National Library of Canada

Acquisitions and Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada

Direction des acquisitions et des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

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

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

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

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

AVIS

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

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

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

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

Canada

Correctly Ordered Postman Tours and Synchronizable Test Sequence Generation For Protocol Conformance Testing

by

Maria Eugenia Perez

A THESIS
submitted to the School of Graduate Studies and Research
in partial fullfillment of the requirement
for the degree of

Master
in
Computer Science

Ottawa-Carleton Institute for Computer Science
Departament of Computer Science
Faculty of Science
University of Ottawa
OTTAWA, ONTARIO

© Maria Eugenia Perez, Ottawa, Canada, 1993

National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services Branch

Direction des acquisitions et des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

THE AUTHOR HAS GRANTED AN IRREVOCABLE NON-EXCLUSIVE LICENCE ALLOWING THE NATIONAL LIBRARY OF CANADA TO REPRODUCE, LOAN, DISTRIBUTE OR SELL COPIES OF HIS/HER THESIS BY ANY MEANS AND IN ANY FORM OR FORMAT, MAKING THIS THESIS AVAILABLE TO INTERESTED PERSONS.

L'AUTEUR A ACCORDE UNE LICENCE IRREVOCABLE ET NON EXCLUSIVE PERMETTANT A LA BIBLIOTHEQUE NATIONALE DU CANADA DE REPRODUIRE, PRETER, DISTRIBUER OU VENDRE DES COPIES DE SA THESE DE QUELQUE MANIERE ET SOUS QUELQUE FORME QUE CE SOIT POUR METTRE DES EXEMPLAIRES DE CETTE THESE A LA DISPOSITION DES PERSONNE INTERESSEES.

THE AUTHOR RETAINS OWNERSHIP OF THE COPYRIGHT IN HIS/HER THESIS. NEITHER THE THESIS NOR SUBSTANTIAL EXTRACTS FROM IT MAY BE PRINTED OR OTHERWISE REPRODUCED WITHOUT HIS/HER PERMISSION.

L'AUTEUR CONSERVE LA PROPRIETE DU DROIT D'AUTEUR QUI PROTEGE SA THESE. NI LA THESE NI DES EXTRAITS SUBSTANTIELS DE CELLE-CI NE DOIVENT ETRE IMPRIMES OU AUTREMENT REPRODUITS SANS SON AUTORISATION.

ISBN 0-315-95965-7

Canadä

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Dr. Sylvia Boyd. Since the undertaking of my thesis, Dr. Boyd has been a constant source of support, guidance, and ideas for me. I am most appreciative of her patient assistance.

I would also like to thank Dr. Hasan Ural, who kindly allowed me to consult him on several opportunities.

Lastly, I would like to thank my husband Alfredo, my family and friends for their constant moral support and encouragement. This would have been a daunting task if they had not been there for me.

Abstract

In this thesis, we will consider the distributed architecture for communication protocol conformance testing, where there are two remote testers, called the lower tester and the upper tester. During the application of a predetermined test sequence in the distributed architecture, the upper and the lower testers are bound to synchronize with each other only through their interaction with the implementation. However, this requirement may lead to a synchronization problem when the upper and the lower tester processes are forced to consider the timing of an interaction in which they have not taken part.

We present a heuristic for generating a minimum length synchronizable test sequence for protocol conformance testing. The approach is based on a variation of a well-known problem in graph theory known as the Chinese postman problem. In this variation, for each edge e=(i,j) in a digraph D, we specify a subset of edges leaving j as the eligible successors for e, and require that each edge in the postman tour be followed by an eligible successsor. We present a heuristic to solve this variation of the Chinese postman problem, and develop a specialized version of this heuristic for the problem of generating a minimum length synchronizable test sequence. We show empirically that our specialized heuristic has superior performance compared to other known heuristics for this problem.

Contents

Acknowledgement i

Abstract ii

Table of Contents iii

1 Introduction **1**

1.1 Protocol conformance testing and the Chinese postman problem 1

1.2 The synchronization problem 4

1.3 The correctly-ordered Chinese postman problem and its application to the synchronization problem 6

1.4 Overview of the thesis 7

2 Background **9**

2.1 Graph theory definitions 9

2.2 Background on the rural Chinese postman problem 11

2.3 The synchronization problem in protocol testing 16

2.4 Modelling the synchronization problem as a correctly-ordered postman tour 19

3 A heuristic for finding a synchronizable test sequence for protocol conformance testing **22**

3.1 Creating the duplex digraph 23

3.2 Characteristics of the duplex digraph 25

3.3 An algorithm for synchronizable test sequence generation 30

4 **Implementation** 39

4.1 Digraph generator GENGRAPH 39

4.2 Synchronizable test sequence generator STSG 40

4.3 Description of the algorithms implemented 41

5 **Performance** 43

5.1 Complexity analysis 43

5.2 Comparison with Wang's heuristic 45

5.2.1 Checking for existence of a COPT in D 47

5.2.2 Creating a duplex digraph 48

5.2.3 Finding the RSA D* 51

5.2.4 Comparison of performance 53

5.3 Empirical Testing of the heuristic 55

5.4 An example where the heuristic does not find the optimal solution 59

6 **A heuristic for finding a correctly-ordered Chinese postman tour** 61

6.1 The digraph D' 62

6.2 Characteristics of the digraph D' 63

6.3 An Algorithm for the problem of finding a COPT in D 64

7 **Conclusions** 69

Appendix A. Programs GENGRAPH and STSG 71

References 112

Chapter 1

Introduction

1.1 Protocol conformance testing and the Chinese postman problem

A protocol is a set of rules stating just how two or more parties are supposed to interact by sending messages to each other. The term "testing" indicates that some operations are carried out to certify the conformity of a protocol implementation to its specification. In data communication, protocol testing means that the test object is a protocol implementation. When testing a protocol implementation to see if it conforms to its specification, we assume that the implementation is a black box. The protocol implementation is tested by externally generating a sequence of inputs from two logically distinct elements, which are called the upper and the lower testers because of their relationship to the interfaces of the implementation, and then verifying that these inputs cause an expected sequence of outputs.

It is well established that conformance testing of protocol implementations is needed to ensure reliable communications in computer networks. Recently, it became apparent that protocols require formal methods to define the conformance test.

A protocol can be specified as a deterministic and minimal finite state machine (FSM) [12], where the FSM is in a specific state and an input message can cause the FSM to generate output messages and undergo a change from the current state to a new state. During the conformance test of a protocol implementation, every state transition of the FSM that represents the protocol should be tested. Testing a state transition from state v_i to state v_j takes place in three steps:

1. Put the implementation into state v_i.
2. Apply the required input and compare the output generated with the one defined in the specification.
3. Verify that the new state of the FSM is v_j.

Generally, the problem of finding a minimum length complete test sequence (also known as a checking sequence) is NP-complete [4]. Several formal methods have been introduced for test sequence generation from FSM-based specifications[18]. Some of these methods are: The T-method, the D-method, the W-method, the UIO-method and several improvements of the UIO-method.

In protocol conformance testing, the concept of a transition tour in an FSM has been applied and is known as the T-method. A transition tour is an input sequence which starts with the initial state and covers all transitions defined in the protocol specification at least once. The T-method detects all operation errors (errors in the output function) but it does not detect all the transfer errors (errors in the next state function), since the state that the FSM of the protocol implementation reaches after the execution of a transition is not checked, as specified in step 3 of the state transition testing procedure. The T-method is

generally applicable; the application of some other methods, such as the W-method, require the existence of a sequence of inputs, called the characterizing sequence, which can be used to verify that the FSM has reached a designated state. The T-method also gives shorter test sequences than the D-method and the W-method because it is directly formed by checking the FSM, and does not need to add any extra characterizing sequence.

Uyar and Dahbura[19] showed that finding a minimum length transition tour of a protocol specification is equivalent to finding a Chinese postman tour of a digraph D, where D represents the FSM of the protocol specification. The objective of minimizing the length of a test sequence stems from the consideration of reducing the total cost of the input/output operations and hence the cost of the testing process.

The Chinese postman problem [3,11] was first considered by a Chinese mathematician, Guan Meigu (1962) and can be described as follows. A postman wishes to find a shortest route in a digraph which starts from some node, traverses each edge at least once, and then returns to the starting node. In a weighted digraph D, we define the weight of a tour $v_0e_1v_1...e_nv_0$ to be $\Sigma(w(e_i)$: for $i=1,...,n)$. The Chinese postman problem is that of finding a minimum weight tour traversing every edge at least once in a weighted connected digraph with non-negative weights. Such a minimum weight tour is called a Chinese postman tour.

The Chinese postman problem and its solution has a number of potential applications, for example, refuse collection, the delivery of milk and post, the inspection of electric power, telephone or railway lines, etc. This problem has been shown to be solvable in polynomial time by Edmonds and Johnson [11].

1.2 The Synchronization Problem

There are two main classes of end-system test methods[13,14,16], the local and the external. The local test methods are really only applicable to in-house testing by suppliers, whereas the external test methods are also applicable to testing carried out by users and third party test centers. Theoretically, the local test method is the best of test methods as it assumes that it has direct access to both the upper and the lower interfaces of the implementation. However, in practice, even if a tester has direct access to both interfaces, any software adopting a local test method would be too complex to be implemented.

There are three types of external test methods: distributed, coordinated and remote. The remote test method assumes no explicit test coordination procedures in the real open system in which the implementation resides, called the system under test. The remote method also tests the system under test as a "black box" from another machine. Therefore, it is incapable of control and observation of the upper interface of the implementation. The distributed and coordinated test methods both require implementation of a complex upper tester within the system under test, and are deficient in guaranteeing synchronization between the upper and the lower testers which reside on two different machines. As reported in [21], Zeng defined an alternative to both the distributed and coordinated methods, called the ferry testing approach. In this method the upper tester is moved from the system under test to the test laboratory's system. The ferry concept was first intendted to simplify the test software in the system under test and enhance the synchronization between an upper and a lower tester. However, this solution requires an additional protocol to facilitate communication between the upper and the lower tester. The necessity for such coordination is eliminated by constructing

a synchronizable test sequence such that the corresponding sequence of transitions causes no synchronization problem.

In this thesis, we will consider the distributed test method. During testing in this method, the upper and the lower testers provide control and observation of the upper and the lower boundaries of the implementation respectively. The lower tester may be miles away from the upper tester and a test sequence may be nonexecutable due to the synchronization problem between the lower and the upper tester. This means that, when applying a test sequence to an implementation, a problem may occur because one of the testers (upper or lower) may not be able to determine when to send an input to the implementation because it did not take part in the previous transition. Formally, we could say that a test sequence [I1/O1, I2/O2,...,Ii/Oi, Ii+1/Oi+1,...,In/On] is synchronizable if (and only if) considering two consecutive transitions from the test sequence, both operations Ii+1 and Oi are related to the same tester (lower or upper) or both operations Ii+1 and Ii are related to the same tester (lower or upper). Since the algorithm described by Uyar and Dahbura[19] for finding a minimum length transition tour may find a test sequence which is not synchronized and therefore unusable in this situation, new techniques for generating synchronizable test sequences for protocol conformance testing have been studied. In [17], Sarikaya and Bochmann discuss an algorithm for generating a transition tour without synchronization problems, in which they make no attempt to obtain a minimum length test sequence. In [7], Chen, Lu, Wang, and Lee describe a heuristic algorithm for generating an approximately minimum length synchronizable test sequence assuming that there is no specific start node, which means that the first edge in their transition tour must be synchronizable with their last edge. Thus there may exist a synchronizable test sequence which starts and ends at a specified node which cannot be generated by their heuristic. Zhiping

Wang in his thesis[20] describes a heuristic method which combined with the T-method can be employed to generate a synchronizable test sequence for protocol conformance testing. Wang also combines this heuristic with other formal methods for test sequence generation, namely the W-method, D-method and UIO-method, to generate synchronizable test sequences for protocol conformance testing.

Wang uses a technique which was first proposed by W.H.Chen, C.S.Lu, L.Chen, and J.T.Wang[6]. This technique involves the conversion of the digraph representing the protocol specification into a duplex digraph so as to generate a synchronizable test sequence for the protocol. However, the technique proposed in [6] suffers some deficiencies which have been improved by Wang. First, the technique used in [6] can only be applied to protocols which are tighly synchronizable with both the upper and the lower tester, i.e. Ii+1 and Oi are related to the same tester. Second, some types of transitions are not covered.

Another approach for dealing with the synchronization problem is discussed in [8], where a link is established between the upper and lower tester, and a synchronizable test sequence may include some use of this external synchronization operation at some specified cost.

1.3 The correctly ordered Chinese postman problem and its application to the synchronization problem

We will consider a variation of the Chinese postman problem, where for each edge e=(i,j) in a digraph D, we specify a subset of edges leaving j as the eligible successors for e. This type of digraph is called an order-specified digraph.

specified digraph D, where for every consecutive pair of edges ei and ek in the tour, ek has been specified as an eligible successor of ei. Such a tour is called a correctly-ordered Chinese postman tour. Due to the fact that finding a correctly-ordered Chinese postman tour is NP-hard[5], it is unlikely that an efficient algorithm will be found to solve it. We present a heuristic method for finding a correctly-ordered Chinese postman tour in an order-specified digraph D.

The problem of finding a minimum length synchronizable test sequence can be modelled as a correctly-ordered Chinese postman problem for which the edges out of each vertex v can be partitioned into two sets, say A and B, such that the eligible successors for each edge into v are the edges in A, B or $A \cup B$. We develop a specialized version of our general heuristic for the correctly-ordered Chinese postman problem for solving the problem of finding a synchronizable test sequence. As in Wang's heuristic[20] we create a duplex digraph, but of much smaller size. Our algorithm also has a lower complexity and improved performance.

1.4 Overview of the thesis

In Chapter 3 of this thesis we describe our heuristic adapted to the problem of finding a synchronizable test sequence for protocol conformance testing. In Chapter 4 we describe our implementation of this heuristic. In Chapter 5 we discuss the complexity of our algorithm and we compare the performance of it with Wang's algorithm. In Chapter 6 we describe our heuristic for the general problem of finding a correctly-ordered Chinese postman tour in a digraph D. Finally in Chapter 7, we make our conclusions and give ideas for future work.

In the next chapter we will introduce some basic concepts and background in graph theory, the rural Chinese postman tour , and protocol testing.

Chapter 2

Background

2.1 Graph theory definitions

For the purpose of this thesis, a *graph* G=(V,E) is defined as a finite set of *vertices* V which are interconnected by a finite set of *edges* E. Each edge $e \in E$ corresponds to two vertices in V, called the *ends* of e. An edge $e \in E$ with ends u and v is sometimes denoted by uv. Two vertices $u,v \in V$ are said to be *adjacent* if $uv \in E$, and if e = uv, e is said to be *incident* with u and v. A weighted graph is one in which every edge is assigned a number called its *cost*.

A *path* from v1 to vi is a sequence P = v1,e1,v2,e2,...,ei-1,vi alternating in vertices and edges such that for $1 \leq j < i-1$, ej is incident with vj and vj+1. If v1 = vi then P is said to be a *tour*. The *length* of a path P is equal to the number of edges in P. The *cost* of a path P is equal to the sum of the costs of its edges.

A graph G is *connected* if every two nodes in G are joined by a path. A *subgraph* G' of G is a graph obtained by removing a number of edges and/or vertices of G. A *component* of a graph G is any maximal connected subgraph of G. An *edge induced subgraph* G[E'] of G for $E' \subseteq E$ is the graph consisting of the edges in E' and the set of end vertices of the edges in E'. A *spanning* subgraph of a graph G is a subgraph which contains all the vertices of G. A *tree* is a connected

graph containing no tours. A *spanning tree* of a connected graph G is a subgraph which is both a tree and spanning.

A *digraph* D = (V,E) is a graph to which we assign a direction to each of its edges . An edge e in D which is directed from vertex i to vertex j is represented as (i,j), where i is called the *head* of e and j is called the *tail* of e, denoted by head(e) and tail(e), respectively. (Note that these definitions for head and tail are standard in the area of protocol testing, whereas in graph theory they are usually defined in the opposite way). Given $X \subseteq V$, $\delta(X)=\{e \in E$: head(e)$\in X$, tail(e)$\notin X\}$.

A path in a digraph D is a sequence P=v1,e1,v2,e2,...,ei-1,vi alternating in vertices and edges such that for $1 \leq j \leq i-1$, ej has head vj and tail vj+1. A digraph D is said to be *strongly connected* if for each vertex v of D there exists a path from v to any other vertex of D. We say D is *weakly connected* if, when we remove the orientation from the arcs of D, a connected graph remains.

For a vertex v, the *out-degree* $d_{out}(v)$ and the *in-degree* $d_{in}(v)$ are, respectively, the number of edges directed out of v and the number of edges directed into v. A digraph D is *symmetric* if for every vertex v, the out-degree of v is equal to the in-degree of v.

A *cut* in a digraph D is the set of all edges e in E with one end in S, and the other not in S, for some $S \subseteq V$. If all of the edges in the cut are directed out of S, then we call the cut *directed.*

If it is possible to partition the vertices of a graph G into two subsets, V1 and V2, such that every edge of G connects a vertex in V1 to a vertex in V2, then G is said to be *bipartite.*

Given a set E, we denote the cardinality of E by |E|. Given another set J, we use the notation E\J to denote the members of E not in J.

2.2 Background on the rural Chinese postman problem

An *Euler tour* in a digraph is a tour which traverses each edge exactly once. The following is a well known theorem which gives necessary and sufficient conditions for the existence of an Euler tour.

Theorem 2.1:[11] A digraph D contains an Euler tour if and only if D is strongly connected and symmetric.

A postman tour (or path) in a digraph D=(V,E) is a tour (or path) which traverses every edge in D at least once. *A Chinese postman tour* (or path) in D is an optimal (minimum-cost) postman tour (or path) in D. Note that by Theorem 2.1[11] in, when D is strongly connected and symmetric, the Chinese postman tour problem can be reduced to the problem of finding an Euler tour in D.

Given a digraph D=(V,E) and a specified set of edges Ec $\subseteq$ E, a *rural postman tour* (RPT) or path (RPP) of D over Ec is a tour (or path) which traverses each edge in Ec at least once. A *rural Chinese postman tour* (RCPT) or path (RCPP) is a rural postman tour (or path) of minimum cost.

Given a digraph $D=(V,E)$ and $E_c \subseteq E$, let $D^*=(V^*,E^*)$ be a digraph formed from D by taking $\lambda_e \geq 0$ copies of each edge e in E to form E^*, and letting V^* be the vertices in V incident with edges in E^*. Then D^* is a *rural symmetric augmentation* (RSA) of D[Ec] if $\lambda_e \geq 1$ for all $e \in E_c$, D^* is symmetric, and the sum of the cost of the edges in E^* is minimized. Note that some of the edges in D may not appear in D^*.

The following two theorems show that the problem of finding a RCPT in a digraph D over Ec can be reduced to the problem of finding an Euler tour of a RSA D^* of D[Ec] if D[Ec] is weakly connected.

Theorem 2.2:[1] Given a digraph $D=(V,E)$ and $E_c \subseteq E$, if the edge-induced subgraph D[Ec] is a weakly connected subgraph of D, then any RSA D^* of D[Ec] is strongly connected.

Theorem 2.3:[1] Given a digraph $D=(V,E)$ and $E_c \subseteq E$, if a RSA D^* of D[Ec] is strongly connected, then D^* has an Euler tour which is an RCPT of D over Ec.

If D is strongly connected, a RSA $D^*=(V^*,E^*)$ of a digraph D[Ec] can be obtained by solving a minimum-cost maximum flow problem on a digraph $D_f = (V_f,E_f)$ constructed from D as follows (see also [1]). Define the *net degree* d(v) of a vertex v in D as the in-degree minus the out-degree of v in D[Ec]. Let $V_f = V \cup \{s,t\}$, where s and t are the source and the sink of D_f respectively, and let $E_f = E \cup \{(s,v_i) : v_i \in V+\} \cup \{(v_j,t) : v_j \in V-\}$ where V+ (V-) is the set of vertices with positive (negative) net degrees. Assign to each edge (s,v_i) cost zero and capacity

equal to d(vi), and to each edge (vj,t) assign cost zero and capacity equal to -d(vj). All the other edges in E_f have the same cost as in D and have infinite capacity.

Since all the edges in E have infinite capacity in D_f , a minimum-cost maximun flow F on D_f saturates all the edges incident to s, and thus to t as well, since $\Sigma(d(v): v\in V+) = \Sigma(d(v): v\in V-)$.

Let $x=(x_e:e\in E_f)$ be the minimum-cost maximum flow found for D_f. A RSA D* is formed by letting $E^*=Ec\cup(x_e$ copies of each $e\in E)$ and letting V* be the set of vertices in V incident with edges in E*.

In Figures 2.1-2.3 we show an example of the above process. Figure 2.1 shows a digraph D with edge subset Ec and net degrees as shown. For this example, all edges are considered to have cost 1. Figure 2.2 shows the corresponding flow problem and optimal flow x. Finally, Figure 2.3 shows the RSA D* obtained from x. Since D[Ec] is weakly connected in this case, D* is strongly connected, and from D* we obtain the RCPT 0, 1, 2, 7, 6, 0, 1, 3, 5, 4, 8, 7, 6, 0 of D over Ec.

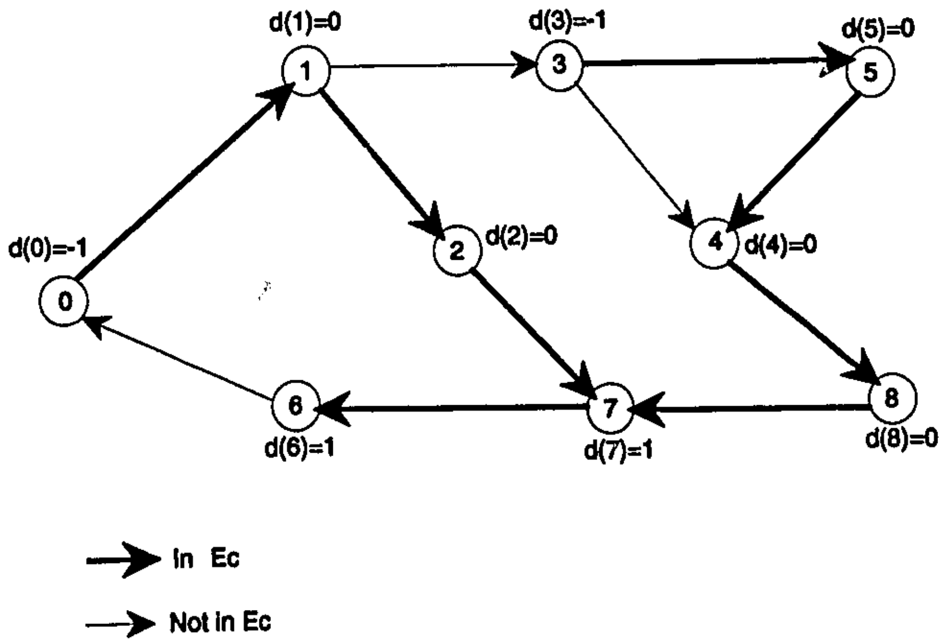


Figure 2.1 Digraph D.

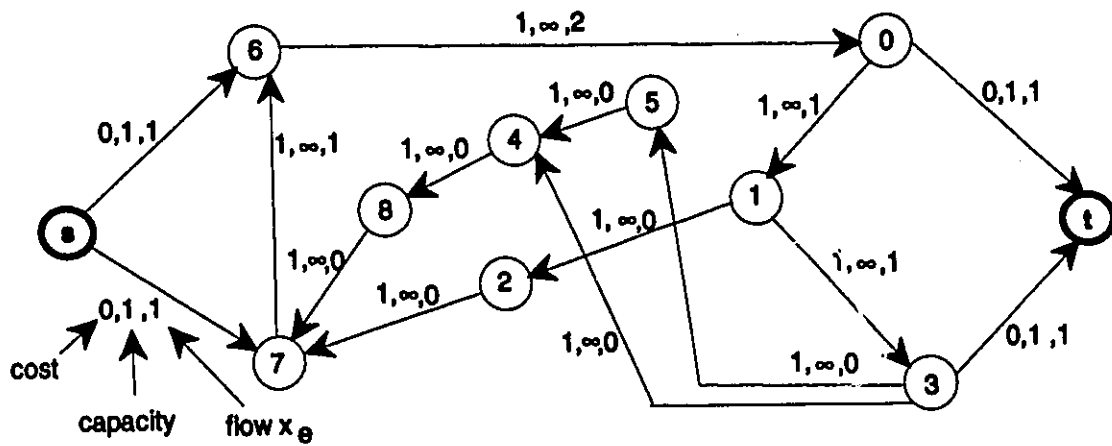
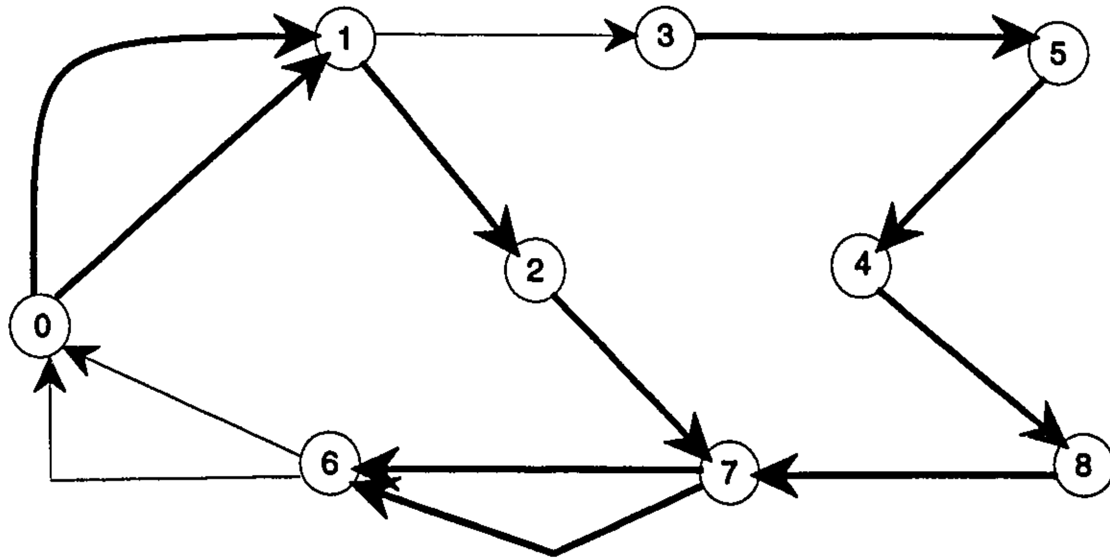


Figure 2.2 Corresponding flow problem $D_f=(V_f,E_f)$ with minimum cost maximum flow x shown.



