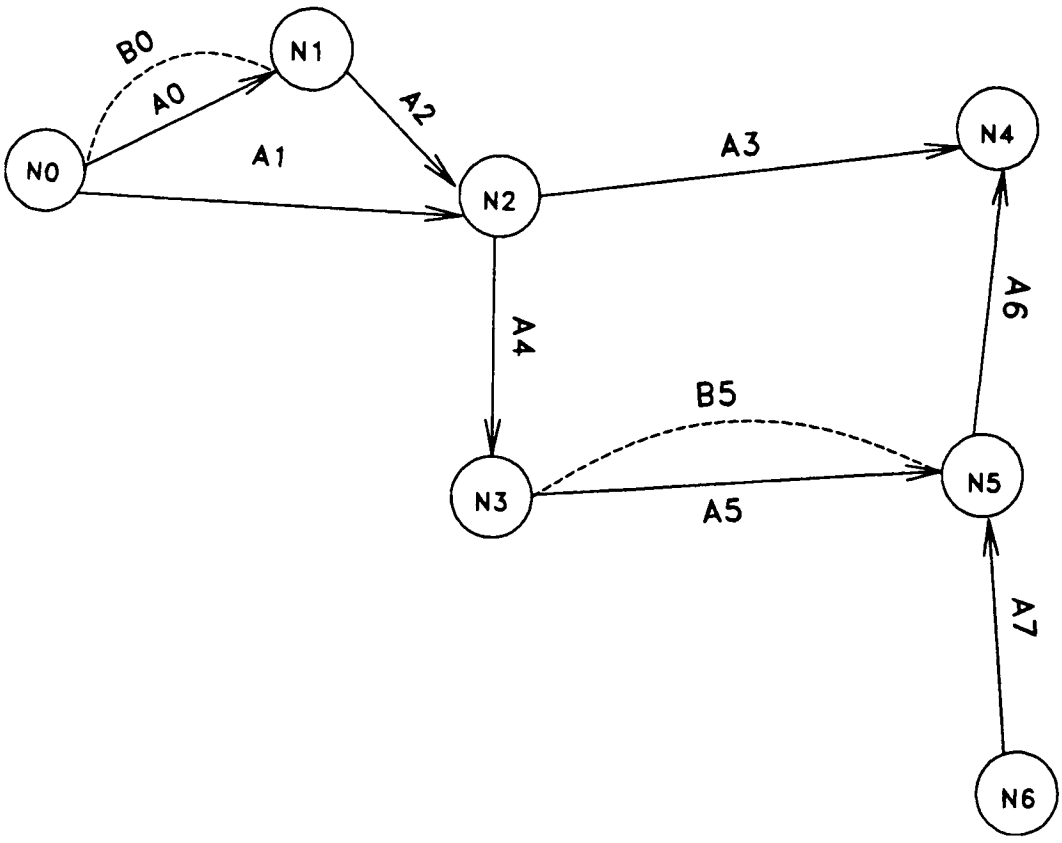


Figure 3.4

Network of Example III

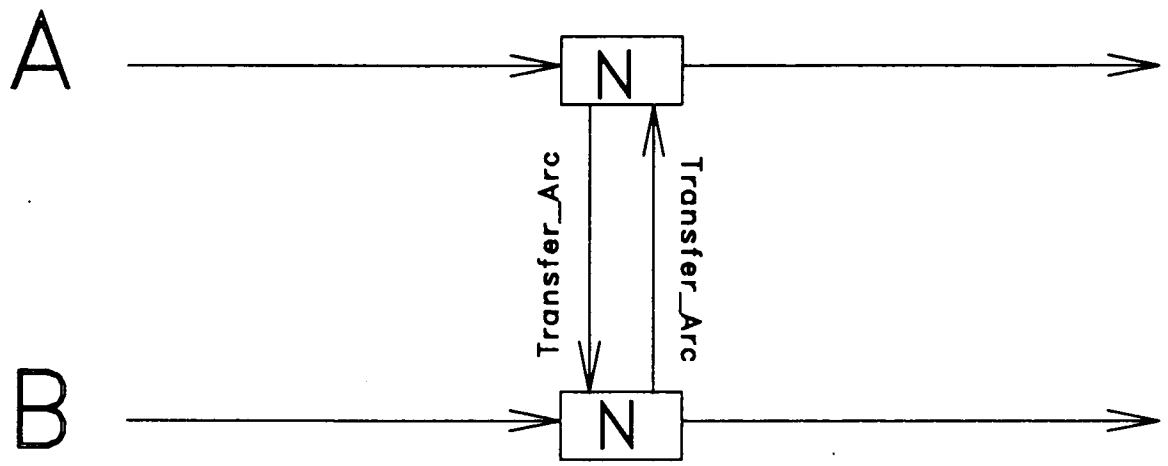


Figure 3.5

Logical Nodes and Logical Links

B(i): flow amount on B mode arc i.

AB(i): transfer amount from A to B mode at node i.

BA(i): transfer amount from B to A mode at node i.

CA(i): cost of A mode arc i.

CB(i): cost of B mode arc i.

CAB(i): transfer arc cost from A to B mode at node i.

CBA(i): transfer arc cost from B to A mode at node i.

Z(ij) : 1 if transfer arc i enters logical node j, -1 if it leaves, 0 otherwise.

X(ij) and Y(ij) are defined the same as in example I and II.

This formulation may seem to be much more complicated than that of the previous examples. However, the structure of the matrix stays the same, the columns stand for decision variables, arcs in this case, and the rows represent constraints on the entities of the network (see Figure 3.6).

The upper left part of the matrix is similar to the matrix of example I. Columns B0 and B5 represent the second mode arc that are parallel with arcs A0 and A5. The transfer arcs are to the right. The rest of columns are the same as those in example I and II.

There are a few changes in the rows. First, a set of constraints is added to ensure the conservation of flow at each node by every mode. For example,

A5 - A6 + A7 - AB5 + BA5 >= -800.0

is the constraint for row MA5. It makes sure the amount of

	A 0	B 0	A 1	A 2	A 3	A 4	A 5	B 5	A 6	A 7	AB 0	BA 0	AB 1	BA 1	AB 3	BA 3	AB 5	BA 5	
Obj	10	6	12	6	7	10	9	6	8	11	3	3	3	3	3	3	3	3	
N0	-1		-1																>=-100.0
N1	1			-1															>= 0.0
N2			1	1	-1	-1													>= 300.0
N3						1	-1												>=1200.0
N4					1				1										>= 500.0
N5							1		-1	1									>=-800.0
N6										-1									>= 0.0
AC0	1																		<= 200.0
AC1			1																<= 150.0
AC2				1															<= 200.0
AC5							1												<= 500.0
AC6									1										<= 500.0
MA0	-1									-1	1								>=-100.0
MB0		-1									1	-1							>= 0.0
MA1	1			-1									-1	1					>= 0.0
MB1		1											1	-1					>= 0.0
MA3						1	-1								-1	1			>=1200.0
MB3								-1							1	-1			>= 0.0
MA5							1		-1	1							-1	1	>=-800.0
MB5								1									1	-1	>= 0.0

Figure 3.6

LP matrix of Example III

flow into node 5 by mode A and the amount transferred from mode B minus the amount of flow out by mode A and transferred out to mode B at least satisfies the demand at node 5.

IV. OOP Models

The three LP tableaus have several characteristics in common. First, the columns and the rows all represent some spatial entity, arcs or nodes. Each column has a name, and a sequence number associated with its position in the matrix. It is related to a decision variable and has a corresponding cost. Each row describes a constraint of an entity, it has a direction and a right hand side representing the constraint inequality. Finally, the matrix is made up of coefficients that are based on the relationship between the columns and the rows. The columns and the rows can be encapsulated in a class hierarchy of C++, and the relationships between them can be modeled by a message passing process among the instances of the classes.

A generic class COLUMN for all the columns of the LP matrix can be defined as:

```
class COLUMN{
public:
   char Name[9];
   float Cost;
```

```
    int Seqnum;
    COLUMN    *Next;
    virtual float Action(class ROW *rw);
    COLUMN(poly_or_point *q, int j);
    COLUMN(line *q, int j);
    virtual ~COLUMN(){};
};

float COLUMN :: Action ( ROW *rw)
{
    return(rw->Row_Action(this));
}
```

The data members and functions of COLUMN are common for all the columns in a LP matrix. 'Name' stores the arc name, and 'Cost' is the unit cost of transporting goods along the arc. 'Seqnum' identifies a column in the program, which can be used as an ID. In order to link all the columns together, there is a pointer to the next column in the matrix for every column. 'COLUMN' has two constructor functions that build a column either by line or by polygon-or-point entity. Since 'polygon' and 'point' are treated as same kind of entity in ArcInfo, a point is a polygon with 0 area and 0 perimeter, they share one constructor function. The two constructor functions are overloaded functions and are distinguished only by the entity by which they are passed, a line or a polygon-

or-point. The destructor function releases memory when an instance of the class goes out of scope. The member function 'Action' sends the address of an instance of COLUMN to a row. The destructor and Action function are declared as 'virtual', which means a derived class of COLUMN can override these functions by defining its own version of them. When a virtual function is called in the program, the version in the caller's class level will respond. If it is not defined at this level, the parent's is used.

Each specific type of column needs to be modeled by a derived class of COLUMN with its own data members and functions. In example I, the following derived class 'Arc' defines all the physical arcs, which are all the columns of that example.

```
class Arc:public COLUMN{
public:
   int From;
   int To;
   virtual float Action(class Row *rw);
   Arc(line *q, int j);
   virtual ~Arc(){};
}
float Arc :: Action (ROW *rw)
{
   return(rw->Row_Action(this));
```

```
}
```

The data members 'From' and 'To' store the from node and the to node IDs of an arc. They are initialized in Arc's constructor function. This information is obtained from ArcInfo's Arc Attribute Table (AAT) file.

```
Arc::Arc(line *q,int j):COLUMN(*q, j)
{
     From = q->anode;
     To   = q->bnode;
}
Arc  c1;
```

"c1" is declared as an instance of Arc, but it also inherits all the data members and functions of COLUMN. Suppose c1 is an arc from node 0 to node 1. The Action function sends messages about what it is to a row, a non_zero coefficient is constructed if there exists a relationship between the row and the column. Because it is virtual, the row receives the specific type of the column. In this case, the row knows the column is an Arc.

Parallel with that of columns in the LP matrix, Class 'ROW' is the generic class for all the rows. It includes a constraint's name, right hand side value, direction and a pointer to the next row as data members.

```
class ROW{
public:
   char Name[9];
   float RHS;
   char Direction;
   ROW *Next;
   virtual float Row_Action(COLUMN *cl) {return(0.0)}
   virtual float Row_Action(Arc *cl) {return(0.0)}
   ROW(poly_or_point *q, int j);
   ROW(line *q, int j);
   virtual ~ROW(){}
}
```

Member function 'Row_Action' responds to the messages that columns pass to it. In order to be able to define its own Row_Action function of a derived class, all the Row_Action functions of ROW are virtual. In class ROW, a virtual Row_Action declaration for every class of column is defined, which returns the default value of 0.0.

As with child classes of COLUMN, there is a specific derived class definition for every type of row in a matrix. In the first example's matrix, except for the row that describes the objective function, all the other rows represent demand constraints of nodes in the network. Thus, there are two derived classes of ROW; Demand, and the objective function.

The Row_Action function of Demand, which reacts with messages from Arc, is constructed as follows. Suppose c1 sends the message about what it is to Demand rows. Since the rows that represent node 0 and node 1 are the from and to node of c1, a coefficient of -1 is generated between node 0 and c1, and a coefficient of 1 is generated between node 1 and c1. Non-zero coefficients of other columns with their corresponding rows are produced by the Row_Action functions in a similar way. The following is the formal definition of Demand and its Row_Action function:

```
class Demand:public ROW{
public:
   int Place;
   Demand(poly_or_point *q, int j);
   virtual float Row_Action(Arc *c1);
   virtual ~Demand(){};
}
Demand::Demand(poly_or_point *q, int j):ROW(*q, j)
{
   Place = q->node_id;
}
float Demand :: Row_Action (Arc *c1)
{
   if (Place == c1->From)
      return(-1.0);
```

```
    if (Place == cl->To)
       return(1.0);
    return(0.0);
}
```

Class 'OBJ' is automatically defined as a derived class of ROW. It includes all the virtual Row_Action functions for every column class, which generates a non-zero coefficient if there exists a cost for a column.

With the necessary class definitions and function prototypes, the core code walks through the list of rows and columns.

```
ROW *row, *rtop;
COLUMN *col, *ctop;

col = ctop;
while(col)
   {
   row = rtop;
   while(row)
      {
       col->Action(row);
       row = row->Next;
      }
   col = col->Next;
```

```
}
```

The double loop executes a message passing process. The address of a column is sent to every row in the row list. Each row first checks the type of column it is passed. If it is the proper type of column, the row's Row_Action function is then executed. If there is interaction between the data members of the row and the column defined in the Row_Action function, a non-zero coefficient may be generated. Otherwise, the coefficient will be zero.

