Routing Models and Solution Procedures for Regional Less-Than-Truckload Operations

by

Daeki Kim

Bachelor of Architectural Engineering
Korea University (1987)

Master of Urban Planning
University of Kansas (1989)

Submitted to the Department of Civil and Environmental Engineering
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE
in Transportation
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1994

© Massachusetts Institute of Technology 1994
All rights reserved

The author hereby grants to MIT permission to reproduce and to
distribute publicly paper and electronic copies of this thesis
document in whole or in part.

Author ______________________________
Department of Civil and Environmental Engineering
February 1, 1994

Certified by ______________________________
Cynthia Barnhart
Professor of Civil and Environmental Engineering
Thesis Supervisor

Accepted by ______________________________
Joseph M. Sussman
Chairman, Departmental Committee on Graduate Studies

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

MAR 21 1994

LIBRARIES

ARCHIVES

Routing Models and Solution Procedures for Regional Less-Than-Truckload Operations

by
Daeki Kim

Submitted to the Department of Civil and Environmental Engineering on
February 1, 1994,
in partial fulfillment of the requirements
for the degree of Master of Science in Transportation

Abstract

Less-than-Truckload (LTL) carriers are required on a daily basis to solve Intra-Group Line-Haul (IGLH) problems. IGLH problems require the determination of routes to service required pickups and deliveries (i.e., 28-foot trailers) at End-Of-Line (EOL) terminals. The objective is to minimize the total tractor miles traveled, given that tractors are able to simultaneously transport two trailers and that all pickups and deliveries must be accomplished. In this paper, an approximate IGLH solution approach is presented.

Given pickup and delivery requirements together with relevant distance data, a matching network is constructed in which nodes correspond to sets of pickups and deliveries and links to routes. A minimum weight non-bipartite matching algorithm is solved over this network and the result is an IGLH solution. This solution is improved by again applying a minimum weight matching algorithm, this time to a matching network in which nodes correspond to routes and links to improved routes. Finally, the routes are sequenced so as to achieve balance at each EOL terminal (i.e., empty trailers must be delivered or picked up as necessary to ensure that each EOL terminal has the same number of pickups and deliveries) and to minimize the inventory of empty trailers. The new IGLH solution procedure is tested on randomly generated data. Computational tests show that near-optimal solutions are generated rapidly.

Thesis Supervisor : Cynthia Barnhart

Title : Assistant Professor of Civil and Environmental Engineering

Acknowledgments

I would like to express my heartfelt thanks to my advisor, Professor Cynthia Barnhart. Her extensive guidance and constant support contributed in a large measure to the success of this research. From the start till the end of the research, she had friendly corrected me in numerous ways.

I am also grateful to the help and encouragement of Professor Yosef Sheffi. His encouragement and valuable comments had kept me from straying from the right path.

I wish to thank to my fellow students, faculty and staff of the Transportation Systems Division and the Center for Transportation Studies at M.I.T. Among them, Ohkyoung Kwon and Christopher Caplice deserve special thanks for their incessant academic and personal advice.

Thanks also go to my acquaintances at the Korea Transport Institute, where I learned the importance of transportation planning. Knowledge earned from that institute had helped me surviving at M.I.T.

Nonetheless to say, my best and deepest thanks go to my parents - Yunchul Kim and Ilyeol Sung. Without their wonderful financial and spiritual support as well as platonic love, nothing would be achieved. I affectionately dedicate this thesis to my parents.

Finally, I would like to unconditionally thank my wife, Soojeong Lee, for her steadfast love, tender care and considerable patience. I love you.

Table Of Contents

Abstract 2

Acknowledgments 3

Chapter 1 Introduction 7

1.1 The Intra-Group Line-Haul Problem 8

Chapter 2 The Intra-Group Line-Haul Solution Procedure 13

2.1 Step 1: IGLH_MATCH Preprocess 13

2.2 Step 2: IGLH_MATCH Construct Routes 15

2.2.1 Solution Procedure 18

2.3 STEP 3: IGLH_MATCH Sequence Routes 27

Chapter 3 Computational Experiences 35

Chapter 4 Model Flexibility and Adaptability 39

Chapter 5 Conclusions and Future Research 41

Bibliography 42

Appendix 1 IGLH_MATCH.C Code 43

Appendix 2 IGLH_MATCH.C Input and Output Files 80

List of Figures

Figure 1.1	Empty Trailer Balancing	11
Figure 2.1	Examples of Alternative Solutions	17
Figure 2.2	Example of Optimal Solution with Load Splitting	18
Figure 2.3	Example I Match Network and Solution	21
Figure 2.4	Example I Re-Match Network and Solution	25
Figure 2.5	Example I Match and Re-Match Solutions	27
Figure A-1	IGLH_MATCH Input File Format	80
Figure A-2	Preprocess (Step 1 of IGLH_MATCH) Output for Example I	81
Figure A-3	Intermediate Input File for Match (NETFILE1.TMP)	82
Figure A-4	Original Match Network and Solution of Example I	83
Figure A-5	IGLH_MATCH Step 2 (Match) Output	84
Figure A-6	Re-Match Network Information	85
Figure A-7	Intermediate Input File for Re-Match (NETFILE2.TMP)	86
Figure A-8	Re-Match Output	86
Figure A-9	IGLH_MATCH Step 3 (Sequence Routes) Output of Example I	87
Figure A-10	Another Example of IGLH_MATCH Step 3 Output	89

List of Tables

Table 2.1	Preprocessing of Example I	15
Table 2.2	Maximal Sets of Example I	19
Table 2.3	Cost Matrix of Example I	20
Table 2.4	Weights Assigned in the Match Procedure	20
Table 2.5	All Possible Legal Route Combinations	24
Table 2.6	Sequence Routes Procedure	34
Table 3.1	Probability Mass Function for Pickups and Deliveries	35
Table 3.2	IGLH_MATCH Computational Results for Randomly Generated Problems	38

Chapter 1 Introduction

In typical Less-than-truckload (LTL) operations, where motor carriers haul freight from many origins to many destinations, service is provided using multiple transportation moves or legs. The first and last legs involve transporting shipments a short distance between their origins/destinations and *end-of-line* (EOL) terminals. Each origin-destination pair of EOL terminals, (called a market), has an associated demand specifying the shipments that the carrier must haul from the origin terminal to the destination terminal. Because the shipment demand for an individual market is typically much smaller than the capacity of a trailer, it is economical to combine the demand for several markets at an origin terminal. The consolidated demand is then transported (i.e., the second transportation leg) to an intermediate terminal, referred to as a consolidation center (*CC*). At the *CC*, consolidated shipments on incoming trucks are unloaded, sorted, and each shipment is reloaded onto an outgoing truck bound for the *CC* nearest its destination. (This third transportation leg is referred to as *main-line* transportation.) At the destination *CC*, each shipment is unloaded, sorted, repacked, and transported to an EOL terminal near its destination before the final transportation leg to its destination.

Regulatory changes in recent years, such as relaxed restrictions on the use of twin-trailer trucks (i.e., one tractor pulling two 28-foot trailers, called doubles) or triples (i.e., one tractor pulling three 28-foot trailers), have provided opportunities for LTL carriers to become more efficient. In main-line operations, for example, it is documented in Sheffi and Powell (1985) that the relaxed regulations on 28-foot trailers (or pups) allow desired service levels to be achieved with reduced cost. Pups provide extra carrying capacity per driver and allow flexibility since a tractor may travel alone (bobtail) or may pull one, two, or in some cases, even three pups.

Although the use of pups in local city pick-up and delivery operations is

impractical, pups have the potential for reducing *intra-group line-haul* costs, that is costs associated with transportation between EOL terminals and regional *CC*'s. (The set of EOL terminals served by one *CC* constitutes a *group*.) In contrast to main-line operations, however, the magnitude of these savings is unknown. It is our objective in this thesis, therefore, to describe models and solution techniques for the intra-group line-haul problem in order to estimate savings that can be achieved by using pups. In Chapter 1, the intra-group line-haul problem is defined, a problem formulation is presented and relevant literature is reviewed. An approximate model and solution procedure is presented in Chapter 2. Computational results achieved using randomly generated data are presented in Chapter 3. Issues concerning model flexibility and adaptability are discussed in Chapter 4, and finally, conclusions and future research are presented in Chapter 5.

1.1 The Intra-Group Line-Haul Problem

The intra-group line-haul problem is characterized by a single *CC* and the EOL terminals it serves. Associated with each EOL terminal, is the number of pups to be picked-up and transported to the *CC* (called pickups) and the number of pups to be delivered from the *CC* (called deliveries). If, for some EOL terminal i, the number of pickups exceeds the number of deliveries, say by e pups, then to achieve *balance*, e empty pups must be delivered from the *CC* or from another terminal to i. Similarly, if the number of deliveries at i exceeds the number of pickups by e, e empty pups must be picked up at i and delivered to the *CC* or to another terminal. The objective is to determine tractor routes and pup assignments to the tractor routes such that: 1) total tractor miles traveled are minimized; 2) each route begins and ends at the same location; 3) each pickup and delivery is accomplished; 4) tractor capacity (i.e., the number of pups that can be pulled) is

never exceeded; 5) trailers are balanced; 6) pickup and delivery time windows are satisfied; 7) level of service requirements are achieved; and 8) driver work restrictions are not violated. We begin by considering a *simplified* version of the intra-group line-haul problem, referred to as the *core problem*, that satisfies only requirements 1, 2, 3, 4 and 5. The enhanced problem including the omitted requirements is discussed later.

Before presenting the core intra-group line-haul problem formulation, we introduce the following:

Notations :

G: = (N,L): complete directed network G with node set N and arc set L, where each node corresponds to an EOL terminal or a *CC*;

A: node-arc incidence matrix for G;

p : vector of number of trailers to be picked up from each EOL terminal;

d : vector of number of trailers to be delivered to each EOL terminal;

c : vector of distances (or costs) incurred by driving a tractor between two nodes of G; and

k : number of trailers that can be pulled simultaneously by a tractor.

Decision Variables :

t : vector of the number of tractors traveling on each arc in G;

x : vector of the number of *outbound* trailers, i.e., trailers loaded with freight to be delivered to an EOL terminal, traveling on each arc in G;

y : vector of the number of *inbound* trailers, i.e., trailers loaded with freight to be delivered to the *CC*, traveling on each arc in G; and

e : vector of the number of *empty* trailers (needed for balance) traveling on each arc in G.

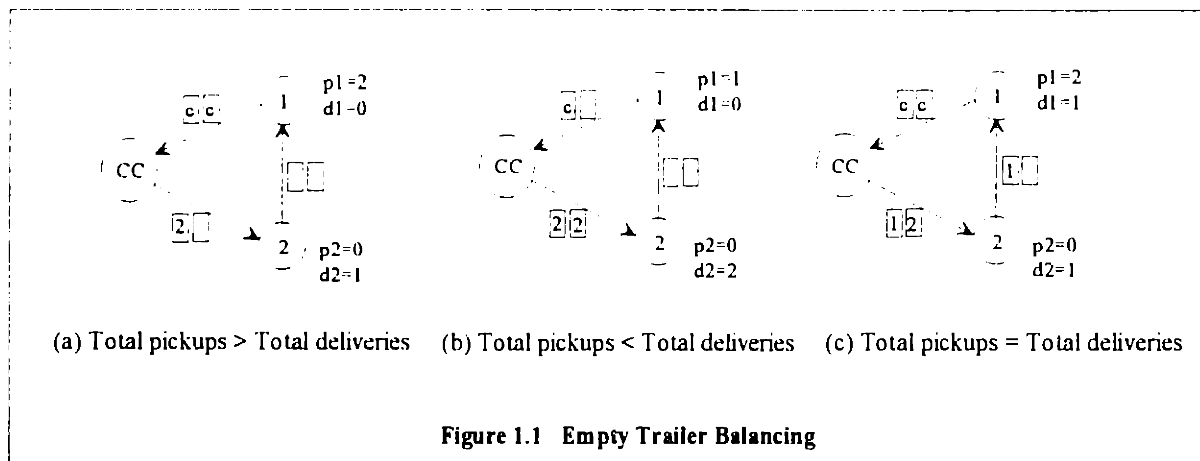
The core intra-group line-haul problem (IGLH) is then formulated as an integer program as follows:

$$
\begin{aligned}
&Min \quad c^T t \\
&\text{subject to} \\
&\quad At = 0 && (1) \\
&\quad Ax = -d && (2) \\
&\quad Ay = p && (3) \\
&\quad Ae = d - p && (4) \\
&\quad kt - x - y - e \geq 0 && (5) \\
&\quad t, x, y, e \geq 0 && (6) \\
&\quad t, x, y, e \ \text{integer} && (7)
\end{aligned}
$$

Constraints (1), (2) and (3) ensure that tractors both enter and leave each location they visit and that all trailer deliveries and pickups are accomplished, respectively. Constraints (4) require that empty trailers are delivered to or picked up from locations in order to achieve balance. Constraints (5) enforce tractor capacity limitations. Except where otherwise noted, we assume that $k = 2$, i.e., that two is the maximum number of trailers that a tractor can carry at any one time. Constraints (6) and (7) ensure that the decision variables are nonnegative and integer, respectively. The objective is to minimize the total number of tractor miles. It is assumed that all distances c: 1) are nonnegative; 2) are symmetric, that is, the distance from node i to node j is the same as that from j to i; and 3) satisfy the triangle inequality, that is, the distance from i to k does not exceed the sum of the distances from i to j and from j to k.

To achieve balance in the group, the total number of pickups must equal the total number of deliveries. Thus, it is assumed for the group that if total pickups exceed total

deliveries by e, e empty trailers are available at the CC for delivery to EOL terminals and similarly, if total deliveries exceed total pickups by e, it is assumed that e empty trailers are required to be delivered to the depot. To illustrate the notion of empty trailer balancing, consider the example in Figure 1.1. Nodes CC, 1, and 2 represent the consolidation center and EOL terminals 1 and 2, respectively. Network arcs correspond to tractor movements and the boxes on each arc represent trailers assigned to that tractor movement. A labeled box is a loaded trailer destined for the terminal indicated by the label, while an unlabeled box is an empty trailer.



(a) Total pickups > Total deliveries (b) Total pickups < Total deliveries (c) Total pickups = Total deliveries

Figure 1.1 Empty Trailer Balancing

In Case (a) of Figure 1.1 where total pickups (i.e., 2) exceed total deliveries (i.e., 1), empty trailer balancing is achieved by transporting one empty trailer from terminal 2 to terminal 1 and another from the CC to terminal 1. In Case (b) where total deliveries exceed total pickups, one of the two empties to be picked up at terminal 2 is delivered to terminal 1 and the other is delivered to the CC. Finally, in Case (c) where the total number of pickups and deliveries is equal, empty trailers are neither picked up from nor delivered to the CC. Empty trailer balancing is achieved instead by transporting one empty trailer from terminal 2 to terminal 1.