In Ec

Not in Ec

Figure 2.3 RSA D* of D[Ec].

2.3 The synchronization problem in protocol testing

A *finite state machine* (FSM) [12] can be defined as follows. It consists of a finite set of states, one of which is designated as the initial or start state, and some (maybe none) of which are designated as final states. A FSM also includes a set of posible inputs and a finite set of transitions which describe for a given state and for a given input which state to go to and which output to produce.

In this thesis we will use an FSM to represent the control portion of a communication protocol. A FSM model of a protocol is characterized by a quintuple $M=(S,I,O,\delta,\lambda)$, where:

S is a finite set of states {s1,s2,...,sn};
I is a finite set of inputs {i1,i2,...,iq};
O is a finite set of outputs {o1,o2,...,or};
δ is the transfer function $SxI \rightarrow S$;
λ is the output function $SxI \rightarrow O$.

A *state transition* t_{jk}=(sj,sk;i/o) is defined as a transition from state sj to state sk, where i is an input symbol, o an output symbol, and i/o is the label of the transition.

The aim of protocol testing is to demostrate that an implementation under test (IUT) conforms to the protocol specification it implements. This involves testing that every transition in the protocol specification is implemented correctly by the IUT; i.e., for each transition, checking that the IUT generates the expected output and transfers to the expected state upon receiving the input

associated with that transition. To this end , the tester(s) supply a sequence of inputs to the IUT to verify that they cause an expected sequence of outputs. This sequence of inputs is called a *test sequence*, and is generated using the protocol specification.

Various formal methods [18] have been proposed for generating test sequences from protocol specifications modelled as FSMs. We will use one of the major formal testing methods called the T-method. In the T-method , a *transition tour* (TT) of an FSM is generated which takes the FSM from its start state, executes every transition at least once, and then returns the FSM to the start state. The corresponding test sequence is obtained by simply taking the inputs corresponding to these transitions. This sequence is then applied to the IUT, and the outputs produced are checked against the expected outputs. Note that the T-method only detects output errors in the IUT, since the state that the FSM reaches after the execution of a transition is not checked.

In this thesis, we will consider the distributed test architecture [13,14,16], which consists of an upper tester U and a lower tester L. During testing the upper and the lower tester provide control and observation of the upper and lower service boundaries of the IUT respectively. These testers supply a sequence of inputs (the test sequence) to the IUT to verify that they cause the expected sequence of outputs.

In a test architecture which consists of an upper and a lower tester, a transition t_{ij} consists of the IUT in the state i receiving an input from the upper or the lower tester, then transfering to state j and sending an output to the upper tester

or lower tester or both, or in some cases sending no output at all. More precisely, each of the transitions will be one of the following types,i.e. *type*(t_{ij}):

R^{IL}: IUT receives an input from L and does not send any output (i.e., output is null).

R^{IU}: IUT receives an input from U and does not send any output (i.e., output is null).

$R^{IL}S^{IL}$: IUT receives an input from L and sends an output to L.

$R^{IU}S^{IU}$: IUT receives an input from U and sends an output to U.

$R^{IL}S^{IU}$: IUT receives an input from L and sends an output to U.

$R^{IU}S^{IL}$: IUT receives an input from U and sends an output to L.

$R^{IL}S^{IL,U}$: IUT receives an input from L and sends an output to L and U.

$R^{IU}S^{IU,L}$: IUT receives an input from U and sends an output to U and L.

Note that the upper and the lower testers do not communicate with each other. This means that when applying a test sequence to an IUT a problem may occur because one of the testers (upper or lower) may not be able to determine when to send an input to the IUT. More precisely, considering two consecutive transitions from the test sequence, one of the testers faces a *synchronization problem* if it did not take part in the first transition and if the second transition requires that it sends a message to the IUT. For example, if a transition of type $R^{IL}S^{IL}$ is followed by a transition of type R^{IU}, a synchronization problem occurs because the upper tester has no way of knowing when to send its input to the IUT.

Any two consecutive transitions t_{ij} and t_{jk} form a *synchronizable pair* of transitions if t_{jk} can follow t_{ij} without generating a synchronization problem. A

test sequence is called a *synchronizable test sequence* if any two consecutive transitions in this sequence form a synchronizable pair.

2.4 Modelling the synchronization problem as a correctly ordered postman tour

We will represent an FSM by a digraph D =(V,E) where V represents the states of the FSM, E represents the transitions of the FSM, and each edge has an input-output label i/o corresponding to the input-output label for the transition it represents. The problem of finding a shortest TT of an FSM representing a protocol specification corresponds to finding a CPT in D starting at a specified start vertex, where the cost on every edge is 1.

We say that a digraph D=(V,E) is *order specified* if for each edge e in E we specify a subset of the edges leaving tail(e) as the *eligible succesor* edges for e. We call a path in D *correctly ordered (CO)* if for every consecutive pair of edges ei and ek in the path, ek has been specified as an eligible successor of ei. A *correctly ordered tour,* a *correctly ordered postman tour(COPT),* a *correctly ordered Euler tour(COET),* and a *correctly ordered Chinese postman tour (COCPT)* are similarly defined.

Given a digraph D representing an FSM model of a protocol, we can ensure only synchronizable pairs of transitions occur in a test sequence by making D an order specified digraph in which the eligible successors of an edge into a vertex v would be the edges out of v representing transitions for which no synchronization problem occurs. Thus, a synchronizable test sequence corresponds

to a COPT in D which starts at a specified state. A minimum length test sequence corresponds to a COCPT in D which starts and ends at a specified state.

As in Wang[20], in order to specify the correct eligible successors for each edge in D, the following notation is convenient. For each vertex v in D=(V,E), the edges leaving v partition into 2 sets: $Leave^U[v]$, which contains the edges corresponding to transitions in which the IUT receives an input from the upper tester, and $Leave^L[v]$, which contains the edges corresponding to transitions in which the IUT receives an input from the lower tester. The edges arriving at v partition into 3 sets: $Arrive^U[v]$, which contains the edges corresponding to the transitions involving the upper tester only, $Arrive^L[v]$, which contains the edges corresponding to the transitions involving the lower tester only, and $Arrive^{U,L}[v]$, which contains the edges corresponding to the transitions involving the lower and upper testers. In other words, $\mathrm{Leave}^U[v]=\{e\in E$: head(e)=v and type(e) $\in \{R^{IU}, R^{IU}S^{IL}, R^{IU}S^{IU}, R^{IU}S^{IU,L}\}\}$, $\mathrm{Leave}^L[v]=\{e\in E$: head(e)=v and type(e) $\in \{R^{IL}, R^{IL}S^{IL}, R^{IL}S^{IU}, R^{IL}S^{IU,L}\}\}$, $\mathrm{Arrive}^U[v]=\{e\in E$: tail(e)=v and type(e) $\in \{R^{IU}, R^{IU}S^{IU}\}\}$, $\mathrm{Arrive}^L[v]=\{e\in E$: tail(e)=v and type(e) $\in \{R^{IL}, R^{IL}S^{IL}\}\}$ and $\mathrm{Arrive}^{U,L}[v]=\{e\in E$: tail(e)=v and type(e) $\in \{R^{IU}S^{IL}, R^{IL}S^{IU}, R^{IU}S^{IU,L}, R^{IL}S^{IL,U}\}\}$.

From the definitions above, it follows that

(i) the eligible successors for edges in $\mathrm{Arrive}^L[v]$ are the edges in $\mathrm{Leave}^L[v]$,

(ii) the eligible successors for edges in $\mathrm{Arrive}^U[v]$ are the edges in $\mathrm{Leave}^U[v]$,

(iii) the eligible successors for edges in $Arrive^{U,L}[v]$ are the edges in $Leave^{U}[v]$ and $Leave^{L}[v]$.

Chapter 3

A heuristic for finding a synchronizable test sequence for protocol conformance testing

In this chapter we examine the problem of finding a minimum length synchronizable test sequence for protocol conformance testing . As discussed in Chapter 2, this can be modelled as the problem of finding a correctly-ordered Chinese postman tour (COPT) in a digraph D starting at a specified vertex v_0. Since this problem is NP-hard[5], it is unlikely that we will find an efficient algorithm to solve this problem exactly. We describe a heuristic for this problem. This heuristic is based on the idea of creating an auxillary digraph D' with a specified edge set Ec and edge e*, which has the property that there exists a COPT in D starting at v_0 if and only if there exists a Rural Postman Tour (RPT) over Ec in D' which uses edge e* exactly once. After creating D', we check to see if D'[Ec] is weakly connected, and if not, add edges to Ec to make it weakly connected. We provide a method for adding these edges, which tries to minimize the length of the resulting RPT, while not affecting the feasibility of the problem.

Once D'[Ec] is weakly connected, we use the methods described in Section 2.2 to create a RSA D* of D'. If we cannot create D*, then we show there is no RPT in D' and therefore no COPT in D starting at v_0. Finally, we obtain a RPT of D' over Ec from D*, and create the corresponding COPT in D starting at v_0 which corresponds to a synchronizable test sequence.

Note that the heuristic described here is infact an improved version of the heuristic developed by Wang in [20]. In Chapter 5 we describe Wang's heuristic and discuss the differences between the methods. For ease of comparison, we will use Wang's notation and definitions wherever possible in the following description of our algorithm.

3.1 Creating the duplex digraph

Given an ordered specified digraph D = (V,E) with specified vertex v_0, we now describe how to create a corresponding duplex digraph D' = (V',E'), where V' = $V^U \cup V^L \cup V^{U,L}$ and E' = Ec $\cup$ F. Without lost of generality, we will assume that D has no isolated vertices. In D', the edges in Ec are in one-to-one correspondence with the edges in D, and for any path P' in D', if we take the path P in D formed by the edges corresponding to Ec $\cap$ P', then P is correctly ordered (CO) in D.

We know that all edges leaving a vertex v in D can be partitioned into two sets: $Leave^U[v]$ and $Leave^L[v]$. All the edges directed into v have either the edges in $Leave^U[v]$ or $Leave^L[v]$ or both as their eligible successors. Thus, in the duplex digraph D', we replace each vertex v by three vertices: v^U, v^L, and $v^{U,L}$. The edges directed out of v^U in D' correspond to $Leave^U[v]$, and the edges directed out of v^L in D' correspond to $Leave^L[v]$. The edges directed into v^U in D' correspond to those whose eligible successors are edges in $Leave^U[v]$, and the edges directed into v^L in D' correspond to those whose eligible successors are edges in $Leave^L[v]$. Edges directed into $v^{U,L}$ in D' correspond to edges whose eligible successors are edges in $Leave^U[v]$ and $Leave^L[v]$. We then also add two edges

$(v^{U,L}, v^U)$ and $(v^{U,L}, v^L)$ so that we can traverse from $v^{U,L}$ to either set of edges.

Note that it is unnecessary to create a "split vertex" $v^{U,L}$ in D' if $Leave^U[v]$ or $Leave^L[v]$ is empty. We handle this situation in steps (e) and (f) of the construction described below. Also, it is not always neccessary to create a vertex v^U or v^L. In such cases these vertices will be isolated in D' and we remove them in step (g) of the construction.

Stage 1 of the duplex digraph D'=(V',E') construction:

(a) For every vertex v in V, we create a vertex v^U in V^U and a vertex v^L in V^L.

(b) For every vertex v in V, if $Arrive^{U,L}[v]$ is not empty, then we create a vertex $v^{U,L}$ in $V^{U,L}$ and two directed edges $(v^{U,L},v^U)$ and $(v^{U,L},v^L)$ in F.
We call $(v^{U,L},v^U)$ and $(v^{U,L},v^L)$ the *child edges.* Also, edges $(v^{U,L},v^U)$ and $(v^{U,L},v^L)$ are called *twins.*

(c) For each edge $(u,v) \in Arrive^U[v]$ (which implies $(u,v) \in Leave^U[u]$), we create a directed edge, in Ec from u^U to v^U. Similarly, for each edge $(u,v) \in Arrive^L[v]$, we create a directed edge in Ec from u^L to v^L.

(d) For each edge $(u,v) \in Arrive^{U,L}[v]$, we create a directed edge in Ec from u^U to $v^{U,L}$ if $(u,v) \in Leave^U[u]$, and if $(u,v) \in Leave^L[u]$, we create a directed edge in Ec from u^L to $v^{U,L}$. We call all such edges *parent edges.*

(e) For each vertex $v^{U,L} \in V^{U,L} \backslash \{v_0{}^{U,L}\}$, if $Leave^U[v]$ is empty, then direct all edges whose tail is $v^{U,L}$ into v^L, and remove $v^{U,L}$ and the twin edges out of $v^{U,L}$.

(f) For each remaining vertex $v^{U,L} \in V^{U,L} \backslash \{v_0{}^{U,L}\}$, if $Leave^L[v]$ is empty, then direct all edges whose tail is in $Arrive^{U,L}[v]$ into v^U, and remove $v^{U,L}$ and the twin edges out of $v^{U,L}$.

(g) Remove all isolated nodes in D'.

(h) Give each edge in Ec cost one and give each edge in F cost zero.

(i) Let $E'=Ec \cup F$, and let $V'=V^U \cup V^L \cup V^{U,L}$.

The edge set E' consists of the twin edges and the edges in Ec, where the edges in Ec and the edges in D are in one-to-one correspondence. We call an edge e in Ec which is not a parent edge an *inherent edge.*

3.2 Characteristics of the duplex digraph

Lemma 3.1: An ordered-specified digraph D=(V,E) with specified vertex v_0 has a COPT of length k starting at v_0 if and only if there exists a RPP P of cost k over Ec in D' with end vertices in $\{v_0{}^L, v_0{}^U\}$.

Proof: Suppose we have a COPT T in D. From the way the duplex digraph is created, we know that each edge e in D is represented in D' by either a single inherent edge in Ec or by a parent edge p. In the latter case, the tail of this edge is adjacent to a pair of twin edges. Since the twin edges have the same head vertex but different tail vertices, as long as the tail vertex in D' and the edge e in D are specified, the segment in D' corresponding to e can be determined. This segment will consist of the twin edge whose tail is the specified tail vertex, and the parent

edge p. If edge e in D corresponds to an inherent edge e' in Ec, then we will let the segment corresponding to e in D' be simply the edge e'.

Let ei be the ith edge of T for i=1,2,...,k, where k is the length of T. To obtain path P, we construct segments {si: i=1,2,...,k} as follows.

Step 1. If ek gives rise to an inherent edge in D', we select the tail vertex of the inherent edge (i.e., v_0^U or v_0^L) as the last vertex of the path P, i.e., the vertex at which P ends. Otherwise ek corresponds to a parent edge in D', and we select v_0^U as the last vertex of P. Initially, let i=k and v_{last} be the last vertex in P. Go to step 2.

Step 2. Construct segment si in D' corresponding to edge ei in D with the specified tail vertex v_{last}. Go to Step 3.

Step 3. If i=1 then stop. Otherwise, update v_{last} to be the head vertex of si and decrement i by one . Go to Step 2.

The path P is readly constructed from segments {si: i=1,2,...,k} as

P=s1 @ s2 @ ... @ sk,

where @ is used to concatenate the segments.

Since each edge e' in Ec is constructed for some edge e in E, and e=ei for some i=1,2,...,k in T, e' is contained in si. Hence P contains every edge in Ec, where each of these edges have cost 1. Since all other edges in P have cost zero, P is a RPP of cost k over Ec in D' with end vertices in {v_0^U, v_0^L}, as required.

Now suppose there is a RPP P' in D' over Ec with cost k and end vertices in $\{v_0^U, v_0^L\}$. We know from the way the duplex digraph is created that there is a one-to-one correspondence between the edges in Ec and the edges in D such that if we take the path P in D formed by the edges corresponding to Ec $\cap$ P', then P is correctly ordered. Since P' traverses all the edges in Ec and P' starts at v_0^U or v_0^L and ends at v_0^U or v_0^L, the corresponding path P in D will be a COPT in D of length k which starts and ends at v_0. ■

In our algorithm for finding a RPP in D' with end vertices in $\{v_0^L, v_0^U\}$, it will be much more convenient to search for a rural postman tour rather then a path. In order to make this possible, we will need to add an edge e* to D' in some cases, where e* is (v_0^L, v_0^U) or (v_0^U, v_0^L). When this is not necessary (i.e. when D' is strongly connected), we will simply add the loop $e^*=(v_0^L, v_0^L)$ or $e^*=(v_0^U, v_0^U)$. Later we will show that there exists a RPP in D' if and only if there exists a RPT in D' $\cup$ {e*} which uses e* exactly once.

Lemma 3.2: If a duplex digraph D'=(V',E') has a RPP over Ec with end vertices in $\{v_0^U, v_0^L\}$, then either D' is strongly connected, or D' with one of the edges (v_0^U, v_0^L) or (v_0^L, v_0^U) added is strongly connected.

Proof: Suppose there exists a RPP P over Ec in D', which starts at a vertex x and ends at vertex y, where $x,y \in \{v_0^U, v_0^L\}$.

Since the subgraph D'[Ec] is a spanning subgraph, P visits every vertex $v \in V'$ in D'. Thus if x=y, D' is strongly connected (Note this will be the case when only one of v_0^U or v_0^L exist). If $x \neq y$, then if we add an edge from x to y, clearly

every pair of distinct vertices will have a path between them. Therefore, D' with the edge (x, y) is strongly connected . ■

Stage 2 of the duplex digraph D'=(V',E') construction:

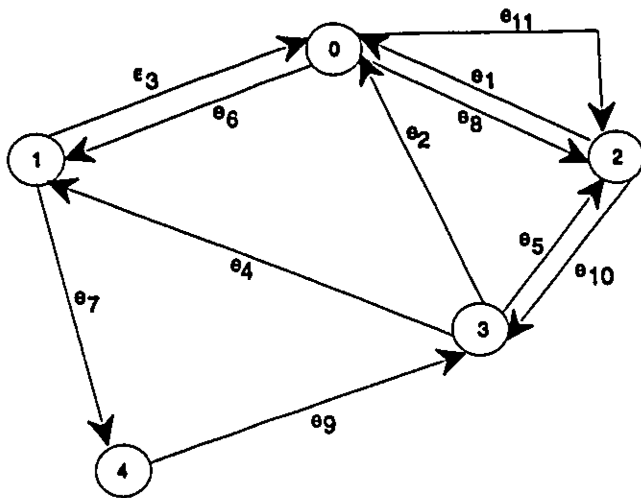
a) Check to see if D' is strongly connected. If it is, then let $e^*=(v_0^L, v_0^L)$ and go to step d). Note that if v_0^L does not exist, we will simply add $e^*=(v_0^U,v_0^U)$ instead.

b) Check to see if D' with edge (v_0^L, v_0^U) added is strongly connected. If it is, let $e^*=(v_0^L, v_0^U)$ and go to step d).

c) Check to see if D' with edge (v_0^U, v_0^L) added is strongly connected. If it is, then let $e^*= (v_0^U, v_0^L)$. If it is not, then by Lemma 3.1 and Lemma 3.2, there does not exist a COPT in D.

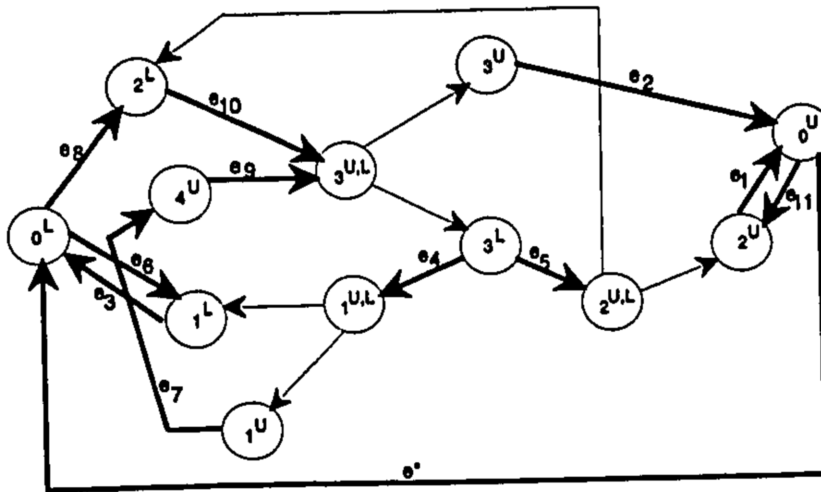
d) Let $Ec=Ec \cup \{e^*\}$, and let e^* have cost 0.

Figures 3.1 and 3.2 shows an example of the duplex digraph construction, Stages 1 and 2.



Edge	Type	Eligible Successor
e1=(2,0)	$R^{IU}S^{IU}$	e11
e2=(3,0)	$R^{IU}S^{IU}$	e11
e3=(1,0)	$R^{IL}S^{IL}$	e6, e8
e4=(3,1)	$R^{IL}S^{IU,L}$	e3, e7
e5=(2,0)	$R^{IL}S^{IU}$	e6, e8,e11
e6=(0,1)	$R^{IL}S^{IL}$	e3
e7=(1,4)	$R^{IU}S^{IU}$	e9
e8=(0,2)	$R^{IL}S^{IL}$	e5, e10
e9=(4,3)	$R^{IU}S^{IL,U}$	e2, e4
e10=(2,3)	$R^{IL}S^{IU}$	e2, e4
e11=(0,2)	$R^{IU}S^{IU}$	e1

Figure 3.1 Digraph D with v^0 =vertex 0.



edges in Ec (cost 1)

edges not in Ec (cost 0)

Duplex Digraph D'=(V',E')

Figure 3.2 An example of the duplex digraph construction.

3.3 An Algorithm for synchronizable test sequence generation

We now describe an algorithm which, given an ordered specified digraph D originating from a synchronizable test sequence problem, either finds a COPT in D starting at v_0, or determines that no such tour exists. In the algorithm we first construct the duplex digraph D' as previously described, and then construct a RPT in D' over Ec which uses the specific edge e* (from Stage 2 of the construction) exactly once. Later this edge will be deleted to give us a RPP in D' over Ec with end vertices in $\{v_0^U, v_0^L\}$, and hence a COPT in D by Lemma 3.1.

In order to construct a RPT in D' over Ec, we must construct a RSA D* of D'[Ec] which is strongly connected. Note that since D' is strongly connected (by Stage 2 of the construction), a RSA of D'[Ec] can be obtained by solving a minimum-cost maximum flow problem as described in the background section.We will ensure e* is used only once in the RPT we create by giving it a capacity of zero in the flow problem. Note that if the RSA D* is strongly connected, then it follows from Theorem 2.3 that we have found a RCPT of D' over Ec, and thus a COCPT of D. However D* may not be strongly connected.

Recall from Theorem 2.2 that if the edge-induced subgraph D'[Ec] is weakly connected, then any RSA D* of D'[Ec] is strongly connected. Thus one way to ensure D* is strongly connected is to add edges to Ec, obtaining Ec' such that D'[Ec'] is weakly connected. Since $E' = Ec \cup F$, these added edges are necessarily child edges. Clearly we wish to add these edges in such a way as to add as few edges as possible while at the same time ensuring that the graph D'[Ec'] is as symmetric as possible. However, we must be careful in doing this, because adding certain edges to Ec will create a new RPT problem for which there is no

solution. For example, in any directed cut of D'\{e*}, we want to be sure that Ec' only contains one edge from this cut.

To find the edges to add to Ec, we first decompose D'\{e*} into strongly connected components. If these strongly connected components are not ordered, then clearly there is no RPP over Ec in D'\{e*} and hence no COPT in D by Lemma 3.1. Otherwise, let the strongly connected components be called S_1, $S_2,...,S_k$ for some k, $k\geq 1$, such that for any edge (a,b), if $a\in S_i$, then $b\in S_j$, $j\geq i$. Then, we find a minimum cost spanning tree T_i in every component S_i (with the directions on the edges ignored), where we assign to each edge e in Ec cost zero, and we assign each edge e' in F' cost 0, where F' is the set of edges in E'\Ec whose head vertices have out-degree equal to 1 and/or whose tail vertices have in-degree equal to 1. Note that such edges will automatically be traversed by any RPT over Ec found for D'. The rest of the edges e' in E'\(Ec∪F') are assigned cost 1 if the addition of that edge to Ec∪F' will make D'[Ec∪F'] more symmetric at both ends (i.e. if d_{out}(head(e')) in D'[Ec∪F'] is less than the d_{in}(head(e')) in D'[Ec∪F'] and the d_{in}(tail(e')) in D'[Ec∪F'] is less than the d_{out}(tail(e')) in D'[Ec∪F']), otherwise we assign e' cost 2. We then let Ec'=Ec∪E(T_1)∪E(T_2)...∪E(T_k). Finally, for any directed cut S_i to S_{i+1} which contains no Ec edges, we find a "cheapest" child edge in this cut (using the costs describe above) and add this edge to Ec'.

Since D'[Ec'] is weakly connected, we can next obtain a RSA D* of D'[Ec'] by solving a minimum cost maximum flow problem $D_f=(V_f, E_f)$ as described in Section 2.2. We have the following result.

Theorem 3.3: There exist a flow $x = (x_e : e \in E_f)$ for the flow problem $D_f=(V_f, E_f)$ which saturates all the edges incident to s and t if and only if D' has a RPT over Ec using e* exactly once.

Proof: Suppose there exist a flow $x=(x_e : e \in E_f)$ for D_f which saturates all the edges incident to s and t. Therefore, the flow x_e for each edge e incident to s or to t is equal to the capacity of e. Hence, the digraph D* formed by replicating each edge e' in D' $x_{e'}$ times and adding in the edges in Ec' is symmetric and contains each edge in Ec' at least once. Since $e^* \in$ Ec' and has capacity 0 in D_f, it is contained in D* only once. From D* we can obtain a RCPT of D' over Ec' which uses e* exactly once, and hence a RPT of D' over Ec which uses e* exactly once.

Now suppose D'=(V',E') has a RPT over Ec which uses e* exactly once. Let D*=(V*,E*) be the symmetrc digraph obtained from the RPT; i,e. let V*=V' and let E* be the edges in the RPT in D'(with multiple copies included). We modify D* to create a new digraph which is symmetric and contains Ec' as follows. Let E" be the edges in Ec'\Ec which are not in the RPT and consider edge e=(x,y) in E". If there exists a path P from y to x in D'\{e*} we add e and P to D*. Note that D* is still symmetric. If there does not exist a path from y to x in D'\{e*}, then let X be the set of all vertices v such that there exists a path from v to vertex x in D'\{e*}, and let Y be the set of all vertices v such that there exists a path from vertex y to v in D'\{e*}.

Claim 1. $\delta(X)$ is a directed cut of D'\{e*}.

Proof: Suppose there exists a vertex k in both sets X and Y. Then there exists a path P from y to x which visits the vertex k, and we obtain a contradiction. Thus the sets X and Y form a partition of the vertices.

