5 OBJECT-ORIENTED ANALYSIS

This chapter presents how to implement the algorithms developed in Chapter 4 as a computer code. The discussion presented here focuses on the general philosophies of software design and the principles of programming, rather than technical detail.

5.1 Principles of the Design

The development of a software system generally consists of several processes or phases: requirement analysis, preliminary and detailed designs, coding, unit testing, integration and system testing, etc. It is these processes or phases that govern the life cycle of a software system. The principal philosophy for developing a software is to view the system as a changing process, instead of being static.

For the problem considered in this research, the primary goal is to develop a model that synthesizes the O-D trip tables from the link volumes. Since the mathematical models and algorithms have been determined, the principal goal of computer coding is to implement the developed algorithm efficiently and effectively, focusing on whether the resulting software system can operate within the set of available time resources and space resources in an acceptable manner. On the other hand, the problem considered in this research arises from real-world transportation phenomena, and it possesses a high degree of complexity. The model can hence be evaluated only via actual applications. Therefore, the developed model and resulting software should expect to be changed according to the system life cycle, and other general software engineering requirements, such as modifiability, reliability, as well as understandability.

In order to achieve these goals, commonly used software engineering principles have to be applied, including data abstraction, information hiding, modularity and localization, uniformity, completeness and conformability. These principles are used throughout the entire program design and coding process, leading to a product that is modifiable, efficient, reliable and understandable.

5.2 Choice of Design Style and Programming Language

The traditional (and probably still the most commonly used) software design method in transportation modeling (as well as in other areas) is the so-called top-down structured design. This design technique concentrates on the operations in the solution space with little regard for the design of the data structure. The corresponding programming paradigm is the so-called procedure programming, that is, to decide the preferred procedures and to use the best algorithms that can be found. The focus is on the processing and on the algorithm needed to perform the computations. As a result, data are forced to be global, making the system less reliable because of the possibility of faulty connections between paths. The system is also less resilient because changes in data tend to ripple through the entire structure. A typical, and original, language supporting this procedural programming is FORTRAN, which was widely used in transportation software systems.

A more advanced and rapidly spreading design style is the Object-Oriented Design (OOD), where, an object is characterized by a number of operations and a state which remembers the effect of these operations. With object-oriented design, each module in the system denotes an object or a class of objects from the problem space, and the approach recognizes the importance of software objects as actors, each with its own set of application operations. The object-oriented design approach focuses on the product life cycle instead of on the projects along. Abstraction and information hiding hence form the foundation of all object-oriented design. Programming that supports the object-oriented design is called Object-

Oriented Programming (OOP). Its paradigm may be described as [Stroustrup 1993]: *decide the preferred class, provide full operations for each class and make commonality explicit by using inheritance*.

For the problem studied in this research, the system may be roughly divided into a networking subsystem and a linear programming subsystem. Each subsystem is relatively independent, despite that they artificially communicate, or interact with each other. A detailed study of each subsystem and its components, including their interrelationships, would be very helpful to understand the nature of the entire system, and lead to a reasonably good software system design.

For the networking subsystem, it is very nice to use an object-oriented design. In fact, it is very natural to think of a network as an object (entity) consisting of some other objects, such as nodes, links and paths, etc., each of them having their own domain (states) and operations. These objects are independent of the models and the algorithm studied in this research. They can be reused on any other O-D research model, even other transportation problems such as dynamic design or dynamic O-D, with little or without any modifications. For the linear programming problem, the basic components or objects could be matrices, vectors, and so on. Although the philosophies of object-oriented design are still applicable to this problem, it may be more benificial if a procedural design is also applied, since more characteristics of the algorithm could then be represented. Therefore, the designing style used in the development of this research may be viewed as a combination of procedural design and object-oriented design, with an emphasis on the latter.

The purpose of software design is to provide a strategy to solve the desired problem. This strategy can be carried out via computer coding with suitable (computer) languages as a vehicle. Ideally, the language chosen should directly reflect the view of the problem space. In addition, such a language must provide tools for expressing primitive objects and operations, and be extensible, in order to perform the life cycle development of the software system. In this research, the C++

programming language is chosen as such a vehicle for at least the following reasons.