Without the tractor capacity constraints (5), IGLH decomposes into four, easy network flow problems - one each for the tractors, the pickups, the deliveries, and the empties. With these capacity constraints, however, the pure network structure of IGLH is destroyed and integer programming techniques must be employed to guarantee that an optimal solution is determined. Substantial literature exists describing solution techniques for integer programs, and for vehicle routing and scheduling problems in particular (see for example the comprehensive surveys by Golden and Assad, 1988 and Magnanti, 1981.) For the intra-group line-haul problem in particular, Eckstein and Sheffi(1987) used optimization techniques, specifically Lagrangian relaxation and branch-and-bound. Their procedure worked well for small problems but was impractical (due to extensive memory requirements and running time) for problems of the size encountered by most LTL carriers. This is because the IGLH problem is NP-hard, especially when k in constraint (5) is considered to be part of the input and k becomes large (Eckstein, 1986). Other than that of Eckstein and Sheffi, we are not aware of any other published accounts of work on the intra-group line-haul problem.

Based on a combination of factors, namely the disappointing results obtained by Eckstein and Sheffi in finding optimal IGLH solutions and the potential need to determine IGLH solutions in real-time, we focused on the development of an approximate IGLH solution technique, described in the next chapter.

Chapter 2 The Intra-Group Line-Haul Solution Procedure

Our approximate solution procedure for the intra-group line-haul problem, denoted IGLH_MATCH, consists of the following three major steps:

STEP 1: IGLH_MATCH Preprocess;

STEP 2: IGLH_MATCH Construct Routes; and

STEP 3: IGLH_MATCH Sequence Routes.

We define a *route* as the sequence of stops visited by one tractor, beginning and ending at a *CC* with one or more intermediate stops at EOL terminals. At each stop, a tractor picks up and/or delivers one or more trailers. A route containing only one EOL terminal is called a *direct*, while one containing more that one EOL terminal is called a *via*. A *loaded direct* is a direct where the tractor carries two deliveries on the route's outbound leg and two pickups on its inbound leg. (The outbound leg is from a *CC* to an EOL terminal, while an inbound leg is the reverse.) A *k-via* is a via visiting *k* intermediate EOL terminals on a route. In the first step of IGLH_MATCH, loaded directs are constructed and the pickups and deliveries assigned to these tractor routes are effectively removed from further consideration. Next, directs and vias servicing every remaining pickup and delivery are constructed. The directs and vias constructed in steps 1 and 2 service each pickup and delivery exactly once. In the final step of IGLH_MATCH, directs and vias are merged together in order to satisfy trailer balancing requirements. We show that balancing can be accomplished without increasing the number of tractor miles traveled. In the following sections, each step of the IGLH_MATCH procedure is detailed.

2.1 Step 1: IGLH_MATCH Preprocess

For each EOL terminal $i = 1,2,\ldots,T$, let p_i and d_i represent the number of pickups and deliveries respectively, and arbitrarily designate the trailers to be picked up (delivered) as trailer numbers $P_1, P_2, \ldots, P_{p_i}$ $(D_1, D_2, \ldots, D_{d_i})$. Then, the steps of Preprocess are as follows:

Preprocess Step 1: For EOL terminal $i = 1,2,\ldots,T$, let m_i represent the maximum number of loaded directs between the *CC* and terminal i, that is: $m_i = \text{MIN}(\lfloor p_i/2 \rfloor, \lfloor d_i/2 \rfloor)$;

Preprocess Step 2: For tractor $a = 1,2,\ldots, m_i$, assign tractor a to perform a loaded direct transporting trailers D_{2a-1} and D_{2a} from the *CC* to terminal i and transporting trailers P_{2a-1} and P_{2a} from terminal i to the *CC*; and

Preprocess Step 3: Eliminate from the problem data all pickups and deliveries assigned to the loaded directs constructed in Preprocess Step 2. That is, for EOL terminal $i = 1,2,\ldots,T$, let $p_i = p_i - 2m_i$ and $d_i = d_i - 2m_i$.

Thus, Preprocess builds for each EOL terminal, the maximum possible number of loaded directs. The motivation for doing this is derived from the following observation:

Lemma 1:

If $p_i = d_i = 2m_i$ for EOL terminal $i = 1,2,\ldots,T$, then an optimal IGLH solution is to perform m_i loaded directs for each EOL terminal i.

Proof:

Since the capacity of each tractor is assumed to be two, the minimum number of

tractor trips from the CC to EOL terminal i is m_i and similarly, the minimum number of tractor trips from terminal i to the CC is m_i. Furthermore, since distances are assumed to satisfy the triangle inequality, the minimum distance trip between the CC and terminal i is $c_{CC,i}$, i.e., the direct route between CC and EOL terminal i. Hence, the optimal IGLH objective function value is at least $\sum_{i=1}^{T} c_{cc,i} * 2m_i$. ■

To illustrate the preprocessing step, consider Example I in Table 2.1. EOL terminal 1 requires two pickups and three deliveries. Preprocessing creates one loaded direct to terminal 1, thereby satisfying required pickups and reducing to one the number of remaining deliveries to that location. Preprocessing similarly alters the pickups and deliveries at EOL terminals 2, 3 and 4, as shown in Table 2.1.

Table 2.1 Preprocessing of Example I

EOL Number	Number of Pickups Before Preprocess	Number of Deliveries Before Preprocess	Number of Pickups After Preprocess	Number of Deliveries After Preprocess
1	2	3	0	1
2	3	3	1	1
3	3	1	3	1
4	2	2	0	0

Note that after preprocessing, either the number of pickups or the number of deliveries is reduced to 0 or 1.

2.2 Step 2: IGLH_MATCH Construct Routes

After preprocessing, it is necessary in the route construction phase (Step 2) to build

additional tractor routes and assign the *remaining* pickups and deliveries to these routes. Like preprocessing, route construction is motivated by the observation that it is usually optimal to serve the pickups and deliveries at an EOL terminal with the minimum number of tractors. Consider for example, the problem depicted in Figure 2.1a. Since the total number of pickups equals four and the total number of deliveries is three, at least two tractors are required. The minimum cost solution, depicted in Figure 2.1a, uses exactly two tractors: one performing the 2-via visiting EOL terminals 1 and 3, and the other performing the direct to EOL terminal 2. Observe that the minimum number of tractors required to service each EOL terminal is one and that exactly this minimum number of tractors visits each EOL terminal. Consider all alternate solutions (presented in Figures 2.1b - 2.1e). In Figure 2.1b, two tractors visit EOL terminal 2, effectively replacing the optimal tractor movement from terminal 1 to terminal 3 in Figure 2.1a with the more costly (due to the triangle inequality) two tractor movements, one from terminal 1 to terminal 2 and the other from terminal 2 to terminal 3. Similarly, in Figure 2.1c, both terminals 1 and 3 are visited by two tractors, thereby altering the optimal solution by replacing the direct to terminal 2 with the more costly (again due to the triangle inequality) 3-via visiting terminals 1, 2 and 3. The solution in Figure 2.1d is a further degradation of the already nonoptimal solution of Figure 2.1b since it inserts another visit to terminal 1 along the movement from the *CC* to terminal 2. Finally, Figure 2.1e represents a solution using 3 tractors, each performing a direct to one of the three terminals. This solution effectively replaces the optimal tractor movement from terminal 1 to terminal 3 with two more costly (due to the triangle inequality) tractor movements, one from terminal 1 to the *CC* and the other from terminal 3 to the *CC*.

Many such examples can be constructed demonstrating that due to the triangle inequality, it is optimal for a tractor visiting an EOL terminal to pickup and deliver a maximal number of trailers. That is, the number of trailers picked up at (delivered to) a terminal by one tractor should equal either the number of pickups (deliveries) required at

that terminal or the capacity of the tractor. Although numerous examples can be constructed validating this observation, it is worth noting that there are instances (such as the one depicted in Figure 2.2) where it is not optimal for each tractor to pickup and deliver a maximal number of trailers at every EOL terminal. Specifically, in order to utilize the minimum number of tractors required and to achieve an optimal solution, it is necessary to serve EOL terminal 1 with two tractors each delivering one trailer, rather than one tractor delivering two trailers.

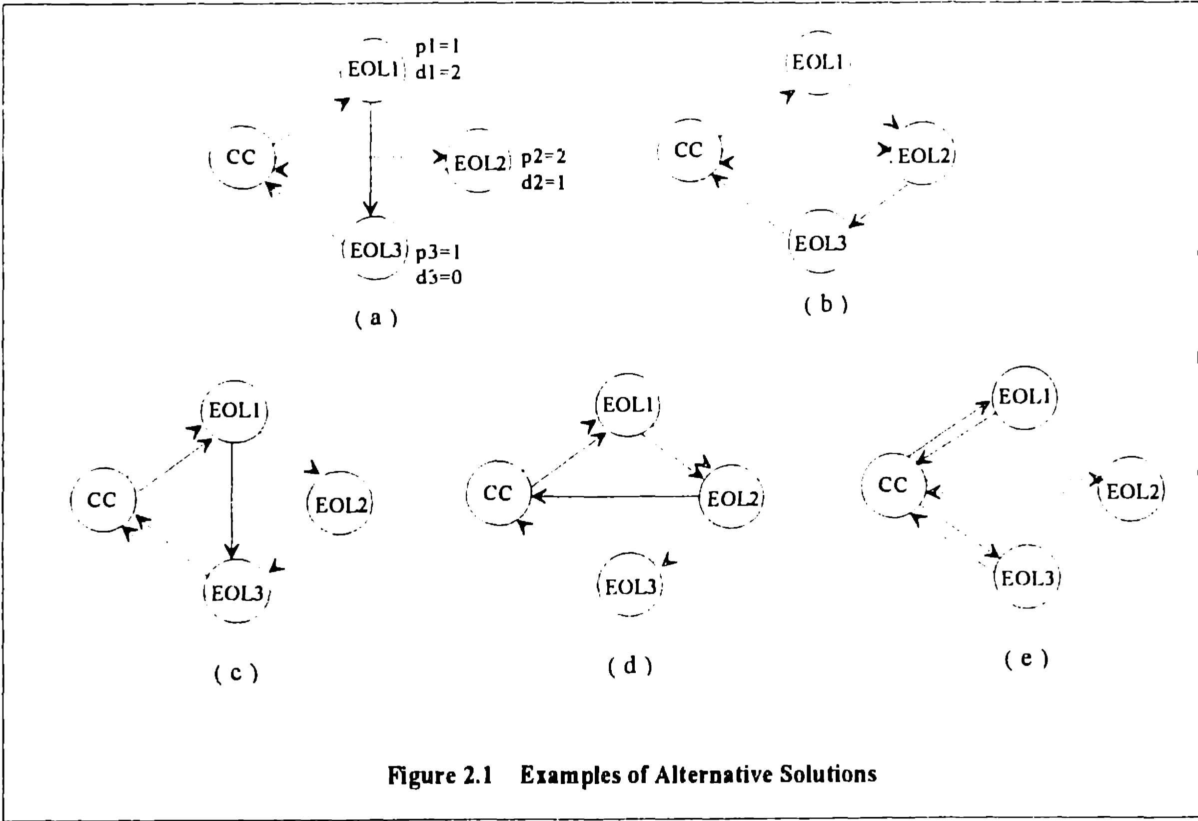


Figure 2.1 Examples of Alternative Solutions

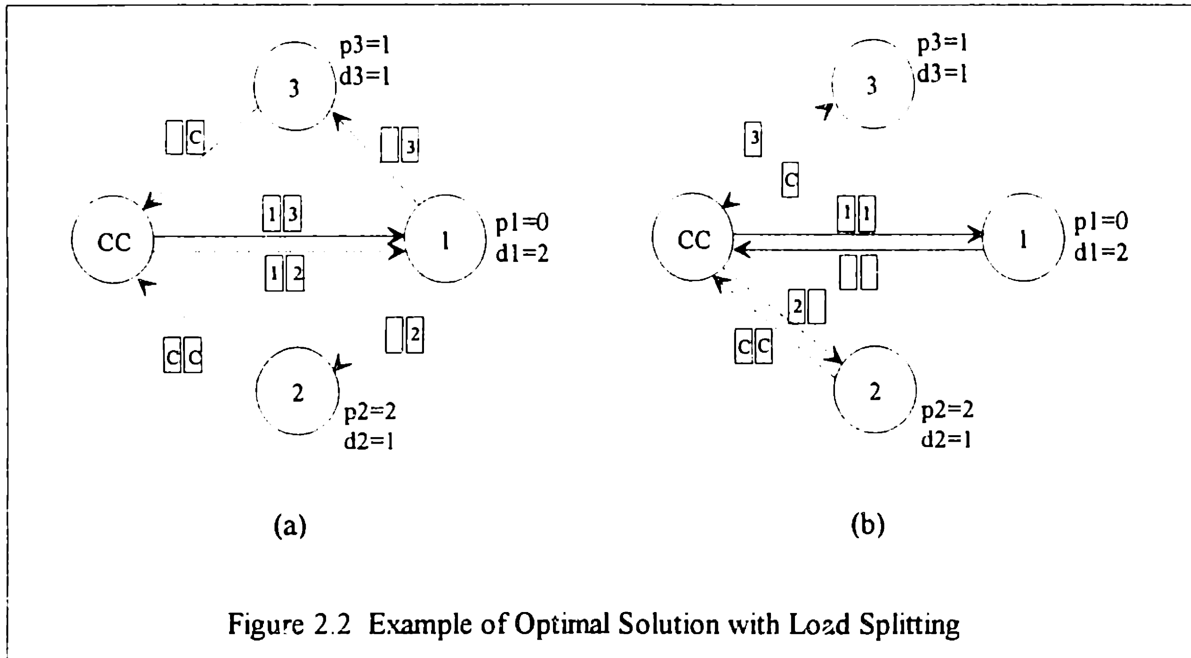


Figure 2.2 Example of Optimal Solution with Load Splitting

The observation that it is *usually* optimal for a tractor visiting an EOL terminal to pickup and deliver a maximal number of trailers motivated the use of a matching-based procedure in the route construction step of the IGLH_MATCH procedure. The matching-based procedure is described in the following sections.

2.2.1 Solution Procedure

In Construct Routes (Step 2 of IGLH_MATCH), additional tractor routes are built and the *remaining* pickups and deliveries are assigned to these routes. This is accomplished with two procedures: *Match* and *Re-Match*. An initial IGLH solution is determined in *Match*, while *Re-Match* is used to generate an improved solution. We illustrate the *Match* and *Re-Match* procedures using Example I and follow-up with detailed descriptions.

The *Match* Procedure

In the first step of *Match*, the remaining pickups and deliveries (p_i, d_i) at EOL

terminal i = 1,2,...,T, are divided into *maximal sets*, that is, into sets where the number of trailers picked up at (delivered to) a terminal by one tractor equals the minimum of: 1) the number of pickups (deliveries) required at that terminal; and 2) tractor capacity. For example, if the remaining number of pickups at terminal 3 is three and the remaining number of deliveries is one, i.e., $(p_3, d_3) = (3, 1)$, then the resulting maximal sets are $(p_3^1, d_3^1) = (2, 1)$ and $(p_3^2, d_3^2) = (1, 0)$. Table 2.2 shows the maximal sets for Example I (introduced in Table 2.1).