For the first example problem, a simple non-OOP C program can be written to formulate the LP matrix. The object oriented approach may seem to be overkill. However, suppose the problem is changed slightly by introducing flow capacity constraints on some of the arcs in the network, such as the second example. In the C program, the structure of the main program needs to be changed. In each stage of the program, whenever the arc list is looped, if-statements must be added to categorize capacitated arcs and uncapacitated arcs. With the OOP approach, the change requires two new derived classes: a derived class of COLUMN for capacitated arcs, and a derived class of ROW to represent the constraints associated with the capacity of each arc:

```
class Cap_Arc:public Arc{
    float capacity;
```

```
    Cap_Arc ( line *q, int j):Arc(*q, j)
                {capacity = q->capacity};
    virtual float Action (ROW *rw);
    virtual ~Cap_Arc(){};
}
float Cap_Arc :: Action (ROW *rw)
{
   return(rw->Row_Action(this));
}
```

This Action function will send the address of an instance of Cap_Arc to rows.

```
class Arc_Cap:public ROW{
    Arc_Cap(line *q, int j):ROW(*q, j);
    virtual float Row_Action(Cap_Arc *cl);
    virtual ~Demand(){};
}
float Arc_Cap :: Row_Action (Cap_Arc *cl)
{
    if(SeqNum == cl->SeqNum)
        return(1.0);
    return(0.0);
}
```

The Row_Action of Arc_Cap is defined to return a non-zero

value if the arc that sends a message to it is a capacitated arc and the row and column arc IDs match. **The rest of the class definitions and the core code are unchanged.** Compared to the C program, the modifications of the OOP model are much less complicated and the logical structure of the program remains the same. This flexibility is one of the characteristics of the OOP approach.

In real world networks, the nodes and arcs are much more complex than that of the first and second examples. Normally, there are multiple modes like those in the third example. Thus, some logical arcs that don't exist physically, need to be created. For this problem, two new derived classes are necessary to incorporate multiple modes, a derived class of COLUMN for transfer arcs and a derived class of ROW representing Demand_by_Mode constraints at each node for every mode. Here is the definition of transfer arcs:

```
class Transfer_Arc: public COLUMN{
public:
    int Fmode;
    int Tmode;
    int Place;
    Transfer_Arc (poly_or_point *q; int j, int i, int k):
            COLUMN(*q,k)
            {Fmode = i; Tmode = k; Place = q->node_id;}
    virtual ~Transfer_Arc(){};
```

```
        virtual float Action(class ROW *rw);
    }
```

Demand_by_Mode constraints make sure no flow is created or lost while transferring from mode to mode.

```
    class Demand_by_Mode: public ROW{
    public:
        int Mode;
        int Place;
        Demand_by_Mode(poly_or_point *q, int j, int k):
                ROW(*q,k){ Mode = k; Place = q->node_id;}
        virtual float Row_Action(Arc *cl);
        virtual float Row_Action(Cap_Arc *cl);
        virtual float Row_Action(Transfer_Arc *cl);
        virtual ~Demand_by_Mode(){};
    }
```

The rows of this class react to all the columns in the matrix. Thus, a Row_Action function is defined to interact with each type of column. The Row_Action functions that deal with columns of Arc and Transfer_Arc are given here:

```
    float Demand_by_Mode :: Row_Action (Arc *cl)
    {
        if (Mode == cl->Mode)
```

```
        {
            if(Place == cl->From)
                return(-1.0);
            if(Place == cl->To)
                return(1.0);
        }
        return(0.0);
    }
    float Demand_by_Mode :: Row_Action (Transfer_Arc *cl)
    {
        if (SeqNum == cl->SeqNum)
        {
            if(Mode == cl->Fmode)
                return(-1.0);
            if(Mode == cl->Tmode)
                return(1.0);
        }
        return(0.0);
    }
```

When a column of type Arc is sent to a row that is an instance of Demand_by_Mode, it first checks whether the Arc is of the appropriate mode. If it is, it then checks if the arc leaves or enters the node the row represents. For the column representing a Transfer_Arc, the SeqNum of the node is checked to see if it matches the row, then -1.0 is returned if the arc

is transferring from the mode and 1.0 if transferring to the mode.

There are a few modifications in the existing definition of row classes. Every virtual function in the derived classes of ROW needs to be defined in ROW. Thus, the following functions are added in ROW as member functions:

```
virtual float Row_Action(Cap_Arc *cl) {return(0.0)};
virtual float Row_Action(Transfer_Arc *cl) {return(0.0)};
```

In the Obj class, these Row_Action functions are also defined to return costs of different columns. For instance, the definition of a Transfer_Arc is,

```
virtual float Row_Action (Transfer_Arc *cl)
{
   if (SeqNum == cl->SeqNum)
      return (cl->cost);
   return(0.0);
}
```

The core code, which walks through every column to contact all the rows, and generates the coefficients in the matrix, remains the same.

LP problems are well structured problems which can be represented by matrices. For any LP, the data members and

constructor functions in columns and rows can be defined similar to those in examples I, II and III. In the OOP approach, the core code is of a fixed structure. Thus, a tool can be built to generate the necessary class definitions, function prototypes, and core code based on the spatial primitives of points, polygons and lines. The next chapter will introduce a such tool, the OOP generator.

CHAPTER 4

THE OOP GENERATOR

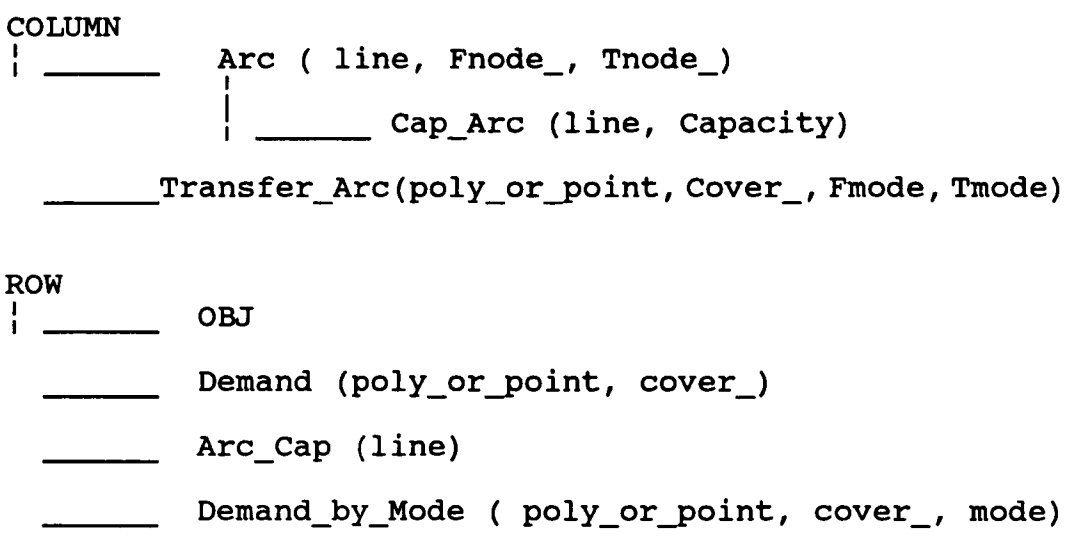
This chapter introduces the OOP generator, and shows how it constructs programs of the type discussed in the previous chapter. The generator is composed of two parts. The first part is a query system which asks the user to describe his or her problem. This part requires a user to enter information about a problem by naming the necessary classes (row and column types), the GIS information each class needs, and the relationships among the classes. The second part of the generator writes an OOP program for formulating linear programs (LPs) based on the information obtained from the previous part. To illustrate how the generator works, I will use example III from the last chapter.

To construct an OOP program of an LP problem, the user needs to provide the following information:

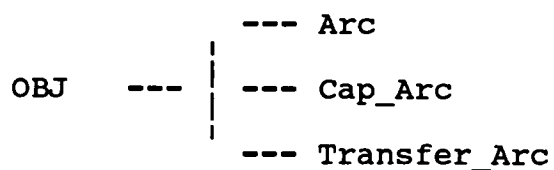
- the types of rows and columns in the LP matrix;
- the spatial basis for each row and column (a poly_or_point or a line);
- the GIS information to be used as data members of each class;
- the parent of each class; and
- the interaction between the row and column classes.

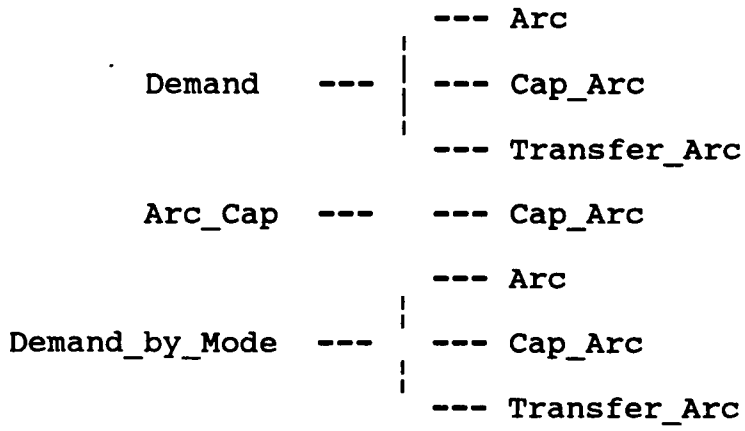
Consider example III from the previous chapter. The

information about class definitions that the generator needs is summarized in the following class family trees.



Every class is directly derived from the one to its left in the above figure. Classes 'Arc' and 'Transfer_Arc' are children of 'COLUMN' while 'Cap_Arc' is a child of 'Arc'. 'ROW' is the parent of class 'OBJ', 'Demand', 'Arc_Cap' and 'Demand_by_Mode'. The first item in the parenthesis after a class name determines what kind of spatial entity the class represents, and the rest of the items are the data members of that class. For instance, class Cap_Arc is generated from a line entity and has a data member called Capacity. For each derived row class, the user needs to enter the column classes with which it interacts. They are given in the following family relationships:





For LP problems, the generic classes ROW and COLUMN, and OBJ are built automatically. The generator then asks the user to define the derived classes that are needed. After the user indicates the row or column he wants to create, its name and the spatial entity on which it is based, the generator lists all the attributes from ArcInfo's Polygon Attribute Table (PAT) file or Arc Attribute Table file. If the class is generated from both lines and points_or_polys, attributes common to both the AAT and PAT files are listed. The user chooses the data members of the class from these listed attributes. The process is repeated until all the necessary classes are constructed.

After the first derived class is created, the user needs to assign a parent for a newly derived class, since it could be derived from a generic class or other derived classes. For instance, the user needs to specify that 'Cap_Arc' is a child of 'Arc' not 'COLUMN' in example III. To help the user identify its parent, the generator provides a "family tree" every time a new class is about to be created. A parent class

of line entity cannot have a child class of point_or_poly entity nor can a point_or_poly entity have a child class of line entity. However, if the parent is an entity defined by both poly_or_point and line, the child has the choice of being generated by either one or both. The attributes that are chosen as data members of a parent won't be listed again for its child, since a child automatically inherits them from the parent.

After the user defines all the classes of row and column, he specifies the column classes with which each row class interacts. For every row, the generator prints the family tree of columns to help the user determining the columns.

Consider example III. The generator starts by asking the user **Do you want to create a row or a column?** (Figure 4.1). Suppose the user wants to define the column classes first. He enters **c**, and when prompted for the class name, he enters **Arc**. The generator then asks **Is this class generated by 1> polys, 2> lines, and 3> both.** Since Arc represents a line entity, the user needs to enter option **2**. The generator then lists the default with data types stored in the GIS.

Fnode_

Tnode_

Lpoly_

Rpoly_

Cover_

Cover_id

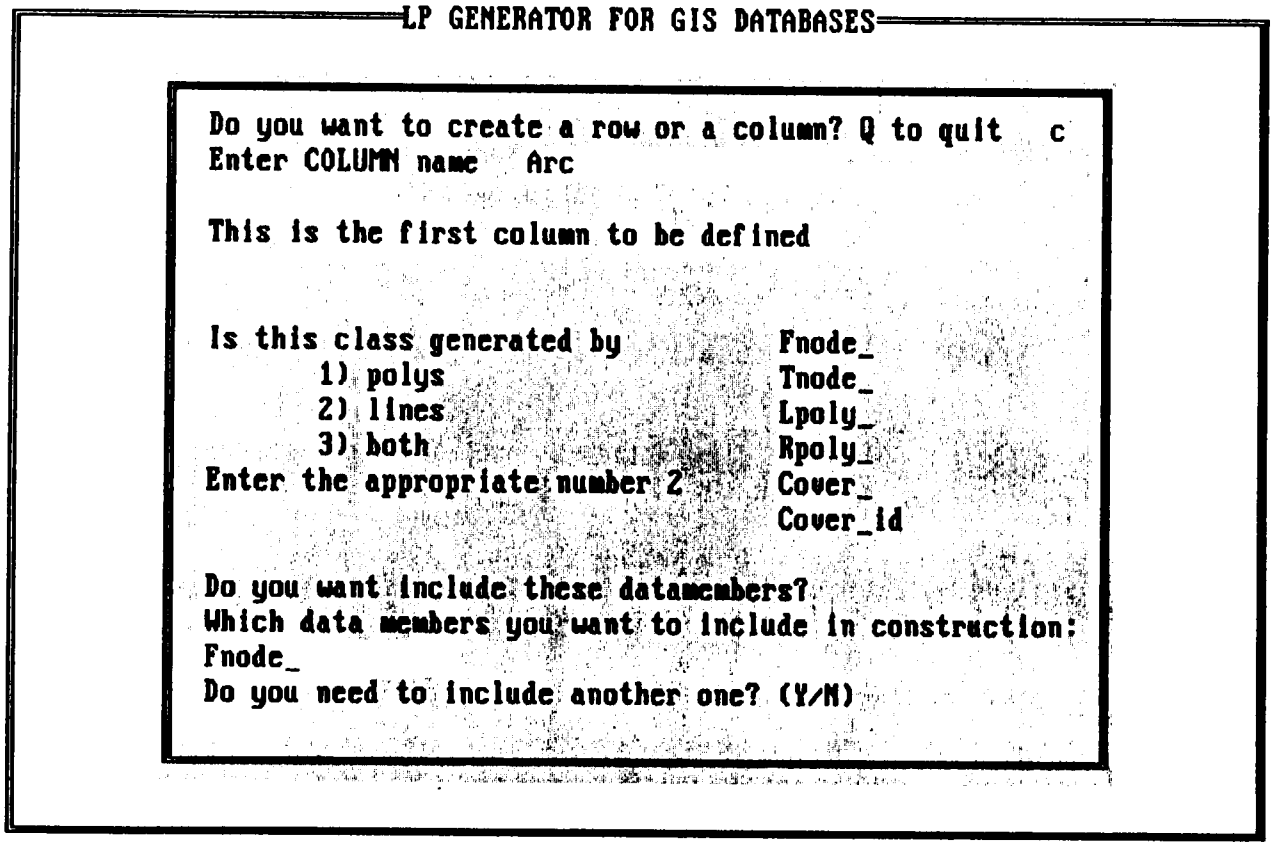


Figure 4.1

LP Generator for GIS Databases: Creating Column with Line Entity

The user can choose which attributes should be saved as data members of **Arc**. In the present example he needs to select Fnode_ and Tnode_ to specify the starting and ending nodes of an instance of **Arc**.