1) It is consistent with the predetermined design style. C++ is an Object-Oriented Language (OOL). It supports data abstraction and encapsulation. Users may define their own data types via classes, and information hiding is carried out through private and protected membership. The basic data operations (functions and operators) for the user-defined data types can be overloaded. It also supports the inheritance, borrowed from Simula, resembling the feature of parameterized types or generics (template) from Clu and Ada, and inherits the polymorphism via pointers from C. All of these features form the backbone of an object-oriented program. Most of the terminology mentioned here will be illustrated with examples in the following subsections.

2) C++ is very efficient and portable. C++ chooses C as its base language. As a procedural language, C has proved to be successful because it is versatile, terse, relatively low-level, and because of its adequacy for most system programming tasks, run everywhere and on everything; C++ was designed as a "better C", and it inherits the basic procedural programming, ingredients, such as functions, arithmetic, selection statements, and looping constructs, from C. This makes the resulting software package more efficient and portable.

3) C++ is becoming more popular than any other Object-Oriented language. It has become the representative of OOL. For example, a large number of Graphic User Interface (GUI) source codes are available written in C++. With a few modifications, a very nice user interface can be added on a particular application, thereby constructing a user friendly software.

In the following sections, a few examples are provided in order to show the basic concepts, or terminology, of object-oriented design (programming and languages) mentioned above, and how to use these principles to complete the desired tasks.

5.3 Object and Abstract Data Type

An object is characterized by a number of operations and a state which remembers the effect of these operations [Jacobson, et al., 1995]. An object can be constructed from, or consists of, other objects, and the resulting relation is often called partition or aggregate hierarchy. For example, in a traffic network, the network is an object--it consists of a set of links (objects) and a set of nodes (objects). In object-oriented models, each object is attached with some behavior and information. A link, for example, should contain some information such as where it comes from and where it goes to, its distance, its free-flow speed and the traffic volume through it, etc., and its structure such as the nodes that it connects. The behavior and information are *encapsulated* in the object, and the only way to affect the object is to perform an operation on it. In this way, an object can hide its internal structure from its surroundings. This results in the concept of *information hiding*, the central characteristic of an object-oriented software system. With these concepts, one may define some abstract data types. *An abstract data type is a model (structure), with a number of operations that affect this model* [Jacobson, et al., 1995].

In C++ and most other OOL, the abstract data types are defined via *classes*. *A class represents a template for several objects and describes how these objects are structured internally. Objects of the same class have the same definition both for their operations and for their information structures* [Jacobson, *et al.*, 1995]. In C++, the information hiding is performed via the key words *private* and *protected*. The private members and functions can only be operated or used by the members of the same class. Protected members and functions, on the other hand, may be accessed by the members of the same classes, its derived classes (subclasses), or friend classes.

To illustrate these ideas, consider a node class, called TrafNode, in a networking model.

```
Class TrafNode
{
        friend class TrafLink;
        friend class DirLink;
        friend class NetWork;
private:
        int  tot_cm_ln_no;       //total number of (directional coming links and
        int  tot_lv_ln_no;       //and leaving links
        int  avail_cm_ln_no;    // total available coming 1links number
        int  avail_lv_ln_no;     // ..... leaving .....
        int  miss_cm_ln_no;
        int  miss_lv_ln_no;
        int  assign_cm_ln_no;
        int  assign_lv_ln_no;
        DirLink ** coming_link;       //pointers of links coming to this node
        DirLink ** going_link;        //…….leaving from …….
        DirLink ** avail_cm_ln;       // pointers of links coming to this node with available volume
        DirLink ** miss_cm_ln;        //   …….. missing ……..
        DirLink ** avail_lv_ln;       // … . …….. leaving from …… Available …..
        DirLink ** miss_lv_ln;        // ….. …….. missing …….
        DirLink ** assign_cm_ln;     // … …… coming to …….. assigned ……
        DirLink ** assign_lv_ln;      //    …….. leaving from …….
protected:
        int  node_no;                // the node number
public:
        int  node_type;              //maybe intern, extern, centroid and other node
        TrafNode(int node_num=0);     //constructor, need the node_number
        TrafNode(const TrafNade &);              // constructor, copy the object of Node args to here
        void write_node_no(int n){node_no=n;}
        int read_node_no(){return node_no;}
        DirLink **path_to_node(TrafNode *dest_node);          //a path is a set of links
        DirLink **min_path_to_node(TrafNode *destNode);   //pointers of links forming min path to
destNode
        friend TrafNode operator +(TrafNode&, TrafNode&);            //operator overloading, resulting
new node
        friend TrafNode operator +(TrafNode &, TrafLink &);      //overloading, connecting  new links to
the node
        friend ostream& operator <<(ostream &, TrafNode &);  //operator overloading,
        friend int operator ==(TrafNade, TrafNade);
        TrafNode&  operator =(const TrafNode&);             //assign operator
        ~TrafNode(){};  //destructor, may need delete those connected links
};
```