Suppose there exists an edge (a,b) such that $a \in Y$ and $b \in X$. By definition of the sets X and Y, there exist a path from b to x, and a path from vertex y to a, and thus a path from y to x, which leads to a contradiction. This completes the proof of Claim 1.

Claim 2. For any directed cut $\delta(U)$ in $D' \backslash \{e^*\}$ if $e \in Ec' \backslash Ec$ is in the cut, then this implies that there is no other edge in Ec' in the cut.

Proof: By our construction of Ec', we know that if $e \in Ec' \backslash Ec$ is in a directed cut of D'/e^*, then it is in a directed cut from S_i to S_{i+1} which contains no Ec edges. Furthermore, it is the only edge in $Ec' \backslash Ec$ in this cut. This completes the proof of Claim 2 .

Since D'[Ec] is spanning, we know by Claim 1 that the RPT in D' over Ec must cross the cut $\delta(X)$ in $D' \backslash \{e^*\}$ using an edge (u,v) such that $u \in X$, $v \in Y$. By Claim 2, we know that the edge (u,v) is not in Ec', therefore we can remove the edge (u,v) form D* and add the edge (x,y), a path from u to x, and a path from y to v (these paths exist by the definitions of the sets X and Y) to our digraph D* and still have a digraph which is symmetric, covers the same subset of Ec' as before, and also now covers edge e=(x,y). By repeating the above modifications to D* for all edges $e \in E''$, we obtain a digraph D* which is symmetric, strongly connected, uses every edge in Ec' at least once, and uses e* exactly once. Remove one copy of each edge $e \in Ec'$ from D* to obtain D**. We create a feasible flow $x=(x_e: e \in E_f)$

for the flow problem $D_f=(V_f,E_f)$ from D** as follows. For each edge $e \in E_f$ which corresponds to an edge in D', let x_e equal the number of copies of edge e in D**. Let the value of x_e equal the capacity for edges e in D_f directed out of s and edges e directed into t. Then x is a feasible flow for the flow problem D_f, moreover it saturates all the edges out of s and into t as required. ■

Let D* = (V*,E*) be the RSA of D'[Ec'] obtained from the flow x found for D_f . Then for each vertex $v \in V^*$, $d_{in}(v) = d_{out}(v)$, where V*= V' and E* contains every edge in Ec' at least once and every edge in E'\Ec zero or more times such that the total cost of the edges in E* is minimized. Then from Theorems 2.2 and 2.3, the RSA D*has an Euler tour which is a RCPT T* of D' over Ec'. When we remove e*, we have a RCPP over Ec' with end vertices in $\{v_0^U,v_0^L\}$, and this corresponds to a RPP over Ec in D', and a COPT T in the digraph D starting at v_0 . T is obtained from T* by removing vertices in $V^{U,L}$ (and child edges) from T*, removing and eliminating the superscripts from the upper and the lower vertices.

Given an order specified digraph D, we propose the following algorithm for constructing an RPP P* in D' over Ec with end vertices in $\{v_0^U , v_0^L\}$.

Step 1. Construct the duplex digraph D' using Stages 1 and 2 as described previously (page 23). Note that we may conclude in Stage 2 that no COPT exists in D.

Step 2. Check to see if D'[Ec] is weakly connected. If not we find a set of edges in E'\Ec to add to Ec to make D'[Ec] weakly connected in the following way:

i) Let F' be the edges $e=(u,v)$ in E'\Ec such that $d_{out}(u)$ is 1 and/or $d_{in}(v)$ is 1.

ii) We define a cost function C' for every edge $e=(u,v)$, where

- If $e \in Ec \cup F'$, then C'e is zero.
- If $d_{out}(u)$ in D'[Ec∪F'] is less than $d_{in}(u)$ in D'[Ec∪F'] and $d_{in}(v)$ in D'[Ec∪F'] is less than $d_{out}(v)$ in D'[Ec∪F'], then C'e is 1.
- Otherwise, C'e is 2.

iii) Let Ec'=Ec.

iv) Decompose D'\{e*} into strongly connected components. If these components are not ordered, then there is no RPP in D'\{e*} over Ec and hence, by Lemma 3.1, no COPT in D starting at v_0. Let the ordered strongly connected components be $S_1, S_2, \ldots, S_k$ for some k, $k \geq 1$.

v) For each component S_i, find a minimum cost spanning tree T_i for the underlying undirected graph using the cost function C'. Let $Ec' = Ec' \cup T_i$.

vi) For each directed cut S_i to S_{i+1}, check to see if any Ec edges are in the cut. If not choose the edge e in the cut with minimum cost c'_e and add it to Ec'.

Step 3. Generate the RSA D* of D'[Ec'] by solving the minimum cost maximum flow problem on graph $D_f=(V_f,E_f)$ as described in the background section. In this flow problem, the edge e* is given capacity zero. Let $x=(x_e : e \in E_f)$ be the minimum cost maximum flow found. If x saturates all the edges incident with s, then we obtain a strongly connected RSA of D' [Ec'] which contains e* exactly once, and we go to step 4. Otherwise, by Theorem 3.3 and Lemma 3.1 we conclude that there exists no COPT inD starting at v_0.

Step 4. Find a RCPT in D* over Ec'. Note that it will traverse e* exactly once.

Step 5. Delete the edge e*. This will give a RPP in D* over Ec with end nodes in $\{v_0^U, v_0^L\}$.

Step 6. Find the corresponding COPT in D using the method described in Lemma 3.1.

We will now demonstrate the above heuristic on the digraph D shown in Figure 3.1, whose duplex digraph D' is shown in Figure 3.2. Figure 3.3 demonstrate Step 2 of the heuristic , showing the strongly connected components S_1, S_2, S_3 of D', the costs C', and the edges which are added to Ec to make D[Ec] weakly connected. Figure 3.4 shows the flow problem $D_f=(V_f,E_f)$ used to find the RSA D* of D[Ec'] and the minimum cost maximum flow x. Figure 3.5 shows the RSA D* of D'[Ec']. The RCPT of D' over Ec' is 0^L, 1^L, 0^L, 2^L, $3^{U,L}$, 3^L, $2^{U,L}$, 2^L, $3^{U,L}$, 3^L, $1^{U,L}$, 1^U, 4^U, $3^{U,L}$, 3^U, 0^U, 2^U, 0^U, 0^L, and from this we obtain the COPT 0,1,0,2,3,2,3,1,4,3,0,2,0 for D starting at v_0 .

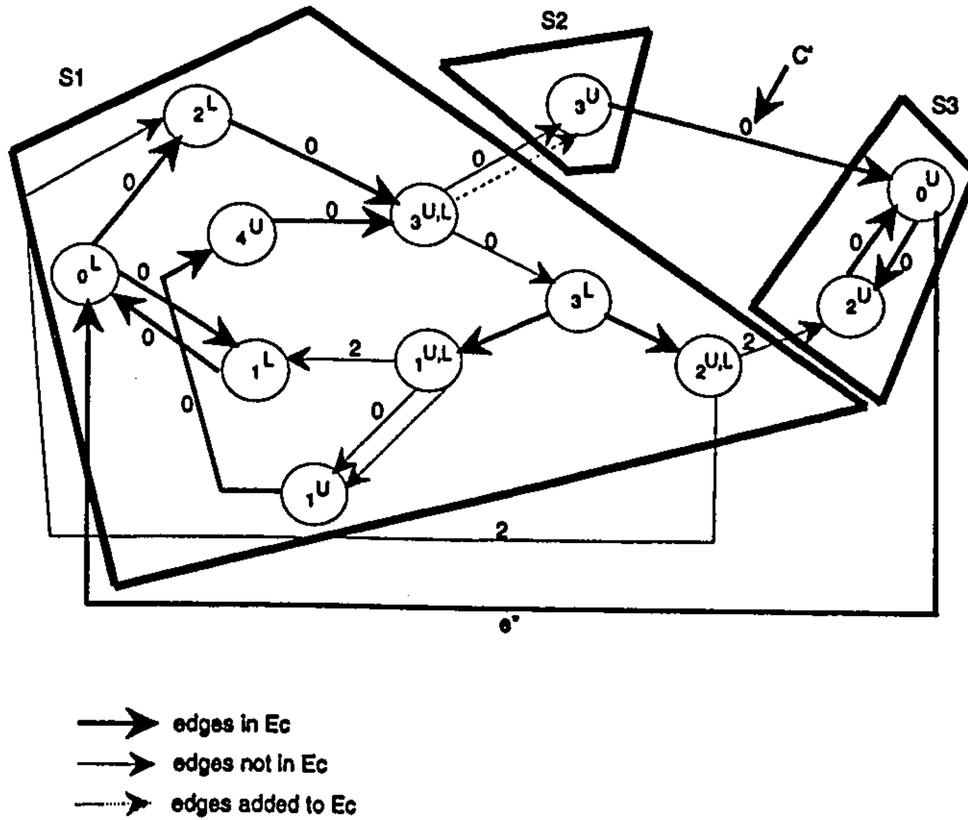


Figure 3.3

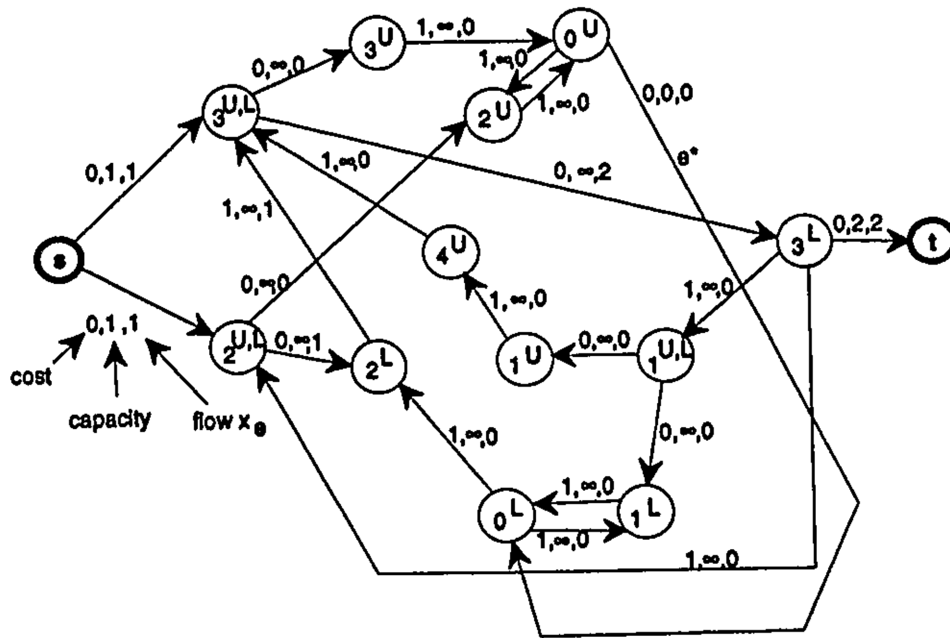


Figure 3.4 Flow problem $D_f=(V_f, E_f)$

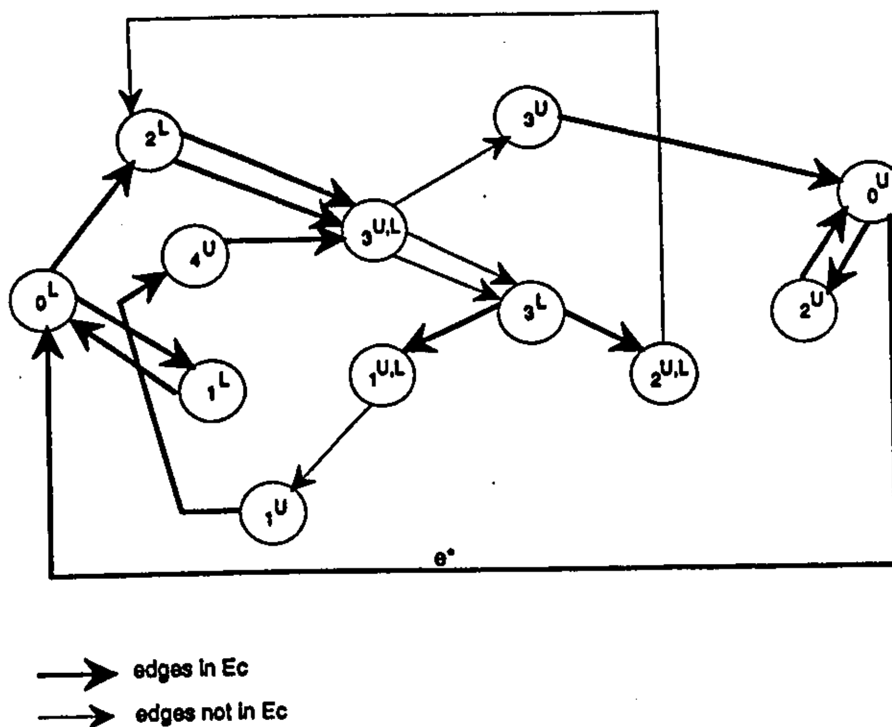


Figure 3.5 The RSA D* of D'[Ec']

Chapter 4

Implementation

4.1 Digraph generator GENGRAPH

In order to test our implementation, we require a way to randomly generate test digraphs. GENGRAPH is a program written in Turbo Pascal Version 6.0 which generates order-specified digraphs which have no loops, and for which a COPT does exist. The user specifies the following parameters:

NUMNODE: The number of vertices in the digraph.
LENGTH: The length of the main path generated.
NUMPATH: The number of paths to be added to the main path.
DIST: The maximum length of an added path.

GENGRAPH first creates a correctly-ordered main path P of length LENGTH by randomly generating the vertices along P, and ensuring the type of each edge (also randomly generated) results in P being correctly ordered. Next, GENGRAPH generates and adds NUMPATH correctly ordered paths to P, each of length <= DIST. Each of these paths Q start at some vertex u in P and end at some vertex v in P such that v comes before u in P. These start and end vertices are randomly generated for each path Q. Also, the edge-type for the first edge in Q is chosen such that it is an eligible successor for the edge into u in P, and the edge-

type for the last edge in Q is chosen such that the edge out of v in P is an eligible successor for it.

A copy of the program GENGRAPH can be found in Appendix A.

4.2 Synchronizable test sequence generator STSG

STSG is a program written in Turbo Pascal version 6.0, which is used to find a COPT for a given order-specified digraph D stored in a file called Digraph.txt.

Our computer program stores all the information contained in the text file Digraph.txt in a matrix called Orig_probl. The row indices in this matrix are the edges_type and the column indices are the number label of the vertices. This matrix contains in each position Vi an array of records containing all the information of the edges incident with vertex Vi.

The first step of our algorithm is to create a duplex digraph D'. Our computer program will create D' by inserting all the information available into a matrix called matrix_duplex. The row indices in this matrix are the vertex_type and the column indices are the number labels of the vertices. This matrix contains in each position Vi a record whose fields are used by different procedures of our computer program. This record also includes two arrays, the array in_vertices which contains all the information of the edges going into Vi and the array out_vertices which contains the information of the edges coming out of Vi.

A vertex Vi is in the duplex digraph D' if the field active in the position Vi is equal to true. Othewise, we ignore that position of the matrix in all the procedures of our program.

A copy of the program STSG can be found in Appendix A.

4.3 Description of algorithms implemented

Our program STSG uses several standard algorithms to accomplish all the steps needed in our algorithm to generate a synchronizable test sequence. These algorithms have been adapted to our data structure, which in most cases is different from the one used originally in the algorithms.

First, we have used an algorithm described in [2,10] to decompose D' into strongly connected components. This procedure is called STRONGCOMPONENTS in our program.

After decomposing the digraph D' into strongly connected components, we use the procedure CHECK_STRONGLY_CONNECTED to check if D' is strongly connected after adding an extra edge e*. this is done by using the STRONGLYCOMPONENTS procedure and checking if there is only one strongly connected component in D'. If this is not the case we exit the program.

Our next algorithm is Minimum Spanning Tree (Dijkstra/Prim) [2]. We find a minimum spanning tree for each strongly connected component. Our computer program uses the MST_EDGE_COST procedure to assign costs to the edges.

The procedure LINK_EDGES finds the edges linking a component Si and a component Si+1 of minimum cost, by comparing the costs of all the edges going into Si+1 that are comming from Si.

The next step in our computer program is to add an extra edge e* and calculate the net degree of each vertex in order to continue with our next procedure which is the minimum cost max flow . Here we use the Busacker-Gowen algorithm [9]. In the minimum cost max flow procedure we will use an algorithm proposed by Ford and Bellman[9] which calculates the shortest paths between a vertex s and all the other vertices. We will use an auxiliary matrix instead of the duplex_matrix to store all the information in this procedure. We do not add a vertex s and a vertex t to our matrix. The flow coming from s and the flow going to t are stored in a field called flow_from_s and flow_to_t, respectively.

The last two procedures in our computer program are RPT (see [15]) and COPT. The RPT procedure uses a standard greedy algorithm to find the Euler tour for the RSA of D'[Ec] we have created. It creates a list called vertex_list with the RPT of D'. The COPT procedure creates a text file with the COPT in D and the length of this COPT.

Chapter 5

Performance

5.1 Complexity analysis

Recall that our original order-specified digraph is D=(V,E) and D'=(V',E') is the corresponding duplex digraph. We have that $|V'| \approx 3|V|$ and $|E'| \approx |E|+2|V|$. The total time required for the work done in creating the duplex digraph D' is $O(|E'|)$. The total number of steps executed in decomposing D'\{e*} into strongly connected components is $O(|E'|)$. The worst case running time for the minimum cost spanning tree algorithm for all the strongly connected components is in $O(|V'||E'|)$. The number of instructions executed in finding the link edges in all directed cuts Si to Si+1 is $O(|E'|)$. Finding the shortest path using the Bellman-Ford algorithm takes time in $O(|V'||E'|)$. Let K be the set of nodes with net degree greater than zero. Since in every iteration of the minimum cost maximum flow algorithm the flow increases by at least 1, and

$$
\begin{aligned}
\Sigma\,(\text{capacities of edges } e : e \in \delta(s)) &= \Sigma\,(\text{net degrees of nodes } v: v \in K) \\
&\leq \Sigma\,(d_{in}(v): v \in K) \\
&\leq |Ec'|,
\end{aligned}
$$

we have that we will search for a flow augmenting path at most $O(|Ec'|)$ times. Since we use Bellman-Ford for each search for a flow augmenting path, the overall amount of work done in the minimum cost maximum flow algorithm is $O(|E'|^2|V'|)$. Finding the RPT in D' takes time $O(|E'||V'|)$ and obtaining the corresponding COPT in D takes time $O(|E'|)$. Hence the complexity of our

synchronizable test sequence generator algorithm is $O(|E'|^2|V'|)$. In terms of E and V this is $O(|E|^2|V|)$. (Note that a complexity analysis of each portion of the implementation is provided in Table 5.1.)

Complexity chart

Algorithm	*O( )*
Clean_orig_probl	$O(\|V\|)$
Fill_original	$O(\|E\|)$
Initiallize_duplex	$O(3\|V'\|)$
Create_Ec_edge	$O(\|E'\|)$
Remove_edges_and_vertices	$O(\|E'\|)$
StrongComponents	$O(\|E'\|)$
Check_ordered	$O(\|V'\|\|E'\|)$
MST_vertex_degree	$O(2\|E'\|)$
MST_edge_cost	$O(\|E'\|)$
Check_strongly_connected	$O(\|E'\|)$
MinSpanningTree	$O(\|V'\|\|E'\|)$
Link_edges	$O(\|E'\|)$
Add_extra_edge	Constant
Net_degree	$O(2\|E'\|)$
Bellman_Ford	$O(\|V'\|\|E'\|)$
Min_cost_maxflow	$O(\|E'\|^2\|V'\|)$
RPT	$O(\|E'\|\|V'\|)$
COPT	$O(\|E'\|)$

Table 5.1

5.2 Comparison with Wang's Heuristic

For comparison purposes and for completeness, we now describe Wang's heuristic for synchronizable test sequence generation[20]. Before Wang describes his algorithm for synchronizable test sequence generation, he describes an algorithm of complexity of $O(|E|^2)$, which determines if there exists of COPT in D. Wang then describes a heuristic for the problem of finding a synchronizable test sequence using the T-method for protocol conformance testing which can be modelled as the problem of finding a COPT in D.

Wang's heuristic

Step 1. Construct the duplex digraph D' from D.

Given a digraph D=(V,E) with specified vertex v_0 and for which a COPT exists, create a duplex digraph D'=(V',E'), where $V'=V^U \cup V^L \cup H$ and $E'=Ec \cup F$, as follows.

(a) For each vertex v in V, there are two sets of edges leaving v: $Leave^U[v]$ and $Leave^L[v]$. Create a vertex v^U in V^U if $Leave^U[v] \neq \emptyset$ and a vertex v^L in V^L if $Leave^L[v] \neq \emptyset$. In addition for vertex v_0, if $Leave^U[v_0]=\emptyset$ but $Arrive^U[v_0] \neq \emptyset$, create a vertex $v_0{}^U$ in V^U; similarly, if $Leave^L[v_0]=\emptyset$ but $Arrive^L[v_0] \neq \emptyset$, create a vertex $v_0{}^L$ in V^L.

(b) For each edge $(u,v) \in Arrive^U[v]$ (that implies $(u,v) \in Leave^U[u]$), create a directed edge, in Ec, from u^U to v^U. Similarly, for each edge (u,v) $\in Arrive^L[v]$, create a directed edge, in Ec, from u^L to v^L.

(c) For each edge $(u,v) \in Arrive^{U,L}[v]$, one of the following is performed:

i) In the case that vertex v^U exists but vertex v^L does not, create a directed edge, in Ec, from u^U to v^U, if $(u,v) \in Leave^U[u]$; if $(u,v) \in Leave^L[u]$, create a directed edge, in Ec, from u^L to v^U.

ii) In the case that vertex v^L exists but vertex v^U does not, create a directed edge, in Ec, from u^U to v^L, if $(u,v) \in Leave^U[u]$; if $(u,v) \in Leave^L[u]$, create a directed edge, in Ec, from u^L to v^L.

iii) In case that both v^U and v^L exist, create two subcases: If $(u,v) \in Leave^U[u]$, create, in H, a vertex $U_{u,v}$, a directed edge $(u^U,U_{u,v})$ in Ec, and two directed edges $(U_{u,v},v^U)$ and $(U_{u,v},v^L)$ in F. Similarly if $(u,v) \in Leave^L[u]$, create, in H, a vertex $L_{u,v}$, a directed edge $(u^L,L_{u,v})$ in Ec, and two directed edges $(L_{u,v},v^U)$ and $(L_{u,v},v^L)$ in F.

(d) Each edge in Ec is given cost one and each edge in F is given cost zero.

Step 2. Find the strongly connected components $\{H_i:i=1,2,...,k\}$ of D'. If k=1 then go to Step 3 else go to Step 4.

Step 3. Construct an RPT T* in D' over Ec starting either from v_0^U or v_0^L according to the method described in [20] in section 2.6.1. Since T* corresponds to a COPT in D, stop.

Step 4. Sort $\{H_i:i=1,2,...,k\}$ into $\{G_i:i=1,2,...,k\}$, the ordered decomposition of D'; obtain subgraph $D_i=(G_i,E_i)$, where $E_i = \{(u,v) \in E':u,v \in G_i\}$, and $E_{ic} = E_i \cap E_c$ for i=1,2,...,k.

Step 5. Determine $\Delta(G_i,G_{i+1})$ and remove redundant edges from $\Delta(G_i,G_{i+1})$ for i=1,2,...,k-1.

Step 6. Identify the links (x_i,y_i) from G_i toward G_{i+1} for i=1,2,...,k-1. Let $y_0=v_0^U$ and $x_k=v_0^L$ when $v_0^U \in G_1$ $v_0^L \in G_k$, otherwise let $y_0=v_0^L$ and $x_k=v_0^U$.

Step 7. Construct an RPP P_i in D_i over E_{ic} from y_{i-1} to x_i for i=1,2,...,k according to the method described in [20] in section 2.6/ Then construct an RPP T* in D" over Ec:

$T^*=P_1 @x_1,y_1 @ P_2 @x_2,y_2 @...@P_{k-1} @x_{k-1},y_{k-1}@ P_k$,

which corresponds a COPT in D.

Our heuristic for finding a COPT in D is infact an improved version of the heuristic developed by Wang [20] for this problem. Below we outline the major areas of improvement, and give an example which compares the two methods.

5.2.1 Checking for the existence of a COPT in D

Wang's heuristic requires the existence of a COPT in D starting at v_0. He checks for the existence of such a tour in a separate algorithm which creates what is called a "reachability matrix". The complexity of this algorithm is $O(|E|^2)$.

In comparison, our heuristic either finds a COPT in D starting at v_0, or determines that no such tour exists. Discovering that such a tour doesn't exist is a natural part of the heuristic, and does not increase the complexity of the algorithm. There are three places in our algorithm where we can discover that no COPT in D exists:

1) In Stage 2 of the construction of the duplex digraph, when edge e* is added.

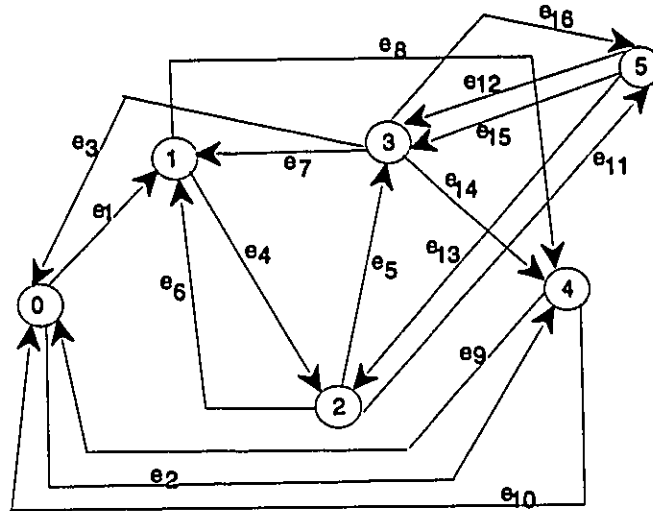
2) In step 2 of the heuristic , when we find the strongly connected components of D'\{e*}.

3) In step 3 of the heuristic, in the case where the flow found does not saturate the edges out of vertex s.