Table 2.2 Maximal Sets of Example I

EOL Number	Number of Pickups After Preprocess	Number of Deliveries After Preprocess	Number of Pickups in Maximal Set	Number of Deliveries in Maximal Set
1	0	1	0	1
2	1	1	1	1
3^1	3	1	2	1
3^2			1	0

In the next step, pairs of maximal sets are evaluated to determine if there exists a tractor route that can *legally* service both sets in the pair. To be legal, the route must satisfy tractor capacity limitations. (Additional constraints may also define legality, e.g. driver work and rest rules mandated by government and contractual agreements, time windows restricting the scheduling of service and enforcing minimum service level requirements, etc.) Consider, for example, the pair of maximal sets (2, 1) and (1, 1). This pair is *illegal* since tractor capacity is violated by the required pickup of 3 trailers on the single route.

Every pair is assigned a *weight* equal to the minimum distance legal route associated with that pair. (If the pair is illegal, the assigned weight is set to infinity.) Table 2.3 contains the distances between EOL terminals and the *CC* in Example I. Assuming for now that legality is defined solely by tractor capacity restrictions, Table 2.4

reports the weights assigned to each pair for Example I.

Table 2.3 Cost Matrix of Example I

from \ to	*CC*	EOL 1	EOL 2	EOL 3	EOL 4
CC	0	37	50	8	26
EOL 1	37	0	15	30	22
EOL 2	50	15	0	44	31
EOL 3	8	30	44	0	20
EOL 4	26	22	31	20	0

Table 2.4 Weights Assigned in the Match Procedure

from \ to	*CC*	EOL 1	EOL 2	EOL 3^1	EOL 3^2
CC	∞	74	100	16	16
EOL 1	74	∞	102	75	75
EOL 2	100	102	∞	∞	102
EOL 3^1	16	75	∞	∞	∞
EOL 3^2	16	75	102	∞	∞

Given the maximal sets and assigned weight for each pair, an *undirected matching* network is constructed: each node corresponds to a maximal set, links are pairs, and each link cost equals the weight of the pair represented by the link. This *base* network is augmented by adding for each base node n, one *copy* node n' and one link from n' to node n, with cost equal to twice the distance between the *CC* and the EOL terminal represented by node n. Like the other network links, these additional links correspond to tractor routes. In particular, the link from n' to n represents a direct serving only the trailers at n. Finally, the network is made complete by adding zero-cost links between every pair of copy nodes and arcs with infinite cost between each pair of nodes with no adjoining link. The Example I matching network is presented in Figure 2.3 (with some infinite cost links omitted for clarity).

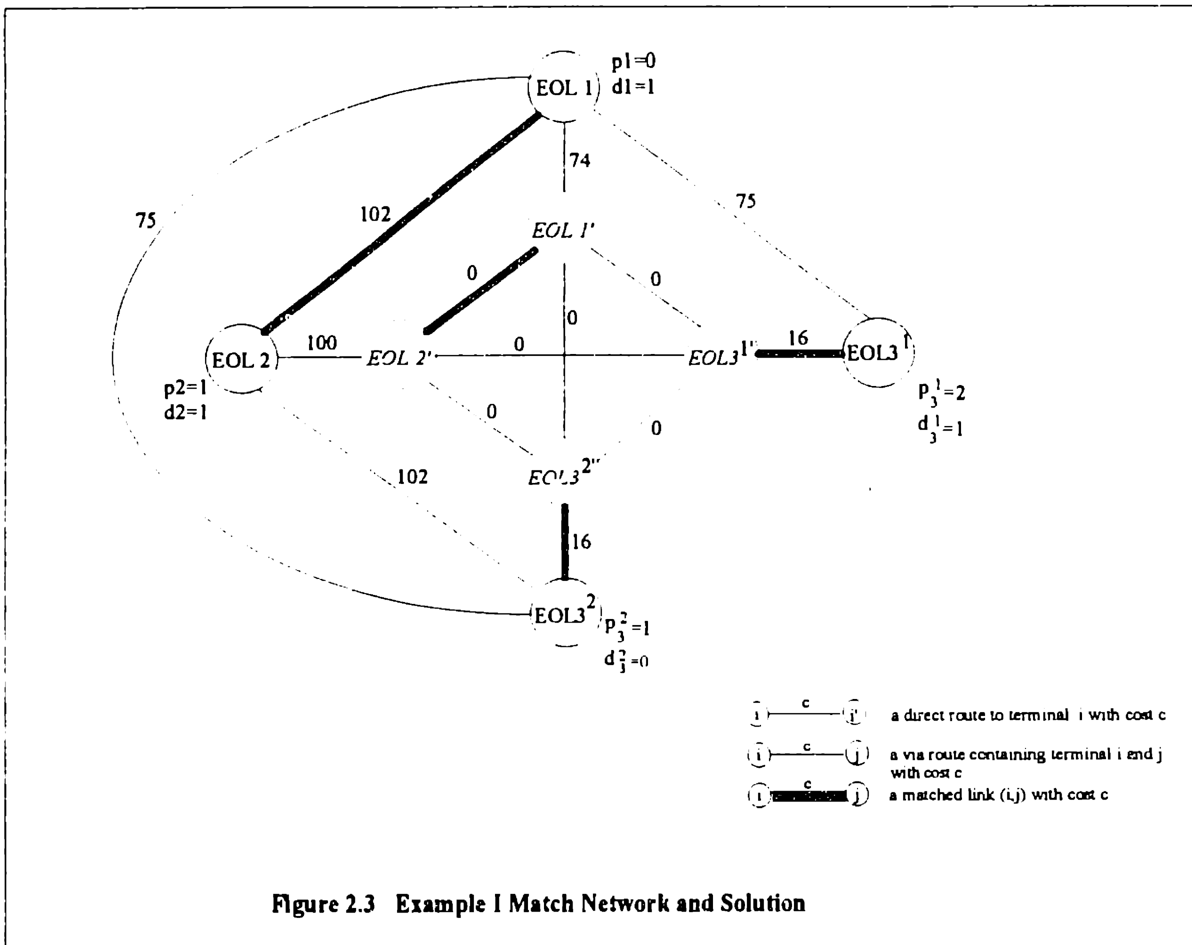


Figure 2.3 Example I Match Network and Solution

Next, a non-bipartite minimum weight matching algorithm (Gabow, 1973) is used to determine an IGLH solution. By definition, a *matching* in a graph is a set of edges, where no two edges in the matching share the same node. Given weights for the edges, the ***minimum weight matching problem*** is to find the matching with the smallest sum of weights.

Rather than directly solving minimum weight matching problem, we use maximum weighted matching algorithm by assigning $w_{ij} = W - c_{ij} > 0 \ \forall \ (i,j)$ where W is larger than the biggest c_{ij}. Since the matching network is complete and it contains an even number of nodes (i.e., a copy node is added for each base network node), the matching solution is *perfect*, that is, every node is incident to exactly one arc in the matching. This implies that each pickup and delivery is included in exactly one route. Furthermore, if the weight on each matched link is less than infinity, each pickup and delivery is included in exactly one

legal route. Since weights on the matching network links equal the distances of the corresponding tractor routes, the optimal matching solution minimizes the number of tractor miles needed to service all pickups and deliveries when it is required that: 1) each tractor route contain two or fewer EOL terminals; and 2) each terminal be served by its minimum required number of tractors, or said another way, each tractor serves the maximum possible number of trailers at every terminal.

To illustrate, consider the minimum weight solution in Figure 2.3 for the Example I matching network. The matched links represent tractor routes that are to be performed. Hence, two tractors perform directs to EOL terminal 3, with one tractor picking up two trailers and delivering one, and the other tractor picking up one trailer. The only other tractor performs a 2-via, first delivering and picking up one trailer each at terminal 2, and then delivering one trailer to terminal 1. The zero-weight matched link connecting copy nodes *EOL 1'* and *EOL 2'* does not correspond to a tractor route, but rather its purpose is to achieve a perfect matching solution. The total number of miles traveled by the three tractors is 134, the value of the minimum weight matching.

The *Match* procedure is summarized as follows:

***Match* Step 1:** For EOL terminal $i = 1,2,\ldots,T$, let p_i (d_i) represent the number of pickups (deliveries) remaining at terminal i after preprocessing and let m_i represent the minimum number of tractors needed to service these required pickups and deliveries, that is:

$$m_i = MAX(\lceil p_i/2 \rceil, \lceil d_i/2 \rceil);$$

***Match* Step 2:** For EOL terminal $i= 1,2,\ldots,T$, partition the total number of pickups and deliveries (p_i, d_i) into m_i maximal trailer sets, denoted $\{(p_i^1, d_i^1), (p_i^2, d_i^2), \ldots, (p_i^{m_i}, d_i^{m_i})\}$, as follows:

$$(p_i^k, d_i^k) = \{MAX[0, MIN(2, p_i^0-2(k-1))], MAX[0, MIN(2, d_i^0-2(k-1))]\},$$

for k=1, 2,...,m_i , where $p_i^0 = p_i$ and $d_i^0 = d_i$.

Match **Step 3:** Construct the undirected *matching* network as follows: Add one *base* node *n* and one *copy* node *n'* for each maximal set (p_i^k, d_i^k), for k=1, 2, ..., m_i and i= 1,2,...,T; for each base node *n*, add a link between *n* and *n'* with weight equal to twice the distance between the *CC* and the associated EOL terminal; add a link between each pair of base nodes with weight equal to the minimum distance legal route associated with that pair; add zero-weight links between every pair of copy nodes; and add infinite-weight links between every pair of nodes without an adjoining link.

Match **Step 4:** Find the minimum weight matching on the matching network.

The *Re-Match* Procedure

The *Match* procedure limits the number of EOL terminals visited in any route to at most two. Although non-optimal solutions may result, if tractor capacity is the only constraint defining route legality, this limit is not particularly restrictive since 76% of all legal IGLH routes contain two or fewer stops (Table 2.5). Even so, to reduce the occurrence of non-optimal solutions, a second matching-based procedure, called *Re-Match*, identifies routes that contain up to four EOL terminals and improve the IGLH solution. This improvement is achieved by merging any two routes generated in *Match* into one route *if* the merger is legal and the merged route results in fewer total miles. (It is not necessary to consider routes visiting more than four EOL terminals because there is at least one pickup or delivery at each stop and a tractor can haul at most two pickups and two deliveries in a single route.)

Table 2.5 All Possible Legal Route Combinations

Number of Stops	Possible Pickups / Deliveries at				No. of Possible Routes
	EOL Terminal 1	EOL Terminal 2	EOL Terminal 3	EOL Terminal 4	
k = 1	(0,1) (0,2) (1,0) (1,1) (1,2) (2,0) (2,1) (2,2)				8
k = 2	(0,1)	(0,1) (1,0) (1,1) (2,0) (2,1)			
	(0,2)	(1,0) (2,0)			
	(1,0)	(1,0) (1,1) (1,2)			
	(1,1)	(1,1)			11
k = 3	(0,1)	(0,1)	(1,0) (2,0)		
		(1,0)	(1,0) (1,1)		
	(0,2)	(1,0)	(1,0)		5
k = 4	(0,1)	(0,1)	(1,0)	(1,0)	1

To illustrate the idea of merging routes, consider the Example I solution generated by *Match*. The 2-via containing EOL terminals 1 and 2, denoted 2V1, and the direct to EOL terminal 3, denoted D1, have a total weight of 118. The pickups and deliveries serviced by these two routes, however, can be serviced with a single *legal* 3-via (denoted 3V1) visiting EOL terminals 2, 1 and 3 (in that order) with total weight of 103, a reduction of about 13%.

The *Re-Match* solution procedure identifies such improvements to the *Match*

solution. It does this by first constructing a *Re-Match* network and then solving a minimum weight non-bipartite matching algorithm on this network. Given the *Match* solution consisting of tractor routes and associated pickups and deliveries, an *undirected re-matching* network is constructed: each node corresponds to a tractor route and trailer assignment in the *Match* solution, links are (merged) tractor routes and corresponding trailer assignments, and each link cost equals the minimum distance legal route associated with the (merged) tractor route. For example, routes 2V1 and D1 in the Example I *Match* solution are each represented in the re-match network by a node, and route 3V1 is represented by a link between nodes 2V1 and D1 with weight equal to 103.

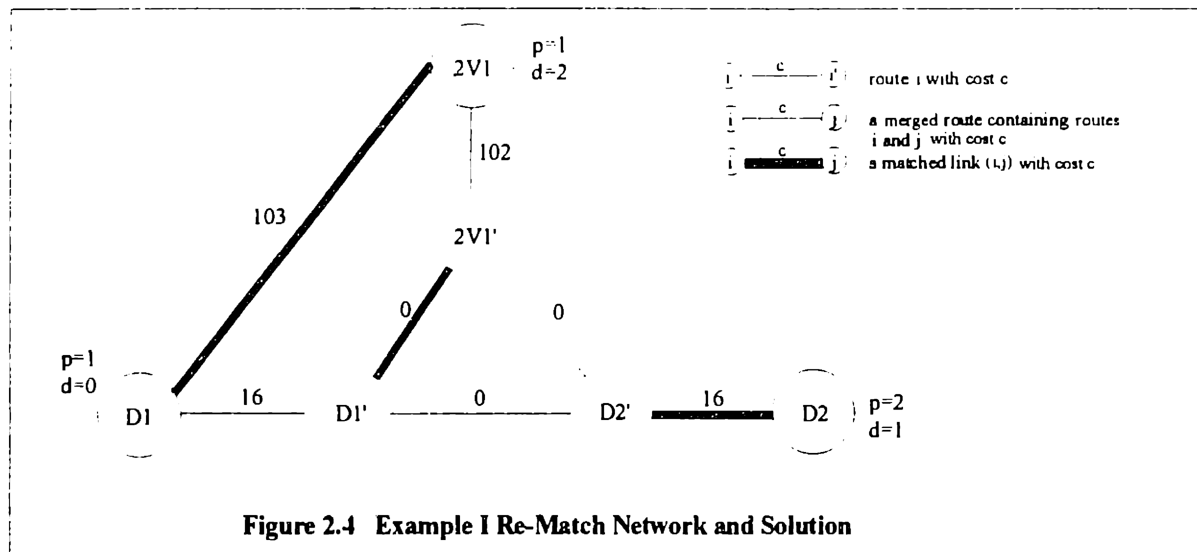


Figure 2.4 Example I Re-Match Network and Solution

This *base* re-matching network is augmented by adding for each *base* node *r*, one *copy* node *r'* and one link from *r'* to *r*, with cost equal to the weight of the route associated with node *r*. A link from node *r'* to *r* represents exactly the route associated with *r*. Finally, the network is completed by adding zero-cost links between every pair of copy nodes and infinite-cost links between every pair of nodes without an adjoining link. The Example I re-matching network is presented in Figure 2.4 (with some infinite cost links omitted for clarity).