In this example, a TrafNode carries at least the following information. What is the sequence of the node in the network (node_no)? What is the type (node_type) of the node? How many (directional) links are connected to this node (tot_cm_ln_no) and which ones (DirLink** coming_link)? How many and which of them have available volumes, and so on, and how many and which of the links are connected from this node, etc. All this information, except the node type and node number, is basically used for internal operations of assigning volumes to those links which have their volumes missing and are connected to this node (see the algorithm of Chapter 3). They are also used to find the minimum path discussed in Section 4.6 and other storage functions. These internal operations and information do not need to be known by other objects and are hidden in the object. The link number is assigned as a protected member, it can be altered directly by the members of a friend class for convenience. Other general classes may also operate the member node_no, but need to do this through the public write function write_node_no() or the constructors. Note that the encapsulation is not only applied to some members of a class, it may also be applied to functions (operations) too. For example, when a class of traffic link "object" is defined, the BPR traffic time calculation (see Chapter 3) should be hidden in the class itself.

As stated before, in object-oriented design, the only way to effect an object is to perform operations on it. A perfect example to show the exact meaning of this concept is the *operator overloading*. In the TrafNade class defined above, several operators have been overloaded, including the assign operation "=", the equivalent operation "==", output stream operation ">>" and the addition operation "+". Note that there are two different versions of the addition operator "+": $(Node)\ a + (Node)\ b$ and $(Node)a + (TrafLink)b$. They are two different operators because their signatures are different (in functional language, the signature of a function gives the name and type information necessary to invoke that function, or to recognize an invocation of that function). For example, in the following function

```
void Plus1(){
        TrafNade  a, b, d;
        … …;   //some codes here
        d = a + b; }
```

$d = a$ if a and b have different node numbers, and otherwise, d results in a new node which has the same node number as a (and b) and connects to all links that connect with a and/or b. On the other hand, in the following function,

```
void Plus2(){
        TrafNade  a,  d;
        TrafLink b;
        … …;   //some codes here
        d = a + b; }
```

d is the node which results from attaching the (non-directional) TrafLink b to a if a is an A_node or a B_node of TrafLink b (a non-directional link consists of two nodes, called A_node and B_node in the transportation literature, that are connected through one or two directional links), and all of the internal information is updated in the objects. This operation is very attractive in the present application and shows the beauty of the design style.

5.4 Template and Inheritance

Template and inheritance are two main features of an object-oriented language. Both of these are used to find the commonality among several objects, but with opposite emphasis. A template, also called parameterized type or generic, is a specification of how to construct a family of related classes, and defines a family of types or functions. It focuses, in most cases, on different kinds of objects that have the same algorithmic structures of operations. For example, to solve the linear programming problem and many other mathematical problems, it is very useful to define various matrix objects and their operations. The entries of these matrices may be integer, float, double-precision real or complex numbers, but their