After each class is built, the generator prompts the user for the next class. In this case, he enters **c** again to build **Cap_Arc**. Because a child of COLUMN is defined, the generator presents the family tree of COLUMN and asks the user to **Enter the column's parent** (Figure 4.2). For example III, he would enter **Arc** since Cap_Arc is a special case of Arc. Next, the generator lists the attributes that Cap_Arc might use as data members:

Lpoly_

Rpoly_

Cover_

Cover_id

Notice that Fnode_ and Tnode_ are not listed. This is because Cap_Arc is a child of Arc and Arc has data members Fnode_ and Tnode_. By inheritance, Cap_Arc also has them as data members. The data member 'Capacity' that Cap_Arc needs is not a built-in attribute of arcs in the AAT file, thus, it is not listed here. The user needs to add it separately. None of the attributes listed above is required for Cap_Arc.

Since the class **Transfer_Arc** is constructed by an poly_or_point entity, the generator provides the attributes of a PAT file (Figure 4.3).

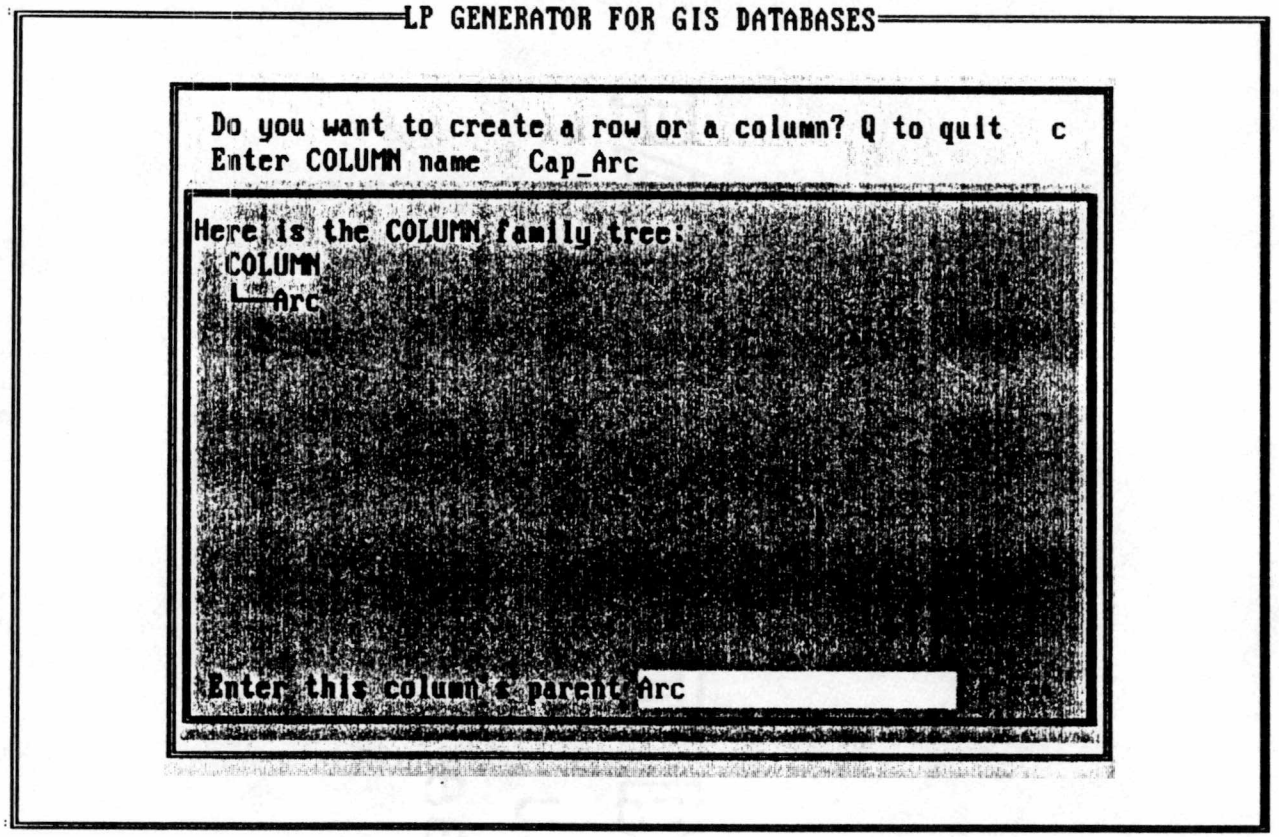


Figure 4.2

LP Generator for GIS Databases: Building Family Tree

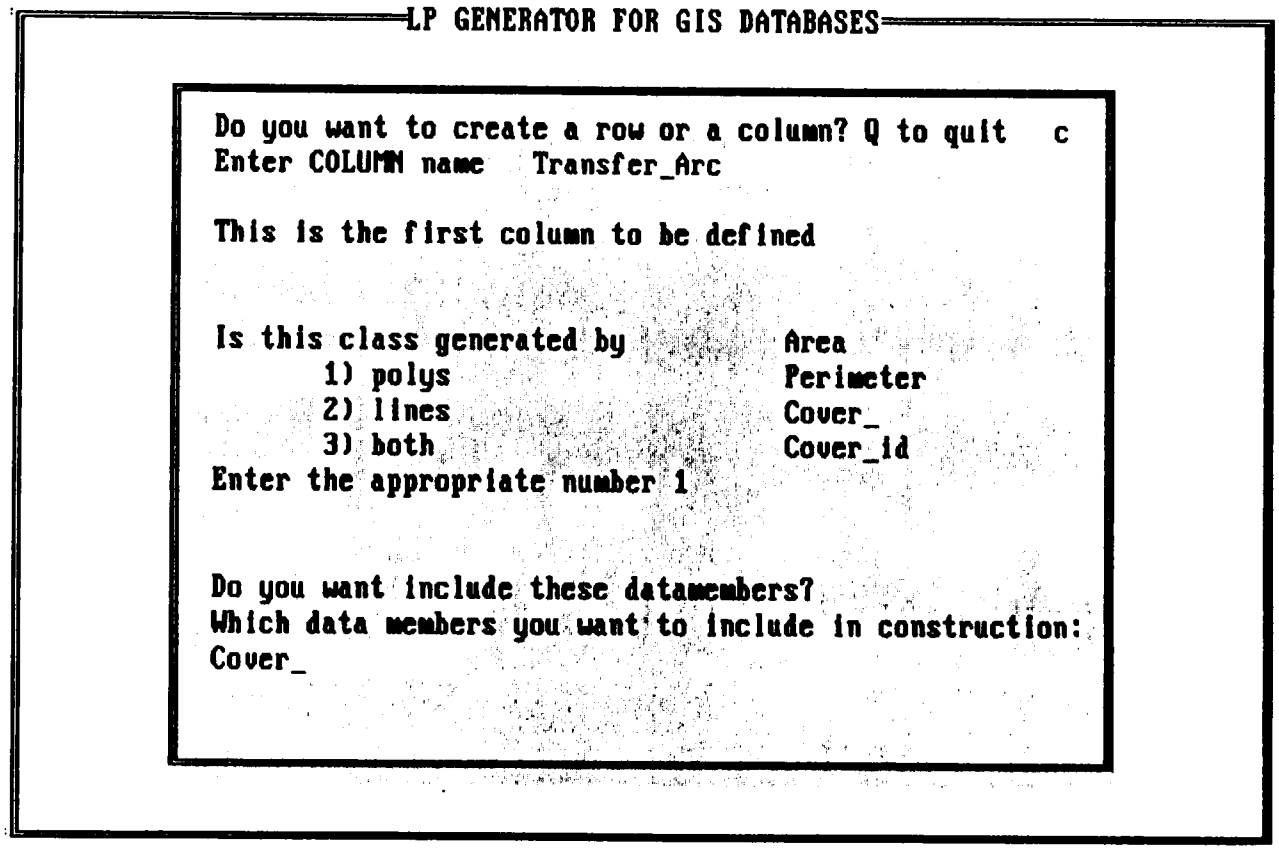


Figure 4.3

LP Generator for GIS Databases: Creating Column with Poly Entity

Area

Perimeter

Cover_

Cover_id

Cover_ is selected as a data member of **Transfer_Arc** because it stores the sequence number of an poly_or_point entity. As with the data member Capacity of **Cap_Arc**, Fmode and Tmode required by **Transfer_Arc** need to be added later on. After building the columns, the user enters **r** for the first question and defines the row classes similar to that of columns (Figure 4.4).

After the user defines all the classes for columns and rows of example III's LP matrix, the generator starts the second stage of the query system,'**DEFINING INTERACTIONS**' (Figure 4.5). It first asks with which columns the **Demand** row interacts. The user needs to enter **Arc**, **Cap_Arc** and **Transfer_Arc**, since it reacts to all the columns. For row class **Arc_Cap**, he would enter **Cap_Arc** as the only interaction. **Demand_by_Mode** is defined exactly the same as class **Demand** since it also responds to all the columns.

In addition to creating the needed class definitions, the generator declares and builds prototypes for virtual Row_Action functions for each row class. A Row_Action function is required for every column class with which the row class interacts. Consider the Demand row in example III. The generator builds the prototypes of the following:

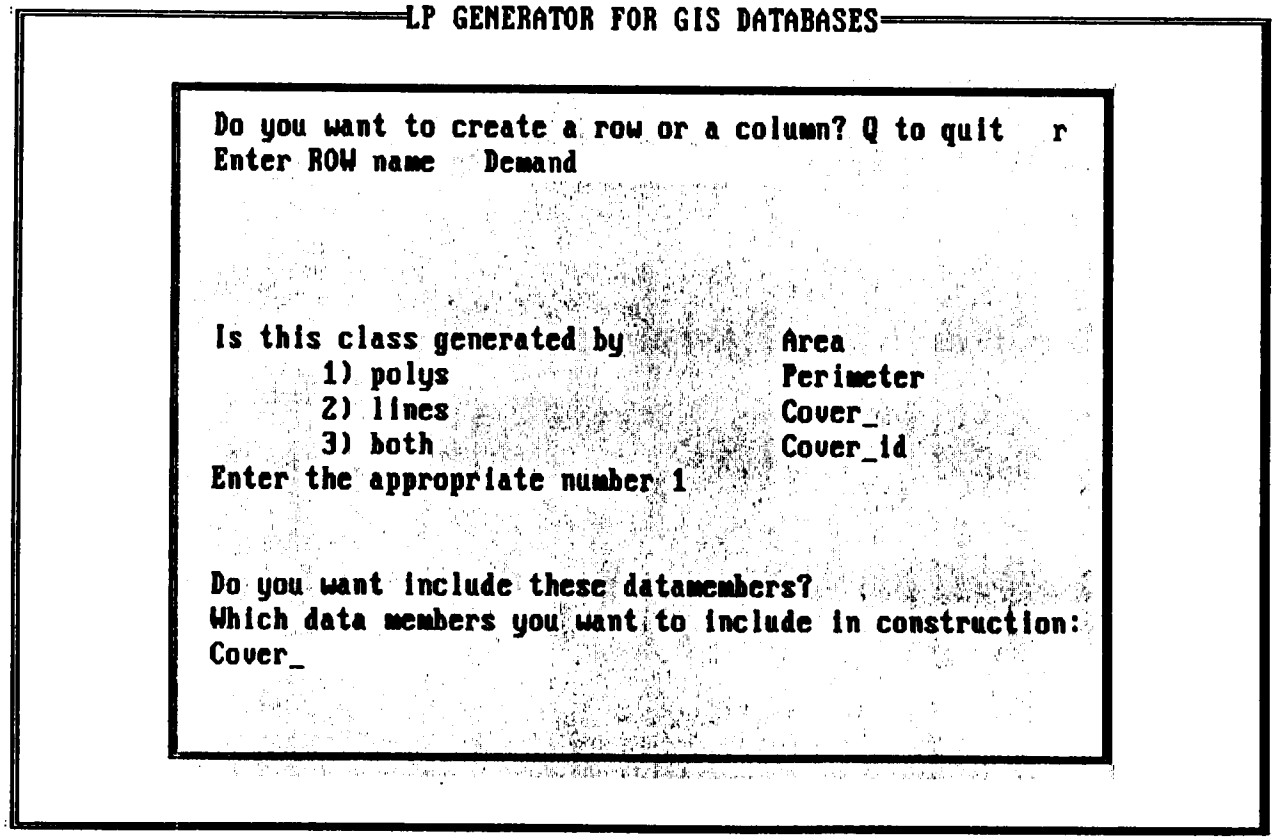


Figure 4.4

LP Generator for GIS Databases: Creating Row

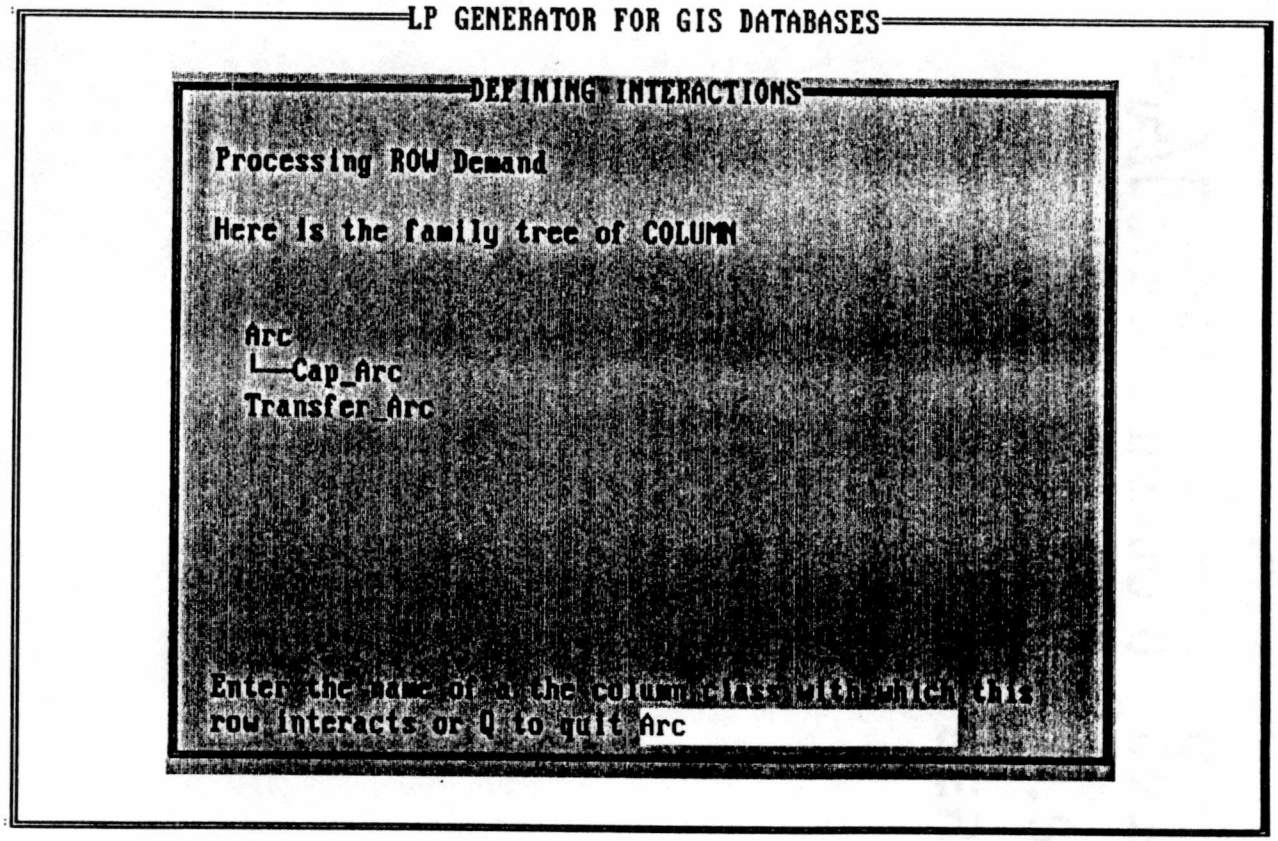


Figure 4.5

LP Generator for GIS Databases: Defining Interactions

```
float Demand :: Row_Action (Arc *cl)
float Demand :: Row_Action (Cap_Arc *cl)
float Demand :: Row_Action (Transfer_Arc *cl)
```

For the row Arc_Cap, it builds

```
float Demand :: Row_Action (Cap_Arc *cl)
```

In order to help the user define the relationship between a row and a column, the generator provides the specific data members for the specific row and column in the prototype of each Row_Action function. For example,

```
float Demand :: Row_Action(Arc *cl)
{
/* Here are the Demand's data members :
    Cover_
   Here are the Arc's data members :
    Fnode, Tnode
*/
    return (0.0);
}
```

In example III, the user would define the following relationship in above Row_Action:

```
{
     if(Cover_ == cl->Fnode)
        return(-1.0);
     if(Cover_ == cl->Tnode)
        return(1.0);
}
```

All the Row_Action functions return coefficients of the matrix based on the relationship that the user defines between rows and columns.

It is the user's responsibility to open the proper attribute files to get information on the poly_or_point and line entities. For example, one could use Codebase++, a commercial library of functions which can open and read from "DBF" files, to open ArcInfo's AAT and PAT files.

```
for(AAT.top; !AAT.eof(); AAT.skip())
    classify_line(p,int j++);
for(PAT.top; !PAT.eof(); PAT.skip())
    classify_poly(p, int j++);
```

The functions 'classify_line()' and 'classify_poly()' transfer the information stored in entity **p** and the sequence id **j** into instances of row and column classes by calling the constructor functions. These two functions build the instances onto the row and column trees simultaneously. For

example, a line entity is classified as an Arc as follows:

```
if (CTop == NULL)
    CTop = CBottom = new Arc(p, j);
else
    CBottom = CBottom->Next = new Arc(p, j);
```

where CTop and CBottom are pointers to COLUMN.

Appendix B includes the program that the generator constructed for example III with twin arcs. It is written into two files. The h_file contains all the class definitions and function prototypes and the .cpp file includes the main program. For example, the column class Arc is defined as following in h_file:

```
class Arc:public COLUMN{
  long  Fnode_;
  long  Tnode_;
  virtual float Action(class ROW *rw);
  Arc(line *q, int j);
  virtual ~Arc(){};
};
Arc::Arc(line *q, int j):COLUMN(line *q, int j)
{/* constructor function goes here. */
  Fnode_=p->Fnode_;
  Tnode_=p->Tnode_;
```

```
}
```

The core code inside the main procedure not only walks through the row and column trees, but also writes the LP matrix into a MPS file, which is a standard format used by most commercial LP solvers.

```
ROW *Rows;
COLUMN *Cols;
Rows = RTop->Next;
   while (Rows)
{
fprintf(out," %c %-8s\n", Rows->Direction, Rows->Name);
Rows = Rows->Next;
}
Cols = CTop;
while (Cols)
{
Rows = RTop;
while (Rows)
   {
   value = Cols->Action(Rows);
   if (value != 0.0)
      {
      fprintf(out,"    %-8s  %-8s  %12.3f\n",
     Cols->Name, Rows->Name, value);
```

```
            }
         Rows = Rows->Next;
         }
      Cols = Cols->Next;
      }
      Rows = RTop;
      while (Rows)
      {if (Rows->Rhs != 0.0)
         {fprintf(out,"      RHS          %-8s  %12.3f\n",
                                   Rows->Name,Rows->Rhs);
      }
           Rows = Rows->Next;
      }
}
```

Compared to the complete program for the same problem in Appendix C (which reads data from ASCII files that could be dumped out from ArcInfo), the generator writes 86% of the entire code.

With the OOP approach the user can concentrate more on the relationships among the spatial entities, the rows and the columns, and less on coding the program. However, there is a limit to the program that can be generated. The generator cannot provide a complete program. For most functions, only the prototype and the parameters that are passed are given. The user has to complete the program by coding the details of

each function and any special classification conditions.

CHAPTER 5

SUMMARY

Current specialized GIS and Spatial Decision Support Systems are limited to specific applications. The OOP models and the OOP generator developed here are methods for integrating mathematical models which require some form of input expressed in matrix form with GIS packages. I introduced the OOP language C++ and problems associated with matrix formulation in Chapter 2. In Chapter 3 I showed how OOP models would attack geographical LP problems, which are typical matrix-based problems, and in Chapter 4 I presented an OOP generator that would automate the same task.

OOP increases the flexibility of spatial models. First, it enables the models to be more easily implemented as changes in the definition of problems arise. Second, OOP programs can be applied to different GIS environments with little modification. The OOP models integrate with a GIS system by accessing its data base. Therefore, modifying the function which reads input data files will enable the system work with different GIS environments. Third, because of the structure of OOP languages, such as the seldom changed core code, and the known type of spatial primitives (points, polygons, and lines) in a GIS, the OOP generator developed for this thesis can construct source code for other matrix-based models. This enables the generator to be used with variety of problems.

The programs created by the generator focus on how to formulate problems rather than how to solve them. Other models have been developed to efficiently get solutions and they often assume the problems are well defined and easily formulated. Most of the literature on specialized GIS and SDSS concern the efficiency of solving a particular problem. However, in practical applications, users often find that it is hard to define and formulate their problems. The OOP generator presented in this thesis is a query system that forces the user to focus on the relationships between spatial entities. The generator takes care of setting up the class structures and writing most of the code.

This process is similar to Vermont Views software (Meadows, 1991). Vermont Views is a user interface development system, which helps users to create data forms, menus, and windows interactively. It then designs the program structure and writes part of the programs for users. It does not write any user-defined functions. It only suggests prototypes for them. This is similar to the generator defining action and reaction functions of classes. It passes all the available variables for each class, but allows the user to define the class relationships.

The OOP generator constructs an LP driver program that reads GIS data and formulates the LP tableau. A separate program is needed to bring the results of a problem back to the GIS system so the user can examine them with the GIS's

display capability. For an LP problem, a report writer that reads LP results and writes information back to the GIS's attribute files could use much of the code and class definitions of the LP driver. Some new classes or structures need to be added to hold data such as primal and dual values of the LP solution. In addition, procedures need to be developed to construct look up tables that GIS packages read.

Not much was said about the contiguity checking problems in the previous chapters. The rows and columns of the contiguity matrix represent the zones in a region. If a row is adjacent to a column, a non-zero coefficient value is returned to the matrix. This problem involves a third dimension, the arc attribute file, beside rows and columns based on polygons. A zone is known to be adjacent to another zone only if they share a common border. For example, the program will have the following pseudo code in the reaction function:

```
for each arc in AAT file
{
        if ((leftpoly = rowid) && (rightpoly = columnid))
             return(1);
 }
 return(0);
```

Further work can be done to make the OOP generator more

user friendly than it is now. One extension is to add a function to read an existing header file, which would include all the class definitions of a previously generated problem. This would allow users to add new columns and rows without having to retype all the column and row names again. The generator in this thesis only lists the default attributes of ArcInfo's AAT and PAT files from which users choose data members of a class. It could be modified to read attributes from a user specified AAT or PAT file. For instance, in example II and III of chapter 3, a user could build CAPACITY as an attribute in AAT file and choose it as a data member of Cap_Arc when the generator lists it. Finally, more error checking procedures could be added to eliminate infeasible options.

Data Hiding was mentioned in chapter 2, but all the variables in the generator and LP models are declared as public. One major reason is that hidden data are protected from unnecessary modification, but they are also insulated from necessary access if no extra or middle functions are introduced. These functions often slow down program execution.

Despite of the drawbacks of some of the OOP language features, there is much to recommend it. As the Object Oriented Programming approach continues to be more accepted, the way we model phenomena will reflect more directly the way we think about those phenomena.

BIBLIOGRAPHY

Berrill, A. and G. Moon, 1992, " An Object Oriented Approach to An Integrated GIS System," GIS/LIS' 92 Proceedings, Vol.1, 59-63.

Church, R., 1990, NCGIA Technical Paper, 90-95.

Entsminger, G., The Tao of Objects: A Beginner's Guide to Object-Oriented Programming, Redwood City, CA: M&T Books.

Goodchild, M. F., 1991, "Spatial Analysis With GIS: Problems and Prospects," GIS/LIS' 91 Proceedings, Vol.1, 40-48.

Meadows, P., 1991, Vermont Views, Richford, VT: Vermont Creative Software.

Newell, R. G., 1992, " Practical Experience of Using Object-Orientation to Implement A GIS," GIS/LIS' 92 Proceedings, Vol.2, 624-629.

Ralston, B. A. and C. Liu, 1990, The Bangladesh Transportation Modeling System, The International Bank for Reconstruction and Development, Washington, DC.

Ralston, B. A., C. Liu, M. Zhang and L. Ralston 1992, Transportation and Inland Logistics Manager, The Agency for International Development, Famine Early Warning System.

Ralston, B. A., " Object Oriented Spatial Analysis." forthcoming in GIS and Spatial Analysis, S. Fotheringham and P. Rogerson, eds.

Sequiter Software (1991) Codebase++.

Stevens, A., 1990, Teach Yourself C++, Portland, OR: MIS Press.

APPENDICES

APPENDIX A

THE OOP GENERATOR AND ITS HEADER FILE

APPENDIX A1

THE OOP GENERATOR

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "screens.h"
#define PERCENTAGE '%'

FILE *out, *out2, *tmp;
int SPACES, P_ROW, CHECK;
typedef struct{
   long Fnode_;
   long Tnode_;
   long Lpoly_;
   long Rpoly_;
   long Cover_;
   long Cover_id;
   }line;
typedef struct{
   double Area;
   double Perimeter;
   long Cover_;
   long Cover_id;
   }poly_or_point;

struct memstr{
   char name[15];
   int  on;
   struct memstr *next;
   };
typedef struct memstr datamember;

datamember *linemem;
datamember *polymem;
datamember *common;

class CLASSTYPE *CTop, *RTop;
CLASSTYPE *Find(char *, CLASSTYPE *);
datamember *getdatamem (CLASSTYPE *,int);
void turnon(datamember *dm);

class CLASSTYPE{
public:
   char Name[31];
   int Gis_basis;
   CLASSTYPE *Parent;
   CLASSTYPE *Child;
   CLASSTYPE *Next;
```

```
    datamember *datamem;
    CLASSTYPE(char * name);
    CLASSTYPE(char * name, char *pname, char type, int
gis_prim);
    ~CLASSTYPE(){};
};

// The following two functions are the two constructor
// functions of CLASSTYPE.