5.2.2 Creating a duplex digraph

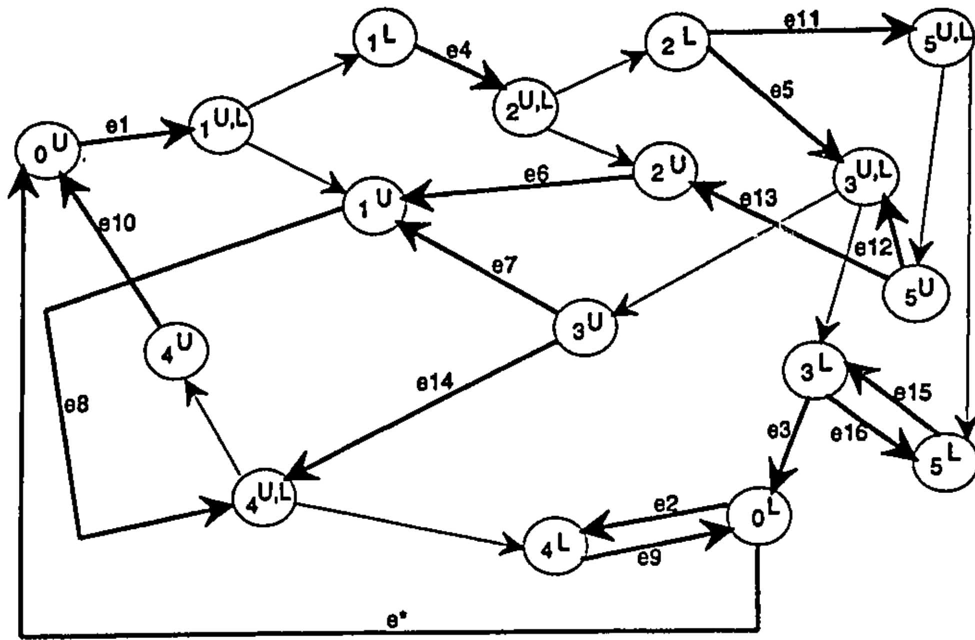
Both heuristics create a duplex digraph D', and search for a RPP over Ec in D'. The main difference in the formation of D' is that, whenever v^U and v^L exist, Wang creates a vertex $v^{U,L}$ and twin edges $(v_e^{U,L}, v^L)$, $(v_e^{U,L}, v^U)$ for every edge $e \in Arrive^{U,L}(v)$, whereas our D' has one such vertex $v^{U,L}$ and a set of twin edges for all edges in $Arrive^{U,L}(v)$. Hence Wang's duplex digraph contains $2|V|+\Sigma (|Arrive^{U,L}(v)| : v \in V)$ vertices, whereas ours contains $3|V|$ vertices. Not only does this means that Wang's duplex digraph D' is much larger, it also means that D'[Ec] potentially contains a much larger number of weakly connected components in D[Ec], which adversely affects the number of additional edges which must be added to Ec in the next stage of the heuristic, thus increasing the length of the tour which is found.

Figure 5.1 shows an order specified digraph D, and Figure 5.2 shows the duplex digraph D' constructed by each heuristic.



Edge	Type	Eligible Successor
e1=(0,1)	$R^{IU}S^{IL}$	e4, e8
e2=(0,4)	$R^{IL}S^{IL}$	e9
e3=(3,0)	$R^{IL}S^{IL}$	e2
e4=(1,2)	$R^{IL}S^{IU}$	e5, e6, e11
e5=(2,3)	$R^{IL}S^{IU}$	e3, e7, e14
e6=(2,1)	$R^{IU}S^{IU}$	e8
e7=(3,1)	$R^{IU}S^{IU}$	e8
e8=(1,4)	$R^{IU}S^{IL}$	e9, e10
e9=(4,0)	$R^{IL}S^{IL}$	e2
e10=(4,0)	$R^{IL}S^{IU}$	e1
e11=(2,5)	$R^{IL}S^{IU,L}$	e12, e13, e15
e12=(5,3)	$R^{IU}S^{IU,L}$	e3, e7, e14, e16
e13=(5,2)	$R^{IU}S^{IU}$	e6
e14=(3,4)	$R^{IU}S^{IU,L}$	e9, e10
e15=(5,3)	$R^{IL}S^{IL}$	e3
e16=(3,5)	$R^{IL}S^{IL}$	e15

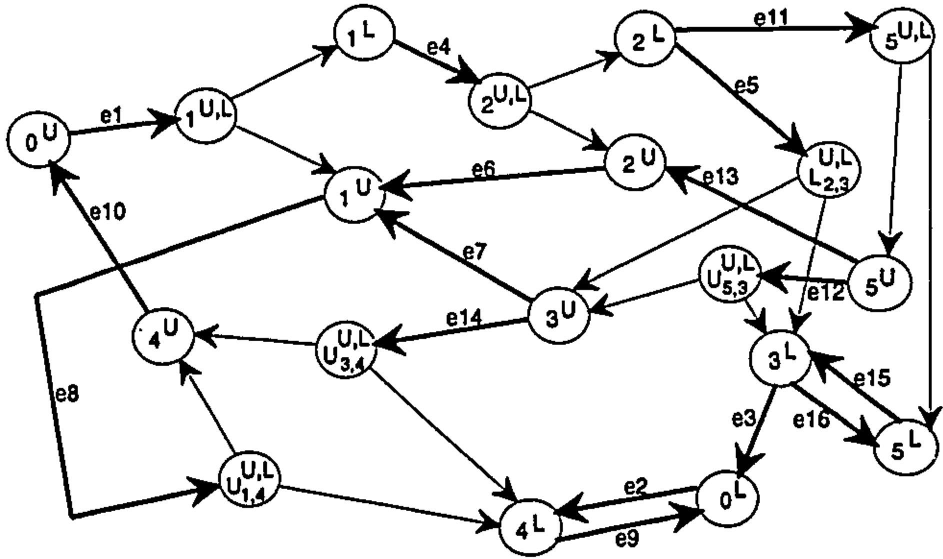
Figure 5.1 Digraph D.



In Ec

Not in Ec

Our duplex digraph D'



In Ec

Not in Ec

Wang's duplex digraph D'

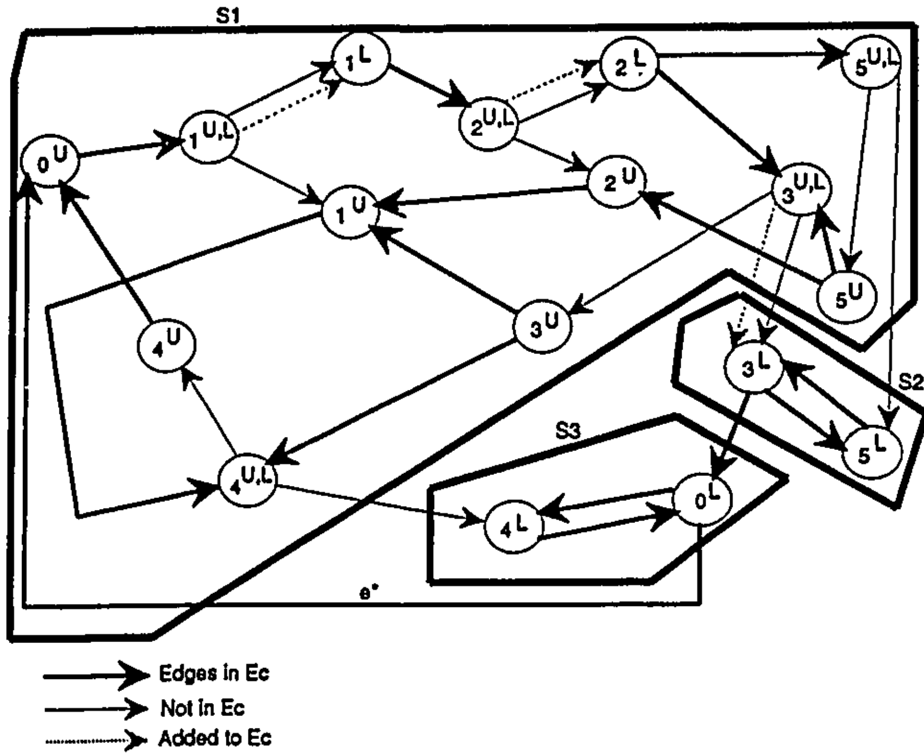
Figure 5.2

5.2.3 Finding the RSA D*

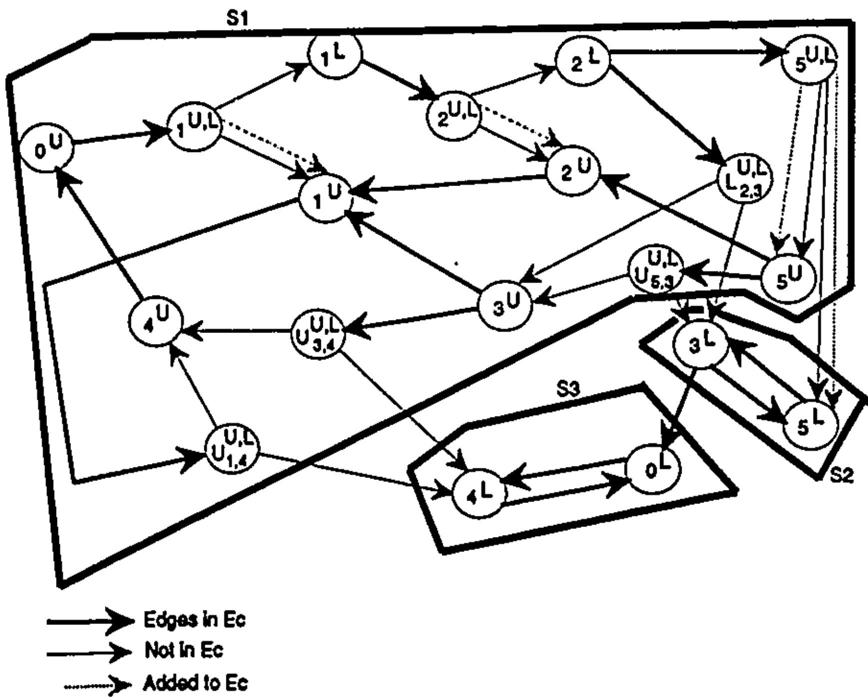
Both heuristics add edges to Ec to make D'[Ec] weakly connected, and then find the RSA D* of D'[Ec] by using a minimum cost maximum flow algorithm. In Wang's heuristic, a flow problem is formulated and solved for each strongly connected component in D'[Ec], whereas we formulate and solve a single flow problem, which is more time efficient.

In adding edges to Ec to make D'[Ec] weakly connected, Wang chooses the edges randomly. We attempt to add edges in such a way as to add as few edges as possible, while keeping D'[Ec] as symmetric as possible. This is an important improvement over Wang's heuristic, as any edge added to Ec now must be traversed by the RPP found for D', which directly affects the length of the COPT found for D.

Figure 5.3 shows the edges added to Ec by each of the heuristics. In choosing edges to add for Wang's heuristic, we have tried to make "bad" choices to illustrate the effect that this can have.



Our digraph D'

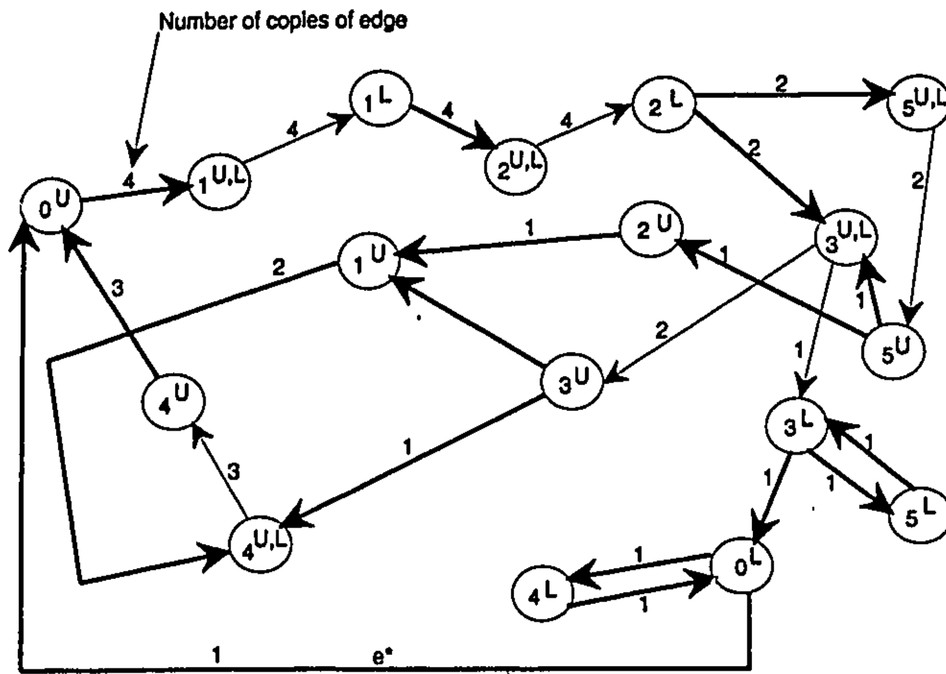


Wang's digraph D'

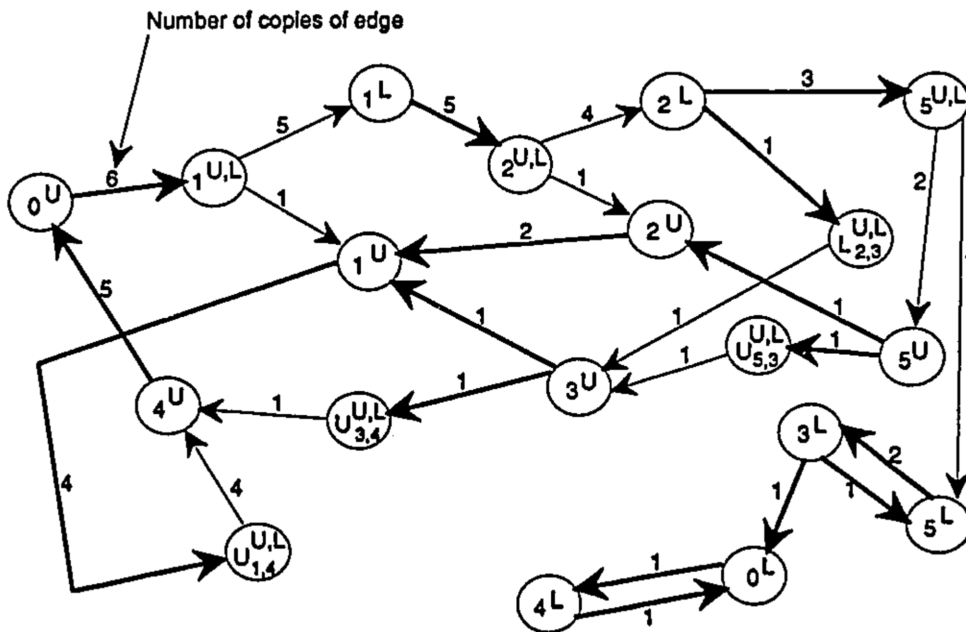
Figure 5.3

5.2.4 Comparison of performance

Our heuristic will always find a COPT in D whose length is less than or equal to the one found by Wang's heuristic. In the example shown in Figure 5.4, the COPT in D found by Wang's heuristic (using bad choices for the edges added to Ec) is 0, 1, 2, 5, 2, 1, 4, 0, 1, 2, 5, 3, 1, 4, 0, 1, 4, 0, 1, 2, 3, 4, 0, 1, 2, 1, 4, 0, 1, 2, 5, 3, 5, 3, 0, 4, 0 and has length 36. The COPT found by our heuristic is 0, 1, 2, 5, 3, 4, 0, 1, 2, 3, 1, 4, 0, 1, 2, 5, 2, 1, 4, 0, 1, 2, 3, 5, 3, 0, 4, 0 and has length 27. Thus for this example, our heuristic out-performs Wang's heuristic. In the next section we will demonstrate empirically that this is generally the case.



Our RSA D* of D'[Ec']



Wang's RSA D* of D'[Ec']

Figure 5.4

5.3 Empirical Testing of the heuristic

We empirically compared our heuristic STSG and our implementation of Wang's heuristic[20]. Our implementation of Wang's heuristic creates a duplex digraph as described in [20] but the rest of this implementation we adopted it from the implementation of our heuristic (STSG) with some changes in the way we choose the edges added to Ec. Since Wang's heuristic chooses this edges randomly, we implemented Wang's heuristic in such a way that it will always make bad choices of the edges added to Ec.

Using GENGRAPH, we generated 41 digraphs D=(V,E), where the number of vertices varied between 3 and 10.

For convenience, we implemented Wang's heuristic using the same data structure that was used for STSG. Although this data structure was satisfactory for our heuristic, its space requirement was much larger for our implementation of Wang's heuristic due to the fact that his duplex digraph is much larger than ours in a worst case scenario. In the worst case, our duplex digraph has $3|V|$ vertices, and hence $O(|V|^2)$ edges. However, Wang's duplex digraph may have $O(|E|)$ vertices in the worst case, and thus $O(|V|^4)$ edges. This meant that we were unfortunately unable to run our implementation of Wang's heuristic on problems larger than 10 vertices, as our data structure required the static allocation of memory for the worst case scenario. The results of our testing are reported in Table 5.3.

In the column labelled STSG and the column labelled Wang, we report the length of the COPT in D found by our heuristic algorithm and by Wang's heuristic

algorithm respectively. In column labelled % Gap, we report the percentage of difference between these lengths, which is calculated using the formula

$$\% \text{ Gap} = 100\ ((\text{Wang-STSG})/\text{Wang}).$$

In the column labelled Density, we report the percentage of the number of possible edges present in the digraph D, which is calculated using the formula

$$\text{Density} = |E|/3|V|(|V|-1).$$

As can been seen from the test results, our algorithm always did as well as or better than Wang's. The maximum percentage gap found in these examples was 52%, and the average percentage gap over all examples was 16.6%.

In our testing, we found no clear relationship between %Gap and Density, or %Gap and |V|. However, we did notice a relationship between % Gap and the number of strongly connected components in the duplex digraph D' for D. Because these test problems are small, D was strongly connected in most cases. However, in the problems with high percentage Gap, the number of strongly connected components was higher than 1.

For example, the problem that gave 52 %Gap had 3 strongly connected components in D'. This leads us to believe that for larger problems for which D' has a larger number of strongly connected components, the %Gap will be higher.

|V|	Density	STSG	Wang	% Gap
3	33%	6	8	25%
3	33%	6	7	14%
3	50%	11	13	15%
3	56%	10	15	33%
3	78%	14	15	7%
3	83%	18	20	10%
3	100%	19	20	5%
4	19%	8	13	38%
4	39%	17	27	37%
4	50%	19	20	5%
4	67%	25	30	17%
4	81%	33	35	6%
4	97%	30	35	14%
5	30%	20	23	13%
5	37%	25	28	11%
5	45%	27	38	29%
5	50%	33	33	-
5	70%	46	46	-

Table 5.3

|V|	Density	STSG	Wang	% Gap
6	19%	8	12	33%
6	19%	20	23	13%
6	30%	32	40	20%
6	34%	35	41	15%
6	40%	39	43	9%
7	7%	11	14	21%
7	18%	26	30	13%
7	21%	28	34	18%
7	25%	32	37	14%
7	27%	38	42	10%
8	6%	13	18	28%
8	7%	11	14	21%
8	8%	14	16	13%
8	9%	16	18	11%
8	14%	26	27	4%
9	5%	15	16	6%
9	7%	11	19	42%
9	8%	20	23	13%
9	10%	24	24	-
10	5%	15	15	-
10	6%	15	20	25%
10	7%	20	25	20%
10	9%	14	29	52%

Table 5.3

5.4 An example where our heuristic does not find an optimal solution

Although our algorithm works very well in practice, it is not guaranteed to always find the optimal solution. As an example, consider the digraph D shown in Figure 5.5. In Figure 5.6, we show the corresponding duplex digraph D', as well as the strongly connected components and the cost C'e for every edge e for the minimum spanning tree which is defined in Step 2 of our heuristic algorithm. Note that for the minimum spanning tree we have to choose the edge $3^{U,L}$ - 3^{U} and any one of the edges $3^{U,L}$ - 3^{L}, $1^{U,L}$ - 1^{L}, or $1^{U,L}$ - 1^{U}. In Figure 5.6 a we choose $3^{U,L}$ - 3^{U} and $1^{U,L}$ - 1^{U} and in Figure 5.6 b we choose $3^{U,L}$ - 3^{U} and $1^{U,L}$ - 1^{L}. The choice of Figure 5.6 a results in the COPT 0 2 0 1 0 1 3 5 3 5 1 4 3 1 4 1 3 2 3 0 2 0 of length 21, whereas the choice of Figure 5.6 b results in the COPT 0 2 0 2 3 5 3 5 1 0 1 3 1 4 1 4 3 1 3 2 3 0 2 0 of length 23, showing that our algorithm may obtain a non optimal solution.

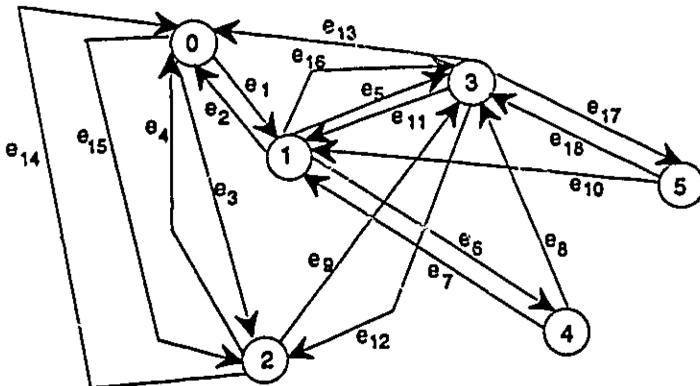


Figure 5.5 Digraph D.

Edge	Type	Eligible Successor
e1=(0,1)	$R^{IL}S^{IL}$	e2, e5
e2=(1,0)	$R^{IL}S^{IL}$	e1, e14
e3=(0,2)	$R^{IU}S^{IU}$	e4
e4=(2,0)	$R^{IU}S^{IU}$	e3
e5=(1,3)	$R^{IL}S^{IU,L}$	e11, e12, e13, e17
e6=(1,4)	$R^{IU}S^{IU}$	e7, e8
e7=(4,1)	$R^{IU}S^{IU}$	e6, e16
e8=(4,3)	$R^{IU}S^{IL}$	e11, e12, e13, e17
e9=(2,3)	$R^{IL}S^{IU,L}$	e11, e12, e13, e17
e10=(5,1)	$R^{IL}S^{IU}$	e2, e5, e6, e16
e11=(3,1)	$R^{IL}S^{IU,L}$	e2, e5, e6, e16
e12=(3,2)	$R^{IL}S^{IU}$	e4, e9, e15
e13=(3,0)	$R^{IU}S^{IL}$	e3
e14=(0,2)	$R^{IL}S^{IL}$	e9, e15
e15=(2,0)	$R^{IL}S^{IL}$	e14, e1
e16=(1,3)	$R^{IU}S^{IU,L}$	e11, e12, e13, e17
e17=(3,5)	$R^{IL}S^{IL}$	e10, e18
e18=(5,3)	$R^{IL}S^{IL}$	e17, e12, e11

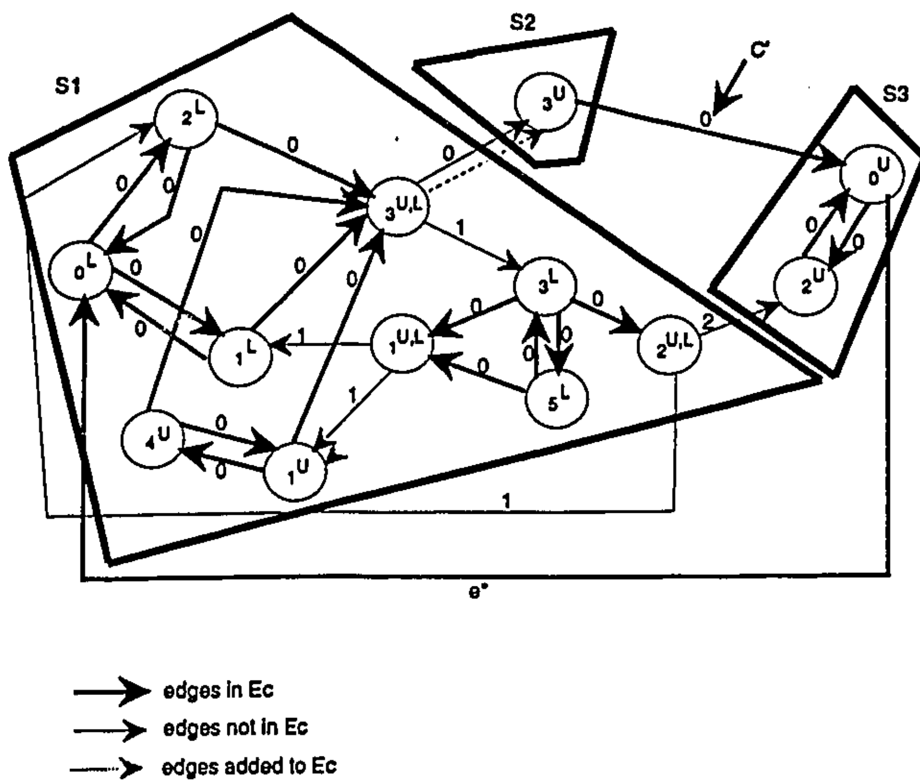


Figure 5.6 a Adding $1^{U,L}$ - 1^{U} and $3^{U,L}$ - 3^{U} to make D'[Ec] weakly connected.

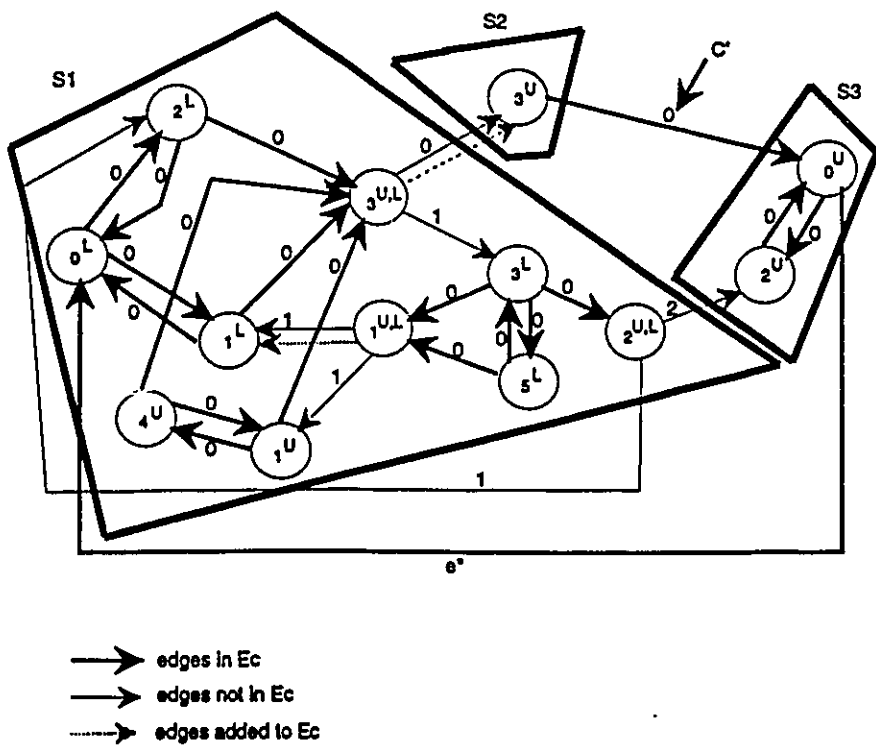


Figure 5.6 b Adding $1^{U,L}$ - 1^{L} and $3^{U,L}$ - 3^{U} to make D'[Ec] weakly connected.