operations such as addition, subtraction, and multiplication, can be defined via the same symbolic algorithm. In traditional procedure programming, each operation, indirectly defined through procedures or functions, should be reprogrammed once for each data type, although the codes are almost the same. With templates, these objects and their operations can be defined in the following uniform way.

```
template <class PT>  class T_Matrix
{
 private:
        int rows, cols;
        PT  *  ptr;
 public:
        T_Matrix(const int  m, const int  n) {  //constructor for a  mxn dimensional matrix
                rows =m;
                cols = n;
                ptr = new PT[rows*cols];
                if (ptr == NULL)  error_mem(); }
        T_Matrix(const T_Matrix &a) {  //copy constructor
                rows=a.rows;
                cols=a.cols;
                ptr= new PT[rows*cols];
                if (ptr == NULL) error_mem();
                memcpy(ptr, a.ptr, rows * cols * sizeof(PT)); }
        virtual ~T_Matrix(){ delete ptr;}
        T_Matrix& operator = (const T_Matrix &a) {……;}  //assign operator

        friend T_Matrix operator +(const T_Matrix& a1,const  T_Matrix& a2){  //define  a1 +
a2
                if (a1.rows!=a2.rows||a1.cols!=a2.cols)
                        { cerr <<"T_Matrix is not in the same size!"<<endl;
                          exit(EXIT_FAILURE); } //checking size
                T_Matrix res(a1.rows, a1.cols);
                for (int i=1; i<= res.rows; i++) {
                        for(int j=1; j<=res.cols; j++)
                          res(i,j)=a1(i,j)+a2(i,j); }
                return res; }
        friend T_Matrix operator - (T_Matrix& a1, T_Matrix& a2) {……; }//define  a1 - a2
        friend T_Matrix operator *(T_Matrix& a1, T_Matrix& a2) {……;} //define  a1*a2
        PT &operator()(int _row, int _col){
                return ptr[(_row-1)*cols + _col -1]; }
        void error_mem(){ cerr <<"Failed to allocate T_Matrix." <<endl;
                        exit(EXIT_FAILURE); }
};
```

With this template, defining integer and real matrices and their operations becomes very easy. Look at the following function:

```
void foo(){
        T_Matrix<int>          Im1(20,10), Im2(20,10), Im3, Im4;  //declare integer matrix
        T_Matrix<float>        R1(32,23), R2(23,22), R3;      //declare real matrix
        Im1(2, 3)=-5;          // the [2][3]-th entry of Im1 is assigned a value -5
        … …;
        Im3=Im1+Im2;
        Im4=Im2 - Im1;
        R3 = R1*R2;        //R3 now is a 32x22 matrix and = R1*R2
}
```

It looks just like a regular mathematical expression in linear algebra textbooks! For the indexed cost matrix considered in Section 4.6, the template can also be easily used. First, introduce the following two-parameter template:

```
Template<class T1, class T2> class T_Index
{
  private:
        T1 id1;
        T2 id2;
 public:
        T_Index(T1 _id1, T2 _id2){ id1=_id1; id2= _id2;}   //constructor
        ……;  //other public function and operator, including +, -, <, ==, >, =
};
```

Then, the following simple program shows how to define and use the indexed cost matrix.

```
void foo(){
        class T_Index<int, float> CostIndex;
        T_Matrix<CostIndex> costM1(50, 50), costM2(50,50), costM3;
        costM1(1,2)=CostIndex(0,2.8);
        …….;
        costM3=costM1 * costM2;
        ……; }
```

Note that if it is used for determining the shortest path via generalized Dijkstra's method as shown in Section 4.6, then the multiplication operator "∗" in the last

program needs to be redefined. In fact, if one studies the algorithm there carefully, it is easy to see that the operator "+" and "∗" are only formally used, and the meaning for these operators are totally different from the meaning for the same operators in the regular linear algebra sense. Since the attributes in the object representing a cost matrix, for a given parameter class, is just the same as a regular matrix, there are two ways to deal with the above differences. The first method is to redefine a new class (of objects). Another method is only to modify or update the operations and information structures in the class that are different from an existing class. These modifications or updates can be carried out in the object-oriented language in terms of the *inheritance* hierarchy.