CLASSTYPE::CLASSTYPE(char * name)
{
    strcpy(Name,name);
    Gis_basis = 3;
    Parent = NULL;
    Child = NULL;
    Next = NULL;
    datamem = NULL;
}

CLASSTYPE::CLASSTYPE(char *name, char *pname, char type, int
gis_prim)
{
    CLASSTYPE *address;

    strcpy(Name,name);
    Gis_basis = gis_prim;

    Child = NULL;
    Next = NULL;
    if (type == 'C')
       Parent = Find(pname,CTop);
    else
       Parent = Find(pname,RTop);
    if (Parent->Child == NULL)
       Parent->Child = this;
    else
       {address = Parent->Child;
       while(address->Next)
       {address = address->Next;
       }
       address->Next = this;
       }

    if(strcmp(name,"OBJ")==0)
       datamem = NULL;
    else
    {
       datamem = getdatamem( Parent, Gis_basis);
       switch(Gis_basis)
       {
          case 1:
```

```
            turnon(polymem);
            break;
          case 2:
            turnon(linemem);
            break;
          case 3:
            turnon(common);
            break;
        }
    }

}

// This function finds the location of a class in a family
// tree.

CLASSTYPE *Find(char *name, CLASSTYPE *present)
{
  CLASSTYPE * value;

  if (strcmp(name, present->Name) == 0)
     {
     return(present);
     }
  if(present->Child)
     {
     if ((value = Find(name,present->Child)) != NULL)
     return(value);
     }
  if(present->Next)
     {
     if ((value = Find(name,present->Next)) != NULL)
     return(value);
     }
  return(NULL);
}

void build_row(void);
void build_col(void);
void write_column_header(CLASSTYPE *);
void write_row_header(CLASSTYPE *);
void write_virtual_header(CLASSTYPE *);
void write_classify(void);
void write_classify_detail(char, CLASSTYPE *, int);
void write_core(void);
void Print_Tree(CLASSTYPE *present, int col);
int checkduplicate(CLASSTYPE *value, char * name);
void write_construct_comment(CLASSTYPE *p);
void write_reaction_comment(CLASSTYPE *p, char *name);
int turnoff(CLASSTYPE *p, datamember *dm);
int checkon(datamember *dm);
void builddatamember ();
```

```
int checkdatamem (char *name, int gs);
int printdatamem(int gisbase);
void classini(CLASSTYPE *p);
void appfile();

void main(void);
void main()
{
  char ans;

  CLASSTYPE *present;

  builddatamember();
  getch();
  disp_open();
  disp_usebios();
  z_sgr(WHITE,BLUE,DISP_INTENSITY);
  z_cls();
  disp_box(0,z_sgr(WHITE,BLUE,DISP_INTENSITY),0,0,23,79);
  z_ctr80(0,"LP GENERATOR FOR GIS DATABASES");

  disp_box(0,z_sgr(WHITE,CYAN,DISP_INTENSITY),2,10,21,69);
  z_sgr(WHITE,CYAN,DISP_INTENSITY);
  z_clearbox(3,11,21,68);

  CTop = new CLASSTYPE("COLUMN");
  RTop = new CLASSTYPE("ROW");
  new CLASSTYPE("OBJ","ROW",'R',0);

  out = fopen("classes.h","w");
  out2 = fopen("classify.cpp","w");
  tmp = fopen("temp.out","w");
  z_sgr(WHITE,BLUE,DISP_INTENSITY);
  while (1){
     z_sgr(WHITE,CYAN,DISP_INTENSITY);
     z_clearbox(3,11,21,68);
     disp_move(3,13);
     disp_printf("Do you want to create a row or a column? Q
to quit ");
     z_sgr(WHITE,BLUE,DISP_INTENSITY);
     disp_move(3,66);
     disp_flush();
     disp_putc(' ');
     ans = getch();
     disp_move(3,66);
     disp_putc(ans);

     if ((ans == 'q') || (ans == 'Q'))
       break;
     switch(ans)
     {
     case 'r':
```

```
      case 'R':
         build_row();
         break;
      case 'c':
      case 'C':
         build_col();
         break;
      default:
         printf("\x7");
      }
      }
     write_column_header(CTop);
     write_row_header(RTop);

     write_core();
     write_classify();


   appfile();
   fclose(out);
   fclose(out2);
}

// This function reads contents of temp.out and writes to
// classes.h file.

void appfile()
{
   FILE *in;
   char buffer[50];

   fclose(tmp);
   in = fopen("temp.out","r");
   while(fgets((char *)buffer,41,in))
       fprintf(out,"%s",buffer);
   fclose(in);
   unlink("temp.out");
}

// This function builds attributes onto datamember link
// lists.

void builddatamember ()
{
    datamember *temp, *temp1, *temp2, *temp3;
    int i;
    char lm[6][15] = { "Fnode_","Tnode_", "Lpoly_",
"Rpoly_", "Cover_", "Cover_id"};
    char pm[4][15] = { "Area", "Perimeter", "Cover_",
"Cover_id"};
```

```
linemem = NULL;
temp1 = NULL;
temp = new datamember;
strcpy(temp->name, lm[0]);
temp->on = 1;
temp->next = NULL;
linemem = temp1 = temp;

for (i =1; i<6; i++)
{
   temp = new datamember;
   strcpy(temp->name, lm[i]);
   temp->on = 1;
   temp->next = NULL;
   temp1->next = temp;
   temp1 = temp1->next;
}

temp = new datamember;
strcpy(temp->name, pm[0]);
temp->on = 1;
temp->next = NULL;
polymem = temp1 = temp;

for (i =1; i<4; i++)
{
   temp = new datamember;
   strcpy(temp->name, pm[i]);
   temp->on = 1;
   temp->next = NULL;
   temp1->next = temp;
   temp1 = temp1->next;
}

temp = new datamember;
common = temp;
temp1 = polymem;
while(temp1)
{
   temp2 = linemem;
   while(temp2)
   {
      if(strcmp(temp1->name,temp2->name)==0)
      {
           strcpy(temp->name, temp1->name);
           temp->on = 1;
           temp->next = NULL;
           temp3 = temp;
           temp = new datamember;
           temp3->next = temp;
      }
      temp2 = temp2->next;
```

```
        }
        temp1 = temp1->next;
    }
    free(temp);
    temp3->next = NULL;
}

// This function checks if user entered 'name' is valid.

int checkdatamem ( char *name, int gs)
{
    datamember *temp;
    datamember *temp2;

    switch (gs)
    {
      case 1:
        temp = polymem;
        while(temp)
        {
           if(strcmp(temp->name,name)==0)
           return (1);
           else
           temp = temp->next;
        }
        return(0);
        break;
      case 2:
        temp = linemem;
        while(temp)
        {
           if(strcmp(temp->name,name)==0)
           return (1);
           temp = temp->next;
        }
        return(0);
        break;
      case 3:
           temp = common;
           while(temp)
           {
              if(strcmp(temp->name, name)==0)
                 return(1);
              temp = temp->next;
           }
           return(0);
        break;
         default:
           printf(" gisbase is wrong \n ");
           return(0);
       }
}
```

```
// This function prints current avaliable attributes onto
// the screen.

int printdatamem(int gisbase)
{
   datamember *temp;
   datamember *temp2;
   int flag=0;

   disp_move( 9, 0);
   switch(gisbase)
      {
      case 1:
         temp = polymem;
         while(temp)
         {
                if(temp->on)
              disp_printf("\t\t\t\t\t\t %s\n", temp->name);
                else flag = 1;
          temp = temp->next;
         }

         break;
      case 2:
         temp = linemem;
         while(temp)
            {
                if(temp->on)
              disp_printf("\t\t\t\t\t\t %s\n", temp->name);
                else flag = 1;
          temp = temp->next;
         }
         break;
      case 3:
         temp = common;
         while(temp)
         {
                if(temp->on)
                    disp_printf("\n\t\t\t\t\t\t%s\n",
temp->name);
                else flag = 1;
                temp = temp->next;
            }
           break;
      }
      return(flag);
}

// This function monitors the user interface while user
// entering datamembers of an object.

datamember  *getdatamem(CLASSTYPE *Parent, int gisbase)
```

```
{
   char buff[15];        datamember *dm;
   static datamember *datamem;
   char yorn;            datamember *dmptr, *temp1;
   int flag = 0;
   datamem = NULL;
   switch(gisbase)
   {
       case 1:
          flag = turnoff(Parent,polymem);
          break;
       case 2:
          flag = turnoff(Parent,linemem);
          break;
       case 3:
          flag = turnoff(Parent, common);
          break;
   }
   if(flag == 0)
     return(NULL);
   flag = 0;

   flag = printdatamem(gisbase);

   if(!flag)
   {
   disp_move(16,13);
   disp_printf("Do you want include these datamembers? \n");
   disp_printf("\t\tA:all  N:none: Y:yes      : ");
   fflush(stdin);
   yorn = getchar();
   if ((yorn == 'A')||(yorn =='a'))
      switch(gisbase)
      {
         case 1:
            return(polymem);
            break;
         case 2:
            return(linemem);
            break;
         case 3:
            return(common);
            break;
       }
   }
   flag = 0;
   if ((yorn == 'N')||(yorn=='n'))
      return(NULL);
      while(1)
      {
        disp_move(17,13);
        dm = new datamember;
```

```
        disp_printf("Which data members you want to include
in construction:\n");
        disp_move(18,13);
        disp_puts("
  ");
        disp_move(18,13);
        z_sgr(WHITE,BLUE,DISP_INTENSITY);
        disp_puts("                    ");
        disp_move(18,13);
        fflush(stdin);
        disp_flush();
        gets(buff);
        switch (gisbase)
        {
          case 1:
             flag = checkdatamem(buff, 1);
             break;
          case 2:
             flag = checkdatamem(buff, 2);
             break;
          case 3:
             flag = checkdatamem(buff, 3);
             break;
        }
        z_sgr(WHITE,CYAN, DISP_INTENSITY);

       if (flag)
       {
         strcpy(dm->name, buff);
         dm->next = NULL;
         if(!datamem)
         datamem = dm;
         else
         {
              dmptr = datamem;
            while(dmptr->next)
            {
               dmptr = dmptr->next;
            }
            dmptr->next = dm;
         }
       }
       else
         disp_printf("                -%s- Not a datamember.
", buff);

       disp_move(19,13);
       disp_printf("Do you need to include another one?
(Y/N) ");
       disp_flush();
       z_sgr(WHITE,BLUE,DISP_INTENSITY);
       yorn = getchar();
```

```
        z_sgr(WHITE,CYAN, DISP_INTENSITY);

        if ((yorn == 'n')||(yorn =='N'))
      break;
      }
      return(datamem);
}

// This function builds row's family tree.

void build_row()
{
  unsigned short buff[16*58];
  int ans;
  char name[31], parent[31];
  CLASSTYPE *present;
  while(1)
  {
     z_sgr(WHITE,CYAN,DISP_INTENSITY);
     disp_move(4,13);
     disp_printf("Enter ROW name ");
     z_sgr(WHITE,BLUE,DISP_INTENSITY);
     disp_move(4,30);
     disp_puts("                    ");
     disp_move(4,30);
     disp_flush();
     gets(name);
     if(strlen(name)==0)
     {
        disp_move(5, 13);
        disp_printf(" Column or Row name cannot be zero
length. \n");
     }
     else if(!Find(name,RTop))
        break;
  }
  SPACES = 0;
  z_sgr(WHITE,CYAN,DISP_INTENSITY);
  disp_move(6,13);

  if (RTop->Child)
     {
     z_sgr(WHITE, RED, DISP_INTENSITY);
     disp_peekbox(buff, 5, 11, 20, 68);
     disp_box(0,z_sgr(WHITE,RED,DISP_INTENSITY),5,11,20,68);
     z_clearbox(6,12,20,67);
     disp_move(6,12);
     disp_printf("Here is the ROW family tree:");
     P_ROW = 7;
     Print_Tree(RTop, 14);
     while(1)
     {
```

```
            disp_move(19,13);
            disp_printf("Enter this row's parent ");
            z_sgr(WHITE,BLUE,DISP_INTENSITY);
            disp_printf("                          ");
            disp_move(19,37);
            disp_flush();
            gets(parent);
            if(Find(parent,RTop))
               break;
            else
            {
               disp_move (20,13);
               disp_printf(" Parent doesn't exit. \n");
            }
         }
         disp_pokebox(buff, 5,11, 20, 68);
         z_sgr(WHITE,CYAN, DISP_INTENSITY);
         }
      else
         {disp_printf("This is the first row to be defined");
         strcpy(parent,"ROW");
         }
      disp_move(8,13);
      present = Find(parent,RTop); /*at this point present
   points to parent*/
      if (present->Gis_basis == 1)
         {disp_printf("This class must be generated by polys");
         ans = 1;
         }
      else if (present->Gis_basis == 2)
         {
         disp_printf("This class must be generated by lines");
         ans = 2;
         }
      else
         {
         disp_move(9,13);
         disp_printf("Is this class generated by");
         disp_move(10,20);
         disp_printf("1) polys");
         disp_move(11,20);
         disp_printf("2) lines");
         disp_move(12,20);
         disp_printf("3) both");
         disp_move(13,13);
         disp_printf("Enter the appropriate number ");
         disp_move(13,42);
         disp_flush();
         scanf("%d",&ans);
         }

      present = new CLASSTYPE(name, parent, 'R', ans);
```

```
}

// This function builds column's family tree.