Chapter 6

A heuristic for finding a correctly-ordered Chinese postman tour

In this chapter we consider the problem of finding a correctly-ordered Chinese postman tour in a digraph D. Because this problem is NP-hard[5], we have little prospect of success for finding an efficient algorithm to solve this problem exactly. Therefore, we describe a heuristic for this problem.

Our heuristic is based on the idea of creating a digraph D' in which each vertex in D is replaced by a bipartite component, and D' has the property that there exists a correctly-ordered postman tour (COPT) in D if and only if there exists a Rural Postman Tour (RPT) over a set of specified edges Ec in D'.

An important step in finding the RPT in D' requires checking to see if D'[Ec] is weakly connected, and if not, adding edges to Ec to obtain Ec' such that this is true. We provide a method for adding these edges which attempts to add as few edges as possible. We then create a RSA D* of D'[Ec'], and from this a COPT of D. If we cannot create D*, then we show there is no RPT in D' over Ec and therefore there is no COPT in D.

6.1 The digraph D'

Given a ordered specified digraph D = (V,E) , we now describe how to create the corresponding digraph D' = (V',E'), where $V' = V^A \cup V^L$ and E' = Ec ∪ F. Without loss of generality, we will assume that D has no isolated vertices. In D', the edges in Ec are in a one-to-one correspondence with the edges in D. We obtain the vertices in D' by essentially replacing each vertex v in D by a bipartite graph in D' whose vertices partition into two sets: v^L and v^A. There is a vertex v^{iA} in v^A for each edge ei coming into v in D, and a vertex v^{jL} in v^L for each edge ej leaving v in D. We include edge (v^{iA},v^{jL}) if edge ej is an eligible successor for edge ei in D. For any path P' in D', if we take the path P in D formed by the edges corresponding to Ec ∩ P', then P is correctly ordered (CO) in D.

Such a graph D' is created as follows.

Digraph D'=(V',E') construction:

(a) For every edge e_i=(u,v) in E, we create a vertex u^{iL} in V^L , a vertex v^{iA} in V^A and an edge e_i' in Ec from u^{iL} to v^{iA}.

(b) For every edge e_i=(u,v) in E, we create an edge in F from v^{iA} to x^{jL} for every edge e_j=(x,y) which is an eligible successor of e_i in D.

(c) Give each edge in Ec cost one and give each edge in F cost zero.

(d) Let $V'=V^L \cup V^A$, and let E'=F∪Ec.

Figure 6.1 gives an example of a digraph D and its corresponding digraph D'.

6.2 Characteristics of the digraph D'

Theorem 6.1: An ordered-specified digraph D=(V,E) has a COPT T* of length k if and only if there exist a RPT T of cost k over Ec in D'.

Proof: Suppose we have a COPT T* in D. From the way the digraph D' is created, we know that each edge e in D is represented by a segment which consists of the corresponding edge e' in Ec and an edge in F. The segment in D' corresponding to an edge e in D can be determined, if the successor of e in T* is specified.

Let ei be the ith edge of T* for i=1,2,...,k, where k is the length of T*. For i=1,...,k-1, let ei" be the edge in F in D' that goes from tail(ei') to head(ei+1').(We know that ei" exists since ei+1 is an eligible successor of ei in D.) Let ek" be the edge that goes from tail (ek') to head(e1'). Then

e1',e1",e2',e2",...,ek',ek"

is a RPT in D' over Ec with cost k.

We also know from the way the digraph D' is created that there is a one-to-one correspondence between the edges in Ec and the edges in D such that for any tour T in D', if we take the tour T* in D formed by the edges corresponding to $E\cap$ T, then T* is correctly ordered. Thus if T is a RPT in D' over Ec of cost k, since T traverses all the edges in Ec, the corresponding tour T* in D will be a COPT in D of length k. ■

It follows from Theorem 6.1 and the fact that Ec is spanning in D' that if D' is not strongly connected, then there can be no COPT in D. In our algorithm for finding a RPT of D' over Ec we check this.

6.3 An Algorithm for the problem of finding a COPT in D

We now describe an algorithm which, given an ordered specified digraph D, either finds a COPT in D, or determines that no such tour exists. In the algorithm we first construct the digraph D' as previously described, and then construct a RPT in D' over Ec.

In order to construct a RPT in D' over Ec, we must construct a RSA D* of D'[Ec] which is strongly connected. Note that since D' is strongly connected, a RSA of D'[Ec] can be obtained by solving a minimum-cost maximum flow problem as described in Section 2.2. If this RSA D* is strongly connected, then it follows from Theorem 2.3 that we have found a RCPT of D' over Ec, and thus a COCPT of D. However D* may not be strongly connected.

Recall from Theorem 2.2 that if the edge-induced subgraph D'[Ec] is weakly connected, then any RSA D* of D'[Ec] is strongly connected. Thus one way to ensure D* is strongly connected is to add edges to Ec, obtaining Ec' such that D'[Ec'] is weakly connected. Clearly we wish to add these edges in such a way as to add as few edges as possible while at the same time ensuring that the graph D[Ec'] is as symmetric as possible.

To find the edges to add to Ec, we find a minimum cost spanning tree T in D' (with the directions on the edges ignored), where we assign to each edge e in Ec a cost C'e equal to zero, and we assign each edge e' in F' cost 0, where F' is the set of edges in E'\Ec whose head vertices have out-degree equal to 1 and/or whose tail vertices have in-degree equal to 1. Note that such edges will automatically be

traversed by any RPT over Ec found for D'. The rest of the edges e' in E'\(Ec∪F') are assigned cost 1 if the addition of that edge to Ec∪F' will make D'[Ec∪F'] more symmetric at both ends, otherwise we assign e' cost 2. We then let Ec'=Ec∪(T\Ec).

D* = (V*,E*) is a RSA of D'[Ec'] if for each vertex $v \in V^*$, din(v) = dout(v), where V*= V' and E* contains every edge in Ec' at least once and every edge in E'\Ec' zero or more times such that the total cost of the edges in E* is minimized. Then from Theorem 1, a minimum cost RSA D*has an Euler tour which is an RCPT T* of D' over Ec', which corresponds to a RPT of D' over Ec, and a COPT T in the digraph D. T is obtained from T* by removing the edges not in E from T* and eliminating the superscripts from the vertices.

Given an order specified digraph D, we propose the following algorithm for constructing an RPT T* in D' over Ec .

Step 1. Construct the digraph D' as described previously.

Step 2. Check to see if D' is strongly connected. If not, we may conclude that no COPT exists in D.

Step 3. Check to see if D'[Ec] is weakly connected. If not we find a set of edges in E'\Ec to add to Ec to make it weakly connected in the following way:

i) We find a minimum cost spanning tree T in D' for the underlying undirected graph using the following cost function C' for every edge e=(u,v).

- If $e \in Ec$, then C'e is zero.

- If $e \in E' \setminus Ec$ and the out-degree of u is 1 and/or the in-degree of v is 1, then $e \in F'$ and C'e is zero.

- If the out-degree of u in $D'[Ec \cup F']$ is less than the in-degree of u in $D'[Ec \cup F']$ and the in-degree of v in $D'[Ec \cup F']$ is less than the out-degree of v in $D'[Ec \cup F']$, then C'e is 1.

- Otherwise, C'e is 2.

ii) We add the edges $T \setminus Ec$ to Ec, to get Ec'.

Step 4. Generate the RSA D* of D'[Ec'] by formulating a mimimum cost maximum flow problem, as described in the background section.

Step 5. Find a RCPT in D*over Ec'.

Step 6. Find the corresponding COPT in D using the method described in Theorem 6.1.

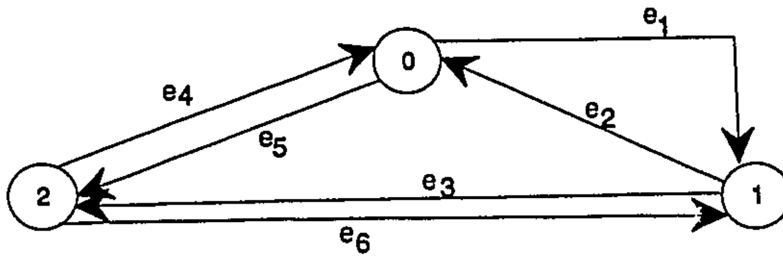
Figure 6.2 shows the digraph D' from 6.1, along with the edges added to Ec to form Ec'. Figure 6.3 shows the RSA D* of D[Ec'] found. From D* we obtain the RCPT

$T^* = 0^{4A}, 0^{1L}, 1^{1A}, 1^{2L}, 0^{2A}, 0^{5L}, 2^{5A}, 2^{6L}, 1^{6A}, 1^{3L}, 2^{3A}, 2^{4L}, 0^{4A}.$

of D' over Ec', and the COPT

T= 0, 1, 0, 2, 1, 2, 0

of D of length 6.



e1 = (0,1), Eligible Successor : e2,e3.
e2 = (1,0), Eligible Successor : e5.
e3 = (1,2), Eligible Successor : e4.
e4 = (2,0), Eligible Successor : e1,e5.
e5 = (0,2), Eligible Successor : e6.
e6 = (2,1), Eligible Successor : e3.

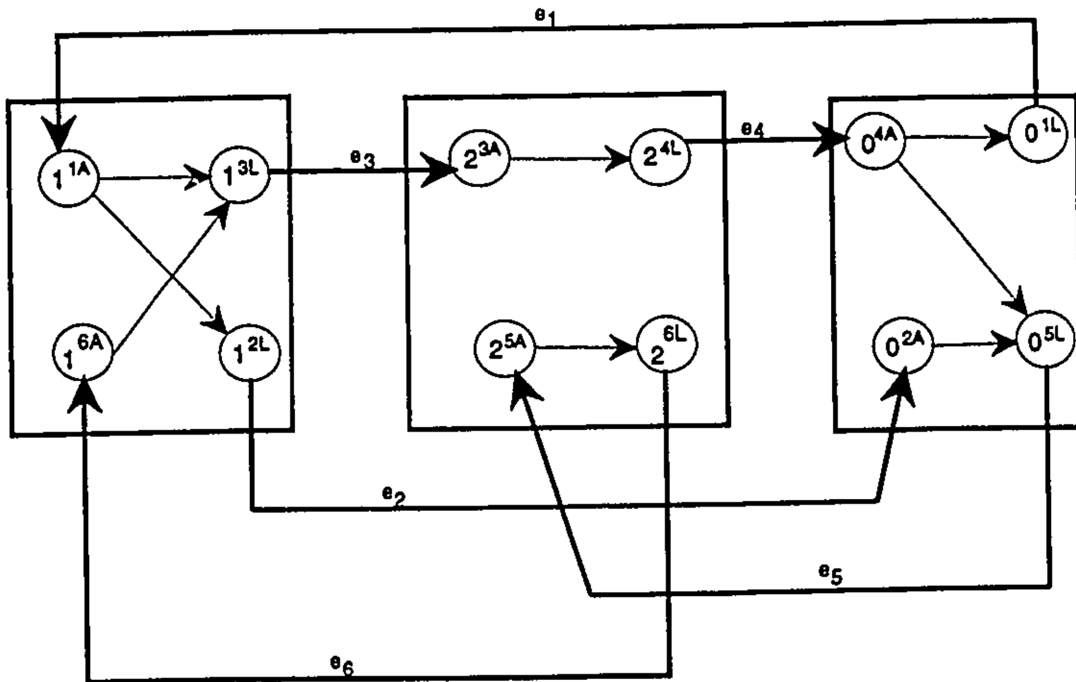


Figure 6.1 Digraph D and corresponding digraph D'.

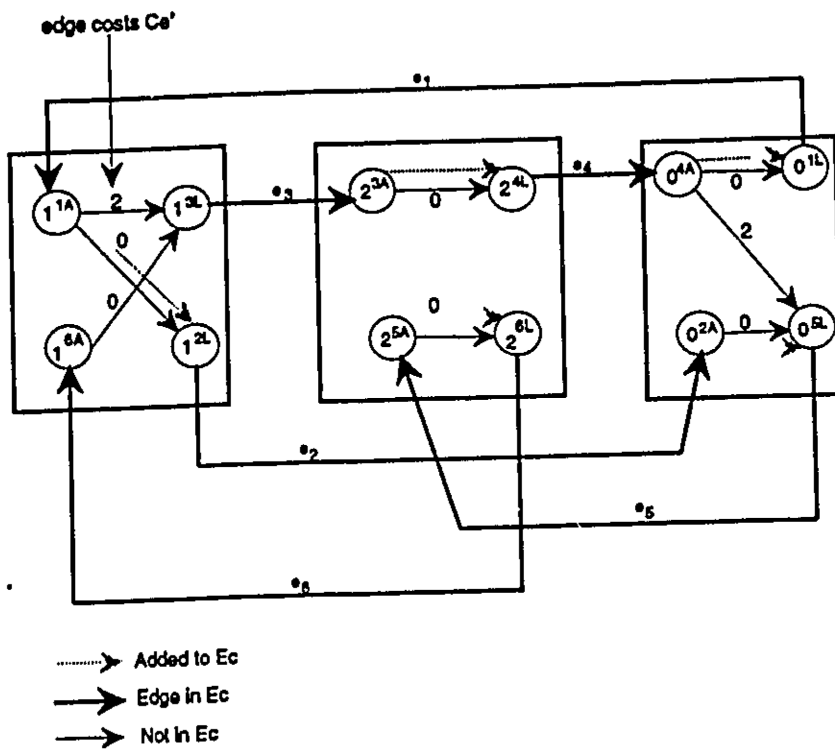


Figure 6.2 Digraph D' with added edges to Ec.

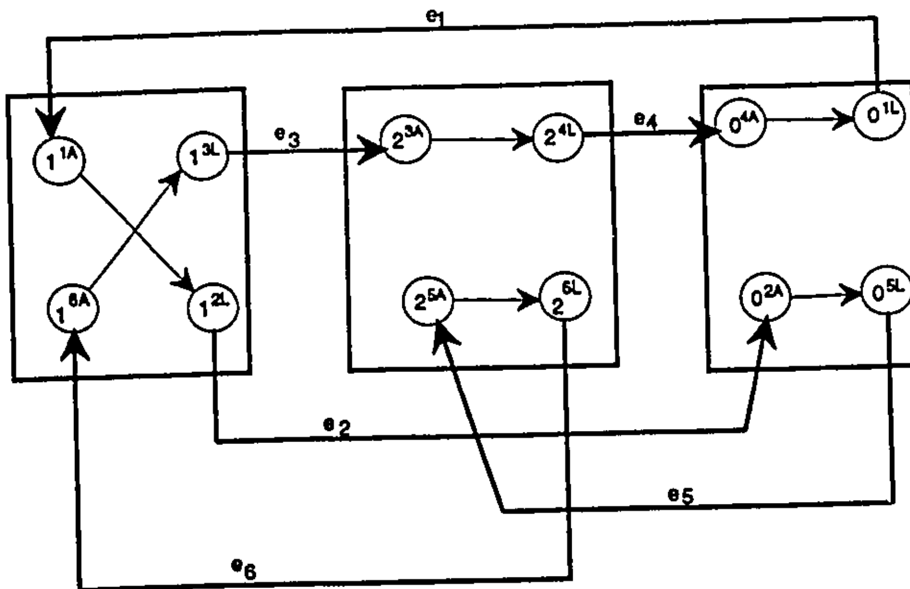


Figure 6.3 RSA D* of D[Ec'].

Chapter 7

Conclusions

In this thesis, we formulated the problem of finding a Chinese postman tour in an order-specified digraph and proposed a heuristic to find an approximate solution. We also formulated a specialized version of this heuristic for generating a minimum length synchronizable test sequence that tests every transition of a protocol at least once.

Although the correctly-ordered Chinese postman problem (COCPP) is NP-hard[5], this does not necessarily mean that it is NP-hard for the special form which needs to be solved for the protocol conformance testing application. Recall that for this application, the eligible successors have a special form (as described in Section 1.3, page 7).The proof used in [5] for the COCPP could not be adopted to this application, as it transforms the Travelling Salesman Problem into an instance of the COCPP in which the eligible successors do not have the required special form. An open problem is to find a polynomial time algorithm to generate a minimum length synchronizable test sequence in an order-specified digraph, or to prove that this problem is NP-hard.

There are several formal testing methods proposed for generating test sequences from FSM-based specifications. Wang[20] describes a heuristic to generate a synchronizable test sequence in an order-specified digraph which is

extended to the W-method, the D-method, and the UIO-method. We have adapted the specialized version of our heuristic to the T-method, and obtain improved performance over Wang's heuristic. A future research topic could be to extend our heuristic to other formal testing methods.

In this thesis, we have concentrated on developing the special version of the general heuristic. Perhaps a future topic of research would be to adapt some of the specializations done for the generation of a synchronizable test sequence in an order-specified digraph to the general problem to improve the current heuristic. For example, when creating D', if the bipartite graph that corresponds to a vertex v in D is complete, we could replace it by a single vertex v' in D'. Other more complicated simplifications of our auxillary graph D' are also possible.

In [7], Chen, Lu, Wang, and Lee described a heuristic algorithm for the general problem. As a suggestion, it would be interesting to compare the performance of this heuristic algorithm with our heuristic algorithm to see which one is more useful.