Inheritance is used to develop a new class merely by stating how it differs from other, already existing, classes. The new class then inherits the existing class. *If class B inherits class A, then both the operations and the information structure described in class A will become part of class B* [Jacobson et al., 1995]. By means of inheritance, the similarities between classes can be shown and these similarities can be described in a class which other classes can inherit. The more general class is then called a base class or a superclass and the specified one is then called a derived class or subclass. The main advantage with inheritance is that existing classes can be reused to a great extent. It is one of the most powerful tools in object-oriented languages. It also simplifies the process of reuse. Note that the reusability also occurs in various programming languages. In procedural programming, for example, the reused levels are procedures, but the levels are classes in object-oriented programming.

The following code shows how to derive a cost matrix class from the T_Matrix class defined above.

```
typedef   T_Index<int, float>  CostIndex;

class IdxPathMatrix : public T_Matrix<CostIndex>
{
        private:
                CostIndex  NonZeroMin(CostIndex  a, CostIndex  b)
```

```
        {
            if (a == CostIndex(0,0) || a>b)
                return b;
            else
                return a;
        }
    public:
        IdxPathMatrix(int m, int n) : public T_Matrix<CostIndex>(m,n){};
        ……;/
        friend IdxPathMatrix operator + (IdxPathMatrix& a1, IdxPathMatrix& a2)
        {
            ……;  //first checking dimension
            IdxPathMatrix   res(a1.rows, a1.cols);
            int  i, j;
            for ( i=1;  i<= res.rows;  i++) {
                for( j =1; j<=res.cols;  j++)
                    res(i,j) = NonZeroMin(a1(i,j),a2(i,j)); }
            return res;
        }
        friend IndexPathMatrix operator *(IndexPathMatrix& a, IndexPathMatrix& b)
        {
            … …;  //first check dimension
            IndexPathMatrix   res(a.rows, b.cols);
            int  i, j, k;
            for ( i=1;  i<= res.rows;  i++) {
                for( j =1; j<=res.cols;  j++){
                    for ( k=1; k<=a.cols; k++)
                    {
                        if (a==CostIndex(0,0) || b==CostIndex(0.0))
                            res( i, j) = CostIndex(0,0);
                        else
                            res( i, j) = a(i,j) + b(i, j);
                    }
                }
            }
        }
}
```

As discussed before, the minimum cost path matrix has its own operation algorithm for "+" and "∗", but the other information and structures, including the operations, are just the same as for ordinary matrices. Therefore, the former matrix can inherit an ordinary matrix. Moreover, look at the constructor - it just passes the information from its parent object.

It may be worth noting that only one special class obtained via the T_Matrix with parameter T_Index<int, real> is inherited. Since the algorithm presented there can

also be used for a shortest path problem having ordinary real costs, it may not be a bad idea to derive the subclass as a template instead of using only one single class. This kind of inheritance is supported in C++, and will not be repeated here. Furthermore, it is very common to declare some operations in the base class as a virtual function, and redefine those operations in the derived class.

5.5 Dynamic Link and Binding Polymorphism

A very important concept in object-oriented design and programming is polymorphism. *Polymorphism* means that the sender of a stimulus does not need to know the receiving instance's class [Jacobson, et al. 1995]. It is the receiver of a stimulus that determines how a stimulus will be interpreted, not the transmitter. In other words, polymorphism is the genie in OOP, taking an instruction from a client and properly interpreting its wishes [Pokl 1993].

In a general sense, polymorphism has been widely used in traditional procedural languages. For example, in FORTRAN and ANSI C, the return type of division operator is dependent on the argument type. If the arguments to the division operator are integer, then the integer division is used. On the other hand, if one or both arguments are floating points, then floating point division is used. Furthermore, a function in ANSI C is called based on its signature, and its operations and return type depends on both its name and argument types. These are some examples of *ad hoc* polymorphism (coercion and overloading).

Another example in OOP is the so called parametric polymorphism, that is, the type is left unspecified and is later instantiated. Manipulation of generic pointers and templates provides this in C++, as discussed in the previous examples. For example, the T_Matrix class defined in the last section uses a pointer as its container. The size of this container is only known after the column and row

numbers have been passed into the constructor. Meanwhile, the T_Matrix is also provided as a template.