Again, since the re-matching network contains an even number of nodes, the *Re-Match* solution is perfect. As in the *Match* solution, this implies that each pickup and delivery is included in exactly one route and, if the weight on each matched link is less than infinity, each pickup and delivery is included in exactly one *legal* route. Since each route *r* in the *Match* solution is represented by a link and two end nodes, and since the weight of that link is equal to cost of route *r*, the optimal *Re-Match* solution corresponds to tractor routes and associated pickups and deliveries with total distance not greater than that of the *Match* solution. Hence, the *Re-Match* procedure identifies and creates 3- and 4-vias that lead to improved IGLH solutions.

The optimal minimum weight *Re-Match* solution for Example I is depicted in Figure 2.4. The matched link between nodes *2V1* and *D1*, i.e., the link corresponding to route *3V1*, indicates that the Example I *Match* solution is improved by replacing routes *2V1* and *D1* with route *3V1*. Similarly, the matched link between node *D2* (i.e., the node representing the direct route, denoted *D2*, to EOL terminal 3 with 1 delivery and two pickups) and its copy node *D2'*, indicates that the *Match* solution is unaltered with respect to route *D2*. Finally the link between the two copy nodes *2V1'* and *D1'* is included in the *Re-match* solution only in order to achieve a perfect matching.

The *Re-Match* procedure is as follows:

***Re-Match* Step 1:** Construct the undirected *re-matching* network as follows: Add one *base* node *r* and one *copy* node *r'* for each route generated in the *Match* procedure; for each base node *r*, add a link between *r* and *r'* with weight equal to the distance of route associated with *r*; add a link between each pair of base nodes with weight equal to the minimum distance legal route associated with that pair (let this weight equal infinity if no legal route exists); add zero-weight links between every pair of copy nodes; and add infinite-weight links between every pair of nodes without an adjoining link.

Re-Match Step 2: Find the minimum weight matching on the re-matching network.

Figure 2.5 illustrates the Example I solutions obtained by *Match* and *Re-Match*.

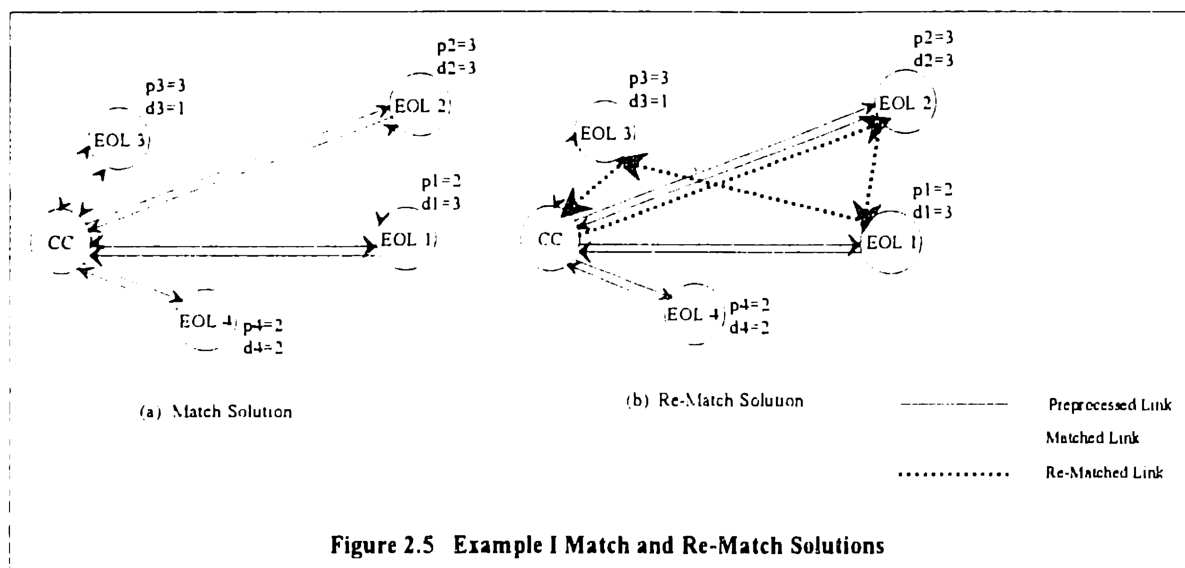


Figure 2.5 Example I Match and Re-Match Solutions

2.3 STEP 3: IGLH_MATCH Sequence Routes

Constraints (4) of the IGLH problem formulation require the movement of empty trailers in order to achieve balance at EOL terminals. In this section, we present a route ordering procedure called *Sequence Routes.* Using the routes generated in Step 2 of IGLH_MATCH, the *Sequence Routes* procedure accomplishes *empty trailer balancing without additional tractor miles*. Before detailing the procedure, we first provide some notations and assumptions that will facilitate the proof of this result.

<u>Notations :</u>

R: the set of routes generated in Steps 1 and 2 of IGLH_MATCH;

v_r: imbalance of route $r \in \boldsymbol{R}$, i.e., $v_r = (\sum_{i=1}^{N^r} p_i^r - \sum_{i=1}^{N^r} d_i^r)$, where i = 1,...,$N^r$ denote the EOL terminals visited in route *r* (assume that the number of each terminal in *r* is its

stop number and 0 and N^{r+1} denote the *CC*), and p^r_i (d^r_i) denote the number of pickups (deliveries) at terminal i accomplished by route $r \in R$;

$\boldsymbol{R}^{\geq}(\boldsymbol{R}^{<})$: the subset of routes contained in $\boldsymbol{R}$ with $v_r \geq (<) 0$;

$X^r_{i,j}$: $= \text{MAX}(\sum_{i=1}^{N^r} p_i^r, \sum_{i=1}^{N^r} d_i^r)$, i.e., the total number of loaded and empty trailers assigned to the location i to location j tractor movement along route $r \in \boldsymbol{R}$;

$x^r_{i,j}$: total number of loaded trailers (assigned in Steps 1 and 2 of IGLH_MATCH) on the location i to location j tractor movement along route $r \in \boldsymbol{R}$; and

$e^r_{i,j}$: total number of empty trailers on the location i to location j tractor movement along route $r \in \boldsymbol{R}$, where $e^r_{i,j} = X^r_{i,j} - x^r_{i,j}$.

Assumptions:

Group Balance Assumption: For each route $r \in \boldsymbol{R}^{\geq}$, assume that there are v_r empty trailers available to route *r* for pickup at the *CC*; otherwise, assume that $-v_r$ empty trailers must be delivered by *r* to the *CC*.

Deliver First Assumption: If the matched link of a *Match* or *Re-Match* solution can represent two routes, each with the same total weight (since distances are assumed to be symmetrical) but with the order of the stops on one route the reverse of the other, the matched link represents the route with the maximum difference between the number of deliveries and pickups at the first stop.

For example, consider constructing a route visiting EOL terminal 1 with zero pickups and one delivery and EOL terminal 2 with one pickup and zero deliveries. It is possible to create two legal 2-vias with the same total weight: the first 2-via, denoted r_1, visits EOL terminal 1 followed by EOL terminal 2, and the second, denoted r_2, reverses this direction and visits terminal 2 before terminal 1.

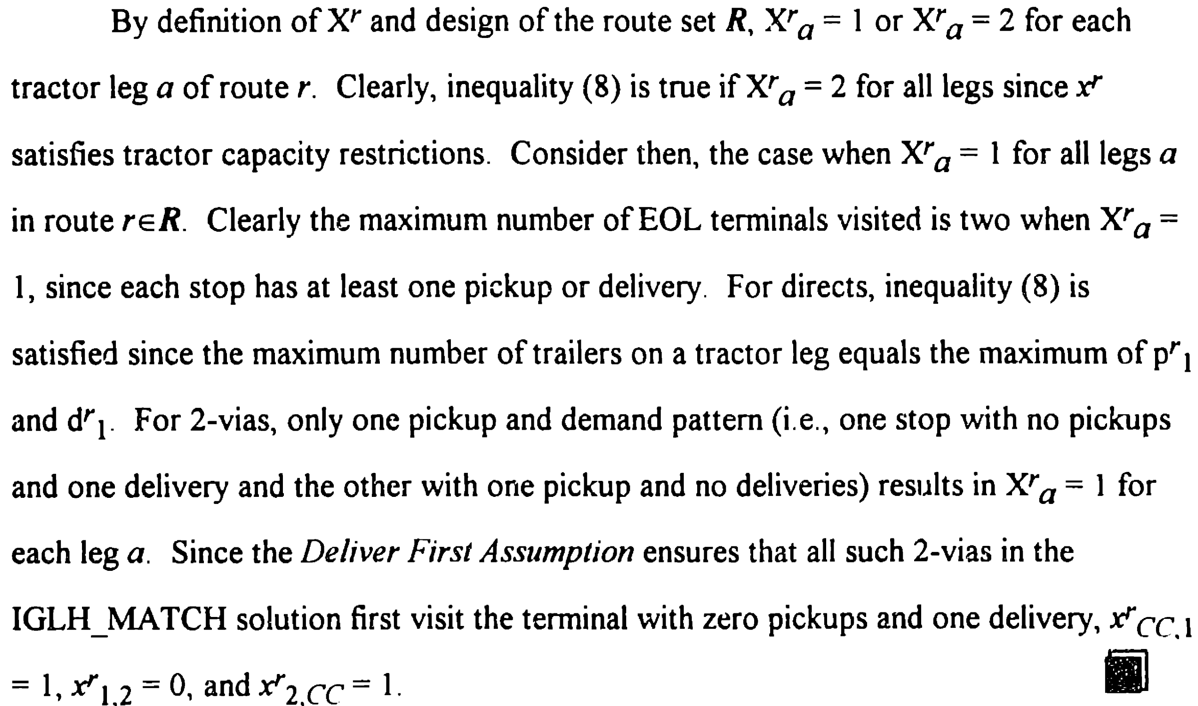
The *Deliver First Assumption* states that route r_1 will be generated by IGLH_MATCH since 1 > -1, i.e., the number of deliveries minus the number of pickups at terminal 1 is greater than that same difference at terminal 2.

The following results can now be stated:

Lemma 2: For each route $r \in \boldsymbol{R}$, the total number of loaded trailers assigned to any tractor leg a is not greater than X^r_a, i.e.,

$$x^r_{k,k+1} = \sum_{i=1}^{k} p_i^r + \sum_{i=k+1}^{N^r} d_i^r \le X^r_{k,k+1}, \quad \forall\, k = 0,1,\ldots,N^r,\ \forall\, r \in R \tag{8}$$

Proof:



By definition of X^r and design of the route set $\boldsymbol{R}$, $X^r_a = 1$ or $X^r_a = 2$ for each tractor leg a of route r. Clearly, inequality (8) is true if $X^r_a = 2$ for all legs since x^r satisfies tractor capacity restrictions. Consider then, the case when $X^r_a = 1$ for all legs a in route $r \in \boldsymbol{R}$. Clearly the maximum number of EOL terminals visited is two when $X^r_a = 1$, since each stop has at least one pickup or delivery. For directs, inequality (8) is satisfied since the maximum number of trailers on a tractor leg equals the maximum of p^r_1 and d^r_1. For 2-vias, only one pickup and demand pattern (i.e., one stop with no pickups and one delivery and the other with one pickup and no deliveries) results in $X^r_a = 1$ for each leg a. Since the *Deliver First Assumption* ensures that all such 2-vias in the IGLH_MATCH solution first visit the terminal with zero pickups and one delivery, $x^r_{CC,1} = 1$, $x^r_{1,2} = 0$, and $x^r_{2,CC} = 1$. ∎

Lemma 3: For each route $r \in \boldsymbol{R}$, let $e^r = X^r - x^r$ where the values of X^r and x^r are as defined above. Given the *Group Balance and Deliver First Assumptions*, the trailer assignments x^r and e^r for each $r \in \boldsymbol{R}$ satisfy the following:

a) loaded trailer pickup and delivery requirements are satisfied;

b) the total number of loaded and empty trailers assigned each tractor leg is less than tractor capacity;

c) empty trailer conservation of flow requirements are satisfied; and

d) the total number of empty trailers assigned each tractor leg is nonnegative.

Hence, empty balancing can be achieved without additional tractor miles.

Proof:

Point a) follows directly from the fact that the loaded trailer assignments for each route $r \in \boldsymbol{R}$ are determined in Steps 1 and 2 of IGLH_MATCH.

By design, each route $r \in \boldsymbol{R}$ is legal and contains no stops with zero pickups and deliveries. Point b) follows from the fact that the total number of trailers a tractor transports between any links of a route r is set to the $\mathrm{MAX}(\sum_{i=1}^{N^r} p_i^r, \sum_{i=1}^{N^r} d_i^r) \leq \mathrm{MAX}(2,2) = 2$.

From point a), it follows that the flow of loaded trailers x^r on route r satisfies conservation of flow requirements, i.e.:

$$x^r_{i-1,i} + p^r_i - d^r_i = x^r_{i,i+1}, \qquad \forall\ i = 1,2,\ldots, N^r,\ \forall\ r \in \boldsymbol{R}.$$

Using the definition $x^r = X^r - e^r$ and substituting, we obtain:

$$X^r_{i-1,i} - e^r_{i-1,i} + p^r_i - d^r_i = X^r_{i,i+1} - e^r_{i,i+1}, \quad \forall\ i = 1,2,\ldots, N^r,\ \forall\ r \in \boldsymbol{R}.$$

Noticing that $X^r_{i-1,i} = X^r_{i,i+1}$ by definition and rewriting:

$$e^r_{i-1,i} - p^r_i + d^r_i = e^r_{i,i+1}, \quad \forall\ i = 1,2,\ldots, N^r,\ \forall\ r \in \boldsymbol{R}.$$

Hence, point c) follows since the total number of empties transported on route r into terminal i (i.e., $e^r_{i-1,i}$) plus the number of empties picked up at (i.e., $- p^r_i + d^r_i \geq 0$) or delivered to (i.e., $- p^r_i + d^r_i < 0$) terminal i equals the number of empties transported out of terminal i (i.e., $e^r_{i,i+1}$).