Appendix A

```
Program GenGraph (input, output);
{This program generates an order-specified digraph which has a COPT}
{First it generates a main path, then it generates a number of paths}
{along that path.  It will contain no loops, and may or may not result}
{in a strongly connected duplex digraph.}

uses crt;

const
  MaxNode = 10;
  MaxGraph = 100;

type
  EdgeKind = (L,LU,U,UL);

  Edge = record
    head,tail: integer;
    kind: edgekind;
    end; {record}

  patharray = array[1..MaxGraph] of edge;
  Grapharray = array [1..MaxGraph] of edge;
  kindarray = array[0..3] of edgekind;

var
  Length,Position,NumNode,NumPath,Dist,I: integer; {Dist is max length of paths}
  PLength: integer;                    {Nodes are 0,1,...,NumNode}
  EdgeType:  kindarray;
  EdgeTypeSt:  array [0..3] of string;
  Graph:  grapharray;
  MPath, Path:  patharray;
  Digraph:text;

  {*************************GETINPUT***************************}
  Procedure GetInput(var NumNode, Length, NumPath, Dist: integer);
  begin
    Write('Please enter number of nodes (must be >=3 and <=',MaxNode:0,'):  ');
    readln(NumNode);
    write('Please enter length of main path:  ');
    readln(Length);
    Write('Please enter number of added paths:  ');
    readln(NumPath);
    Write('Please enter max. length for these paths (must be >=2):  ');
    Readln(Dist);
  end;

  {***********************CHOOSEKIND****************************}
```

```
Function ChooseKind(GivenKind: EdgeKind): EdgeKind;
{Find an edge kind which is an eligible successor of GivenKind}

var k,z: integer;
begin
 if GivenKind = U then begin
  K:= random(2);
  ChooseKind := EdgeType[k+2];
  end; {if}
 if GivenKind = L then begin
  k:= random(2);
  ChooseKind:= EdgeType[k];
  end;{if}
 if (GivenKind = UL) or (GivenKind = LU) then begin
  k:= random(3);
  if k=0 then ChooseKind:=L;
  if k=1 then ChooseKind:=U;
  if k=2 then begin
   k:=random(2);
   ChooseKind:=EdgeType[2*k+1];
  end; {if}
 end;{if}
end; {Proc}

{***************************MAINPATH****************************}
Procedure MainPath(NumNode,Length: integer; var MPath: PathArray);
{Generates the Main Path from 0 to 0 which is Length long.}
{The path is stored as a list of nodes. MPath[i].tail is the ith node}
{in the path, and MPath.kind is the edge kind of the edge}
{ (MPath[i-1], MPath[i]).}

var j,N: integer;
begin
 MPath[1].tail:= 0;
 MPath[length+1].tail:=0;
 Mpath[1].kind:=UL;
 for j:= 2 to length do begin
  repeat
   N:= random(NumNode);
   MPath[j].tail:= N;
  until N<>MPath[j-1].tail; {i.e. no loops in path}
  MPath[j].kind:= ChooseKind(MPath[j-1].kind);
 end; {for}
 MPath[Length+1].kind:= ChooseKind(MPath[Length].kind);
 if MPath[Length].tail = MPath[Length+1].tail then begin {avoid loop at end}
  repeat
   N:= random(NumNode);
   MPath[Length].tail:= N;
  until (N<>MPath[Length+1].tail) and (N<>Mpath[Length-1].tail);
 end; {if}
end; {Proc}
{************************UPDATEGRAPH************************}
Procedure UpdateGraph(LengthIn: integer; PathIn: patharray;
            var Position: integer;
            var Graph: grapharray);
```

```
{Graph is a list of the edges (head and tail) and their kind in the graph.}
{This procedure adds the edges from the latest path produced to Graph.}
{Position is the position of the last entry in Graph.}

var j: integer;
  Current: integer;

begin
 for j:= 1 to LengthIn do begin
   Current:= Position + j;
   Graph[Current].tail:= PathIn[j+1].tail;
   Graph[Current].head:= PathIn[j].tail;
   Graph[Current].kind:= PathIn[j+1].kind;
 end; {for}
 Position:= Position + LengthIn;
end; {Procedure}

{************************MAKEPATHI********************}
Procedure MakePathI(MPath: Patharray;
            NumNode,NumPath, Dist, Length: integer;
            var Path:  patharray;
            var PLength: integer);
{Produces a path of length PLength<= Dist from a randomly chosen start position
in the main path to a finish position which occurs before
the start position in the main path.}

var start,finish: integer;
  k,N: integer;
  Temp: edgekind;
begin
 PLength:= random(Dist-1) + 2; {ensure PLength>=2,<=dist}
 {Choose a start and finish position in MPath}
 start:= random(length+1)+1;
 finish:= random(start)+1;
 Path[1].tail:= Mpath[start].tail;
 Path[1].kind:= Mpath[start].kind;  {ensure can traverse MPath onto Path}
 Path[PLength+1].tail:= MPath[finish].tail;
 for k:=2 to PLength do begin
  repeat
   N:= random(NumNode);
   Path[k].tail:=N;
  until N<>Path[k-1].tail;  {prevent loops}
  Path[k].kind:= ChooseKind(Path[k-1].kind);
 end; {for}
 {Choose kind for final edge such that can traverse MPath}
 Temp:=ChooseKind(Path[PLength].kind);
 If Temp=L then Temp:=LU;
 If Temp=U then Temp:=UL;
 Path[PLength+1].kind:= Temp;
 if (path[PLength].tail= path[PLength+1].tail) and (PLength>=2) then begin
  repeat
   N:= random(NumNode);        {Prevent loop at end}
   Path[Plength].tail:=N;
  until (N<>Path[PLength+1].tail) and (N<>Path[PLength-1].tail);
 end; {if}
```

```
end; {Procedure}


{**********************WRITEPATH*********************}
Procedure WritePath(InLength: integer; InPath: patharray);
var k:integer;
begin
  writeln;
  writeln('Path');
  for k:= 1 to InLength+1 do
    write(Inpath[k].tail,' ',EdgeTypeSt[ord(Inpath[k].kind)]:2,'     ');
  end;

{****************************DELETEDUPLICATE*********************}
procedure DeleteDuplicate(var position:integer;var Graph: Grapharray);
var y,x,z,head,tail:integer;
kind:edgekind;
begin
 y:=1;
 while y<position do
 begin
  head:=graph[y].head;
  tail:=graph[y].tail;
  kind:=graph[y].kind;
  x:=y+1;
  while x<position do
   if ((head=graph[x].head) and (tail=graph[x].tail)) and
      (kind=graph[x].kind) then
   begin
    z:=x;
    while z<position do
    begin
     graph[z]:=graph[z+1];
     z:=z+1;
    end;
    position:=position-1;
   end
   else
    x:=x+1;
  if ((head=graph[position].head) and (tail=graph[position].tail)) and
     (kind=graph[position].kind) then
   position:=position-1;
  y:=y+1;
 end; {while y}
end; {proc}

{*****************************WRITEGRAPH**************************}
Procedure WriteGraph(Position: integer; Graph: Grapharray);
var edge: integer;
begin
 writeln;
 writeln('List of Graph Edges');
 for edge:= 1 to Position do
  write(Graph[edge].head:3, Graph[edge].tail:3,EdgeTypeSt[ord(Graph[edge].kind)]:3,'   ');
 writeln;
```

```
end; {Proc}

{****************************CREATEDIGRAPH**********************}
procedure CreateDigraph(Position:integer; Graph: Grapharray; var Digraph:text);
var
 edge,N:integer;
 drive:string;
 filename,output_file:string;
begin
  filename:=' ';
  write('Drive where the output file will be located: ');
  readln(drive);
  write('Name of the output file: ');
  readln(output_file);
  filename:=drive+':'+output_file;
  assign(Digraph,filename);
  rewrite(Digraph);
  for edge:=1 to position do
  begin
   write(Digraph,'(',graph[edge].head,',',graph[edge].tail,'),');
   if graph[edge].kind=L then
    writeln(Digraph,'RilSil');
   if graph[edge].kind=LU then
   begin
     N:=random(2);
     if N=0 then
      writeln(Digraph,'RilSiu')
     else
      writeln(Digraph,'RilSiul');
    end;
    if graph[edge].kind=UL then
    begin
     N:=random(2);
     if N=0 then
      writeln(Digraph,'RiuSil')
     else
      writeln(Digraph,'RiuSiul');
    end;
    if graph[edge].kind=U then
     writeln(Digraph,'RiuSiu');
  end;  {for}
  writeln(Digraph,' ');
  close(Digraph);
 end; {proc}

 {****************************MAIN PROGRAM*******************}
 begin {Main Program}
    clrscr;
    randomize;
    Position:= 0;
    EdgeType[0] := L; EdgeTypeSt[0] := 'L';
    EdgeType[1]:= LU; EdgeTypeSt[1] := 'LU';
    EdgeType[2]:= U;  EdgeTypeSt[2] := 'U';
    EdgeType[3]:= UL; EdgeTypeSt[3] := 'UL';
    GetInput (NumNode, Length, NumPath, Dist);
```

```
 MainPath (NumNode, Length, MPath);
 WritePath (Length, MPath);
 UpdateGraph(length,Mpath,Position,Graph);
 For I:= 1 to NumPath do begin
   MakePathI(Mpath, NumNode, NumPath, Dist, Length, Path, PLength);
   WritePath(PLength, Path);
   UpdateGraph(PLength,Path,Position, Graph);
 end; {for}
 writeGraph(position,graph);
 Deleteduplicate(position,Graph);
 WriteGraph(Position, Graph);
 CreateDigraph(Position, Graph, Digraph);
end. {Main program}
```

```
{This program is used to find a COPT for a given order-specified digraph D
 stored in the input file. The input file is generated by a program called
 GENGRAPH}


program STSG;
{$M 65520,0,655360}

uses crt;

const
  max_vertices=10; {|V|+1}
  max_nodes=50; {maximum of positions in an array used in some procedures}
  max_e_orig=30; {maximum number of edges in each position of original}
  max_e_duplex=30;{maximum number of edges going in or coming out of each
              vertex in the duplex digraph D'}

type

 edges_type=(arrive_up,arrive_low,arrive_u_l,leave_up,leave_low);
 { this are the rows for matrix_original}

 rec_edges=
    record
      vertex:integer;
      type_edge:edges_type;
      num_vertex :integer;
 end;

 vertex_type=(upper,lower,up_low);
 {this are the rows for matrix_duplex}

 pointer=^node;
 node=
   record
     next:pointer;
     vertex:integer;
     v_type:vertex_type;
 end;
 {this list is used in the  procedures RPT and COPT}
 list=
    record
      head:pointer;
 end;

 set_of_edge=(Ec,f); {This indicates if the edge is in Ec
              or if is a twin edge}

 vertices=array[1..max_e_orig] of rec_edges;
 {this is array containing the edges in arrive_up, arrive_low,...,leave_low}

 matrix_original=array[0..max_vertices,arrive_up..leave_low] of vertices;
```

```
{this matrix contains all the information of the digraph D}

edge_rec=
   record
     out_of_vertex:integer;
     out_of_type:vertex_type;
     into_vertex:integer;
     into_type:vertex_type;
end;
{this record contains the information of the edge e*}

rec_vertex=
   record
     num_vertex:integer;
     vertex:integer;
     type_vertex:vertex_type;
     mst_e_cost:integer;
     edge_set:set_of_edge;
     duplicate:boolean;
     capacity:integer;
end;
{this record in matrix_duplex to store the information
 of each edge going in or coming out of every vertex in D'}

statustype=(intree,fringe,unseen);
{this is used in the minimum spanning tree procedure}

rec_duplex=
    record
      active:boolean; {this is true if the vertex is in the duplex digraph}
      in_vertices:array[1..max_e_duplex] of rec_vertex;
      {this array contain the information of the edges going into a vertex in D'}

      out_vertices:array[1..max_e_duplex] of rec_vertex;
      {this array contains the information of the edges coming out of a vertex in D'}

      {the next five fields are used for the strongly connected components procedure}
      component:integer;
      dfsNumber:integer;
      low:integer;
      removed:boolean;
      ptr:integer;

      {the next three fields are used to store the information on
       the link edges between components}
      link:boolean;
      link_vertex:integer;
      link_type:vertex_type;

      {the next five fields are used for the minimum spanning tree procedure}
      status:statustype;
      mst_v_degree:integer;
      fringe_cost:integer;
      parent_vertex:integer;
      parent_type:vertex_type;
```

```
      {the next fields are used in the minimum cost maximum flow procedure}
      net_degree:integer;
      old_label:integer;
      new_label:integer;
      old_pre_vertex:integer;
      new_pre_vertex:integer;
      old_pre_type:vertex_type;
      new_pre_type:vertex_type;
      flow_from_s:integer;
      flow_to_t:integer;
end;
  matrix_duplex=array[0..max_vertices,upper..up_low] of rec_duplex;
  {this matrix contains all the information of the duplex digraph D'}
var
  prueba,num_comp:integer;
  edge:edge_rec;
  orig_probl:matrix_original;
  duplex:matrix_duplex;
  original,tour:text;
  vertex_list:list;

{****************************CLEAN_ORIG_PROBL*****************************}
procedure clean_orig_probl(var orig_probl:matrix_original);
var
 i:integer;
 j:edges_type;
begin
 for i:=0 to max_vertices do
   for j:=arrive_up to leave_low do
     orig_probl[i,j][1].num_vertex:=0;
end;

{******************FILL_ORIGINAL**************************************}
procedure fill_original(var original:text;var orig_probl:matrix_original);
var
  numvertex:char;
  i,j,k,m,code,v,u,num_char:integer;
  line,type_edge,file_name,drive,input_file:string;
  s:edges_type;
begin
  file_name:=' ';
  write('Name of the input file: ');
  readln(input_file);
  write('Drive where input file is located: ');
  readln(drive);
  file_name:=drive+':'+input_file;
  assign(original,file_name);
  reset(original);
  readln(original,line);
  m:=0;
  k:=0;
  clean_orig_probl(orig_probl);
  while not eof(original) do
  begin
```

```
j:=2;
val(line[j],u,code);
j:=j+2;
val(line[j],v,code);
j:=j+2;
num_char:=length(line)-j;
type_edge:=copy(line,j+1,num_char);
if (type_edge = 'RiuSil')or (type_edge ='RiuSiul') then
         begin
          k:=orig_probl[u,leave_up][1].num_vertex +1;
          orig_probl[u,leave_up][k].vertex:= v;
          orig_probl[u,leave_up][k].type_edge:=arrive_u_l;
          orig_probl[u,leave_up][1].num_vertex:= k;
          m:=orig_probl[v,arrive_u_l][1].num_vertex +1;
          orig_probl[v,arrive_u_l][m].vertex:= u;
          orig_probl[v,arrive_u_l][m].type_edge:=leave_up;
          orig_probl[v,arrive_u_l][1].num_vertex:= m;
         end
 else
   if type_edge='RiuSiu' then
         begin
          k:=orig_probl[u,leave_up][1].num_vertex +1;
          orig_probl[u,leave_up][k].vertex:= v;
          orig_probl[u,leave_up][k].type_edge:=arrive_up;
          orig_probl[u,leave_up][k].num_vertex:= k;
          m:=orig_probl[v,arrive_up][1].num_vertex+1;
          orig_probl[v,arrive_up][m].vertex:= u;
          orig_probl[v,arrive_up][m].type_edge:=leave_up;
          orig_probl[v,arrive_up][1].num_vertex:= m;
         end
    else
     if type_edge ='RilSil' then
         begin
          k:=orig_probl[u,leave_low][1].num_vertex +1;
          orig_probl[u,leave_low][k].vertex:= v;
          orig_probl[u,leave_low][k].type_edge:=arrive_low;
          orig_probl[u,leave_low][1].num_vertex :=k;
          m:=orig_probl[v,arrive_low][1].num_vertex +1;
          orig_probl[v,arrive_low][m].vertex:= u;
          orig_probl[v,arrive_low][m].type_edge:=leave_low;
          orig_probl[v,arrive_low][1].num_vertex:= m;
         end
      else
       if (type_edge ='RilSiu') or (type_edge='RilSiul') then
          begin
           k:=orig_probl[u,leave_low][1].num_vertex +1;
           orig_probl[u,leave_low][k].vertex:= v;
           orig_probl[u,leave_low][k].type_edge:=arrive_u_l;
           orig_probl[u,leave_low][1].num_vertex:= k;
           m:=orig_probl[v,arrive_u_l][1].num_vertex +1;
           orig_probl[v,arrive_u_l][m].vertex:= u;
           orig_probl[v,arrive_u_l][m].type_edge:=leave_low;
           orig_probl[v,arrive_u_l][1].num_vertex:= m;
          end;
    readln(original,line);
```

```
   end;
  close(original);
 end;

{********************INITIALLIZE_DUPLEX**********************************}
 Procedure initiallize_duplex(orig_probl:matrix_original;var duplex:matrix_duplex);
 var
  i,x:integer;
  j:vertex_type;
 begin
   for i:=0 to max_vertices do
    for j:=upper to up_low do
     begin
      duplex[i,j].active:=false;
      if (j=up_low) and (orig_probl[i,arrive_u_l][1].num_vertex<>0) then
       begin
        duplex[i,j].active:=true;
        duplex[i,j].out_vertices[1].vertex:=i;
        duplex[i,j].out_vertices[1].type_vertex:= upper;
        duplex[i,j].out_vertices[1].edge_set:=f;
        duplex[i,j].out_vertices[1].capacity:=1000;
        duplex[i,j].out_vertices[1].duplicate:=false;
        duplex[i,j].out_vertices[2].vertex:=i;
        duplex[i,j].out_vertices[2].type_vertex:= lower;
        duplex[i,j].out_vertices[2].edge_set:=f;
        duplex[i,j].out_vertices[2].capacity:=1000;
        duplex[i,j].out_vertices[2].duplicate:=false;
        duplex[i,j].out_vertices[1].num_vertex:=2;
        duplex[i,j].in_vertices[1].num_vertex:=0;
        duplex[i,j].mst_v_degree:=0;
        duplex[i,j].fringe_cost:=0;
        duplex[i,j].link:=false;
        duplex[i,j].flow_from_s:=0;
        duplex[i,j].flow_to_t:=0;
       end;
      if ((j=upper) or (j=lower)) and ((orig_probl[i,leave_up][1].num_vertex>0) or
          (orig_probl[i,leave_low][1].num_vertex>0)) then
       begin
        duplex[i,j].active:=true;
        duplex[i,j].mst_v_degree:=0;
        duplex[i,j].fringe_cost:=0;
        duplex[i,j].link:=false;
        duplex[i,j].flow_from_s:=0;
        duplex[i,j].flow_to_t:=0;
        if orig_probl[i,arrive_u_l][1].num_vertex>0 then
        begin
         duplex[i,j].in_vertices[1].vertex:=i;
         duplex[i,j].in_vertices[1].type_vertex:=up_low;
         duplex[i,j].in_vertices[1].edge_set:=f;
         duplex[i,j].in_vertices[1].capacity:=1000;
         duplex[i,j].in_vertices[1].duplicate:=false;
         duplex[i,j].in_vertices[1].num_vertex:=1;
         duplex[i,j].out_vertices[1].num_vertex:=0;
        end
        else
```

```
        begin
         duplex[i,j].in_vertices[1].num_vertex:=0;
         duplex[i,j].out_vertices[1].num_vertex:=0;
        end;
      end;
    end;
    for i:=0 to max_vertices do
     for j:=upper to up_low do
      if duplex[i,j].active=true then
      begin
       if j=upper then
        writeln(i,',upper');
       if j=lower then
        writeln(i,',lower');
       if j=up_low then
        writeln(i,',up_low');
       readln;
      end;
end;

{****************************CREATE_Ec_EDGE****************************}
procedure create_Ec_edge(orig_probl:matrix_original;var duplex:matrix_duplex);
var
i,k,m,u,in_pos,out_pos,out_u_pos,out_l_pos:integer;
j,y:edges_type;
begin
 for i:=0 to max_vertices do
  for j:=arrive_up to leave_low do
  begin
    if (j=arrive_up) and (orig_probl[i,j][1].num_vertex > 0) then
     begin
      in_pos:=duplex[i,upper].in_vertices[1].num_vertex;
      m:=orig_probl[i,arrive_up][1].num_vertex;
      for k:=1 to m do
      begin
        u:=orig_probl[i,arrive_up][k].vertex;
        out_pos:=duplex[u,upper].out_vertices[1].num_vertex+1;
        duplex[u,upper].out_vertices[out_pos].vertex:=i;
        duplex[u,upper].out_vertices[out_pos].edge_set:=Ec;
        duplex[u,upper].out_vertices[out_pos].type_vertex:=upper;
        duplex[u,upper].out_vertices[out_pos].capacity:=1000;
        duplex[u,upper].out_vertices[out_pos].duplicate:=false;
        in_pos:=in_pos +1;
        duplex[i,upper].in_vertices[in_pos].vertex:=u;
        duplex[i,upper].in_vertices[in_pos].edge_set:=Ec;
        duplex[i,upper].in_vertices[in_pos].type_vertex:=upper;
        duplex[i,upper].in_vertices[in_pos].capacity:=1000;
        duplex[i,upper].in_vertices[in_pos].duplicate:=false;
        duplex[u,upper].out_vertices[1].num_vertex:=out_pos;
      end;
      duplex[i,upper].in_vertices[1].num_vertex:=in_pos;
    end;
    if (j=arrive_low) and (orig_probl[i,j][1].num_vertex > 0) then
     begin
      in_pos:=duplex[i,lower].in_vertices[1].num_vertex;
```

```
m:=orig_probl[i,arrive_low][1].num_vertex;
for k:=1 to m do
begin
  u:=orig_probl[i,arrive_low][k].vertex;
  out_pos:=duplex[u,lower].out_vertices[1].num_vertex +1;
  duplex[u,lower].out_vertices[out_pos].vertex:=i;
  duplex[u,lower].out_vertices[out_pos].edge_set:=Ec;
  duplex[u,lower].out_vertices[out_pos].type_vertex:=lower;
  duplex[u,lower].out_vertices[out_pos].capacity:=1000;
  duplex[u,lower].out_vertices[out_pos].duplicate:=false;
  in_pos:=in_pos +1;
  duplex[i,lower].in_vertices[in_pos].vertex:=u;
  duplex[i,lower].in_vertices[in_pos].edge_set:=Ec;
  duplex[i,lower].in_vertices[in_pos].type_vertex:=lower;
  duplex[i,lower].in_vertices[in_pos].capacity:=1000;
  duplex[i,lower].in_vertices[in_pos].duplicate:=false;
  duplex[u,lower].out_vertices[1].num_vertex:=out_pos;
 end;
 duplex[i,lower].in_vertices[1].num_vertex:=in_pos;
end;
if (j=arrive_u_l) and (orig_probl[i,j][1].num_vertex > 0) then
begin
 in_pos:=duplex[i,up_low].in_vertices[1].num_vertex;
 m:=orig_probl[i,arrive_u_l][1].num_vertex;
 for k:=1 to m do
 begin
   u:=orig_probl[i,arrive_u_l][k].vertex;
   y:=orig_probl[i,arrive_u_l][k].type_edge;
   if y=leave_up then
   begin
    out_u_pos:=duplex[u,upper].out_vertices[1].num_vertex +1;
    duplex[u,upper].out_vertices[out_u_pos].vertex:=i;
    duplex[u,upper].out_vertices[out_u_pos].edge_set:=Ec;
    duplex[u,upper].out_vertices[out_u_pos].type_vertex:=up_low;
    duplex[u,upper].out_vertices[out_u_pos].capacity:=1000;
    duplex[u,upper].out_vertices[out_u_pos].duplicate:=false;
    in_pos:=in_pos +1;
    duplex[i,up_low].in_vertices[in_pos].vertex:=u;
    duplex[i,up_low].in_vertices[in_pos].edge_set:=Ec;
    duplex[i,up_low].in_vertices[in_pos].type_vertex:=upper;
    duplex[i,up_low].in_vertices[in_pos].capacity:=1000;
    duplex[i,up_low].in_vertices[in_pos].duplicate:=false;
    duplex[u,upper].out_vertices[1].num_vertex:=out_u_pos;
   end
   else
    if y=leave_low then
    begin
     out_l_pos:=duplex[u,lower].out_vertices[1].num_vertex +1;
     duplex[u,lower].out_vertices[out_l_pos].vertex:=i;
     duplex[u,lower].out_vertices[out_l_pos].edge_set:=Ec;
     duplex[u,lower].out_vertices[out_l_pos].type_vertex:=up_low;
     duplex[u,lower].out_vertices[out_l_pos].capacity:=1000;
     duplex[u,lower].out_vertices[out_l_pos].duplicate:=false;
     in_pos:=in_pos +1;
     duplex[i,up_low].in_vertices[in_pos].vertex:=u;
```

```
        duplex[i,up_low].in_vertices[in_pos].edge_set:=Ec;
        duplex[i,up_low].in_vertices[in_pos].type_vertex:=lower;
        duplex[i,up_low].in_vertices[in_pos].capacity:=1000;
        duplex[i,up_low].in_vertices[in_pos].duplicate:=false;
        duplex[u,lower].out_vertices[1].num_vertex:=out_l_pos;
       end
    end;
    duplex[i,up_low].in_vertices[1].num_vertex:=in_pos;
   end;
  end;
end;

{*******************REMOVE_EDGES_AND_VERTICES***************************}
procedure remove_edges_and_vertices(var duplex:matrix_duplex;orig_probl:matrix_original);
var
num_in,num_out,x,i,k,num_u_l,num_low,num_up,posi,j,vertex,m,aux_num:integer;
type_of_vertex:vertex_type;
found:boolean;
begin
 for i:=0 to max_vertices do
  begin
   if ((orig_probl[i,leave_up][1].num_vertex=0) and
     (orig_probl[i,leave_low][1].num_vertex>0))and (i<>0) then
    if duplex[i,up_low].active=true then
    begin
     writeln(i,',lower');
     num_low:=duplex[i,lower].in_vertices[1].num_vertex;
     writeln('num_low: ',num_low);
     if num_low>1 then
     for k:=2 to num_low do
     duplex[i,lower].in_vertices[k-1]:=duplex[i,lower].in_vertices[k];
    duplex[i,lower].in_vertices[1].num_vertex:=num_low -1;
    writeln('num_vertex: ',duplex[i,lower].in_vertices[1].num_vertex);
    readln;
    j:=1;
    posi:=duplex[i,lower].in_vertices[1].num_vertex +1;
    num_u_l:=duplex[i,up_low].in_vertices[1].num_vertex;
    while num_u_l > 0 do
    begin
     vertex:=duplex[i,up_low].in_vertices[j].vertex;
     type_of_vertex:=duplex[i,up_low].in_vertices[j].type_vertex;
     aux_num:=duplex[vertex,type_of_vertex].out_vertices[1].num_vertex;
     m:=1;
     found:=false;
     while (m<=aux_num) and not found do
     if (duplex[vertex,type_of_vertex].out_vertices[m].vertex=i) and
       (duplex[vertex,type_of_vertex].out_vertices[m].type_vertex=up_low) then
       found:=true
     else
      m:=m+1;
     duplex[vertex,type_of_vertex].out_vertices[m].type_vertex:=lower;
     duplex[i,lower].in_vertices[posi]:=duplex[i,up_low].in_vertices[j];
     j:=j+1;
     num_u_l:=num_u_l - 1;
     posi:=posi+1;
```

```
 end;
 duplex[i,lower].in_vertices[1].num_vertex:=posi-1;
  writeln('num_vertex: ',duplex[i,lower].in_vertices[1].num_vertex);
  readln;
 orig_probl[i,arrive_u_l][1].num_vertex:=num_u_l;
end;
if ((orig_probl[i,leave_low][1].num_vertex=0) and
   (orig_probl[i,leave_up][1].num_vertex>0)) and (i<>0) then
 if duplex[i,up_low].active=true then
  begin
   num_up:=duplex[i,upper].in_vertices[1].num_vertex;
   if num_up>1 then
   for k:=2 to num_up do
   duplex[i,upper].in_vertices[k-1]:=duplex[i,upper].in_vertices[k];
  duplex[i,upper].in_vertices[1].num_vertex:=num_up -1;
  j:=1;
  posi:=duplex[i,upper].in_vertices[1].num_vertex +1;
  num_u_l:=duplex[i,up_low].in_vertices[1].num_vertex;
  while num_u_l > 0 do
  begin
   vertex:=duplex[i,up_low].in_vertices[j].vertex;
   type_of_vertex:=duplex[i,up_low].in_vertices[j].type_vertex;
   aux_num:=duplex[vertex,type_of_vertex].out_vertices[1].num_vertex;
   m:=1;
   found:=false;
   while (m<=aux_num) and not found do
   if (duplex[vertex,type_of_vertex].out_vertices[m].vertex=i) and
     (duplex[vertex,type_of_vertex].out_vertices[m].type_vertex=up_low) then
     found:=true
   else
    m:=m+1;
   duplex[vertex,type_of_vertex].out_vertices[m].type_vertex:=upper;
   duplex[i,upper].in_vertices[posi]:=duplex[i,up_low].in_vertices[j];
   j:=j+1;
   num_u_l:=num_u_l - 1;
   posi:=posi+1;
  end;
  duplex[i,upper].in_vertices[1].num_vertex:=posi-1;
 orig_probl[i,arrive_u_l][1].num_vertex:=num_u_l;
 end;
if (orig_probl[i,arrive_u_l][1].num_vertex = 0)  then
begin
 {if (orig_probl[i,leave_up][1].num_vertex>0) and (orig_probl[i,leave_low][1].num_vertex>0) then
 begin
  num_up:= duplex[i,upper].in_vertices[1].num_vertex;
  num_low:=duplex[i,lower].in_vertices[1].num_vertex;
  if num_low >1 then
   for k:=2 to num_low do
    duplex[i,lower].in_vertices[k-1]:=duplex[i,lower].in_vertices[k];
  duplex[i,lower].in_vertices[1].num_vertex:=num_low-1;
   if num_up >1 then
   for k:=2 to num_up do
    duplex[i,upper].in_vertices[k-1]:=duplex[i,upper].in_vertices[k];
  duplex[i,upper].in_vertices[1].num_vertex:=num_up -1;
 end; }
```

```
duplex[i,up_low].active:=false;
end;
 if (orig_probl[i,leave_up][1].num_vertex=0) and (i<>0) then
  duplex[i,upper].active:=false;
 if (orig_probl[i,leave_low][1].num_vertex=0) and (i<>0) then
  duplex[i,lower].active:=false;
 if ((orig_probl[i,leave_up][1].num_vertex=0) and
   (orig_probl[i,arrive_up][1].num_vertex=0)) and (i=0) then
  duplex[i,upper].active:=false;
 if ((orig_probl[i,leave_low][1].num_vertex=0) and
   (orig_probl[i,arrive_low][1].num_vertex=0)) and (i=0) then
  duplex[i,lower].active:=false;
end;
for i:=0 to max_vertices do
 for type_of_vertex:=upper to up_low do
 if duplex[i,type_of_vertex].active=true then
 begin
  if type_of_vertex=upper then
  begin
   writeln(i,',upper');
   writeln('in_vertices: ',duplex[i,type_of_vertex].in_vertices[1].num_vertex);
   num_in:=duplex[i,type_of_vertex].in_vertices[1].num_vertex;
   x:=1;
   while x<=num_in do
   begin
    if duplex[i,type_of_vertex].in_vertices[x].type_vertex=upper then
     writeln(duplex[i,type_of_vertex].in_vertices[x].vertex,',upper');
    if duplex[i,type_of_vertex].in_vertices[x].type_vertex=lower then
     writeln(duplex[i,type_of_vertex].in_vertices[x].vertex,',lower');
    if duplex[i,type_of_vertex].in_vertices[x].type_vertex=up_low then
     writeln(duplex[i,type_of_vertex].in_vertices[x].vertex,',up_low');
    x:=x+1;
   end;
   writeln('out_vertices: ',duplex[i,type_of_vertex].out_vertices[1].num_vertex);
   num_out:=duplex[i,type_of_vertex].out_vertices[1].num_vertex;
   x:=1;
   while x<=num_out do
   begin
    if duplex[i,type_of_vertex].out_vertices[x].type_vertex=upper then
     writeln(duplex[i,type_of_vertex].out_vertices[x].vertex,',upper');
    if duplex[i,type_of_vertex].out_vertices[x].type_vertex=lower then
     writeln(duplex[i,type_of_vertex].out_vertices[x].vertex,',lower');
    if duplex[i,type_of_vertex].out_vertices[x].type_vertex=up_low then
     writeln(duplex[i,type_of_vertex].out_vertices[x].vertex,',up_low');
    x:=x+1;
   end;
  end;
  if type_of_vertex=lower then
  begin
   writeln(i,',lower');
   writeln('in_vertices: ',duplex[i,type_of_vertex].in_vertices[1].num_vertex);
   num_in:=duplex[i,type_of_vertex].in_vertices[1].num_vertex;
   x:=1;
   while x<=num_in do
   begin
```

```
    if duplex[i,type_of_vertex].in_vertices[x].type_vertex=upper then
     writeln(duplex[i,type_of_vertex].in_vertices[x].vertex,',upper');
    if duplex[i,type_of_vertex].in_vertices[x].type_vertex=lower then
     writeln(duplex[i,type_of_vertex].in_vertices[x].vertex,',lower');
    if duplex[i,type_of_vertex].in_vertices[x].type_vertex=up_low then
     writeln(duplex[i,type_of_vertex].in_vertices[x].vertex,',up_low');
    x:=x+1;
   end;
   writeln('out_vertices: ',duplex[i,type_of_vertex].out_vertices[1].num_vertex);
   num_out:=duplex[i,type_of_vertex].out_vertices[1].num_vertex;
   x:=1;
   while x<=num_out do
   begin
    if duplex[i,type_of_vertex].out_vertices[x].type_vertex=upper then
     writeln(duplex[i,type_of_vertex].out_vertices[x].vertex,',upper');
    if duplex[i,type_of_vertex].out_vertices[x].type_vertex=lower then
     writeln(duplex[i,type_of_vertex].out_vertices[x].vertex,',lower');
    if duplex[i,type_of_vertex].out_vertices[x].type_vertex=up_low then
     writeln(duplex[i,type_of_vertex].out_vertices[x].vertex,',up_low');
    x:=x+1;
   end;
  end;
  if type_of_vertex=up_low then
  begin
   writeln(i,',up_low');
   writeln('in_vertices: ',duplex[i,type_of_vertex].in_vertices[1].num_vertex);
   num_in:=duplex[i,type_of_vertex].in_vertices[1].num_vertex;
   x:=1;
   while x<=num_in do
   begin
    if duplex[i,type_of_vertex].in_vertices[x].type_vertex=upper then
     writeln(duplex[i,type_of_vertex].in_vertices[x].vertex,',upper');
    if duplex[i,type_of_vertex].in_vertices[x].type_vertex=lower then
     writeln(duplex[i,type_of_vertex].in_vertices[x].vertex,',lower');
    if duplex[i,type_of_vertex].in_vertices[x].type_vertex=up_low then
     writeln(duplex[i,type_of_vertex].in_vertices[x].vertex,',up_low');
    x:=x+1;
   end;
   writeln('out_vertices: ',duplex[i,type_of_vertex].out_vertices[1].num_vertex);
   num_out:=duplex[i,type_of_vertex].out_vertices[1].num_vertex;
   x:=1;
   while x<=num_out do
   begin
    if duplex[i,type_of_vertex].out_vertices[x].type_vertex=upper then
     writeln(duplex[i,type_of_vertex].out_vertices[x].vertex,',upper');
    if duplex[i,type_of_vertex].out_vertices[x].type_vertex=lower then
     writeln(duplex[i,type_of_vertex].out_vertices[x].vertex,',lower');
    if duplex[i,type_of_vertex].out_vertices[x].type_vertex=up_low then
     writeln(duplex[i,type_of_vertex].out_vertices[x].vertex,',up_low');
    x:=x+1;
   end;
  end;
  readln;
 end;
end;
```

```
{**********************STRONGCOMPONENTS************************************}
Procedure StrongComponents(var duplex:matrix_duplex;var num_comp:integer);
type
 nodes=record
      vertex:integer;
      v_type:vertex_type;
     end;
 var
  i,dfn:integer;
  top_vertex,v_vertex:integer;
  j,top_type,v_type:vertex_type;
  SC:array[1..max_nodes] of nodes;
  num_nodes:integer;

 Procedure SCompDFS(v_vertex:integer;v_type:vertex_type);
 var
  w_vertex:integer;
  w_type:vertex_type;
  ptr:integer;

 begin
  dfn:=dfn+1;
  duplex[v_vertex,v_type].dfsNumber:=dfn;
  duplex[v_vertex,v_type].low:=dfn;
  duplex[v_vertex,v_type].removed:=false;
  while duplex[v_vertex,v_type].ptr<=duplex[v_vertex,v_type].out_vertices[1].num_vertex do
  begin
   ptr:=duplex[v_vertex,v_type].ptr;
   w_vertex:=duplex[v_vertex,v_type].out_vertices[ptr].vertex;
   w_type:=duplex[v_vertex,v_type].out_vertices[ptr].type_vertex;
   if duplex[w_vertex,w_type].dfsNumber=0 then
   begin
    SCompDFS(w_vertex,w_type);
    if duplex[w_vertex,w_type].low<duplex[v_vertex,v_type].low then
     duplex[v_vertex,v_type].low:=duplex[w_vertex,w_type].low;
   end
   else
    if not duplex[w_vertex,w_type].removed then
     if duplex[w_vertex,w_type].dfsNumber<duplex[v_vertex,v_type].low then
      duplex[v_vertex,v_type].low:=duplex[w_vertex,w_type].dfsNumber;
    duplex[v_vertex,v_type].ptr:=duplex[v_vertex,v_type].ptr+1;
   end;
   if duplex[v_vertex,v_type].low=duplex[v_vertex,v_type].dfsNumber then
   begin
    num_comp:=num_comp+1;
    duplex[v_vertex,v_type].removed:=true;
    duplex[v_vertex,v_type].component:=num_comp;
    if num_nodes>0 then
    begin
     top_vertex:=SC[num_nodes].vertex;
     top_type:=SC[num_nodes].v_type;
     while (num_nodes>0) and
(duplex[top_vertex,top_type].dfsNumber>duplex[v_vertex,v_type].dfsNumber) do
     begin
```

```
    duplex[top_vertex,top_type].component:=num_comp;
    duplex[top_vertex,top_type].removed:=true;
    num_nodes:=num_nodes-1;
    if num_nodes>0 then
    begin
     top_vertex:=SC[num_nodes].vertex;
     top_type:=SC[num_nodes].v_type;
    end;
   end;
  end;
 end
 else
 begin
  num_nodes:=num_nodes+1;
  SC[num_nodes].vertex:=v_vertex;
  SC[num_nodes].v_type:=v_type;
 end;
end;

begin
 for v_vertex:=0 to max_vertices do
  for v_type:=upper to up_low do
  if duplex[v_vertex,v_type].active=true then
  begin
   duplex[v_vertex,v_type].component:=0;
   duplex[v_vertex,v_type].dfsNumber:=0;
   duplex[v_vertex,v_type].ptr:=1;
  end;
  num_nodes:=0;
  dfn:=0;
  for v_vertex:=0 to max_vertices do
   for v_type:=upper to up_low do
    if (duplex[v_vertex,v_type].active=true) and
       (duplex[v_vertex,v_type].dfsNumber=0) then
      SCompDFS(v_vertex,v_type);
 end;

{**********************CHECK_ORDERED*************************************}
Procedure check_ordered(duplex:matrix_duplex;num_comp:integer);
var
 found:boolean;
 i,num,n,k,component,num_no_link:integer;
 j,l:vertex_type;
begin
 num_no_link:=0;
 component:=1;
 while component<=num_comp do
 begin
  found:=false;
  for i:=0 to max_vertices do
   for j:=upper to up_low do
   if not found and ((duplex[i,j].active=true)and
      (duplex[i,j].component=component)) then
   begin
    n:=1;
```

```
   num:=duplex[i,j].out_vertices[1].num_vertex;
   found:=false;
   while (n<=num) and not found do
   begin
    k:=duplex[i,j].out_vertices[n].vertex;
    l:=duplex[i,j].out_vertices[n].type_vertex;
    if duplex[k,l].component<>component then
     found:=true
    else
     n:=n+1;
   end;
  end;
 if not found then
   num_no_link:=num_no_link+1;
 component:=component+1;
end;
if num_no_link>1 then
begin
 writeln('THERE IS NO COPT');
 readln;
 halt;
end;
end;

{********************MST_VERTEX_DEGREE*************************************}
Procedure MST_vertex_degree(var duplex:matrix_duplex);
var
 vertex,i,num_out_u,m,num_in_u,n,in_u,out_u:integer;
 j,type_of_vertex:vertex_type;
begin
  for i:=0 to max_vertices do
   for j:=upper to up_low do
    if duplex[i,j].active=true then
    begin
     in_u:=0;
     num_in_u:=duplex[i,j].in_vertices[1].num_vertex;
     if (num_in_u=1) and (duplex[i,j].in_vertices[1].edge_set=f) then
      in_u:=in_u+1
     else
      for n:=1 to num_in_u do
       if duplex[i,j].in_vertices[n].edge_set = Ec then
         in_u:=in_u+1
       else
       begin
        vertex:=duplex[i,j].in_vertices[n].vertex;
        type_of_vertex:=duplex[i,j].in_vertices[n].type_vertex;