In OOP references, the terminology polymorphism is used mostly to emphasize the pure inclusion. In other word, in object-oriented methodology, polymorphism mostly focuses on the fact that one type is a subtype of another type, i.e., it satisfies the inheritance hierarchy discussed in the last section. Functions available for the base type will work on the subtype. Such a function can have different implementations that are invoked by a run-time determination of subtype. In C++, the pure inclusive polymorphism is mostly implemented through *dynamic binding* and *dynamic instantiation*.

Dynamic instantiation means that new objects are created by the program at run-time, and dynamic binding means that the compiler arranges a mechanism to select at run-time the method to be used to handle an invocation on a given object. To illustrate this concept, consider the following example.

```
//////////////////          define the base class BLink          ////////////////////////////////////////////
class BLink{
        private:
                TrafNade *A_Node, *Bnode;
                float     Dist, FrFlSpeed;
                int        Capacity;
        public:
                float      BPR_EQ(float sp, float dis, float cap, float vol){ \* ……;*\}
                TLink( TrafNade * _aNd, TrafNade * _bNd, float _dis, float _cap, float
_spd){…};
                virtual   void TheCost()=0;
                ……;
};

////////////////          define the derived class TrafLink    ////////////////////////////////////////////
class TrafLink : public BLink{
        private:
                int        A2Bvol, B2Avol;
        public:
                TrafLink( TrafNade* _a, TrafNade *_b, float _d, float _c, float _s, float _v1,
float _v2);
                void      TheCost();
```

```
};

TrafLink( TrafNade*_a, TrafNade*_b, float _d, int _c, float _s, int _v1,int _v2)
        :  public BLink(* _a, *_b, _d, _c, _s)
{
        A2Bvol = _v1;
        B2Avol = _v2;
}

void  TrafLink :: TheCost():public Blink{
        cout <<"A2B Cost = "<<  BPR_EQ( FrFlSpeed,  Dist,  Capacity,  A2Bvol) <<endl;
        cout <<"B2A Cost = "<<  BPR_EQ( FrFlSpeed,  Dist,  Capacity,  B2Avol) <<endl;
}
///////////////         define the derived class TrafLink    ////////////////////////////////////////
class DirLink : public BLink{
        private:
                int         vol;
        public:
                TrafLink( TrafNade* _a, TrafNade *_b, float _d, float _c, float _s, float _v);
                void       TheCost();
};

DirLink( TrafNade* _a, TrafNade*_b, float _d, int _c, float _s, int _v) :  public BLink( _a, _b, _d,
_c,  _s)
{
        vol = _v;
}

void  DirLink :: TheCost():public Blink{
        cout <<"The Cost = "<<   BPR_EQ( FrFlSpeed,  Dist,  Capacity,  vol) <<endl;
}
```

The key word *virtual* used in the base class implies that a method having the same signature in a derived class will override/shadow/hide the base class method. With the above definition, then, the following program shows how to choose to initiate each object and choose a right method to implement during the run time.

```
Main(){
        Blink     * link1, *link2;
        TrafNade          *n1, *n2;
        n1 = new TrafNade(1);      //create a TrafNade object
        n2 = new TrafNade(3);          //create a TrafNade object
        link1 = new TrafLink (n1,n2, 0.6, 3000, 55, 1900, 824);     //create a TrafLink object
```

```
        link2 = new DirLink (n2,n1, 0.6, 3000, 55, 824);      //creat a DirLink object
        link1->TheCost();     //access member function, the implementation of TrafLink
        link2->TheCost();     //access member function, the implementation of DirfLink
}
```

5.6 Summary and Technical Remarks

So far, philosophies of software design and programming used in this research have been briefly introduced. In short, the object-oriented design and programming are used as the backbone in the computer coding part of this research, with an implementation using the language C++. The choice of C++ as the programming language is based on the following considerations.

- C++ supports the object-oriented design and programming, a style chosen to be used in this research.

- C++ uses C language as its base language, which is a relatively lower-level language and has proved to be very efficient in various operating and hardware systems. Since the efficiency of programming is the main concern in the software design of this research, it is hoped that the choice of C++ will result in a program that executes in an efficient manner.