Finally, point d) follows directly from Lemma 2 and the definition $e^r = X^r - x^r \geq 0$, $\forall\ r \in \boldsymbol{R}$. ■

To summarize, if $\sum_{r \in R^{<}} v_r$ empty trailers are available at the *CC*, then Lemma 3 shows that empty balancing can be achieved with the routes and loaded trailer assignments generated in IGLH_MATCH Steps 1 and 2 without additional tractor miles. Empty trailer balancing can be accomplished, however, with an initial inventory of empty trailers at the *CC* that may be significantly less than $\sum_{r \in R^{<}} v_r$ empty trailers and similarly, with a final inventory that is less than $\sum_{r \in R^{<}} v_r$. Specifically, let the total group imbalance, denoted *IMB*, equal $\sum_{r \in R^{<}} v_r + \sum_{r \in R^{-}} v_r$. Then, through appropriate sequencing of certain routes, empty balancing can be achieved without additional tractor miles with an initial inventory of MAX(*IMB*, 0) empty trailers at the *CC* and a final inventory of MAX(- *IMB*, 0). Before proving this, we detail the Step 3 IGLH_MATCH procedure, called *Sequence Routes*, as follows:

Sequence Routes **Step 1:** Let $e^r = X^r - x^r$ be the number of empty trailers assigned to each leg of every route $r \in \boldsymbol{R}$. Partition the set $\boldsymbol{R}$ into the following subsets: $\boldsymbol{R}_I = \{r \mid \sum_{r=1}^{k} v_r = IMB,\ r = 1,2,\ldots,k \in \boldsymbol{R},\ k \text{ as small as possible}\}$; $\boldsymbol{R}_0 = \{r \mid v_r = 0,\ r \in \boldsymbol{R} \backslash \boldsymbol{R}_I\}$; $\boldsymbol{R}_1^+ = \{r \mid v_r = 1,\ r \in \boldsymbol{R} \backslash \boldsymbol{R}_I\}$; $\boldsymbol{R}_1^- = \{r \mid v_r = -1,\ r \in \boldsymbol{R} \backslash \boldsymbol{R}_I\}$; $\boldsymbol{R}_2^+ = \{r \mid v_r = 2,\ r \in \boldsymbol{R} \backslash \boldsymbol{R}_I\}$; and $\boldsymbol{R}_2^- = \{r \mid v_r = -2,\ r \in \boldsymbol{R} \backslash \boldsymbol{R}_I\}$. Let $|\boldsymbol{R}_1| = \text{MIN}(|\boldsymbol{R}_1^+|, |\boldsymbol{R}_1^-|)$ and $|\boldsymbol{R}_2| = \text{MIN}(|\boldsymbol{R}_2^+|, |\boldsymbol{R}_2^-|)$ where $|\ \boldsymbol{S}\ |$ denotes the number of elements in set $\boldsymbol{S}$. Initialize $\boldsymbol{P} = \{\boldsymbol{R}_0\}$.

Sequence Routes **Step 2:** Repeat the following steps $|\boldsymbol{R}_1|$ times: 1) select any route, denoted r_1^+, from $\boldsymbol{R}_1^+$ and any route, denoted r_1^-, from $\boldsymbol{R}_1^-$; 2) let r^S

denote the *super-route* formed by performing route r_1^- followed by route r_1^+; 3) let $\boldsymbol{P} = \boldsymbol{P} + r^S$; 4) let $\boldsymbol{R}_1^+ = \boldsymbol{R}_1^+ - r_1^+$; and 4) let $\boldsymbol{R}_1^- = \boldsymbol{R}_1^- - r_1^-$.

Sequence Routes **Step 3:** Repeat the following steps $|\boldsymbol{R}_2|$ times: 1) select any route, denoted r_2^+, from $\boldsymbol{R}_2^+$ and any route, denoted r_2^-, from $\boldsymbol{R}_2^-$; 2) let r^S denote the *super-route* formed by performing route r_2^- followed by route r_2^+; 3) let $\boldsymbol{P} = \boldsymbol{P} + r^S$; 4) let $\boldsymbol{R}_2^+ = \boldsymbol{R}_2^+ - r_2^+$; and 4) let $\boldsymbol{R}_2^- = \boldsymbol{R}_2^- - r_2^-$.

Sequence Routes **Step 4:** Repeat the following steps until $\boldsymbol{R}_1^+$ is empty: 1) select any two routes, denoted r_1^+ and q_1^+, from $\boldsymbol{R}_1^+$ and select any route, denoted r_2^-, from $\boldsymbol{R}_2^-$; 2) let r^S denote the *super-route* formed by performing route r_2^- followed by routes r_1^+ and q_1^+; 3) let $\boldsymbol{P} = \boldsymbol{P} + r^S$; 4) let $\boldsymbol{R}_1^+ = \boldsymbol{R}_1^+ - r_1^+ - q_1^+$; and 4) let $\boldsymbol{R}_2^- = \boldsymbol{R}_2^- - r_2^-$.

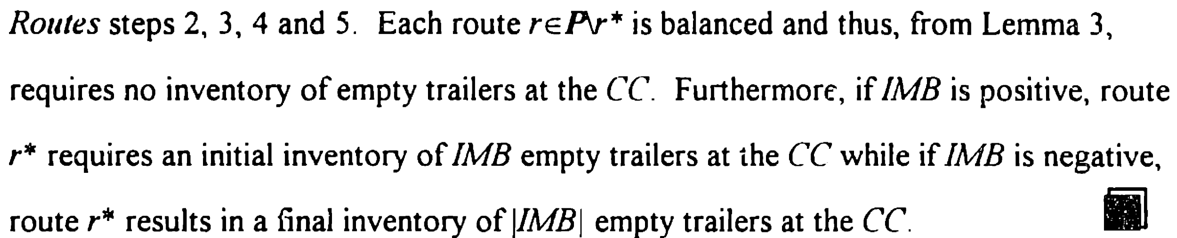
Sequence Routes **Step 5:** Repeat the following steps until $\boldsymbol{R}_2^+$ is empty: 1) select any two routes, denoted r_1^- and q_1^-, from $\boldsymbol{R}_1^-$ and select any route, denoted r_2^+, from $\boldsymbol{R}_2^+$; 2) let r^S denote the *super-route* formed by performing routes r_1^- and q_1^- followed by route r_2^+; 3) let $\boldsymbol{P} = \boldsymbol{P} + r^S$; 4) let $\boldsymbol{R}_1^- = \boldsymbol{R}_1^- - r_1^- - q_1^-$; and 4) let $\boldsymbol{R}_2^+ = \boldsymbol{R}_2^+ - r_2^+$.

Sequence Routes **Step 6:** Let r* = 0 and repeat the following steps $|\boldsymbol{R}_\text{I}|$ times : 1) select from $\boldsymbol{R}_\text{I}$ any route, denoted r_I, with $v_{r_I} < 0$ if one exists, otherwise select any route, denoted r_I, with $v_{r_I} \geq 0$; 2) let r^* denote the super-route formed by performing route r^* followed by route r_I; and 3) let $\boldsymbol{R}_\text{I} = \boldsymbol{R}_\text{I} - r_\text{I}$.

Sequence Routes **Step 7:** Let $\boldsymbol{P} = \boldsymbol{P} + r^*$.

Lemma 4: Given the routes, the loaded trailer assignments generated in IGLH_MATCH Steps 1 and 2, and the empty trailer assignments $e^r = X^r - x^r$ for each route $r \in \boldsymbol{R}$, minimization of the initial and final empty trailer inventories at the *CC* can be achieved with the route set $\boldsymbol{P}$.

Proof: From the definition of v_r, the absolute value of imbalance for any route $r \in \boldsymbol{R}$, i.e., $|v_r|$, is either 0, 1 or 2. Therefore, sets $\boldsymbol{R}_1$, $\boldsymbol{R}_0$, $\boldsymbol{R}_1^+$, $\boldsymbol{R}_1^-$, $\boldsymbol{R}_2^+$, $\boldsymbol{R}_2^-$ partition set $\boldsymbol{R}$. Furthermore, by design, $\sum_{r \in R_I} v_r = \sum_{r \in R} v_r$ and it follows that $\sum_{r \in R \setminus R_I} v_r = 0$, or equivalently, $\sum_{r \in R_1^+ \cup R_2^+} v_r = \sum_{r \in R_1^- \cup R_2^-} v_r$. Thus, there always exist appropriate routes for selection in *Sequence*



Routes steps 2, 3, 4 and 5. Each route $r \in \boldsymbol{P} \setminus r^*$ is balanced and thus, from Lemma 3, requires no inventory of empty trailers at the *CC*. Furthermore, if *IMB* is positive, route r^* requires an initial inventory of *IMB* empty trailers at the *CC* while if *IMB* is negative, route r^* results in a final inventory of $|IMB|$ empty trailers at the *CC*. ∎

The *Sequence Routes* procedure is illustrated in Table 2.6 for two problems: one with a positive group imbalance and another with a negative imbalance. The examples show that through appropriate sequencing and merging of routes, the inventory of empty trailers at the *CC* can be reduced substantially.

Through appropriate sequencing of the routes generated in Steps 1 and 2 of IGLH_MATCH, the required inventory of empty trailers at the *CC* can be reduced from 6 to 2 for Problem 1 and from 4 to 0 for Problem 2.

Table 2.6 *Sequence Routes* Procedure

Steps 1 & 2 IGLH_MATCH routes	Problem 1				Problem 2			
	Route Imbal-ance†	Number of Empties required at the CC	IGLH_MATCH Step 3 Routes (Sequence Routes)		Route Imbal-ance†	Number of Empties required at the CC	IGLH_MATCH Step 3 Routes (Sequence Routes)	
			Sequence	Imbalance			Sequence	Imbalance
r_1	-1	0	$r_1 \rightarrow r_4$	0	0	0	r_1	0
r_2	0	0	r_2	0	-1	0	$r_2 \rightarrow r_3$	0
r_3	2	2	-	-	1	1	-	-
r_4	1	1	-	-	-2	0	$r_4 \rightarrow r_6$	0
r_5	-1	0	$r_5 \rightarrow r_{10}$	0	-2	0	r_5	-2
r_6	0	0	r_6	0	2	2	-	-
r_7	0	0	r_7	0	0	0	r_7	0
r_8	2	2	r_8	2	-1	0	$r_8 \rightarrow r_9$	0
r_9	-2	0	$r_9 \rightarrow r_3$	0	1	1	-	-
r_{10}	1	1	-	-	0	0	r_{10}	0
Total	2	6		2	-2	4		-2

† : Route imbalance equals the total number of pickups minus the total number of deliveries for that route.

Chapter 3 Computational Experiences

The IGLH_MATCH procedure was tested using randomly generated data designed to reflect the IGLH operations of large LTL carriers. The randomly generated problems have from 2 to 100 EOL terminals, and for each number of EOL terminals, a set of five different problems are generated. The EOL terminals are categorized as small, medium, or large, with each size equally likely to occur in the market. The required pickups and deliveries at each terminal are determined from the probability mass function shown in Table 3.1. Based on typical intra-group line-haul operations at various LTL carriers, the maximum number of pickups (deliveries) at any one terminal is assumed to be four. The distances between each pair of terminals and between the tenninal and the *CC* were generated so as to preserve the triangle inequality.

Table 3.1 Probability Mass Function for Pickups and Deliveries

EOL Terminal Size			Small	Medium	Large
Probability of	1	pickup	0.90	0.50	0.30
		delivery	0.75	0.40	0.20
	2	pickups	0.10	0.40	0.40
		deliveries	0.20	0.40	0.30
	3	pickups	0.00	0.10	0.30
		deliveries	0.05	0.20	0.40
	4	pickups	0.00	0.00	0.00
		deliveries	0.00	0.00	0.10

The total number of pickups and deliveries for each problem were not necessarily balanced. This reflects, for example, the fact that the number of deliveries on Monday often exceeds the number of pickups, with the reverse occurring on Friday.

The purpose of the computational experiments was twofold: 1) to evaluate the

quality of the solutions generated by IGLH_MATCH; and 2) to determine both the memory and run time requirements of the IGLH_MATCH procedure.

Implementation

The IGLH_MATCH procedure was implemented in the C programming language (see Appendix 1) and all computational tests were performed on an IBM RS\6000, Model 370 workstation. The *Match* and *Re-Match* procedures in IGLH_MATCH Step 2 used the nonbipartite weighted matching algorithm of Gabow (1973) with its worst case running time $O(n^3)$, where n is the number of network nodes. Therefore, the worst case running time of IGLH_MATCH is $O(\{2N\lceil MAX\ (pickups/2,\ deliveries/2)\rceil\}^3)$, where N is the number of terminals. The total number of network links is, due to complete network, at most $[\{2N\lceil MAX(pickups/2,\ deliveries/2)\rceil\} * \{2N\lceil MAX(pickups/2,\ deliveries/2)\rceil - 1\}] / 2$. Hence, for the randomly generated problems where the maximum number of pickups (deliveries) is four, the worst case running time for the nonbipartite weighted matching algorithm is $O(64N^3)$, and the number of network links is $8N^2 - 2N$.

Examples of the required input file formats and output files that are generated by IGLH_MATCH are shown at Appendix 2.

Computational Results

For the smaller test problems, i.e., up to eight EOL terminals, the objective function values of the IGLH_MATCH solutions were compared with the optimal IGLH solution values obtained using a branch-and-bound procedure(IGLH_BB), implemented using IBM's Optimization Subroutine Library (OSL) (Optimization Subroutine Library, 1992). Table 3.3 shows that the IGLH_MATCH procedure determined optimal solutions for problems containing up to six EOL terminals. For the data sets containing seven and eight EOL terminals, only two of the five test problems could be solved exactly in each

case. For these problems, the average gaps between the IGLH_MATCH and exact solutions were 6.44% and 3.59%, respectively. The average median IGLH_MATCH solution time was 0.0 seconds, compared to 391.7 seconds required for IGLH_BB.

For larger problems containing nine or more EOL terminals, optimal solutions could not be achieved due to insufficient memory. Hence, the IGLH_MATCH objective function value was compared to a lower bound on the optimal IGLH solution, obtained by solving the linear relaxation of IGLH (IGLH_LP), implemented using OSL. For these problems, Table 3.2 shows that the IGLH_MATCH procedure typically generates solutions within ten percent of the IGLH_LP lower bound, with the gap decreasing as the number of EOL terminals increases. The IGLH_MATCH solution time never exceeded 10 seconds, while the average median IGLH_LP solution time was 182.9 seconds. For problems containing 100 EOL terminals, over one thousand seconds were required by the IGLH_LP procedure.

In conclusion, IGLH_MATCH produces optimal or near-optimal solutions for very large IGLH problems in seconds.