void build_col()
{
  unsigned short buff[16*58];
  int ans;
  char name[25], parent[25];
  CLASSTYPE *present;

  while(1)
  {
     z_sgr(WHITE,CYAN,DISP_INTENSITY);
     disp_move(4,13);
     disp_printf("Enter COLUMN name ");
     z_sgr(WHITE,BLUE,DISP_INTENSITY);
     disp_move(4,33);
     //disp_flush();
     disp_puts("                         ");
     disp_move(4,33);
     disp_flush();
     gets(name);
     if(strlen(name)==0)
     {
        disp_move(5,13);
        disp_printf(" Column name cannot be zero length.
\n");
     }
     else if(!Find(name,CTop))
        break;
  }
  z_sgr(WHITE,CYAN,DISP_INTENSITY);
  disp_move(6,13);
  SPACES = 0;

  if (CTop->Child)
     {
     z_sgr(WHITE, RED, DISP_INTENSITY);
     disp_peekbox(buff, 5, 11, 20, 68);
     disp_box(0,z_sgr(WHITE,RED,DISP_INTENSITY),5,11,20,68);
     z_clearbox(6,12,20,67);
     disp_move(6,12);
     disp_printf("Here is the COLUMN family tree:");
     P_ROW = 7;
     Print_Tree(CTop, 14);
     while(1)
     {
        disp_move(19,13);
        disp_printf("Enter this column's parent ");
        z_sgr(WHITE,BLUE,DISP_INTENSITY);
        disp_printf("                         ");
```

```
            disp_move(19,40);
            disp_flush();
            gets(parent);
            if(Find(parent,CTop))
               break;
            else
            {
               disp_move(20,13);
               disp_printf(" Parent doesn't exist. \n");
            }
         }
         disp_pokebox(buff, 5,11, 20, 68);
         z_sgr(WHITE,CYAN, DISP_INTENSITY);
         }
      else
         {
         disp_printf("This is the first column to be defined");
         strcpy(parent,"COLUMN");
         }

      present = Find(parent,CTop); /*at this point present
   points to parent*/
      disp_move(8,13);
      if (present->Gis_basis == 1)
         {disp_printf("This class must be generated by polys");
         ans = 1;
         }
      else if (present->Gis_basis == 2)
         {
         disp_printf("This class must be generated by lines");
         ans = 2;
         }
      else
         {
         disp_move(9,13);
         disp_printf("Is this class generated by");
         disp_move(10,20);
         disp_printf("1) polys");
         disp_move(11,20);
         disp_printf("2) lines");
         disp_move(12,20);
         disp_printf("3) both");
         disp_move(13,13);
         disp_printf("Enter the appropriate number ");
         disp_move(13,42);
         disp_flush();
         scanf("%d",&ans);
         }
      present = new CLASSTYPE(name, parent, 'C', ans);

   }
```

```
// This function writes the definition of a column, the
// prototype and comments for Action functions and
// constructor functions.

void write_column_header(CLASSTYPE *p)
{
   datamember *temp;

   if (p == NULL)
     return;
   fprintf(out,"\nclass %s",p->Name);
   if (p->Parent)
      fprintf(out,":public %s",p->Parent->Name);
   fprintf(out,"{\n   ");
   if (!p->Parent)
      {
      fprintf(out,"char Name[9];\n   ");
      fprintf(out,"float Cost;\n   ");
      fprintf(out,"int Seqnum;\n   ");
      fprintf(out,"COLUMN    *Next;\n   ");
      }
   classini(p);
   fprintf(out,"virtual float Action(class ROW *rw);\n   ");
   switch(p->Gis_basis)
     {
     case 1:
     fprintf(out,"%s(poly_or_point *q, int j);\n
",p->Name);
     break;
     case 2:
     fprintf(out,"%s(line *q, int j);\n   ",p->Name);
     break;
     case 3:
     fprintf(out,"%s(poly_or_point *q, int j);\n
",p->Name);
     fprintf(out,"%s(line *q, int j);\n   ",p->Name);
     }
   fprintf(out,"virtual ~%s(){};\n",p->Name);
   fprintf(out,"};\n\n\n");
   if (p != CTop)
      {
      switch(p->Gis_basis)
     {
     case 1:
        if (p->Parent)
           {
           if (p->Parent->Gis_basis == 2)
           {printf("\nERROR -the parent %s and child %s are
generated by different GIS types",
           p->Parent->Name, p->Name);
           fprintf(out,"\n/*
*****************************************\n");
```

```
            fprintf(out,"\nERROR -the parent %s and child %s
are generated by different GIS types\n",
            p->Parent->Name, p->Name);
            fprintf(out,"\n
****************************************** */\n");
            break;
            }
            fprintf(out,"%s::%s(poly_or_point *q, int
j):%s(poly_or_point *q, int j)\n",
            p->Name, p->Name, p->Parent->Name);
            }
         else
            fprintf(out,"%s::%s(poly_or_point *q, int j)\n",
            p->Name, p->Name);
         break;
      case 2:
         if (p->Parent)
            {
            if (p->Parent->Gis_basis == 1)
            {printf("\nERROR -the parent %s and child %s are
generated by different GIS types",
            p->Parent->Name, p->Name);
            fprintf(out,"\n/*
*****************************************\n");
            fprintf(out,"\nERROR -the parent %s and child %s
are generated by different GIS types\n",
            p->Parent->Name, p->Name);
            fprintf(out,"\n
****************************************** */\n");
            break;
            }
            fprintf(out,"%s::%s(line *q, int j):%s(line *q,
intj)\n",
            p->Name,p->Name,p->Parent->Name);
            }
         else
            fprintf(out,"%s::%s(line *q, int j)\n",p->Name,
p->Name);
         break;
      case 3:
         if (p->Parent)
            {
            if (p->Parent->Gis_basis != 3)
           printf("\n\nERROR - the parent and child are
generated by different GIS types");
            fprintf(out,"%s::%s(poly_or_point *q, int
j):%s(poly_or_point *q, int j)\n",
            p->Name, p->Name, p->Parent->Name);
               write_construct_comment(p);
            fprintf(out,"%s::%s(line *q, int j):%s(line *q,
intj)\n",
            p->Name,p->Name,p->Parent->Name);
```

```
            }
         else
            {
            fprintf(out,"%s::%s(poly_or_point *q, int j)\n{\n
 ",p->Name,p->Name);
               write_construct_comment(p);
            fprintf(out,"%s::%s(line *q, int j)\n{\n
",p->Name,p->Name);
            }
          break;
      }
         write_construct_comment(p);
      }
   else /*at CTop*/
      {
      fprintf(out,"%s::%s(poly_or_point *q, int j)\n{\n
",p->Name,p->Name);
      fprintf(out,"   Next = NULL;\n    sprintf(Name,\"C");
      fprintf(out,"%c", PERCENTAGE);
      fprintf(out,"07d\",j);\n   SeqNum = j;\n   ");
      fprintf(out,"/* you need to assign Cost here or in
child*/\n");
      fprintf(out,"}\n\n");
      fprintf(out,"%s::%s(line *q, int j)\n{\n
",p->Name,p->Name);
      fprintf(out,"   Next = NULL;\n    sprintf(Name,\"C");
      fprintf(out,"%c", PERCENTAGE);
      fprintf(out,"07d\",j);\n   SeqNum = j;\n   ");
      fprintf(out,"/* you need to assign Cost here or in
child*/\n");
      fprintf(out,"}\n\n");
      }

   fprintf(out,"float %s::Action(ROW *rw)\n{\n   ",p->Name);
   fprintf(out,"return(rw->Row_Action(this));\n}\n\n");

   if (p->Child)
     write_column_header(p->Child);
   if (p->Next)
     write_column_header(p->Next);
}

// This function lists data members of a column
// in a constructor function.

void write_construct_comment(CLASSTYPE *p)
{
     datamember *temp;

     fprintf(out,"{/* constructor function goes here. */\n
 ");
     temp = p->datamem;
```

```
     while(temp)
       {
          fprintf(out, "  %s=p->%s;\n     ",
temp->name,temp->name);
          temp = temp->next;
       }
    fprintf(out,"\n}\n\n");

}

// This function writes the definition of a row, the
// prototype and comments for Row_Action functions and
// constructor functions.

void write_row_header(CLASSTYPE *p)
{

   char  name[31];


   fprintf(out,"\nclass %s",p->Name);
   if (p->Parent)
      fprintf(out,":public %s",p->Parent->Name);
   fprintf(out,"{\n   ");
   if (!p->Parent)
      {
      fprintf(out,"char Name[9];\n   ");
      fprintf(out,"float RHS;\n   ");
      fprintf(out,"char  Direction;\n   ");
      fprintf(out,"ROW  *Next;\n   ");
      write_virtual_header(CTop);
      }
   classini(p);
   switch(p->Gis_basis)
     {
     case 1:
     fprintf(out,"%s(poly_or_point *q, int j);\n
",p->Name);
     break;
     case 2:
     fprintf(out,"%s(line *q, int j);\n   ",p->Name);
     break;
     case 3:
     fprintf(out,"%s(poly_or_point *q, int j);\n
",p->Name);
     fprintf(out,"%s(line *q, int j);\n   ",p->Name);
     }
   if(strcmp(p->Name,"OBJ") == 0)
      {if (CTop->Child)
      write_virtual_header(CTop->Child);
      }
   else
```

```
     {if (p != RTop)
     {
     while(1){

disp_box(0,z_sgr(WHITE,MAGENTA,DISP_INTENSITY),2,10,21,69);
        z_sgr(WHITE,MAGENTA,DISP_INTENSITY);
        z_clearbox(3,11,21,68);
        z_ctr80(2,"DEFINING INTERACTIONS");
        disp_move(4,13);
        disp_printf("Processing ROW %s\n", p->Name);
        disp_move(6,13);
        disp_printf("Here is the family tree of COLUMN\n");
        P_ROW = 9;
        SPACES = 0;
        Print_Tree(CTop->Child, 15);
        disp_move(19,13);
        disp_printf("Enter the name of a the column class
with which this");
        disp_move(20,13);
        disp_printf("row interacts or Q to quit");
        disp_move(20,40);
        z_sgr(WHITE,BLUE,DISP_INTENSITY);
        disp_puts("                    ");
        disp_move(20,40);
        disp_flush();
        disp_move(20,40);
        disp_flush();
           gets(name);
        disp_move(20,40);
        disp_puts(name);
        disp_flush();
        if (strcmp(name,"Q") == 0)
           break;
        fprintf(out,"virtual float Row_Action(%s *cl);\n
",name);
           write_reaction_comment(p, name);
        }
     }
     }

   fprintf(out,"virtual ~%s(){};\n",p->Name);
   fprintf(out,"};\n\n\n");

   if (p != RTop)
      {
      switch(p->Gis_basis)
     {
     case 1:
        if (p->Parent)
           {
           if (p->Parent->Gis_basis == 2)
           {printf("\nERROR -the parent %s and child %s are
```

```
generated by different GIS types",
            p->Parent->Name, p->Name);
            fprintf(out,"\n/*
**********************************************\n");
            fprintf(out,"\nERROR -the parent %s and child %s
are generated by different GIS types\n",
            p->Parent->Name, p->Name);
            fprintf(out,"\n
********************************************** */\n");
            break;
            }
            fprintf(out,"%s::%s(poly_or_point *q, int
j):%s(poly_or_point *q, int j)\n",
            p->Name, p->Name, p->Parent->Name);
            }
         else
            fprintf(out,"%s::%s(poly_or_point *q, int j)\n",
            p->Name, p->Name);
         break;
      case 2:
         if (p->Parent)
            {
            if (p->Parent->Gis_basis == 1)
            {printf("\nERROR -the parent %s and child %s are
generated by different GIS types",
            p->Parent->Name, p->Name);
            fprintf(out,"\n/*
**********************************************\n");
            fprintf(out,"\nERROR -the parent %s and child %s
are generated by different GIS types\n",
            p->Parent->Name, p->Name);
            fprintf(out,"\n
********************************************** */\n");
            break;
            }
            fprintf(out,"%s::%s(line *q, int j):%s(line *q,
intj)\n",
            p->Name,p->Name,p->Parent->Name);
            }
         else
            fprintf(out,"%s::%s(line *q, int j)\n",p->Name,
p->Name);
         break;
      case 3:
         if (p->Parent)
            {
            if (p->Parent->Gis_basis != 3)
           printf("\n\nERROR - the parent and child are
generated by different GIS types");
            fprintf(out,"%s::%s(poly_or_point *q, int
j):%s(poly_or_point *q, int j)\n",
            p->Name, p->Name, p->Parent->Name);
```

```
                write_construct_comment(p);
              fprintf(out,"%s::%s(line *q, int j):%s(line *q,
intj)\n",
              p->Name,p->Name,p->Parent->Name);
              }
           else
              {
              fprintf(out,"%s::%s(poly_or_point *q, int j)\n{\n
 ",p->Name,p->Name);
                write_construct_comment(p);
              fprintf(out,"%s::%s(line *q, int j)\n{\n
",p->Name,p->Name);
              }
              break;
        }
           if(strcmp(p->Name, "OBJ")!=0)
              write_construct_comment(p);
         }
      else /*at RTop*/
         {
         fprintf(out,"%s::%s(poly_or_point *q, int j)\n{\n
",p->Name,p->Name);
         fprintf(out,"   Next = NULL;\n    sprintf(Name,\"R");
         fprintf(out,"%c",PERCENTAGE);
         fprintf(out,"07d\",j);\n   SeqNum = j;\n   ");
         fprintf(out,"/* you need to assign Direction and RHS
or in child*/\n");
         fprintf(out,"}\n\n");
         fprintf(out,"%s::%s(line *q, int j)\n{\n
",p->Name,p->Name);
         fprintf(out,"   Next = NULL;\n    sprintf(Name,\"R");
         fprintf(out,"%c",PERCENTAGE);
         fprintf(out,"07d\",j);\n   SeqNum = j;\n   ");
         fprintf(out,"/* you need to assign Direction and RHS
or in child*/\n");
         fprintf(out,"}\n\n");

         }
      if (p->Child)
        write_row_header(p->Child);
      if (p->Next)
        write_row_header(p->Next);

}

// This function writes datamembers of an object in its
// definition.

void classini(CLASSTYPE *p)
{
     datamember *temp;
```

```
      temp = p->datamem;
      while(temp != NULL)
      {

if((strcmp(temp->name,"Area")==0)||(strcmp(temp->name,
"Perimeter")==0))
              fprintf(out, "double %s;\n   ",temp->name);
          else
              fprintf(out, "long  %s;\n   ", temp->name);
          temp = temp->next;
      }
  }