        if (duplex[vertex,type_of_vertex].out_vertices[1].num_vertex=1) and
           (duplex[vertex,type_of_vertex].out_vertices[1].edge_set=f) then
          in_u:=in_u+1;
       end;
     out_u:=0;
     num_out_u:=duplex[i,j].out_vertices[1].num_vertex;
     if (num_out_u=1) and (duplex[i,j].out_vertices[1].edge_set=f) then
      out_u:=out_u+1
     else
```

```
    for m:=1 to num_out_u do
     if duplex[i,j].out_vertices[m].edge_set = Ec then
      out_u:=out_u+1
     else
     begin
      vertex:=duplex[i,j].out_vertices[m].vertex;
      type_of_vertex:=duplex[i,j].out_vertices[m].type_vertex;
      if (duplex[vertex,type_of_vertex].in_vertices[1].num_vertex=1) and
        (duplex[vertex,type_of_vertex].in_vertices[1].edge_set=f) then
        out_u:=out_u+1;
     end;
    duplex[i,j].mst_v_degree:=in_u - out_u;
   end;
end;


{******************MST_EDGE_COST****************************************}
procedure MST_edge_cost(var duplex:matrix_duplex);
var
  i,k,m,edge_out_num,edge_in_num:integer;
  j,n:vertex_type;
begin
  for i:=0 to max_vertices do
    for j:=upper to up_low do
    begin
     if duplex[i,j].active=true then
     begin
      edge_out_num:=duplex[i,j].out_vertices[1].num_vertex;
      edge_in_num:=duplex[i,j].in_vertices[1].num_vertex;
      for k:=1 to edge_out_num do
        if duplex[i,j].out_vertices[k].edge_set=Ec then
          duplex[i,j].out_vertices[k].mst_e_cost:=0
        else
        begin
         m:=duplex[i,j].out_vertices[k].vertex;
         n:=duplex[i,j].out_vertices[k].type_vertex;
         if (duplex[i,j].out_vertices[1].num_vertex=1) or
           (duplex[m,n].in_vertices[1].num_vertex=1) then
          duplex[i,j].out_vertices[k].mst_e_cost:=0
         else
          if (duplex[i,j].mst_v_degree>0) and
          (duplex[m,n].mst_v_degree<0) then
           duplex[i,j].out_vertices[k].mst_e_cost:=1
           else
           duplex[i,j].out_vertices[k].mst_e_cost:=2;
        end;
       for k:=1 to edge_in_num do
        if duplex[i,j].in_vertices[k].edge_set=Ec then
          duplex[i,j].in_vertices[k].mst_e_cost:=0
        else
        begin
         m:=duplex[i,j].in_vertices[k].vertex;
         n:=duplex[i,j].in_vertices[k].type_vertex;
         if (duplex[i,j].in_vertices[1].num_vertex=1) or
           (duplex[m,n].out_vertices[1].num_vertex=1) then
```

```
      duplex[i,j].in_vertices[k].mst_e_cost:=0
     else
     if (duplex[i,j].mst_v_degree<0) and
     (duplex[m,n].mst_v_degree>0) then
      duplex[i,j].in_vertices[k].mst_e_cost:=1
     else
      duplex[i,j].in_vertices[k].mst_e_cost:=2;
    end;
   end;
  end;
 end;

{********************CHECK_STRONGLY_CONNECTED****************************}
Procedure check_strongly_connected(duplex:matrix_duplex;num_comp:integer;var edge:edge_rec);
var
 auxiliar:matrix_duplex;
 comp_num,num_in,num_out:integer;
 begin
  auxiliar:=duplex;
  if (num_comp=1) and (auxiliar[0,lower].active=true) then
  begin
   edge.out_of_vertex:=0;
   edge.out_of_type:=lower;
   edge.into_vertex:=0;
   edge.into_type:=lower;
  end
  else
   if (num_comp=1) and (auxiliar[0,upper].active=true) then
   begin
    edge.out_of_vertex:=0:
    edge.out_of_type:=upper;
    edge.into_vertex:=0;
    edge.into_type:=upper;
   end;
  if num_comp>1 then
  begin
   comp_num:=0;
   num_in:=auxiliar[0,upper].in_vertices[1].num_vertex;
   auxiliar[0,upper].in_vertices[num_in+1].vertex:=0;
   auxiliar[0,upper].in_vertices[num_in+1].type_vertex:=lower;
   auxiliar[0,upper].in_vertices[1].num_vertex:=num_in+1;
   num_out:=auxiliar[0,lower].out_vertices[1].num_vertex;
   auxiliar[0,lower].out_vertices[num_out+1].vertex:=0;
   auxiliar[0,lower].out_vertices[num_out+1].type_vertex:=upper;
   auxiliar[0,lower].out_vertices[1].num_vertex:=num_out+1;
   edge.out_of_vertex:=0;
   edge.out_of_type:=lower;
   edge.into_vertex:=0;
   edge.into_type:=upper;
   StrongComponents(auxiliar,comp_num);
   if comp_num>1 then
   begin
    comp_num:=0;
    auxiliar[0,upper].in_vertices[1].num_vertex:=auxiliar[0,upper].in_vertices[1].num_vertex-1;
    auxiliar[0,lower].out_vertices[1].num_vertex:=auxiliar[0,lower].out_vertices[1].num_vertex-1;
```

```
    num_in:=auxiliar[0,lower].in_vertices[1].num_vertex;
    auxiliar[0,lower].in_vertices[num_in+1].vertex:=0;
    auxiliar[0,lower].in_vertices[num_in+1].type_vertex:=upper;
    auxiliar[0,lower].in_vertices[1].num_vertex:=num_in+1;
    num_out:=auxiliar[0,upper].out_vertices[1].num_vertex;
    auxiliar[0,upper].out_vertices[num_out+1].vertex:=0;
    auxiliar[0,upper].out_vertices[num_out+1].type_vertex:=lower;
    auxiliar[0,upper].out_vertices[1].num_vertex:=num_out+1;
    edge.out_of_vertex:=0;
    edge.out_of_type:=upper;
    edge.into_vertex:=0;
    edge.into_type:=lower;
    StrongComponents(auxiliar,comp_num);
    if comp_num>1 then
     begin
      writeln('THERE IS NO COPT');
      readln;
      halt;
     end;
   end;
  end;
 end;

{**********************MINSPANNINGTREE**********************************}
procedure MinSpanningTree(var duplex:matrix_duplex;num_comp:integer);
 type
  vertexdata=
  record
    vertex:integer;
    vert_type:vertex_type;
  end;
 var
  list_fringe:array[1..max_nodes] of vertexdata;
  parent_vertex,vertex_out_num,vertex_in_num,total_vertex,x,n,actual_v,p,g,y,k,fringelist:integer;
  num,i,edgecount,q,s,pos_min_cost,component,num_parent:integer;
  found,stuck:boolean;
  parent_type,m,j,actual_t_v,r,h:vertex_type;
 begin
  component:=1;
  while component<=num_comp do
  begin
  n:=0;
  found:=false;
  for i:=0 to max_vertices do
   for j:=upper to up_low do
    if (duplex[i,j].active=true) and (duplex[i,j].component=component) then
    begin
     n:=n+1;
     if found=false then
     begin
       found:=true;
       parent_vertex:=i;
       parent_type:=j;
     end;
    end;
```

```
if n=1 then
begin
 duplex[parent_vertex,parent_type].status:=intree;
 duplex[parent_vertex,parent_type].parent_vertex:=parent_vertex;
 duplex[parent_vertex,parent_type].parent_type:=parent_type;
end
else
begin
k:=parent_vertex;
m:=parent_type;
edgecount:=0;
fringelist:=0;
for i:=0 to max_vertices do
 for j:=upper to up_low do
  if (i=parent_vertex)and (j=parent_type) then
  begin
   duplex[i,j].status:=intree;
   duplex[i,j].parent_vertex:=parent_vertex;
   duplex[i,j].parent_type:=parent_type;
  end
  else
  if (duplex[i,j].active=true) and (duplex[i,j].component=component) then
    duplex[i,j].status:=unseen;
 stuck:=false;
 while (edgecount<n-1) and not stuck do
 begin
  vertex_out_num:=duplex[k,m].out_vertices[1].num_vertex;
  vertex_in_num:=duplex[k,m].in_vertices[1].num_vertex;
  total_vertex:=vertex_out_num+vertex_in_num;
  y:=1;
  x:=1;
  while  y<=total_vertex do
  begin
   if y<=vertex_out_num then
   begin
    actual_v:=duplex[k,m].out_vertices[y].vertex;
    actual_t_v:=duplex[k,m].out_vertices[y].type_vertex;
    if duplex[actual_v,actual_t_v].component=component then
    begin
    if (duplex[actual_v,actual_t_v].status = fringe) and
      (duplex[k,m].out_vertices[y].mst_e_cost< duplex[actual_v,actual_t_v].fringe_cost) then
    begin
     duplex[actual_v,actual_t_v].parent_vertex:=k;
     duplex[actual_v,actual_t_v].parent_type:=m;
     duplex[actual_v,actual_t_v].fringe_cost:=duplex[k,m].out_vertices[y].mst_e_cost;
    end;
    if (duplex[actual_v,actual_t_v].status=unseen) and (duplex[actual_v,actual_t_v].active=true) then
    begin
     duplex[actual_v,actual_t_v].status:=fringe;
     fringelist:=fringelist+1;
     list_fringe[fringelist].vertex:=actual_v;
     list_fringe[fringelist].vert_type:=actual_t_v;
     duplex[actual_v,actual_t_v].parent_vertex:=k;
     duplex[actual_v,actual_t_v].parent_type:=m;
     duplex[actual_v,actual_t_v].fringe_cost:=duplex[k,m].out_vertices[y].mst_e_cost;
```

```
 end;
 end;
end
else
begin
 actual_v:=duplex[k,m].in_vertices[x].vertex;
 actual_t_v:=duplex[k,m].in_vertices[x].type_vertex;
 if duplex[actual_v,actual_t_v].component=component then
 begin
 if (duplex[actual_v,actual_t_v].status = fringe) and
  (duplex[k,m].in_vertices[x].mst_e_cost< duplex[actual_v,actual_t_v].fringe_cost) then
 begin
  duplex[actual_v,actual_t_v].parent_vertex:=k;
  duplex[actual_v,actual_t_v].parent_type:=m;
  duplex[actual_v,actual_t_v].fringe_cost:=duplex[k,m].in_vertices[x].mst_e_cost;
 end;
 if (duplex[actual_v,actual_t_v].status=unseen) and (duplex[actual_v,actual_t_v].active=true) then
 begin
  duplex[actual_v,actual_t_v].status:=fringe;
  fringelist:=fringelist+1;
  list_fringe[fringelist].vertex:=actual_v;
  list_fringe[fringelist].vert_type:=actual_t_v;
  duplex[actual_v,actual_t_v].parent_vertex:=k;
  duplex[actual_v,actual_t_v].parent_type:=m;
  duplex[actual_v,actual_t_v].fringe_cost:=duplex[k,m].in_vertices[x].mst_e_cost;
 end;
 end;
 x:=x+1;
end;
y:=y+1;
end;
if fringelist=0 then stuck:=true
else
 begin
  pos_min_cost:=1;
  if fringelist >1 then
  begin
   for s:=2 to fringelist do
   begin
    p:=list_fringe[s].vertex;
    r:=list_fringe[s].vert_type;
    g:=list_fringe[pos_min_cost].vertex;
    h:=list_fringe[pos_min_cost].vert_type;
     if duplex[p,r].fringe_cost<
       duplex[g,h].fringe_cost then
      pos_min_cost:=s;
   end;
  end;
  k:=list_fringe[pos_min_cost].vertex;
  m:=list_fringe[pos_min_cost].vert_type;
  q:=pos_min_cost;
  while q<fringelist do
  begin
   list_fringe[q]:=list_fringe[q+1];
   q:=q+1;
```

```
    end;
    fringelist:=fringelist-1;
    duplex[k,m].status:=intree;
    edgecount:=edgecount+1;
   end;
   end;
   end;
   component:=component+1;
  end;
 end;

{**************************LINK_EDGES************************************}
 procedure link_edges(var duplex:matrix_duplex;num_comp:integer);
 type
  vertexdata= record
           vertex:integer;
           vert_type:vertex_type;
           other_vertex:integer;
           other_type:vertex_type;
           cost:integer;
          end;
 var
  pos_vertex,i,component,edge_link,num_in,num_links,num_parent,position:integer;
  pos_type,
  j,edge_type:vertex_type;
  min_link:array[1..max_vertices] of vertexdata;

 begin
  if num_comp>1 then
  begin
   for i:=1 to max_vertices do
     min_link[i].cost:=3;
   for i:=0 to max_vertices do
    for j:=upper to up_low do
    begin
     if (duplex[i,j].active=true) then
     begin
      num_in:=duplex[i,j].in_vertices[1].num_vertex;
      component:=duplex[i,j].component;
      for num_links:=1 to num_in do
      begin
       edge_link:=duplex[i,j].in_vertices[num_links].vertex;
       edge_type:=duplex[i,j].in_vertices[num_links].type_vertex;
       if (duplex[edge_link,edge_type].component=component+1) and
         (duplex[i,j].in_vertices[num_links].mst_e_cost<min_link[component].cost) then
       begin
        min_link[component].vertex:=i;
        min_link[component].vert_type:=j;
        min_link[component].other_vertex:=edge_link;
        min_link[component].other_type:=edge_type;
        min_link[component].cost:=duplex[i,j].in_vertices[num_links].mst_e_cost;
       end;
      end;
     end;
    end;
```

```
    for i:=1 to num_comp-1 do
    begin
     pos_vertex:=min_link[i].vertex;
     pos_type:=min_link[i].vert_type;
     duplex[pos_vertex,pos_type].link:=true;
     duplex[pos_vertex,pos_type].link_vertex:=min_link[i].other_vertex;
     duplex[pos_vertex,pos_type].link_type:=min_link[i].other_type;
     pos_vertex:=min_link[i].other_vertex;
     pos_type:=min_link[i].other_type;
     duplex[pos_vertex,pos_type].link:=true;
     duplex[pos_vertex,pos_type].link_vertex:=min_link[i].vertex;
     duplex[pos_vertex,pos_type].link_type:=min_link[i].vert_type;
    end;
   end;
  end;

{************************ADD_EXTRA_EDGE**********************************}
procedure add_extra_edge(var duplex:matrix_duplex;edge:edge_rec);
 var
  i,num:integer;
  j:vertex_type;
 begin
  i:=edge.out_of_vertex;
  j:=edge.out_of_type;
  num:=duplex[i,j].out_vertices[1].num_vertex+1;
  duplex[i,j].out_vertices[num].vertex:=edge.into_vertex;
  duplex[i,j].out_vertices[num].type_vertex:=edge.into_type;
  duplex[i,j].out_vertices[num].edge_set:=Ec;
  duplex[i,j].out_vertices[1].num_vertex:=num;
  i:=edge.into_vertex;
  j:=edge.into_type;
  num:=duplex[i,j].in_vertices[1].num_vertex+1;
  duplex[i,j].in_vertices[num].vertex:=edge.out_of_vertex;
  duplex[i,j].in_vertices[num].type_vertex:=edge.out_of_type;
  duplex[i,j].in_vertices[num].edge_set:=Ec;
  duplex[i,j].in_vertices[1].num_vertex:=num;
 end;

{************************NET_DEGREE*****************************************}
Procedure net_degree(var duplex:matrix_duplex);
var
 negative_degree,positive_degree,num_out,num_in,z,x,i,num_out_u,n,num_in_u,in_u,out_u:integer;
 j,y:vertex_type;
 found:boolean;
begin
 for i:=0 to max_vertices do
  for j:=upper to up_low do
  if duplex[i,j].active=true then
  begin
   in_u:=0;
   num_in_u:=duplex[i,j].in_vertices[1].num_vertex;
   for n:=1 to num_in_u do
   begin
    if ((duplex[i,j].link=true) and
      (duplex[i,j].in_vertices[n].vertex=duplex[i,j].link_vertex)) and
```

```
  ((duplex[i,j].in_vertices[n].type_vertex=duplex[i,j].link_type) and
  (duplex[i,j].in_vertices[n].edge_set=f)) then
 in_u:=in_u+1;
if ((duplex[i,j].in_vertices[n].vertex=duplex[i,j].parent_vertex) and
  (duplex[i,j].in_vertices[n].type_vertex=duplex[i,j].parent_type)) and
  (duplex[i,j].in_vertices[n].edge_set=f) then
begin
 num_out:=duplex[i,j].out_vertices[1].num_vertex;
 z:=1;
 found:=false;
 while (z<=num_out) and not found do
  if ((duplex[i,j].out_vertices[z].vertex=duplex[i,j].parent_vertex) and
  (duplex[i,j].out_vertices[z].type_vertex=duplex[i,j].parent_type)) and
  (duplex[i,j].out_vertices[z].edge_set=Ec) then
    found:=true
  else
   z:=z+1;
 if not found then
  in_u:=in_u+1;
end;
x:=duplex[i,j].in_vertices[n].vertex;
y:=duplex[i,j].in_vertices[n].type_vertex;
if ((duplex[x,y].parent_vertex=i) and
   (duplex[x,y].parent_type=j)) and
   (duplex[i,j].in_vertices[n].edge_set=f) then
begin
 num_out:=duplex[i,j].out_vertices[1].num_vertex;
  z:=1;
  found:=false;
  while (z<=num_out) and not found do
   if ((duplex[i,j].out_vertices[z].vertex=x) and
   (duplex[i,j].out_vertices[z].type_vertex=y)) and
   (duplex[i,j].out_vertices[z].edge_set=Ec) then
     found:=true
   else
    z:=z+1;
  if not found then
  in_u:=in_u+1;
 end;
 if duplex[i,j].in_vertices[n].edge_set=Ec then
  in_u:=in_u+1;
end;
out_u:=0;
num_out_u:=duplex[i,j].out_vertices[1].num_vertex;
for n:=1 to num_out_u do
begin
 if ((duplex[i,j].link=true) and
   (duplex[i,j].out_vertices[n].vertex=duplex[i,j].link_vertex)) and
   ((duplex[i,j].out_vertices[n].type_vertex=duplex[i,j].link_type) and
   (duplex[i,j].out_vertices[n].edge_set=f)) then
  out_u:=out_u+1;
 if ((duplex[i,j].out_vertices[n].vertex=duplex[i,j].parent_vertex) and
   (duplex[i,j].out_vertices[n].type_vertex=duplex[i,j].parent_type)) and
   (duplex[i,j].out_vertices[n].edge_set=f) then
 begin
```

```
    num_in:=duplex[i,j].in_vertices[1].num_vertex;
    z:=1;
    found:=false;
    while (z<=num_in) and not found do
     if ((duplex[i,j].in_vertices[z].vertex=duplex[i,j].parent_vertex) and
     (duplex[i,j].in_vertices[z].type_vertex=duplex[i,j].parent_type)) and
     (duplex[i,j].in_vertices[z].edge_set=Ec) then
       found:=true
     else
      z:=z+1;
    if not found then
     out_u:=out_u+1;
   end;
  x:=duplex[i,j].out_vertices[n].vertex;
  y:=duplex[i,j].out_vertices[n].type_vertex;
  if ((duplex[x,y].parent_vertex=i) and
     (duplex[x,y].parent_type=j)) and
     (duplex[i,j].out_vertices[n].edge_set=f) then
   begin
    num_in:=duplex[i,j].in_vertices[1].num_vertex;
    z:=1;
    found:=false;
    while (z<=num_in) and not found do
     if ((duplex[i,j].in_vertices[z].vertex=x) and
     (duplex[i,j].in_vertices[z].type_vertex=y)) and
     (duplex[i,j].in_vertices[z].edge_set=Ec) then
       found:=true
     else
      z:=z+1;
    if not found then
    out_u:=out_u+1;
   end;
   if duplex[i,j].out_vertices[n].edge_set=Ec then
    out_u:=out_u+1;
  end;
  duplex[i,j].net_degree:=in_u-out_u;
  end;
positive_degree:=0;
negative_degree:=0;
for i:=0 to max_vertices do
 for j:=upper to up_low do
  if duplex[i,j].active=true then
  begin
   if j=upper then
    writeln(i,',upper; net degree: ',duplex[i,j].net_degree);
   if j=lower then
    writeln(i,',lower; net degree: ',duplex[i,j].net_degree);
   if j=up_low then
    writeln(i,',up_low; net degree: ',duplex[i,j].net_degree);
   readln;
   if duplex[i,j].net_degree>0 then
    positive_degree:=positive_degree+duplex[i,j].net_degree;
   if duplex[i,j].net_degree<0 then
    negative_degree:=negative_degree+((-1)*duplex[i,j].net_degree);
  end;
```

```
 if negative_degree<>positive_degree then
  begin
   writeln('THERE IS NO COPT');
   readln;
   halt;
  end;
end;

{*********************BELLMAN_FORD*****************************************}
procedure Bellman_Ford(var duplex:matrix_duplex;var spath:boolean);
var
 num,n,i,s,k,num_in,x,min_vertex,min_label,cost:integer;
 cycle,same,changed:boolean;
 j,m,min_type:vertex_type;
begin
  n:=0;
  num:=0;
  for i:=0 to max_vertices do
   for j:=upper to up_low do
   if duplex[i,j].active=true then
   begin
    if (duplex[i,j].net_degree>0) and
      (duplex[i,j].flow_from_s<duplex[i,j].net_degree) then
    begin
     duplex[i,j].old_label:=0;
     duplex[i,j].old_pre_vertex:=i;
     duplex[i,j].old_pre_type:=j;
     num:=num+1;
    end
    else
    begin
     duplex[i,j].old_label:=1000;
     duplex[i,j].old_pre_vertex:=i;
     duplex[i,j].old_pre_type:=j;
    end;
    n:=n+1;
   end;
   writeln('num: ',num);
   readln;
  if num=0 then spath:=false;
  same:=false;
  cycle:=false;
  s:=2;
  while ((not same)  and (not cycle)) and (spath) do
  begin
   changed:=false;
   for i:=0 to max_vertices do
    for j:=upper to up_low do
    if duplex[i,j].active=true then
    begin
     num_in:=duplex[i,j].in_vertices[1].num_vertex;
     x:=2;
     min_vertex:=duplex[i,j].in_vertices[1].vertex;
     min_type:=duplex[i,j].in_vertices[1].type_vertex;
     if (duplex[i,j].in_vertices[1].edge_set=Ec) and
```

```
  (duplex[i,j].in_vertices[1].duplicate=false) then
  cost:=1;
if (duplex[i,j].in_vertices[1].edge_set=Ec) and
  (duplex[i,j].in_vertices[1].duplicate=true) then
 cost:=-1;
if duplex[i,j].in_vertices[1].edge_set=f then
 cost:=0;
min_label:=duplex[min_vertex,min_type].old_label+cost;
while x<=num_in do
begin
  if (duplex[i,j].in_vertices[x].edge_set=Ec) and
    (duplex[i,j].in_vertices[x].duplicate=false) then
   cost:=1;
  if (duplex[i,j].in_vertices[x].edge_set=Ec) and
    (duplex[i,j].in_vertices[x].duplicate=true) then
   cost:=-1;
  if duplex[i,j].in_vertices[x].edge_set=f then
   cost:=0;
  k:=duplex[i,j].in_vertices[x].vertex;
  m:=duplex[i,j].in_vertices[x].type_vertex;
  if duplex[k,m].old_label+cost<min_label then
  begin
   min_vertex:=k;
   min_type:=m;
   min_label:=duplex[k,m].old_label+cost;
  end;
  x:=x+1;
end;
if min_label<duplex[i,j].old_label then
begin
 changed:=true;
 duplex[i,j].new_pre_vertex:=min_vertex;
 duplex[i,j].new_pre_type:=min_type;
 duplex[i,j].new_label:=min_label;
end
else
begin
 duplex[i,j].new_pre_vertex:=duplex[i,j].old_pre_vertex;
 duplex[i,j].new_pre_type:=duplex[i,j].old_pre_type;
 duplex[i,j].new_label:=duplex[i,j].old_label;
end;
end;
if changed and (s=n) then
begin
 cycle:=true;
 writeln('THERE IS A CYCLE');
 readln;
end;
if not changed and (s<=n) then
 same:=true;
if (s<n) and changed then
 for i:=0 to max_vertices do
  for j:=upper to up_low do
   if duplex[i,j].active=true then
   begin
```

```
     duplex[i,j].old_pre_vertex:=duplex[i,j].new_pre_vertex;
     duplex[i,j].old_pre_type:=duplex[i,j].new_pre_type;
     duplex[i,j].old_label:=duplex[i,j].new_label;
    end;
   s:=s+1;
  end;
 end;

{***************************MIN_COST_MAXFLOW*********************************}
procedure min_cost_maxflow(var duplex:matrix_duplex;edge:edge_rec);
var
 aux_graph:matrix_duplex;
 y,z,num,vertex,edges_in,num_in_aux,num_in_duplex,num_out_duplex:integer;
 edges_out,i,num_in,num_out,k,x,pre_vertex,min_capacity,min_label,min_vertex:integer;
 type_of_vertex,j,m,pre_type,min_type:vertex_type;
 found,another_path:boolean;
begin
 aux_graph:=duplex;
 for i:=0 to max_vertices do
  for j:=upper to up_low do
  if ((aux_graph[i,j].active=true) and (i=edge.into_vertex)) and
   (j=edge.into_type) then
  begin
   num_in:=aux_graph[i,j].in_vertices[1].num_vertex;
   found:=false;
   x:=1;
   while not found do
    if (aux_graph[i,j].in_vertices[x].vertex=edge.out_of_vertex) and
     (aux_graph[i,j].in_vertices[x].type_vertex=edge.out_of_type) then
     found:=true
    else
     x:=x+1;
    while x<num_in do
     aux_graph[i,j].in_vertices[x]:=aux_graph[i,j].in_vertices[x+1];
    aux_graph[i,j].in_vertices[1].num_vertex:=aux_graph[i,j].in_vertices[1].num_vertex-1;
   end;
  another_path:=true;
  Bellman_Ford(aux_graph,another_path);
  writeln('After bellman_ford');
  readln;
  while another_path do
  begin
   min_label:=1001;
   for i:=0 to max_vertices do
    for j:=upper to up_low do
     if ((aux_graph[i,j].active=true) and
      (aux_graph[i,j].flow_to_t<aux_graph[i,j].net_degree*(-1))) and
      (aux_graph[i,j].new_label<min_label) then
     begin
      min_vertex:=i;
      min_type:=j;
      min_label:=aux_graph[i,j].new_label;
     end;
     min_capacity:=2000;
     k:=min_vertex;
```

```
m:=min_type;
pre_vertex:=aux_graph[k,m].old_pre_vertex;
pre_type:=aux_graph[k,m].old_pre_type;
while (k<>pre_vertex) or (m<>pre_type) do
begin
 num:=aux_graph[k,m].in_vertices[1].num_vertex;
 x:=1;
 found:=false;
 while (x<=num) and not found do
  if (aux_graph[k,m].in_vertices[x].vertex=pre_vertex) and
   (aux_graph[k,m].in_vertices[x].type_vertex=pre_type) then
  found:=true
  else
   x:=x+1;
 if aux_graph[k,m].in_vertices[x].capacity<min_capacity then
  min_capacity:=aux_graph[k,m].in_vertices[x].capacity;
 k:=pre_vertex;
 m:=pre_type;
 pre_vertex:=aux_graph[k,m].new_pre_vertex;
 pre_type:=aux_graph[k,m].new_pre_type;
end;
if (aux_graph[k,m].net_degree-aux_graph[k,m].flow_from_s)<min_capacity then
 min_capacity:=aux_graph[k,m].net_degree-aux_graph[k,m].flow_from_s;
k:=min_vertex;
m:=min_type;
if (aux_graph[k,m].net_degree*(-1)-aux_graph[k,m].flow_to_t)<min_capacity then
 min_capacity:=aux_graph[k,m].net_degree*(-1)-aux_graph[k,m].flow_to_t;
aux_graph[k,m].flow_to_t:=aux_graph[k,m].flow_to_t+min_capacity;
pre_vertex:=aux_graph[k,m].old_pre_vertex;
pre_type:=aux_graph[k,m].old_pre_type;
while (k<>pre_vertex) or (m<>pre_type) do
begin
 num:=aux_graph[k,m].in_vertices[1].num_vertex;
 x:=1;
 found:=false;
 while (x<=num) and not found do
  if (aux_graph[k,m].in_vertices[x].vertex=pre_vertex) and
   (aux_graph[k,m].in_vertices[x].type_vertex=pre_type) then
   found:=true
  else
   x:=x+1;
 if aux_graph[k,m].in_vertices[x].duplicate=true then
 begin
  aux_graph[k,m].in_vertices[x].capacity:=aux_graph[k,m].in_vertices[x].capacity-min_capacity;
  vertex:=aux_graph[k,m].in_vertices[x].vertex;
  type_of_vertex:=aux_graph[k,m].in_vertices[x].type_vertex;
  num_out:=aux_graph[vertex,type_of_vertex].out_vertices[1].num_vertex;
  x:=1;
  found:=false;
  while (x<=num_out) and not found do
   if ((aux_graph[vertex,type_of_vertex].out_vertices[x].vertex=k) and
    (aux_graph[vertex,type_of_vertex].out_vertices[x].type_vertex=m)) and
    (aux_graph[vertex,type_of_vertex].out_vertices[x].duplicate=true) then
    found:=true
   else
```

```
   x:=x+1;
  aux_graph[vertex,type_of_vertex].out_vertices[x].capacity:=
  aux_graph[vertex,type_of_vertex].out_vertices[x].capacity-min_capacity;
 end
 else
 begin
  num_out:=aux_graph[k,m].out_vertices[1].num_vertex;
  x:=1;
  found:=false;
  while (x<=num_out) and not found do
   if ((aux_graph[k,m].out_vertices[x].vertex=pre_vertex) and
    (aux_graph[k,m].out_vertices[x].type_vertex=pre_type)) and
    (aux_graph[k,m].out_vertices[x].duplicate=true) then
     found:=true
   else
    x:=x+1;
  if found then
  begin  {si el duplicado ya estaba creado}
   aux_graph[k,m].out_vertices[x].capacity:=aux_graph[k,m].out_vertices[x].capacity+min_capacity;
   vertex:=aux_graph[k,m].out_vertices[x].vertex;
   type_of_vertex:=aux_graph[k,m].out_vertices[x].type_vertex;
   num_in:=aux_graph[vertex,type_of_vertex].in_vertices[1].num_vertex;
   x:=1;
   found:=false;
   while (x<=num_in) and not found do
    if ((aux_graph[vertex,type_of_vertex].in_vertices[x].vertex=k) and
     (aux_graph[vertex,type_of_vertex].in_vertices[x].type_vertex=m)) and
     (aux_graph[vertex,type_of_vertex].in_vertices[x].duplicate=true) then
     found:=true
    else
     x:=x+1;
   aux_graph[vertex,type_of_vertex].in_vertices[x].capacity:=
   aux_graph[vertex,type_of_vertex].in_vertices[x].capacity+min_capacity;
  end
  else{if the duplicated edge hasn't been created}
  begin
   num_out:=aux_graph[k,m].out_vertices[1].num_vertex+1;
   aux_graph[k,m].out_vertices[num_out].vertex:=pre_vertex;
   aux_graph[k,m].out_vertices[num_out].type_vertex:=pre_type;
   num_in:=aux_graph[k,m].in_vertices[1].num_vertex;
   x:=1;
   found:=false;
   while not found and (x<=num_in) do
    if (aux_graph[k,m].in_vertices[x].vertex=pre_vertex) and
      (aux_graph[k,m].in_vertices[x].type_vertex=pre_type) then
     found:=true
    else
     x:=x+1;
   aux_graph[k,m].out_vertices[num_out].edge_set:=
   aux_graph[k,m].in_vertices[x].edge_set;
   aux_graph[k,m].out_vertices[num_out].capacity:=min_capacity;
   aux_graph[k,m].out_vertices[num_out].duplicate:=true;
   aux_graph[k,m].out_vertices[1].num_vertex:=num_out;
   num_in:=aux_graph[pre_vertex,pre_type].in_vertices[1].num_vertex+1;
   aux_graph[pre_vertex,pre_type].in_vertices[num_in].vertex:=k;
```

```
     aux_graph[pre_vertex,pre_type].in_vertices[num_in].type_vertex:=m;
     aux_graph[pre_vertex,pre_type].in_vertices[num_in].edge_set:=
     aux_graph[k,m].out_vertices[num_out].edge_set;
     aux_graph[pre_vertex,pre_type].in_vertices[num_in].capacity:=min_capacity;
     aux_graph[pre_vertex,pre_type].in_vertices[num_in].duplicate:=true;
     aux_graph[pre_vertex,pre_type].in_vertices[1].num_vertex:=num_in;
    end;
   end;
   k:=pre_vertex;
   m:=pre_type;
   pre_vertex:=aux_graph[k,m].new_pre_vertex;
   pre_type:=aux_graph[k,m].new_pre_type;
  end;
  if (k=pre_vertex) and (m=pre_type) then
   aux_graph[k,m].flow_from_s:=aux_graph[k,m].flow_from_s+min_capacity;
  another_path:=true;
  Bellman_Ford(aux_graph,another_path);
  writeln('after the second bellman ford');
  readln;
 end;
for i:=0 to max_vertices do
 for j:=upper to up_low do
 begin
  if duplex[i,j].active=true then
  begin
   x:=1;
   num_in_duplex:=duplex[i,j].in_vertices[1].num_vertex;
   num_out_duplex:=duplex[i,j].out_vertices[1].num_vertex;
   while x<=num_out_duplex do
   begin
    if duplex[i,j].out_vertices[x].edge_set=Ec then
     duplex[i,j].out_vertices[x].capacity:=1
    else
    begin
     if ((duplex[i,j].link=true) and
       (duplex[i,j].link_vertex=duplex[i,j].out_vertices[x].vertex)) and
       (duplex[i,j].link_type=duplex[i,j].out_vertices[x].type_vertex) then
     begin
      vertex:=duplex[i,j].link_vertex;
      type_of_vertex:=duplex[i,j].link_type;
      if duplex[i,j].component=duplex[vertex,type_of_vertex].component+1 then
       duplex[i,j].out_vertices[x].capacity:=1;
     end {if link=true}
     else
      if (duplex[i,j].parent_vertex=duplex[i,j].out_vertices[x].vertex) and
        (duplex[i,j].parent_type=duplex[i,j].out_vertices[x].type_vertex) then
      begin
       z:=1;
       found:=false;
       while (z<=num_in_duplex) and not found do
        if ((duplex[i,j].in_vertices[z].vertex=duplex[i,j].parent_vertex) and
         (duplex[i,j].in_vertices[z].type_vertex=duplex[i,j].parent_type)) and
          (duplex[i,j].in_vertices[z].edge_set=Ec) then
         found:=true
        else
```

```
    z:=z+1;
   if found=false then
    duplex[i,j].out_vertices[x].capacity:=1
   else
    duplex[i,j].out_verdices[x].capacity:=0;
  end {if parent_vertex}
  else
  begin
   vertex:=duplex[i,j].out_vertices[x].vertex;
   type_of_vertex:=duplex[i,j].out_vertices[x].type_vertex;
   if (duplex[vertex,type_of_vertex].parent_vertex=i) and
     (duplex[vertex,type_of_vertex].parent_type=j) then
     duplex[i,j].out_vertices[x].capacity:=1
   else
    duplex[i,j].out_vertices[x].capacity:=0;
  end;
 end;{if edge set=f}
 vertex:=duplex[i,j].out_vertices[x].vertex;
 type_of_vertex:=duplex[i,j].out_vertices[x].type_vertex;
 z:=1;
 found:=false;
 num_in:=duplex[vertex,type_of_vertex].in_vertices[1].num_vertex;
 while (z<=num_in) and not found do
  if (duplex[vertex,type_of_vertex].in_vertices[z].vertex=i) and
   (duplex[vertex,type_of_vertex].in_vertices[z].type_vertex=j) then
     found:=true
  else
   z:=z+1;
 duplex[vertex,type_of_vertex].in_vertices[z].capacity:=duplex[i,j].out_vertices[x].capacity;
 x:=x+1;
end;{while x}
end; {if active}
end;{for}


for i:=0 to max_vertices do
 for j:=upper to up_low do
 begin
  if duplex[i,j].active=true then
  begin
   num_in_aux:=aux_graph[i,j].in_vertices[1].num_vertex;
   num_in_duplex:=duplex[i,j].in_vertices[1].num_vertex;
   num_out_duplex:=duplex[i,j].out_vertices[1].num_vertex;
   x:=1;
   while x<=num_in_aux do
   begin
    if aux_graph[i,j].in_vertices[x].duplicate=true then
    begin
     y:=1;
     found:=false;
     while (y<=num_out_duplex) and not found do
      if (duplex[i,j].out_vertices[y].vertex=aux_graph[i,j].in_vertices[x].vertex) and
       (duplex[i,j].out_vertices[y].type_vertex=aux_graph[i,j].in_vertices[x].type_vertex) then
       found:=true
      else
```

```
       y:=y+1;
       duplex[i,j].out_vertices[y].capacity:=
       duplex[i,j].out_vertices[y].capacity+aux_graph[i,j].in_vertices[x].capacity;
       vertex:=duplex[i,j].out_vertices[y].vertex;
       type_of_vertex:=duplex[i,j].out_vertices[y].type_vertex;
       z:=1;
       found:=false;
       num_in:=duplex[vertex,type_of_vertex].in_vertices[1].num_vertex;
       while (z<=num_in) and not found do
        if (duplex[vertex,type_of_vertex].in_vertices[z].vertex=i) and
        (duplex[vertex,type_of_vertex].in_vertices[z].type_vertex=j) then
          found:=true
        else
         z:=z+1;
       duplex[vertex,type_of_vertex].in_vertices[z].capacity:=duplex[i,j].out_vertices[y].capacity;
      end; {if duplicate=true}
      x:=x+1;
     end;{while x}
    end;{if active}
   end; {for}
  end; {procedure}

{***************************RPT*******************************************}
procedure RPT(duplex:matrix_duplex;var vertex_list:list);
var
 vertex,aux_vertex,aux_i,num_out,x,i:integer;
 vertex_t,aux_type,aux_j,j:vertex_type;
 stuck,end_list,found:boolean;
 wp,wq:pointer;
 q,p,aux:pointer;
 aux_list:list;
begin
 found:=false;
 for i:=0 to max_vertices do
  for j:=upper to up_low do
   if (duplex[i,j].active=true) and not found then
   begin
    found:=true;
    aux_i:=i;
    aux_j:=j;
   end;  {if active}
   stuck:=false;
   vertex_list.head:=nil;
   while not stuck do
   begin
    i:=aux_i;
    j:=aux_j;
    new(p);
    p^.vertex:=i;
    p^.v_type:=j;
    p^.next:=nil;
    if vertex_list.head=nil then
    begin
     vertex_list.head:=p;
     wp:=vertex_list.head;
```

```
end
else
begin
 wp^.next:=p;
 wp:=p
end;
found:=false;
num_out:=duplex[i,j].out_vertices[1].num_vertex;
x:=1;
while (x<=num_out) and not found do
 if duplex[i,j].out_vertices[x].capacity>0 then
 begin
  found:=true;
  duplex[i,j].out_vertices[x].capacity:=duplex[i,j].out_vertices[x].capacity-1;
  aux_i:=duplex[i,j].out_vertices[x].vertex;
  aux_j:=duplex[i,j].out_vertices[x].type_vertex;
 end
 else
  x:=x+1;
 if (x>num_out) and not found then
  stuck:=true;
end; {while not stuck}
p:=vertex_list.head;
while (p<>nil) do
begin
 i:=p^.vertex;
 j:=p^.v_type;
 x:=1;
 num_out:=duplex[i,j].out_vertices[1].num_vertex;
 while (x<=num_out) and not found do
  if duplex[i,j].out_vertices[x].capacity>0 then
  begin
   found:=true;
   duplex[i,j].out_vertices[x].capacity:=duplex[i,j].out_vertices[x].capacity-1;
  end
  else
   x:=x+1;
 if not found then
  p:=p^.next
 else
 begin
  end_list:=false;
  aux_list.head:=nil;
  while not end_list do
  begin
   aux_i:=duplex[i,j].out_vertices[x].vertex;
   aux_j:=duplex[i,j].out_vertices[x].type_vertex;
   new(q);
   q^.vertex:=aux_i;
   q^.v_type:=aux_j;
   q^.next:=nil;
   if aux_list.head=nil then
   begin
    aux_list.head:=q;
    wq:=aux_list.head;
```

```
    end
    else
    begin
     wq^.next:=q;
     wq:=q;
    end;
    i:=aux_i;
    j:=aux_j;
    found:=false;
    num_out:=duplex[i,j].out_vertices[1].num_vertex;
    x:=1;
    while (x<=num_out) and not found do
     if duplex[i,j].out_vertices[x].capacity>0 then
     begin
      found:=true;
      duplex[i,j].out_vertices[x].capacity:=duplex[i,j].out_vertices[x].capacity-1;
     end
     else
      x:=x+1;
    if (x>num_out) and not found then
      end_list:=true;
   end; {while not end_list}
   if end_list=true then
   begin
    aux:=p;
    p:=p^.next;
    q^.next:=p;
    q:=aux_list.head;
    aux^.next:=q;
   end;
  end;{else}
 end;{while p<>nil}
end;{procedure}

{*************************COPT*******************************************}
procedure COPT (vertex_list:list;var tour:text);
var
 p:pointer;
 previous,x,vertex,y:integer;
 vertex_t:vertex_type;
 file_name,drive,output_file:string;
begin
 file_name:=' ';
 write('Name of the output file: ');
 readln(output_file);
 write('Drive where the output file is going to be located: ');
 readln(drive);
 file_name:=drive+':'+output_file;
 assign(tour,file_name);
 rewrite(tour);
 x:=0;
 p:=vertex_list.head;
 previous:=p^.vertex;
 write(tour,previous);
 y:=1;
```

```
p:=p^.next;
while p<>nil do
begin
 vertex:=p^.vertex;
 if (vertex<>previous) then
 begin
  x:=x+1;
  previous:=vertex;
  write(tour,', ',previous);
  y:=1;
 end
 else
  y:=y+1;
 if y>2 then
 begin
  x:=x+1;
  write(tour,', ',previous);
 end;
 p:=p^.next;
end;
writeln(tour);
writeln(tour,'Length of the COPT: ',x);
close(tour);
end;{procedure}

{******************************MAIN****************************************}
begin
 clrscr;
 num_comp:=0;
 fill_original(original,orig_probl);
 initiallize_duplex(orig_probl,duplex);
 create_Ec_edge(orig_probl,duplex);
 remove_edges_and_vertices(duplex,orig_probl);
 writeln('Before strongcomponents');
 readln;
 StrongComponents(duplex,num_comp);
 writeln('After strongcomponents');
 readln;
 check_ordered(duplex,num_comp);
 MST_vertex_degree(duplex);
 MST_edge_cost(duplex);
 check_strongly_connected(duplex,num_comp,edge);
 MinSpanningTree(duplex,num_comp);
 link_edges(duplex,num_comp);
 add_extra_edge(duplex,edge);
 net_degree(duplex);
 writeln('Before min_cost_maxflow');
 readln;
 min_cost_maxflow(duplex,edge);
 writeln('After min_cost_maxflow');
 readln;
 RPT(duplex,vertex_list);
 COPT(vertex_list,tour);
 writeln('THE OUTPUT FILE HAS BEEN CREATED');
 readln;
```

```
end.
```