Table 3.2 IGLH_MATCH Computational Results for Randomly Generated Problems

Number of EOL terminals	Median Run Time (sec) †			Solution Gap (%) ‡	
	IGLH_MATCH	IGLH_BB	IGLH_LP	IGLH_MATCH vs. IGLH_BB	IGLH_MATCH vs. IGLH_LP
2	0.0	0.0	0.0	0.00	23.48
3	0.0	0.0	0.0	0.00	19.20
4	0.0	1.0	0.0	0.00	15.38
5	0.0	8.0	0.0	0.00	12.43
6	0.0	155.0	0.0	0.00	7.98
7	0.0	1225.0	0.0	6.44	12.48
8	0.0	1353.0	0.0	3.59	9.44
9	0.0	*	0.5	N/A	11.58
10	0.0	*	1.0	N/A	10.59
15	0.0	*	1.0	N/A	8.93
20	0.0	*	3.0	N/A	12.37
25	1.0	*	5.0	N/A	8.86
30	1.0	*	10.0	N/A	8.24
35	1.0	*	16.0	N/A	6.18
40	1.0	*	26.5	N/A	6.31
45	1.0	*	41.0	N/A	4.81
50	2.0	*	65.0	N/A	6.66
55	3.0	*	93.0	N/A	5.11
60	3.0	*	126.5	N/A	4.26
65	4.0	*	195.0	N/A	5.97
70	4.0	*	255.0	N/A	4.14
75	5.0	*	322.0	N/A	4.82
80	5.0	*	479.5	N/A	4.41
85	6.0	*	568.0	N/A	4.53
90	7.0	*	716.5	N/A	4.45
95	9.0	*	933.5	N/A	4.04
100	9.0	*	1079.5	N/A	4.16
Average	2.30	N/A	182.87	N/A	8.55

† : IBM RS\6000, Model 370 workstation

‡ : Solution Gap (%) = {(IGLH_MATCH solution - IGLH_BB (or IGLH_LP)) / IGLH_BB (or IGLH_LP)} * 100

* : Run time was not available due to insufficient memory.

Chapter 4 Model Flexibility and Adaptability

The core IGLH problem presented and formulated in Chapter 1 is a simplified version of the IGLH problem faced by LTL carriers. Several practical considerations, such as driver work rules, time window restrictions, level of service requirements, etc., are not included in the model formulation (1) - (7). The IGLH_MATCH procedure, however, can easily accommodate such restrictions.

Consider for example, the work rule limiting the total number of hours a driver can be on duty in any given day. This rule can be enforced within the IGLH_MATCH procedure with a simple legality check. That is, since each link in the *Match* and *Re-Match* network corresponds to a route, each route can be evaluated to ensure that its total elapsed time does not exceed the limit. If the limit is exceeded, the link corresponds to an illegal route and its cost is set to infinity. Similarly, any route not satisfying level of service requirements, time windows, etc., can be eliminated from consideration by setting its corresponding link cost to infinity. These legality checks can be performed quickly, with a small increase in the total solution run time.

Our computational experiments show that within a few seconds, the IGLH_MATCH procedure can produce solutions to IGLH problems that are larger than those faced by most American LTL carriers. Thus, IGLH_MATCH can provide a tool for real-time scheduling of IGLH operations. Not all required pickups and deliveries are known at the time driver routes and schedules are developed. There is a need, therefore, to alter certain routes and develop new routes while keeping some routes fixed. The IGLH_MATCH procedure fixes a route by removing from consideration the pickups and deliveries it services, i.e., by eliminating the corresponding nodes (and hence, links) in the matching network. An existing route is altered by including the route as a node in the matching network and adding links to every other network node, with link cost set to

infinity if the corresponding altered route is illegal. Finally, altered and new routes are determined by applying the IGLH_MATCH procedure to the modified network. Hence, IGLH_MATCH can produce very quickly new IGLH solutions that reflect current pickup and delivery information and previously defined routes.

Chapter 5 Conclusions and Future Research

IGLH_MATCH, a simple, flexible procedure relying on matching theory, has been shown to produce quality IGLH solutions extremely quickly on a workstation class computer. These IGLH solutions attempt to minimize simultaneously the number of tractor miles traveled and the empty trailer inventory size at the *CC*. Practical considerations concerning work rules, time windows, level of service requirements, etc., can be modeled easily and incorporated into the IGLH_MATCH solution process with a simple legality check. Furthermore, the IGLH_MATCH procedure is a viable method for real-time solution of IGLH problems: partial solutions can be fixed and re-optimization is extremely fast.

Future work is necessary to quantify the practical value of the IGLH_MATCH procedure. That is, by comparing IGLH_MATCH solutions to those obtained by practitioners solving IGLH problems daily, savings associated with the IGLH_MATCH procedure can be quantified.

Bibliography

1. Sheffi, Y. and W. Powell (1985), "Interactive Optimization for LTL Network Design", Center for Transportation Studies Report 85-17, Massachusetts Institute of Technology, Cambridge, Massachusetts.
2. Golden, B. L. and A. A. Assad (1988), "Vehicle Routing : Methods and Studies", **Studies in Management Science and Systems**, Vol. 16, North-Holland, Amsterdam.
3. Magnanti, T.L. (1981), "Combinatorial Optimization and Vehicle Fleet Planning : Perspectives and Prospects", **Networks, Vol. 11**, pp. 179 - 213.
4. Sheffi, Y. and J. Eckstein (1987), "Optimization of Group Line-Haul Operations for Motor Carriers Using Twin Trailers", **Transportation Research Record 1120**, pp. 12-23.
5. Eckstein, J. (1986), "Routing Methods for Twin-Trailer Trucks", Masters Thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, Operations Research Center, pp. 10.
6. Gabow, N. H. (1973), "Implementation of Algorithms for Maximum Matching on Nonbipartite Graphs", Ph.D. Thesis, Stanford University, Stanford, California.
7. Optimization Subroutine Library (1992), "Guide and Reference, Release 2", Publication No. SC23-0519-03, IBM Corporation, Kingston, NY.