- C++ is among the most popular of object-oriented languages. Therefore, choosing C++ as the programming language ensures that the developed code will be easier to reuse and modify, either for the purpose of modifying the model and algorithm developed in this research or for use in other transportation problems.

The basic features of object-oriented programming and examples on how to realize these characteristics in this research have been provided in the last few subsections. The following paragraphs present some concerns on the

implementation of these general philosophies in this particular application, while emphasizing on the efficiency of the programming.

1. Dynamic and Static Bindings. The dynamic instantiation and binding provide a possibility of creating a new object and selecting an appropriate method during the run time. This gives a great deal of flexibility for the resulting program. On the other hand, since the object is created at run time, it also injects, in most cases, certain overheads into the resulting program. These overheads influence the performance of the program, especially when the same object needs to be created many times or the same operation needs to be enforced with a high frequency. A typical example in this research is that, for solving the linear programming problem by using the modified column generation algorithm, some path searching operations and matrix operations are performed many times, especially when the test network is large-sized. For such repeatedly used objects and methods, a static binding mechanism is better to use, i.e., one that interprets the methods during the compiling operation. In most of this research, these object will be created dynamically and their methods then used statically, in order to operate the resulting program by using the set of available time and space resources in an efficient manner.

2. Array and Linked Lists. Arrays and matrices are basic components in networking and linear programming. They are created statically in traditional procedural programming such as in FORTRAN or C, in the sense that fixed blocks of memories are allocated in the compiling step for each array in each procedure. This programming style has certain drawbacks. For example, consider what is a reasonable maximum number of coming in or coming out links for each node in a network. In the computer code used in the LP model [Sherali et at., 1994], this number is set to be 10. On the other hand, this number is taken to be 4 in the Highway Emulator package [Bromage 1988, 1993]. For LP models, the program guarantees that the allocated space is sufficient to perform desired operations for almost every real network. Since a

much larger sized space than needed is allocated, the allocation of a large fixed size memory wastes available space resources, and in turn, reduces the computation speed, especially when the system memory is limited and a huge amount of memory swapping is forced. For the Highway Emulator model, on the other hand, setting that number to be 4 is suitable for usual networks, but for those networks in which the in or out degree for some nodes are more than 4, the user manual suggests to reconstruct the network. An alternative solution may be to change the source code so that the model is suitable for the particular studied network. This means that users need to change the dimension declaration of arrays in many procedures, and this increases the possibility of dimension-not-matching errors. In C language, the problem of changing the code in several places can be avoided by defining the array sizes as macros, but it has been reported frequently that macro-programming is a trouble-maker style. In object-oriented languages, this dimensional resetting problem can be treated easily at run time by using the dynamic instantiation, and the sizes of the arrays can be set exactly in an as-needed basis. To avoid the shortage of overhead as discussed in the last remark, some advanced techniques, such as binding the operation statically when the object has been initialized, are also used. Another example is the mathematical representation of a path in a given network. In Chapter 3, a path is defined as an $n \times n$ matrix, where n is the total number of nodes, and an entry is set to be 1 when the path goes through that node pair and to be 0 in other cases. Although this representation is correct theoretically, there is no reason to waste space for memorizing those node pairs that are never connected. Therefore, a path will be represented as an array or a linked list (in the terminology of data structures) of links or nodes, and only these connected node pairs are remembered in this array or linked list.

3. Other Data Structures. The column generation algorithm discussed in Section 4.5 is used mainly for solving the linear programming model developed in Chapter 3. To implement this algorithm as a computer code, some other data

structures are also used. For example, to assign a volume to links with missing volume information, Remark 3.6 in Section 3.3 suggests that the assignment should begin with a node that has the most information regarding available volumes on its incident links. Therefore, there is a need to search for that node which has the most information after each assignment is performed. In order to improve the performance, a binary search tree (BST) is used during the entire assignment process. Some similar techniques are also used in other parts of the program.

In summary, in order to solve the linear programming problem using the modified column generation algorithm, a suitable software package is developed based on an object-oriented design. This design style is implemented via the C++ language, and various modern programming techniques and methods, in addition to the advanced algorithm, are used to make the program more efficient.