References

[1] A. V. Aho, A.T. Dahbura, D. Lee, and M. U. Uyar, "An Optimization Technique For Protocol Conformance Test Sequence Generation Based On UIO Sequences And Rural Chinese Postman Tours", in Protocol Specification, Testing, and Verification, VII, S. Aggarwal and K. Sabnani (Editors), Elservier Science Publishers B. V., North-Holland, pp.75-86, 1988.

[2] S. Baase, Computer Algorithm, Addison-Wesley Publishing Company, 1988.

[3] J. A. Bondy and U. S. R. Murty, Graph Theory With Application, New York: Elsevier North-Holland, Inc., 1976.

[4] S. Boyd and H. Ural, "On The Complexity Of Generating Optimal Test Sequences", IEEE Transactions on Software Engineering, Vol. 17, No. 9, pp. 976-978, 1991.

[5] S. Boyd and H. Ural, " The Synchronization Problem In Protocol Testing And Its Complexity", Infor. Proc. Letters, Vol. 40, No. 3, pp. 131-136, 1991.

[6] W. H. Chen, C. S. Lu, L. Chen, and J. T. Wang, "Synchronizable Protocol Test Sequence Generation Via The Duplex Technique", IEEE INFOCOM'90, Vol. 2, pp. 561-563, 1990.

[7] W. Chen, C. Lu, J. Wang, and R. Lee, "Constrained Chinese Postman Problem With Its Application On Synchronizable Protocol Test Generation", Journal of Infor. Science and Engineering, Vol. 6, pp. 149-157, 1990.

[8] W. Chen, C. Tang, and H.Ural, "Minimum-Cost Synchronizable Test Sequence Generation Via The DuplexU Digraph", INFOCOM'93, pp. 128-135, 1993.

[9] N. Christofides, Graph Theory: An Algorithmic Approach, Academic Press, New York, 1975.

[10] T. Cormen, C. Leiserson, and R. Rivest, Introduction To Algorithms, The MIT Press, McGraw-Hill, New York, 1990.

[11] J. Edmonds and E. L. Johnson, "Matching, Euler Tours And Chinese Postman", Math. Prog., Vol. 5, pp. 88-124, 1973.

[12] A. Gill, Introduction To The Theory Of Finite-State Machines, McGraw-Hill, New York, 1962.

[13] ISO TC97/SC21, OSI Conformance Testing Methodology And Framework - Part 1-5, ISO 2nd DP9646-1 revised text, D. Rayner (Editor), Vancouver, December 1987.

[14] R. J. Linn, "Conformance Testing For OSI Protocols", Computer Networks and ISDN System, Vol. 18, pp. 203-219, 1989/90.

[15] C. Papadimitriou, and K. Steiglitz, Combinatorial Optimization, Prentice-Hall, New Jersey, 1982.

[16] D. Rayner, "OSI Conformance Testing", Computer Networks and ISDN Systems, Vol. 14, pp. 79-98, 1987.

[17] B. Sarikaya, and G. V. Bochmann, "Synchronization And Specification Issues In Protocol Testing", IEEE Trans. on Comm., Vol. Comm-32, pp.389-395, 1984.

[18] H. Ural, "Formal Methods For Test Sequence Generation", Computer Communications, Vol. 15, No. 5, pp. 311-325, 1992.

[19] M. U. Uyar and A. T. Dahbura, "Optimal Test Sequence Generation For Protocols: The Chinese Postman Algorithm Applied To Q.931", in Proc. IEEE Global Telecommun. Conf., pp. 68-72, 1986.

[20] Z. Wang, "Synchronizable Test Sequence Generation For Protocol Conformance Testing", Master Thesis, University of Ottawa, Ottawa, Ont., Canada, 1991.

[21] H. X. Zeng, S. T. Chanson and B. R. Smith, "On Ferry Clip Approaches In Protocol Testing", Computer Networks and ISDN Systems, Vol. 17, pp. 77-88, 1989.