Appendix 1 IGLH_MATCH.C Code

```
/**************************************************************************************
 FILENAME : ighlmatch.c

 RF    : GENERALIZED WEIGHTED MATCHING FOR NON-BIPARTITE GRAPH

        This program solves Intra-Group Line-Haul (IGLH_MATCH) problem and heuristically
        achieving empty trailer balancing requirements by sequencing routes.

         NONBIPARTITE WEIGHTED MATCHING PROGRAM (A function called
        "Weighted_Match") returns nodes and their mate node numbers ([i] and Mate[i]) for
        i=1,2,..., size, where size is equal to the total number of nodes(including copy nodes) in
        the network.  In reading intermediate " .tmp" file generated by two input files (input1
        and input2), we used a function called  "ReadGraph".

 RUN :  iglhmatch input1 input2 output1 output2 , where
                input1  - Terminal pickup and delivery finename.
                input2  - Cost matrix.
                output1 - Output filename.
                output2 - route map.

 FORMAT OF INPUTS:

      input1
            total_number_of_EOL_terminals    MAX(supply, delivery)
               EOL_terminal_number    pickups   deliveries

      input2
            for(i=0; i<= total_EOL_number; i++){
                  for(j=0; j <= total_EOL_number; j++){
                     printf("%d", cost[i][j]);
                  }
                  printf("\n");
               }

 AUTHOR :  DAEKI KIM
*************************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <limits.h>

#define NULL 0
#define CALLOC(X, KOUNT) ((X *)calloc((KOUNT), sizeof(X)))
#define MALLOC(X)       ((X *)malloc(sizeof(X)))
#define YES  1
#define NO   0
```

```
typedef struct edge_ent {
  int endpoint;
  int label;
  int label2;
  struct edge_ent *nextedge;
  struct edge_ent *prevedge;
  struct edge_ent *otheredge;
} *Edge;

typedef struct node_entry {
  int degree;
  int label;
  int x;
  int y;
  struct edge_ent *adj_list;
} *Graph;

typedef struct eol {
  int num;
  int index;
  int pickup;
  int delivery;
} Terminal;

typedef struct history {
  int num;
  int i;
  int mate;
  int pickup;
  int delivery;
} Info;

typedef struct route {
  int strgy;
  int from;
  int from_pk;
  int from_dl;
  int mate_from;
  int mate_from_pk;
  int mate_from_dl;
  int to;
  int to_pk;
  int to_dl;
  int mate_to;
  int mate_to_pk;
  int mate_to_dl;
  int min_cost;
} Route_Info;

typedef struct unbalanced {
  int route_num;
  int trailer;
  struct unbalanced *prev;
```

```
 struct unbalanced *next;
} Unbalanced;

typedef struct header {
 Unbalanced *first;
 Unbalanced *last;
 int      count;
} Header;


int            *dum_tractor, *tractor, trac_col;
int            max_node, max_split, *cost, column;
int               *fcost, *re_cost;
double         pre_cost, post_cost;
Info           *match_info;
int               COLUMN, UPPER, route_col;
Route_Info *route_info;
Header       *header_surplus, *header_deficit;

/****************************************************************************************
        MAIN Function
****************************************************************************************/
void main(int argc, char **argv)
{
 double      optimal, optimal_2;
 double      matching_solution(Terminal *, int, int *, FILE *, char *);
 FILE        *fpo, *fpo_map;
 time_t      first, second, third, fourth, fifth, sixth, seventh;
 void        elapsed_time_print(FILE *,time_t,time_t,time_t,time_t,time_t,time_t,time_t);
 Terminal  *temp, *temp_2;
 Terminal  *preprocessing(char *);
 void        matching_network_creation(Terminal *, char *, char *);
 Graph       graph, graph_2;
 int         *Mate, *Mate_2, size, size_2;

 printf("\n\n******** New Matching Procedure *************");
 printf("\nTerminal pickup/delivery filename : %s", argv[1]);
 printf("\nCost(distance) matrix  filename : %s", argv[2]);
 printf("\nOutput filename                  : %s", argv[3]);
 printf("\nRoute Map Filename                : %s", argv[4]);

 fpo = fopen(argv[3], "w");
 if(fpo == NULL){
          printf("\nERROR. Can not write file : %s. No Memory is left.",argv[3]);
          exit(1);
 }

 fprintf(fpo,"\n**********************************");
 fprintf(fpo,"\nInput File  : %s %s",argv[1],argv[2]);
 fprintf(fpo,"\nOutput File : %s",argv[3]);
 fprintf(fpo,"\nRoute Map File : %s",argv[4]);
 fprintf(fpo,"\n**********************************\n");
```

```
first = time(NULL);

pre_cost  = 0.;
post_cost = 0.;

/* Preprocess */
 temp  = preprocessing(argv[1]);
matching_network_creation(temp, argv[2], "netfile1.tmp");

/* Match Starts */
second = time(NULL);
graph = ReadGraph(&size,"netfile1.tmp");

/* removes netfile1.tmp file to save disk space */
i = remove("netfile1.tmp");
if(i != 0) {
 printf("\nERROR. netfile1.tmp must have been there.");
}
Mate  = Weighted_Match(graph,1);
third = time(NULL);

optimal = matching_solution(temp, size, Mate, fpo, argv[4]);

free(temp);
free(Mate);

fourth = time(NULL);

/* Re-Match Network Build */
temp_2  = preprocessing("retest.in");
matching_network_creation(temp_2, "recost.mat", "netfile2.tmp");

fifth = time(NULL);

/* Re-Match Starts */
graph_2 = ReadGraph(&size_2,"netfile2.tmp");

/* removes netfile2.tmp file to save disk space */
i = remove("netfile2.tmp");
if(i != 0) {
 printf("\nERROR. netfile2.tmp must have been there.");
}

Mate_2  = Weighted_Match(graph_2,1);
sixth = time(NULL);

optimal_2 = matching_solution(temp_2, size_2, Mate_2, fpo, argv[4]);

printf("\n\n This program has been successfully executed.");
fprintf(fpo,"\n\n This program has been successfully executed.");
printf("\n Previous matching solution was : %.0lf",optimal);
printf("\n Current matching solution is   : %.0lf", optimal_2);
fprintf(fpo,"\n Previous matching solution was : %.0lf",optimal);
```

```
 fprintf(fpo,"\n Current matching solution is  : %.0lf", optimal_2);

 seventh = time(NULL);
 elapsed_time_print(fpo, first, second, third, fourth, fifth, sixth, seventh);

 fclose(fpo);
}

/*************************************************************************************
 MATCHING-SOLUTION PRINTOUT AND RE-MATCHING STARTS
*************************************************************************************/
double matching_solution(Terminal *temp, int size, int *Mate, FILE *fpo, char *filename_2)
{
 int   i, j, k, ccost;
 double total_cost, ntractor;
 int   next_max_node;
 FILE  *fpo_1, *fpo_2;
 void  rematching_network(int, int, int *, Terminal *, FILE *);
 int   space, route_pickup, route_delivery, route_empty;
 int       network_pickup,network_delivery,network_empty,network_cost;
 void  empty_balancing_1(int, int);
 void  empty_balancing_2(int, FILE *);
 static int step = 0;

 step++;

 trac_col = size+1;

 tractor = CALLOC(int,trac_col*trac_col);
 if(!tractor) {
  printf("\nERROR.  Out of Memory in tractor.");
  exit(1);
 }

 /* preprocessed tractor # assignment */
 for(i=0; i <= max_node; i++)
        tractor[i] = dum_tractor[i];

 /* MATCHING SOLUTION PRINTOUT */
 if(step == 1)
  fprintf(fpo,"****\n1. IGLH_MATCH STEP 1 OUTPUT ****");
 else if(step > 1)
  fprintf(fpo,"\n\f****5. IGLH_MATCH STEP 1 OUTPUT(RE-MATCH NETWORK) ****");

 fprintf(fpo,"\n\n    NUM    Index  Supply  Demand  Tractor(PRE)");

 for (i=1; i <= max_split; i++) {
  fprintf(fpo,"\n%8d %8d %8d %8d %8d", temp[i].num, temp[i].index,
            temp[i].pickup, temp[i].delivery, tractor[i]);
  if(step == 1) pre_cost += cost[temp[i].index] * tractor[i];
 }
 if(step ==1)
  fprintf(fpo, "\n\n2. IGLH_MATCH STEP 2 MATCHED NODES(INITIAL)");
```

```
else
  fprintf(fpo, "\n\n6. IGLH_MATCH STEP 2 MATCHED NODES(RE-MATCH NETWORK)");
fprintf(fpo, "\n\n          FROM                TO\n");

k=0;
for (i=1; i <= size; i++) {
  ccost = cost[temp[i].index * column + temp[Mate[i]].index];
  if (((temp[i].pickup == 0)&&(temp[i].delivery == 0))
        &&((temp[Mate[i]].pickup == 0)&&(temp[Mate[i]].delivery == 0)))
                ccost = 0;
  if (ccost != 0) {

    /* To prevent not printing 0 matched node */
    if(Mate[i] == 0) {
        Mate[i] = i + max_split;
        Mate[i+max_split] = i;
    }

    if (i < Mate[i]) {
        k++;
        fprintf(fpo,"%5d. %8d [%8d]  <--> %8d [%8d]\n",k, i, temp[i].index,
                Mate[i], temp[Mate[i]].index);
    }
    tractor[temp[i].index*trac_col+temp[Mate[i]].index]++;
  }
}

next_max_node = k;

total_cost = 0;

if(step == 1)
  fprintf(fpo, "\n3. IGLH_MATCH INITIAL SOLUTION");
else
  fpritnf(fpo, "\n7. IGLH_REMATCH SOLUTION");
fprintf(fpo, "\n\n       FROM       TO   TRACTORS  COSTS");
k = 0;
for (i=0; i <= size; i++) {
  for (j=1; j <= size; j++) {
    if ((i<j)&&(tractor[i*trac_col+j] != 0)){
        k++;
        fprintf(fpo, "\n%5d. %5d   <--> %5d %8d %8d", k, i, j,
                tractor[i*trac_col+j], cost[i*column + j] * tractor[i*trac_col+j]);
        total_cost += cost[i*column + j] * tractor[i*trac_col+j];
    }
  }
}


if(step == 1)
  post_cost = total_cost - pre_cost;
else {
  post_cost  = total_cost;
```

```
  total_cost = post_cost + pre_cost;
 }

 fprintf(fpo,"\n=================================================================");
 fprintf(fpo, "\n\n    TOTAL TRACTOR COST  = %12.0lf", total_cost);
 fprintf(fpo, "\n    PRE -PROCESSED COST = %12.0lf", pre_cost);
 fprintf(fpo, "\n    POST-PROCESSED COST = %12.0lf", post_cost);

 if(step == 1) {
  fpo_1 = fopen(filename_2, "w");
  if(!fpo_1){
   printf("\nERROR. I can't write RouteMap file of %s", filename_2);
   exit(1);
  }

  /* PREPROCESSED ROUTE MAP PRINTING */
  fprintf(fpo_1, "********************** \n");
  fprintf(fpo_1, "PREPROCESSED ROUTE MAP \n");
  fprintf(fpo_1, "********************** \n\n");
  for(k=0,i=1; i <= max_node; i++) {
   if (dum_tractor[i] > 0) {
       k++;
       fprintf(fpo_1, "%3d.  CC-> %d(2,2)-> CC %5d(UC) %5d(TR) %5d(TC)\n",
              k, temp[i].num,cost[temp[i].index],dum_tractor[i],
              cost[temp[i].index]*dum_tractor[i]);
   }
  }
  fprintf(fpo_1,"\n");

  if (k == 0)
   fprintf(fpo_1, "     No Preprocessed (Loaded Direct) Trips.\n");
  else
   fprintf(fpo_1, "  PreProcessed Trip Cost : %12.0lf.",pre_cost);

  fclose(fpo_1);
  rematching_network(next_max_node,size,Mate,temp,fpo);
 }

 /* VIA TRIP ROUTE MAP PRINTING */
 else {
  fpo_2 = fopen(filename_2, "a");
  if(!fpo_2){
   printf("\nERROR. I can't write RouteMap file of %s", filename_2);
   exit(1);
  }
  k = 0;
  network_pickup = network_delivery = network_empty = network_cost = 0;
  fprintf(fpo_2,"\n\n*****************\n");
  fprintf(fpo_2,  "MATCHED ROUTE MAP\n");
  fprintf(fpo_2,  "*****************\n\n");
  for(i=0; i < route_col; i++) {
   for(j=0; j < route_col; j++) {
       if ((i<j)&&(route_info[i*route_col+j].strgy != 0)&&(tractor[i*trac_col+j] != 0)){
```

```
route_pickup = route_delivery = route_empty = 0;
k++;
space = 0;

fprintf(fpo_2,"%3d.  CC", k);

if(route_info[i*route_col+j].from != 0){
 fprintf(fpo_2,"-> %3d(%2d,%2d)",
        route_info[i*route_col+j].from,
        route_info[i*route_col+j].from_pk,
        route_info[i*route_col+j].from_dl);
}
if(route_info[i*route_col+j].from == 0)
 space++;

if(route_info[i*route_col+j].mate_from != 0){
 fprintf(fpo_2,"-> %3d(%2d,%2d)",
        route_info[i*route_col+j].mate_from,
        route_info[i*route_col+j].mate_from_pk,
        route_info[i*route_col+j].mate_from_dl);
}
if(route_info[i*route_col+j].mate_from == 0)
 space++;

if(route_info[i*route_col+j].to != 0){
 fprintf(fpo_2,"-> %3d(%2d,%2d)",
        route_info[i*route_col+j].to,
        route_info[i*route_col+j].to_pk,
        route_info[i*route_col+j].to_dl);
}
if(route_info[i*route_col+j].to == 0)
 space++;

if(route_info[i*route_col+j].mate_to != 0){
 fprintf(fpo_2,"-> %3d(%2d,%2d)",
        route_info[i*route_col+j].mate_to,
        route_info[i*route_col+j].mate_to_pk,
        route_info[i*route_col+j].mate_to_dl);
}
if(route_info[i*route_col+j].mate_to == 0)
 space++;

fprintf(fpo_2,"->  CC");

switch (space) {
case 0: break;
case 1:
 fprintf(fpo_2,"             ");
 break;
case 2:
 fprintf(fpo_2,"                          ");
 break;
```

```
        case 3:
         fprintf(fpo_2,"                              ");
         break;
        case 4:
         fprintf(fpo_2,"                                     ");
         break;
        default:
         break;
        }

        fprintf(fpo_2," %8d\n",route_info[i*route_col+j].min_cost);

        route_pickup = route_info[i*route_col+j].from_pk +
         route_info[i*route_col+j].mate_from_pk +
          route_info[i*route_col+j].to_pk +
              route_info[i*route_col+j].mate_to_pk;
        route_delivery = route_info[i*route_col+j].from_dl +
         route_info[i*route_col+j].mate_from_dl +
          route_info[i*route_col+j].to_dl +
              route_info[i*route_col+j].mate_to_dl;

        route_empty = route_delivery - route_pickup;


        /* Empty Balancing_1 Subroutine Call */
        if(route_empty != 0) empty_balancing_1(k, route_empty);
        /* End of Empty Balancing_1 Subroutine Call */

        network_pickup += route_pickup;
        network_delivery += route_delivery;
        network_cost   += route_info[i*route_col+j].min_cost;
       }
   }
  }
  network_empty = network_delivery - network_pickup;
  fprintf(fpo_2,"\n   MATCHING_Pickup = %5d, MATCHING_Delivery = %5d\n",
          network_pickup, network_delivery);
  fprintf(fpo_2,"                    MATCHING_Empties    =    %5d,      MATCHING_Cost     =
%8d",network_empty,network_cost);

  if (network_empty == 0)
   fprintf(fpo_2,"    Balanced MATCHING Network\n");
  else if (network_empty > 0)
   fprintf(fpo_2,"    Surplus MATCHING Network\n");
  else
   fprintf(fpo_2,"    Deficit MATCHING Network\n");

  if(k==0)
   fprintf(fpo_2,"\n    No Via Trips Available.");

  /* Empty Balancing_2 Subroutine Call */
  else
   empty_balancing_2(network_empty, fpo_2);
```

```
  /* End of Empty Balancing_2 Subroutine */

  fprintf(fpo_2,"\n\n******************\n");
  fprintf(fpo_2,  "IGLH_MATCH SUMMARY\n");
  fprintf(fpo_2,  "******************\n\n");
  fprintf(fpo_2,"   Preprocessed Cost  : %12.0lf\n",pre_cost);
  fprintf(fpo_2,"   Matched Trip Cost  : %12.0lf\n",post_cost);
  fprintf(fpo_2,"   Total Tractor Cost : %12.0lf",total_cost);

  fclose(fpo_2);

 }

 free(cost);
 free(tractor);
 return(total_cost);          /* returns initial matching solution */
}

/*************************************************************************************
 EMPTY TRAILER BALANCING (FIRST STEP) : CREATING TWO GROUPS
                                        SURPLUS && DEFICIT
*************************************************************************************/
void empty_balancing_1(int k, int route_empty)
{
 Unbalanced *new_route;
 static int  step = 0;

 step++;
 if (step == 1) {
  header_surplus = MALLOC(Header);
  header_deficit = MALLOC(Header);
  if (header_surplus){
   header_surplus->first = header_surplus->last = NULL;
   header_surplus->count = 0;
  }
  else {
   printf("\nERROR. Lack of Memory in EMPTY_TRAILER BALANCING");
   exit(1);
  }

  if(header_deficit){
   header_deficit->first = header_deficit->last = NULL;
   header_deficit->count = 0;
  }
  else {
   printf("\nERROR. Lack of Memory in EMPTY_TRAILER BALANCING");
   exit(1);
  }
 }

 new_route = MALLOC(Unbalanced),
 if (new_route) {
  new_route->prev = new_route->next = NULL;
```

```
  new_route->route_num = k;
  new_route->trailer = route_empty;
 }
 else {
  printf("\nLack  of Memory in Empty Trailer Balancing Step 1_New Route");
  exit(1);
 }
 /* Surplus Route */

 if (route_empty > 0){

  if(header_surplus->last == NULL){          /* The very first item */
   header_surplus->first =          new_route;
   header_surplus->last  = new_route;
   (header_surplus->count)++;
  }
  else if (route_empty == 1) {        /* Add to the end of the queue */
   new_route->prev = header_surplus->last;
   header_surplus->last->next = new_route;
   header_surplus->last = new_route;
   (header_surplus->count)++;
  }
  else if(route_empty == 2) {        /* Add at the first of the queue */
   new_route->next = header_surplus->first;
   header_surplus->first->prev = new_route;
   header_surplus->first = new_route;
   (header_surplus->count)++;
  }
  else {
   printf("\nERROR. ROUTE_EMPTIES MUST BE EITHER 1 OR 2.");
        }
 }

 /* Deficit Route */

 if (route_empty < 0){
  new_route->trailer = -(new_route->trailer);

  if(header_deficit->last == NULL){
                    header_deficit->first =      new_route;
   header_deficit->last = new_route;
   (header_deficit->count)++;
  }
  else if (route_empty == -1) {        /* Add to the end of the queue */
   new_route->prev = header_deficit->last;
   header_deficit->last->next = new_route;
   header_deficit->last = new_route;
   (header_deficit->count)++;
  }
  else if(route_empty == -2) {        /* Add at the first of the queue */
   new_route->next = header_deficit->first;
   header_deficit->first->prev = new_route;
   header_deficit->first = new_route;
```

```
    (header_deficit->count)++;
   }
   else {
    printf("\nERROR. ROUTE_EMPTIES MUST BE EITHER 1 OR 2.");
   }
  }
}

/*****************************************************************************
 EMPTY TRAILER BALANCING (SECOND STEP): SEQUENCING ROUTES
                                    SURPLUS && DEFICIT
*****************************************************************************/
void empty_balancing_2(int network_empty, FILE *fpo_2)
{
 Unbalanced  *tmp_surplus, *tmp_deficit;
 Unbalanced  *dummy_route;
 int         Value, k=0;

 fprintf(fpo_2,"\n\n************************\n");
 fprintf(fpo_2,   "SEQUENCE ROUTES SUMMARY\n");
 fprintf(fpo_2,   "************************\n\n");


 if(network_empty == 0) {

  tmp_surplus = header_surplus->first;
  tmp_deficit = header_deficit->first;