// This function lists data members of a column
// in a Row_Action function.

void write_reaction_comment(CLASSTYPE *p, char *name)
{
    datamember *temp;   CLASSTYPE  *tempclass;
    fprintf(tmp, "float %s::Row_Action(%s *cl)\n", p->Name,
name);
    fprintf(tmp, "{\n   /*Here are the %s's datamembers : \n
                ", p->Name);
    tempclass = p;
    while(tempclass)
    {
       temp = tempclass->datamem;
       while(temp)
       {
          fprintf(tmp, "  %s", temp->name);
          temp = temp->next;
       }
       tempclass = tempclass->Parent;
    }
    if((tempclass = Find(name, CTop))==NULL)
    {  printf(" %s is not a Column class\n", name);
       exit(1);
    }
    fprintf(tmp, "\n     Here are the %s's datamembers : \n
              ", tempclass->Name);
    while(tempclass)
    {
       temp = tempclass->datamem;
       while(temp)
       {
          fprintf(tmp, "  %s", temp->name);
          temp = temp->next;
       }
       tempclass = tempclass->Parent;
    }
    fprintf(tmp, "*/\n     return(0.0); }\n\n");
}
```

```
// This function writes virtual Row_Action functions in
// a class definition.

void write_virtual_header(CLASSTYPE *p)
{

   fprintf(out,"virtual float Row_Action(%s *cl) {
return(0.0);}\n   ",p->Name);
   if(p->Child)
      write_virtual_header(p->Child);
   if (p->Next)
      write_virtual_header(p->Next);
}

// This funcntion writes prototypes of classify_line() and
// classify_poly() functions.

void write_classify()
{
   /*write points case*/
   fprintf(out2,"\n\nvoid classify_poly(point_or_poly *p,
int j)\n");
   fprintf(out2,"{\n");
   fprintf(out2,"/*you may wish to preceed each \"new\"
statement with a condition*/\n");
   write_classify_detail('C', CTop->Child, 1);
   write_classify_detail('R', RTop->Child, 1);
   fprintf(out2,"}\n");
   /*write lines case*/
   fprintf(out2,"\n\nvoid classify_line(line *p, int j)\n");
   fprintf(out2,"{\n");
   fprintf(out2,"/*you may wish to preceed each \"new\"
statement with a condition*/\n");
   write_classify_detail('C', CTop->Child, 2);

   write_classify_detail('R', RTop->Child, 2);
   fprintf(out2,"}\n");
}

// This function writes the contents of classify functions.

void write_classify_detail(char type, CLASSTYPE *p, int
gistype)
{
   if (p == NULL)
      return;
   if ((p->Gis_basis == gistype) || (p->Gis_basis == 3))
      {if (type == 'C')
      {
      fprintf(out2,"\n   if (CTop == NULL)\n");
      fprintf(out2,"       CTop = CBottom = new %s(p,
```

```
j);\n", p->Name);
      fprintf(out2,"   else\n");
      fprintf(out2,"       CBottom = CBottom->Next = new
%s(p, j);\n", p->Name);
      }
      if (type == 'R')
      fprintf(out2,"\n   RBottom = RBottom->Next = new %s(p,
j);\n", p->Name);
      }
   if (p->Child)
      write_classify_detail(type, p->Child, gistype);
   if (p->Next)
      write_classify_detail(type, p->Next, gistype);
}

// This function writes the core code.

void write_core()
{
  FILE *in;
  char buffer[90];

  fprintf(out2,"#include <stdio.h>\n");
  fprintf(out2,"#include <stdlib.h>\n");
  fprintf(out2,"#include \"classes.h\"\n");
  fprintf(out2,"/* if opening coverage files (aat and pat)
with an external library\n");
  fprintf(out2,"   such as Codebase, you need to add the
proper include files here*/\n");

  fprintf(out2,"\nROW     *RTop, *RBottom;\n");
  fprintf(out2,"COLUMN  *CTop, *CBottom;\n");

  fprintf(out2,"\n/*Function prototypes*/\n");
  fprintf(out2,"void main(void);\n");
  fprintf(out2,"void classify_poly(point_or_poly *,
int);\n");
  fprintf(out2,"void classify_line(line *, int);\n");
  fprintf(out2,"\nvoid main()\n{\n");
  fprintf(out2,"   ROW        *Rows;\n");
  fprintf(out2,"   COLUMN     *Cols;\n");
  fprintf(out2,"   int        j;\n");
  fprintf(out2,"/* Depending on the method used for opening
the aat and pat files\n");
  fprintf(out2,"   you will need to enter the proper file
handlers here*/\n");
  fprintf(out2,"\n   CTop = CBottom = NULL;\n");
  fprintf(out2,"   RTop = new OBJ();\n");
  fprintf(out2,"   RBottom = RTop;\n");
  fprintf(out2,"   j = 0;\n");

  fprintf(out2,"\n it is the user's responsibility to
```

```
correctly open the AAT and PAT files*/\n");
  fprintf(out2,"\n /* need PAT file reading loop here");
  fprintf(out2,"\n a Codebase example would be as follows
*/");
  fprintf(out2,"\n for (PAT.top(); !PAT.eof();
PAT.skip())\n");
  fprintf(out2,"\n{classify_poly(p, int j);\n");
  fprintf(out2,"\n   j++;\n}\n");


  fprintf(out2,"\n /* need AAT file reading loop here\n");
  fprintf(out2,"\n a Codebase example would be as
follows*/");
  fprintf(out2,"\n for (AAT.top(); !AAT.eof();
AAT.skip())\n");
  fprintf(out2,"\n{  classify_line(p, int j)\n");
  fprintf(out2,"\n   j++;\n}\n");
  fprintf(out2,"\n\n");
  in = fopen("core.btx","rb");
  while (fread(buffer,sizeof(buffer),1,in))
    {fprintf(out2,"%s",buffer);
    }
  fclose(in);
  fprintf(out2,"}\n");
}

// This function prints family tree of row or column.

void Print_Tree(CLASSTYPE *present, int col)
{
  int i;

  if (present == NULL)
    return;
  disp_move(P_ROW++, col);
  for(i = 0; i < SPACES-3; i++)
    disp_printf(" ");
  if (SPACES >= 3)
    {disp_printf("└");
    for (i=SPACES-2; i < SPACES; i++)
       disp_printf("─");
    }
  disp_printf("%s",present->Name);

  if(present->Child)
     {SPACES += 3;
     Print_Tree(present->Child, col);
     }
  if(present->Next)
     Print_Tree(present->Next, col);
  else
     SPACES -= 3;
```

```
}

// This function checks if there are duplications of row
// and column names.

int checkduplicate(CLASSTYPE *value, char * name)
{
   int  rval;
   if(strcmp(name, value->Name) == 0)
   {
      disp_move(5,13);
      disp_printf(" Duplicated Column or Row Name: %s
    \n", name);
      return(1);
   }
   if (value->Child)
   {
      rval = checkduplicate(value->Child, name);
      return(rval);
   }
   if( value ->Next)
   {
      rval = checkduplicate(value->Next, name);
      return(rval);
   }
   return(0);
}

// This function turns on the flag indicating the
// avaliability of a datamember.

void turnon(datamember *dm)
{
   datamember *temp;

   temp = dm;
   while(temp)
   {
     temp->on = 1;
     temp = temp->next;
   }
}

// This function checks the avaliability of a datamember.


int checkon(datamember *dm)
{
   datamember *temp;

   temp = dm;
   while(temp)
```

```
    {
       if(temp->on != 0)
         return(1);
       temp = temp->next;
    }
    return(0);
}

// This function turns off the flags indicating the
// avaliability of attributes.

int turnoff(CLASSTYPE *p, datamember *plc)
{
  CLASSTYPE *tempclass; datamember *dm, *temp;

  tempclass = p;
  while(tempclass)
  {
     dm = tempclass->datamem;
     while(dm)
     {
        temp = plc;
        while (temp)
        {
           if(strcmp(temp->name, dm->name)==0)
           {
              temp->on = 0;
           }
           temp = temp->next;
        }
        dm = dm->next;
     }
     tempclass = tempclass->Parent;
  }
  return(checkon(plc));
}
```

APPENDIX A2

THE OOP GENERATOR'S HEADER FILE screens.h

```
#include <disp.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>


#define BLACK   0
#define BLUE    1
#define GREEN   2
#define CYAN    3
#define RED     4
#define MAGENTA 5
#define BROWN   6
#define WHITE   7


void z_cls(void);
int z_sgr(int, int, int);
void z_ctr80(int, char []);
void z_place(int, int, char[]);
void z_clearbox(int, int, int, int);

void z_cls()
{
   disp_move(0,0);
   disp_eeop();
}
int z_sgr(int foreground, int background, int attribute)
{
   int color;
   color = background*16+foreground;
   disp_setattr(color|attribute);
   return(color|attribute);
}
void z_ctr80(int row, char text[])
{
   int   length, col;

   length = strlen(text);
   col    = (80 - length) / 2;
   disp_move(row, col);
   disp_puts(text);
}
void z_place(int row, int col, char text[])
{
   disp_move(row, col);
   disp_puts(text);
```

```
}

void z_clearbox(int lrow, int lcol, int rrow, int rcol)
{
   int i, j;
   char s[23][80];
   for (i = 0; i < rrow-lrow; i++)
      {sprintf(s[i]," ");
      for (j = 0; j < rcol-lcol; j++)
          strcat(s[i]," ");
      strcat(s[i],"\0");
      }
  for (i=0; i < rrow-lrow; i++)
      {disp_move(lrow+i,lcol);
      disp_puts(s[i]);
      }
}
```

APPENDIX B

THE OOP PROGRAM FOR EXAMPLE III

APPENDIX B1

classify.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include "classes.h"
/* if opening coverage files (aat and pat) with an external
library such as Codebase, you need to add the proper include
files here*/

ROW     *RTop, *RBottom;
COLUMN  *CTop, *CBottom;

/*Function prototypes*/
void main(void);
void classify_poly(point_or_poly *, int);
void classify_line(line *, int);

void main()
{
   ROW         *Rows;
   COLUMN      *Cols;
   int         j;
/* Depending on the method used for opening the aat and pat
files you will need to enter the proper file handlers here*/

   CTop = CBottom = NULL;
   RTop = new OBJ();
   RBottom = RTop;
   j = 0;

/* it is the user's responsibility to correctly open the AAT
and PAT files*/

 /* need PAT file reading loop here

 a Codebase example would be as follows */
 for (PAT.top(); !PAT.eof(); PAT.skip())

 {  classify_poly(p, int j);

   j++;
 }

 /* need AAT file reading loop here

 a Codebase example would be as follows */
 for (AAT.top(); !AAT.eof(); AAT.skip())

 {   classify_line(p, int j)
```

```
    j++;
  }


      Rows = RTop->Next;

      while (Rows)
      {
      fprintf(out," %c %-8s\n", Rows->Direction, Rows->Name);
      Rows = Rows->Next;
      }
      Cols = CTop;
      while (Cols)
      {
      Rows = RTop;
      while (Rows)
         {
         value = Cols->Action(Rows);
         if (value != 0.0)
            {
            fprintf(out,"    %-8s  %-8s  %12.3f\n",
           Cols->Name, Rows->Name, value);
            }
         Rows = Rows->Next;
         }
      Cols = Cols->Next;
      }
      Rows = RTop;
      while (Rows)
      {if (Rows->Rhs != 0.0)
         {fprintf(out,"           RHS                    %-8s
%12.3f\n",Rows->Name,Rows->Rhs);
         }

         Rows = Rows->Next;
      }
}


void classify_poly(point_or_poly *p, int j)
{
/*you may wish to preceed each "new" statement with a
condition*/

   RBottom = RBottom->Next = new Demand(p, j);
}


void classify_line(line *p, int j)
{
/*you may wish to preceed each "new" statement with a
condition*/
```

```
    if (CTop == NULL)
        CTop = CBottom = new Arc(p, j);
    else
        CBottom = CBottom->Next = new Arc(p, j);

    if (CTop == NULL)
        CTop = CBottom = new Cap_Arc(p, j);
    else
        CBottom = CBottom->Next = new Cap_Arc(p, j);

    RBottom = RBottom->Next = new Arc_Cap(p, j);
}
```

APPENDIX B2

classes.h

```
class COLUMN{
   char Name[9];
   float Cost;
   int Seqnum;
   COLUMN    *Next;
   virtual float Action(class ROW *rw);
   COLUMN(poly_or_point *q, int j);
   COLUMN(line *q, int j);
   virtual ~COLUMN(){};
};


COLUMN::COLUMN(poly_or_point *q, int j)
{
      Next = NULL;
    sprintf(Name,"C%07d",j);
   SeqNum = j;
   /* you need to assign Cost here or in child*/
}

COLUMN::COLUMN(line *q, int j)
{
      Next = NULL;
    sprintf(Name,"C%07d",j);
   SeqNum = j;
   /* you need to assign Cost here or in child*/
}

float COLUMN::Action(ROW *rw)
{
   return(rw->Row_Action(this));
}


class Arc:public COLUMN{
   long  Fnode_;
   long  Tnode_;
   virtual float Action(class ROW *rw);
   Arc(line *q, int j);
   virtual ~Arc(){};
};


Arc::Arc(line *q, int j):COLUMN(line *q, intj)
{/* constructor function goes here. */
       Fnode_=p->Fnode_;
       Tnode_=p->Tnode_;
```

```
}

float Arc::Action(ROW *rw)
{
   return(rw->Row_Action(this));
}


class Cap_Arc:public Arc{
   virtual float Action(class ROW *rw);
   Cap_Arc(line *q, int j);
   virtual ~Cap_Arc(){};
};


Cap_Arc::Cap_Arc(line *q, int j):Arc(line *q, intj)
{/* constructor function goes here. */

}

float Cap_Arc::Action(ROW *rw)
{
   return(rw->Row_Action(this));
}


class Transfer_Arc:public COLUMN{
   long  Cover_;
   virtual float Action(class ROW *rw);
   Transfer_Arc(poly_or_point *q, int j);
   virtual ~Transfer_Arc(){};
};


Transfer_Arc::Transfer_Arc(poly_or_point    *q,    int
j):COLUMN(poly_or_point *q, int j)
{/* constructor function goes here. */
       Cover_=p->Cover_;

}

float Transfer_Arc::Action(ROW *rw)
{
   return(rw->Row_Action(this));
}


class ROW{
   char Name[9];
   float RHS;
   char  Direction;
```

```
    ROW  *Next;
    virtual float Row_Action(COLUMN *cl) { return(0.0);}
    virtual float Row_Action(Arc *cl) { return(0.0);}
    virtual float Row_Action(Cap_Arc *cl) { return(0.0);}
    virtual float Row_Action(Transfer_Arc *cl) { return(0.0);}
    ROW(poly_or_point *q, int j);
    ROW(line *q, int j);
    virtual ~ROW(){};
};


ROW::ROW(poly_or_point *q, int j)
{
       Next = NULL;
     sprintf(Name,"R%07d",j);
    SeqNum = j;
    /* you need to assign Direction and RHS or in child*/
}

ROW::ROW(line *q, int j)
{
       Next = NULL;
     sprintf(Name,"R%07d",j);
    SeqNum = j;
    /* you need to assign Direction and RHS or in child*/
}


class OBJ:public ROW{
    virtual float Row_Action(Arc *cl) { return(0.0);}
    virtual float Row_Action(Cap_Arc *cl) { return(0.0);}
    virtual float Row_Action(Transfer_Arc *cl) { return(0.0);}
    virtual ~OBJ(){};
};



class Demand:public ROW{
    long  Cover_;
    Demand(poly_or_point *q, int j);
    virtual float Row_Action(Arc *cl);
    virtual float Row_Action(Cap_Arc *cl);
    virtual float Row_Action(Transfer_Arc *cl);
    virtual ~Demand(){};
};


Demand::Demand(poly_or_point *q, int j):ROW(poly_or_point *q,
int j)
{/* constructor function goes here. */
        Cover_=p->Cover_;
```

```
}


class Arc_Cap:public ROW{
   Arc_Cap(poly_or_point *q, int j);
   virtual float Row_Action(Cap_Arc *cl);
   virtual ~Arc_Cap(){};
};


Arc_Cap::Arc_Cap(poly_or_point *q, int j):ROW(poly_or_point
*q, int j)
{/* constructor function goes here. */

}


class Demand_by_Mode:public ROW{
   long  Cover_;
   Demand_by_Mode(poly_or_point *q, int j);
   virtual float Row_Action(Arc *cl);
   virtual float Row_Action(Cap_Arc *cl);
   virtual float Row_Action(Transfer_Arc *cl);
   virtual ~Demand_by_Mode(){};
};


Demand_by_Mode::Demand_by_Mode(poly_or_point    *q,    int
j):ROW(poly_or_point *q, int j)
{/* constructor function goes here. */
       Cover_=p->Cover_;

}

float Demand::Row_Action(Arc *cl)
{
   /*Here are the Demand's datamembers :
                     Cover_
     Here are the Arc's datamembers :
                    Fnode_  Tnode_*/
     return(0.0); }

float Demand::Row_Action(Cap_Arc *cl)
{
   /*Here are the Demand's datamembers :
                     Cover_
     Here are the Cap_Arc's datamembers :
                    Fnode_  Tnode_*/
     return(0.0); }

float Demand::Row_Action(Transfer_Arc *cl)
{
```

```
   /*Here are the Demand's datamembers :
                        Cover_
     Here are the Transfer_Arc's datamembers :
                      Cover_*/
     return(0.0); }

float Arc_Cap::Row_Action(Cap_Arc *cl)
{
   /*Here are the Arc_Cap's datamembers :

     Here are the Cap_Arc's datamembers :
                    Fnode_  Tnode_*/
     return(0.0); }

float Demand_by_Mode::Row_Action(Arc *cl)
{
   /*Here are the Demand_by_Mode's datamembers :
                        Cover_
     Here are the Arc's datamembers :
                    Fnode_  Tnode_*/
     return(0.0); }

float Demand_by_Mode::Row_Action(Cap_Arc *cl)
{
   /*Here are the Demand_by_Mode's datamembers :
                        Cover_
     Here are the Cap_Arc's datamembers :
                    Fnode_  Tnode_*/
     return(0.0); }

float Demand_by_Mode::Row_Action(Transfer_Arc *cl)
{
   /*Here are the Demand_by_Mode's datamembers :
                        Cover_
     Here are the Transfer_Arc's datamembers :
                      Cover_*/
     return(0.0); }
```

APPENDIX C

THE PROGRAM OF EXAMPLE III NOT GENERATED BY THE OOP GENERATOR

APPENDIX C1

corr.cpp

```
#include <string.h>
#include "newpre1.h"


void main(void);
void Classify_Link(Link *, int);
void Classify_Node(Node *, int);
void main()
{
   FILE *lnks, *ndes;
   Row *Rows;
   Column *Cols;
   Link p;
   Node q;
   int i;
   char temp[10];

   i = 0;
   CTop = CBottom = NULL;
   RTop = RBottom = new Obj();

   lnks = fopen("Links.Dat","r");
   ndes = fopen("Nodes.Dat","r");

   cls  = fopen("cols.mps","w");

   fprintf(cls,"NAME          USAID\n");
   fprintf(cls,"ROWS\n");
   RBottom->Row_ROWS();

   while(fscanf(lnks,"%s %s %f %f %d %s", p.anode, p.bnode,
   &p.cost, &p.capacity, &p.tflag, p.mode) != EOF)
          Classify_Link(&p,i++);

   fclose(lnks);
   i = 0;
   while(fscanf(ndes,"%s %f",q.name, &q.demand) != EOF)
      Classify_Node(&q, i++);

   fprintf(cls,"COLUMNS\n");
   Cols = CTop;
   while (Cols)
         {
      Rows = RTop;
      while (Rows)
          {
          Cols->Action(Rows);
```

```
            Rows = Rows->Next;
          }
        Cols = Cols->Next;
        }

    fprintf(cls,"RHS\n");
    Rows = RTop;
    while(Rows)
       {
         if ((Rows->Type != 'O')&&(Rows->Rhs != 0.0))
       Rows->Row_RHS();
            Rows = Rows->Next;
       }
     fprintf(cls,"ENDDATA\n");
    fclose(cls);
}

void Classify_Link(Link *p, int j)
{

   if (p->capacity < 0.0)
      {
        if (CTop == NULL)
           CTop = CBottom = new Arc(p, j);
        else
           {
          CBottom = CBottom->Next = new Arc(p, j);
               if (p->tflag != 0)
                CBottom = CBottom->Next = new Arc(p, j, 1);
           }
      }
   else      //a capacititated link
      {
         if (CTop == NULL)
           CTop = CBottom = new Cap_Arc(p, j);
         else
           {
           CBottom = CBottom->Next = new Cap_Arc(p, j);
           RBottom = RBottom->Next = new Arc_Cap(p, j);
           RBottom->Row_ROWS();
             if (p->tflag != 0)
                { CBottom = CBottom->Next = new Cap_Arc(p, j,
                  1);
               RBottom = RBottom->Next = new Arc_Cap(p, j,
                  1);
               RBottom->Row_ROWS(); }
           }
     }
}
void Classify_Node(Node *r, int j)
{
   RBottom = RBottom->Next = new Demand(r, j);
```

```
    RBottom->Row_ROWS();
}
```

APPENDIX C2

newpre1.h

```
#include <stream.hpp>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

FILE *cls;

struct Link
{
   char  anode[10], bnode[10];
   float cost, capacity;
   int tflag;
   char mode[5];
};

struct Node
{
   char name[10];
   float demand;
};

void prt(char, int, char, int, int );

class Column{
public:
   float Cost;
   char Type;
   int  SeqNum;
   Column *Next;
   Column(float x, char t, int j);
   virtual void Action(class Row *p);
   ~Column(){};
};

Column::Column(float x, char t, int j)
{
   Cost = x;
   Type = t;
   SeqNum = j;
   Next = NULL;
}

class Arc : public Column{
public:
   char    From[10], To[10];
   Arc(Link *p, int j);
   Arc(Link *p, int j, int t);
   ~Arc(){};
```

```
   virtual void Action(class Row *p);
};


Arc::Arc(Link *p, int j):Column(p->cost, 'A', j)
{
   strcpy(From,p->anode);
   strcpy(To, p->bnode);
}

Arc::Arc(Link *p, int j, int t):Column(p->cost, 'B', j)
{
   strcpy(From,p->bnode);
   strcpy(To, p->anode);
}

class Cap_Arc : public Arc {
public:
   float Capacity;
   Cap_Arc(Link *p, int j):Arc(p, j){Capacity = p->capacity;}
   Cap_Arc(Link *p, int j, int t):Arc(p, j, 1){Capacity =
p->capacity;}
   virtual void Action(class Row *p);
   ~Cap_Arc(){};
};


class Row{
public:
   char    Type;
   int  SeqNum;
   char Direction;
   float        Rhs;
   Row     *Next;
   Row(char t, char d, int s, float x);
   virtual void obj(Column *p){};
   virtual void Row_ROWS()
     {fprintf(cls," %c  %c%03d\n",Direction,Type,SeqNum);}
   virtual void Row_RHS()
     {fprintf(cls,"                DEMAND                %c%03d
%f\n",Type,SeqNum,Rhs);}
   virtual void Row_Action(Column *p){};
   virtual void Row_Action(Arc *p){};
   virtual void Row_Action(Cap_Arc *p){};
   ~Row(){};
};

Row::Row(char t, char d, int s, float x)
{
   Type = t;
   Direction = d;
   SeqNum = s;
```

```
    Rhs = x;
    Next = NULL;
}

class Demand : public Row {
public:
    char  Name[10];
    Demand(Node *p, int j);
    virtual void Row_Action(Arc *p);
    virtual void Row_Action(Cap_Arc *p);
    ~Demand(){};
};

Demand::Demand(Node *p, int j):Row('Z', 'L', j, p->demand)
{
    strcpy(Name,p->name);
}

void Demand::Row_Action(Arc *p)
{
    if (strcmp(Name, p->From) == 0)
        {
            printf("\n Arc %c%d leaves %s",p->Type, p->SeqNum,
Name);
            prt(p->Type, p->SeqNum, Type, SeqNum, -1);
        }
    if (strcmp(Name, p->To) == 0)
        {
            printf("\n this is a row   %c%d        \n", Type,
SeqNum);
            printf("\n Arc %c%d enters %s",p->Type, p->SeqNum,
Name);
            prt(p->Type,p->SeqNum,Type,SeqNum,1);
        }
}
void Demand::Row_Action(Cap_Arc *p)
{
    if (strcmp(Name, p->From) == 0)
        {
            printf("\n Arc %c%d leaves %s",p->Type, p->SeqNum,
Name);
            prt(p->Type,p->SeqNum,Type,SeqNum,-1);
        }
    if (strcmp(Name, p->To) == 0)
        {
            printf("\n Arc %c%d enters %s",p->Type, p->SeqNum,
Name);
       prt(p->Type,p->SeqNum,Type,SeqNum,1);
        }
}

class Obj : public Row{
```

```
public:
   Obj():Row('O','N',0,0){};
   ~Obj(){};
   virtual void Row_Action(Arc *p)
      {
       printf("\nColumn %c%d costs %f",p->Type, p->SeqNum,
p->Cost);
       fprintf(cls,"    %c%03d      %c%03d      %f\n",
          p->Type,p->SeqNum,Type,SeqNum,p->Cost);
      }

   virtual void Row_Action(Cap_Arc *p)
      {
       printf("\nColumn %c%d costs %f",p->Type, p->SeqNum,
p->Cost);
       fprintf(cls,"    %c%03d      %c%03d      %f\n",
              p->Type,p->SeqNum,Type, SeqNum,p->Cost);
      }
};
class Arc_Cap : public Row {
public:
   Arc_Cap(Link *p, int j): Row('Y','L',j,p->capacity){};
     Arc_Cap(Link *p, int j, int
t):Row('t','L',j,p->capacity){};
   virtual void Row_Action(Cap_Arc *p);
   ~Arc_Cap();
};

void Arc_Cap::Row_Action(Cap_Arc *p)
{
   if (((SeqNum == p->SeqNum)&&(Type == 't')&&(p->Type ==
'B'))||
       ((SeqNum == p->SeqNum)&&(Type == 'Y')&&(p->Type ==
'A')))
      { printf("\nArc %c%d has a capacity of %f", p->Type,
p->SeqNum, Rhs);
        prt(p->Type,p->SeqNum, Type, SeqNum, 1);
      }
}
void Column::Action(Row *q)
{
   q->Row_Action(this);
}
void Arc::Action(Row *q)
{
   q->Row_Action(this);
}
void Cap_Arc::Action(Row *q)
{
   q->Row_Action(this);
}
```

```
void prt(char ptype, int pseqnum, char qtype, int qseqnum, int
one)
{
     fprintf(cls,"    %c%03d       %c%03d       %d\n",
          ptype,pseqnum,qtype,qseqnum,one);
}

Column *CTop, *CBottom;
Row *RTop, *RBottom;
```

VITA

Mei Zhang was born in Lanzhou, P. R. China on May 26th, 1969. She attended elementary schools in the Chengguan (Lanzhou) District, and graduated from #33 Middle High School in May, 1987. The following September she entered University of Tennessee. In May, 1991 received the degree of Bachelor of Science in Mathematics, and received the degree of Master of Science in Geography in 1993.