  fprintf(fpo_2,"* Balanced Network: No additional empties are required.\n\n");

  if(header_surplus->count == header_deficit->count) {
   while(tmp_surplus != NULL) {
       k++;
       fprintf(fpo_2,"%3d.  Sroute[%d] + Droute[%d]\n",k,
               tmp_surplus->route_num,tmp_deficit->route_num);
       tmp_surplus = tmp_surplus->next;
       tmp_deficit = tmp_deficit->next;
   }
  }
  else if(header_surplus->count < header_deficit->count) {
   while(tmp_surplus != NULL) {
       k++;
       if(tmp_surplus->trailer == tmp_deficit->trailer){
        fprintf(fpo_2,"%3d.  Sroute[%d] + Droute[%d]\n",k,
                tmp_surplus->route_num,tmp_deficit->route_num);
        tmp_surplus = tmp_surplus->next;
        tmp_deficit = tmp_deficit->next;
       }
       else if(tmp_surplus->trailer > tmp_deficit->trailer) {
        fprintf(fpo_2,"%3d.  Sroute[%d] + Droute[%d] + Droute[%d]\n",k,
                tmp_surplus->route_num,tmp_deficit->route_num,tmp_deficit->next->route_num);
        tmp_surplus = tmp_surplus->next;
        tmp_deficit = tmp_deficit->next->next;
```

```
        }
        else
           printf("\nError Has Occurred in Sorting EMPTY TRAILER Queueing(BALANCED
NETWORK)");
    }
  }
  else {          /* if(header_surplus->count > header_deficit->count) */
   while(tmp_surplus != NULL){
        k++;
        if(tmp_surplus->trailer == tmp_deficit->trailer){
         fprintf(fpo_2,"%3d.  Sroute[%d] + Droute[%d]\n",k,
                 tmp_surplus->route_num,tmp_deficit->route_num);
         tmp_surplus = tmp_surplus->next;
         tmp_deficit = tmp_deficit->next;
        }
        else if(tmp_surplus->trailer < tmp_deficit->trailer) {
         fprintf(fpo_2,"%3d.  Sroute[%d] + Sroute[%d] + Droute[%d]\n",k,
                 tmp_surplus->route_num,tmp_surplus->next->route_num,tmp_deficit->route_num);
         tmp_surplus = tmp_surplus->next->next;
         tmp_deficit = tmp_deficit->next;
        }
        else
         printf("\nError Occurred in Sorting EMPTY TRAILER Queueing(BALANCED NETWORK)");
    }
  }
 }


/* SURPLUS NETWORK */

else if(network_empty > 0) {
  fprintf(fpo_2,"*  Surplus Network: %d empties are returned to CC.\n\n",network_empty);
  Value = network_empty;
  if(Value <= 2) {
   dummy_route = MALLOC(Unbalanced);
   if(Value == 1) {
        if (dummy_route) {
         dummy_route->prev = dummy_route->next = NULL;
         dummy_route->route_num = 0;
         dummy_route->trailer = 1;

         if(header_deficit->last == NULL) {
          header_deficit->first = dummy_route;
          header_deficit->last = dummy_route;
          (header_deficit->count)++;
         }
         else{
          dummy_route->prev = header_deficit->last;    /* Add at the end */
          header_deficit->last->next = dummy_route;
          header_deficit->last = dummy_route;
          (header_deficit->count)++;
         }
        }
```

```
      else {
        printf("\nLack  of Memory in Empty Trailer Balancing Step 2_New Route");
        exit(1);
      }
                  }
   if(Value == 2) {
      if(dummy_route) {
        dummy_route->prev = dummy_route->next = NULL;
        dummy_route->route_num = 0;
        dummy_route->trailer = 2;

        if(header_deficit->last == NULL){
          header_deficit->first = dummy_route;
          header_deficit->last = dummy_route;
          (header_deficit->count)++;
        }
        else{
          dummy_route->next = header_deficit->first;  /* Add at the front */
          header_deficit->first->prev = dummy_route;
          header_deficit->first = dummy_route;
          (header_deficit->count)++;
        }
      }
      else {
        printf("\nLack  of Memory in Empty Trailer Balancing Step 2_New Route");
        exit(1);
      }
   }
 }
 else {                                              /* If (Value > 2) */
  while(Value > 1){
      dummy_route = MALLOC(Unbalanced);
      dummy_route->prev = dummy_route->next = NULL;
      dummy_route->route_num = 0;
      dummy_route->trailer = 2;

      if(header_deficit->last == NULL){
        header_deficit->first = dummy_route;
        header_deficit->last = dummy_route;
        (header_deficit->count)++;
      }
      else{
        dummy_route->next = header_deficit->first;   /* Add at the front */
        header_deficit->first->prev = dummy_route;
        header_deficit->first = dummy_route;
        (header_deficit->count)++;
      }
      Value -= 2;

  }
  if(Value == 1) {
      dummy_route = MALLOC(Unbalanced);
      dummy_route->prev = dummy_route->next = NULL;
```

```
            dummy_route->route_num = 0;
            dummy_route->trailer = 1;

            if(header_deficit->last == NULL){
              header_deficit->first = dummy_route;
              header_deficit->last = dummy_route;
              (header_deficit->count)++;
            }
            else{
              dummy_route->prev = header_deficit->last;  /* Add at the end */
              header_deficit->last->next = dummy_route;
              header_deficit->last = dummy_route;
              (header_deficit->count)++;
            }
        }
      }

      tmp_surplus = header_surplus->first;
      tmp_deficit = header_deficit->first;

      if(header_surplus->count == header_deficit->count) {
        while(tmp_surplus != NULL) {
            k++;
            fprintf(fpo_2,"%3d.   Sroute[%d] + Droute[%d]\n",k,
                    tmp_surplus->route_num,tmp_deficit->route_num);
            tmp_surplus = tmp_surplus->next;
            tmp_deficit = tmp_deficit->next;
        }
      }
      else if(header_surplus->count < header_deficit->count) {
        while(tmp_surplus != NULL) {
            k++;
            if(tmp_surplus->trailer == tmp_deficit->trailer){
              fprintf(fpo_2,"%3d.   Sroute[%d] + Droute[%d]\n",k,
                      tmp_surplus->route_num,tmp_deficit->route_num);
              tmp_surplus = tmp_surplus->next;
              tmp_deficit = tmp_deficit->next;
            }
            else if(tmp_surplus->trailer > tmp_deficit->trailer) {
              fprintf(fpo_2,"%3d.   Sroute[%d] + Droute[%d] + Droute[%d]\n",k,
                      tmp_surplus->route_num,tmp_deficit->route_num,tmp_deficit->next->route_num);
              tmp_surplus = tmp_surplus->next;
              tmp_deficit = tmp_deficit->next->next;
            }
            else
              printf("\nError Has Occurred in Sorting EMPTY TRAILER Queueing(SURPLUS
NETWORK)");
        }
      }
      else {          /* if(header_surplus->count > header_deficit->count) */
        while(tmp_surplus != NULL){
            k++;
            if(tmp_surplus->trailer == tmp_deficit->trailer){
```

```
        fprintf(fpo_2,"%3d.  Sroute[%d] + Droute[%d]\n",k,
                tmp_surplus->route_num,tmp_deficit->route_num);
        tmp_surplus = tmp_surplus->next;
        tmp_deficit = tmp_deficit->next;
       }
       else if(tmp_surplus->trailer < tmp_deficit->trailer) {
        fprintf(fpo_2,"%3d.  Sroute[%d] + Sroute[%d] + Droute[%d]\n",k,
                tmp_surplus->route_num,tmp_surplus->next->route_num,tmp_deficit->route_num);
        tmp_surplus = tmp_surplus->next->next;
        tmp_deficit = tmp_deficit->next;
       }
       else
        printf("\nError Occurred in Sorting EMPTY TRAILER Queueing(SURPLUS NETWORK)");
   }
  }
 }

/* DEFICIT NETWORK */

else {
 Value = -(network_empty);
 fprintf(fpo_2,"*  Deficit Network: %d empties must be reserved at CC.\n\n",Value);
 if(Value <= 2) {
  dummy_route = MALLOC(Unbalanced);
  if(Value == 1) {
       if (dummy_route) {
        dummy_route->prev = dummy_route->next = NULL;
        dummy_route->route_num = 0;
        dummy_route->trailer = 1;

        if(header_surplus->last == NULL) {
               header_surplus->first = dummy_route;
         header_surplus->last = dummy_route;
         (header_surplus->count)++;
        }
        else {
         dummy_route->prev = header_surplus->last;    /* Add at the end */
         header_surplus->last->next = dummy_route;
         header_surplus->last = dummy_route;
         (header_surplus->count)++;
        }
       }
       else {
        printf("\nLack of Memory in Empty Trailer Balancing Step 2_New Route");
        exit(1);
       }
  }
  if(Value == 2) {
       if(dummy_route) {
        dummy_route->prev = dummy_route->next = NULL;
        dummy_route->route_num = 0;
        dummy_route->trailer = 2;
```

```
          if(header_surplus->last == NULL) {
            header_surplus->first = dummy_route;
            header_surplus->last = dummy_route;
            (header_surplus->count)++;
          }
          else {
            dummy_route->next = header_surplus->first;  /* Add at the front */
            header_surplus->first->prev = dummy_route;
            header_surplus->first = dummy_route;
            (header_surplus->count)++;
          }
        }
        else{
          printf("\nLack  of Memory in Empty Trailer Balancing Step 2_New Route");
          exit(1);
        }
    }
  }
  else {                                                        /* If (Value > 2) */
    while(Value > 1){
        dummy_route = MALLOC(Unbalanced);
        if(dummy_route) {
          dummy_route->prev = dummy_route->next = NULL;
          dummy_route->route_num = 0;
          dummy_route->trailer = 2;

          if(header_surplus->last == NULL){
            header_surplus->first = dummy_route;
            header_surplus->last = dummy_route;
            (header_surplus->count)++;
          }
          else {
            dummy_route->next = header_surplus->first;    /* Add at the front */
            header_surplus->first->prev = dummy_route;
            header_surplus->first = dummy_route;
            (header_surplus->count)++;
          }
          Value -= 2;

        }
        else {
          printf("\nERROR. Lack of Memory.");
          exit(1);
        }
    }
    if(Value == 1) {
        dummy_route = MALLOC(Unbalanced);
        if(dummy_route){
          dummy_route->prev = dummy_route->next = NULL;
          dummy_route->route_num = 0;
          dummy_route->trailer = 1;

          if(header_surplus->last == NULL){
```

```
            header_surplus->first = dummy_route;
            header_surplus->last = dummy_route;
            (header_surplus->count)++;
          }
          else{
            dummy_route->prev = header_surplus->last;     /* Add at the end */
            header_surplus->last->next = dummy_route;
            header_surplus->last = dummy_route;
            (header_surplus->count)++;
          }
        }
        else{
          printf("\nERROR. Lack of Memory.");
          exit(1);
        }
    }
  }

  tmp_surplus = header_surplus->first;
  tmp_deficit = header_deficit->first;

  if(header_surplus->count == header_deficit->count) {
    while(tmp_surplus != NULL) {
        k++;
        fprintf(fpo_2,"%3d.   Sroute[%d] + Droute[%d]\n",k,
                tmp_surplus->route_num,tmp_deficit->route_num);
        tmp_surplus = tmp_surplus->next;
        tmp_deficit = tmp_deficit->next;
    }
  }
  else if(header_surplus->count < header_deficit->count) {
    while(tmp_surplus != NULL) {
        k++;
        if(tmp_surplus->trailer == tmp_deficit->trailer){
          fprintf(fpo_2,"%3d.   Sroute[%d] + Droute[%d]\n",k,
                tmp_surplus->route_num,tmp_deficit->route_num);
          tmp_surplus = tmp_surplus->next;
          tmp_deficit = tmp_deficit->next;
        }
        else if(tmp_surplus->trailer > tmp_deficit->trailer) {
          fprintf(fpo_2,"%3d.   Sroute[%d] + Droute[%d] + Droute[%d]\n",k,
                tmp_surplus->route_num,tmp_deficit->route_num,tmp_deficit->next->route_num);
          tmp_surplus = tmp_surplus->next;
          tmp_deficit = tmp_deficit->next->next;
        }
        else
            printf("\nError   Has   Occurred   in   Sorting   EMPTY   TRAILER   Queueing(DEFICIT
NETWORK)");
    }
  }
  else {          /* if(header_surplus->count > header_deficit->count) */
    while(tmp_surplus != NULL){
        k++;
```

```
        if(tmp_surplus->trailer == tmp_deficit->trailer){
          fprintf(fpo_2,"%3d.  Sroute[%d] + Droute[%d]\n",k,
                  tmp_surplus->route_num,tmp_deficit->route_num);
          tmp_surplus = tmp_surplus->next;
          tmp_deficit = tmp_deficit->next;
        }
        else if(tmp_surplus->trailer < tmp_deficit->trailer) {
          fprintf(fpo_2,"%3d.  Sroute[%d] + Sroute[%d] + Droute[%d]\n",k,
                  tmp_surplus->route_num,tmp_surplus->next->route_num,tmp_deficit->route_num);
          tmp_surplus = tmp_surplus->next->next;
          tmp_deficit = tmp_deficit->next;
        }
        else
          printf("\nError Occurred in Sorting EMPTY TRAILER Queueing(DEFICIT NETWORK)");
    }
   }
  }
}
```

```
/*****************************************************************************************
 REMATCHING NETWORK CREATION I; RETEST4.IN
*****************************************************************************************/
void rematching_network(int next_max_node,int size,int *Mate,Terminal *temp,FILE *fpo)
{
 FILE  *fpo_1,*fpo_2;
 int   i,j,k;
 int   ccost;
 int   rematching_cost(int,int, Terminal *, FILE *, int);
 int   *next_cost;
 static int step = 0;

 step++;

 /* starting of rematching network creation */
 if(step == 1) {
  fpo_1 = fopen("retest.in", "w");
  if(fpo_2 == NULL){
   printf("\nERROR. Can not write file : retest.in. No Memory is left.");
   exit(1);
  }
  fpo_2 = fopen("recost.mat", "w");
  if(fpo_2 == NULL){
   printf("\nERROR. Can not write file : recost.mat. No Memory is left.");
   exit(1);
  }
 }
 else if(step > 1) {
  fpo_1 = fopen("retest_2.in", "w");
  if(fpo_2 == NULL){
   printf("\nERROR. Can not write file : retest_2.in. No Memory is left.");
   exit(1);
  }
  fpo_2 = fopen("recost_2.mat", "w");
```

```
  if(fpo_2 == NULL){
    printf("\nERROR. Can not write file : recost_2.mat. No Memory is left.");
    exit(1);
  }
 }

 match_info = CALLOC(Info, next_max_node+1);
 for(i=0; i <= next_max_node; i++) {
  match_info[i].num   = 0;
  match_info[i].i     = 0;
  match_info[i].mate  = 0;
  match_info[i].pickup = 0;
  match_info[i].delivery = 0;
 }

 if(!match_info) {
  printf("\nERROR. Out of Memory in match_info");
  exit(1);
 }
 fprintf(fpo_1, "%5d      2\n", next_max_node);
 k=0;

 for (i=1; i <= size; i++) {
  ccost = cost[temp[i].index * column + temp[Mate[i]].index];
  if (((temp[i].pickup == 0)&&(temp[i].delivery == 0))
       &&((temp[Mate[i]].pickup == 0)&&(temp[Mate[i]].delivery == 0)))
   ccost = 0;
  if (ccost != 0) {
   if (i < Mate[i]) {
       k++;
       match_info[k].num  = k;
       match_info[k].i    = i;
       if (Mate[i] > max_split) Mate[i] = 0;  /* Breakbulk representation */
       match_info[k].mate = Mate[i];
       match_info[k].pickup = temp[i].pickup+temp[Mate[i]].pickup;
       match_info[k].delivery = temp[i].delivery+temp[Mate[i]].delivery;
   }
  }
 }
for (i=1; i <= next_max_node; i++)
  fprintf(fpo_1,"%5d %8d %8d\n", match_info[i].num, match_info[i].pickup,
         match_info[i].delivery);

/* test */
fprintf(fpo,"\n\f");
fprintf(fpo,"**** Rematching Network ****");
fprintf(fpo,"\n\n===========================================================");
fprintf(fpo,"\n  NUM  [I]  MATE  PICK   DEL I.PIC I.DEL M.PIC M.DEL\n");
for (i=1; i <= next_max_node; i++)
 fprintf(fpo,"%5d %5d %5d %5d %5d %5d %5d %5d %5d\n", match_info[i].num,
        match_info[i].i, match_info[i].mate, match_info[i].pickup,
        match_info[i].delivery,temp[match_info[i].i].pickup, temp[match_info[i].i].delivery,
        temp[match_info[i].mate].pickup, temp[match_info[i].mate].delivery);
```

```
fprintf(fpo,"\n\n==========================================================");

route_info = CALLOC(Route_Info, (next_max_node+1)*(next_max_node+1));
if(!route_info){
 printf("\nERROR. No memory is left.");
 exit(1);
}
next_cost = CALLOC(int, (next_max_node+1)*(next_max_node+1));
if(!next_cost){
 printf("\nERROR. No memory is left.");
 exit(1);
}

for (i=0; i <= next_max_node; i++){
 for(j=0; j <= next_max_node; j++){
  if(i <= j){
      next_cost[i*(next_max_node+1)+j] = rematching_cost(i,j,temp,fpo,next_max_node);
      next_cost[j*(next_max_node+1)+i] = next_cost[i*(next_max_node+1)+j];
      route_info[j*(next_max_node+1)+i] = route_info[i*(next_max_node+1)+j];
  }
 }
}
for (i=0; i <= next_max_node; i++) {
 for(j=0; j <= next_max_node; j++){
  fprintf(fpo_2,"%6d",next_cost[i*(next_max_node+1)+j]);
 }
 fprintf(fpo_2,"\n");
}

route_col = next_max_node + 1;
rewind(fpo_1);
rewind(fpo_2);
}

/******************************************************************************
 REMATCHING NETWORK COST II; RECOST4.MAT
 ******************************************************************************/
int rematching_cost(int i,int j,Terminal *temp, FILE *fpo, int next_max_node)
{
 int   k;
 int   min_fcost, dum_fcost[9];
 int   result;
 int   feasibility_check(int, int, int, int, int, int, Terminal *);


 UPPER++;
 min_fcost  = dum_fcost[0] = 5 * UPPER;

 if (i == j) {
  min_fcost = 0;
  return(min_fcost);
```

```
}

/* 1 */
result = feasibility_check(i,j,match_info[i].i, match_info[i].mate,
                           match_info[j].i, match_info[j].mate, temp);
if (result == YES){
  dum_fcost[1] =          fcost[temp[match_info[i].i].index]
    + fcost[temp[match_info[i].i].index * column+temp[match_info[i].mate].index]
        + fcost[temp[match_info[i].mate].index * column+temp[match_info[j].i].index]
         + fcost[temp[match_info[j].i].index * column+temp[match_info[j].mate].index]
          + fcost[temp[match_info[j].mate].index * column];
  if(dum_fcost[1] < min_fcost)
    min_fcost = dum_fcost[1];
}
/* 2 */
result = feasibility_check(i,j,match_info[i].i, match_info[i].mate,
                           match_info[j].mate, match_info[j].i, temp);
if(result == YES) {
  dum_fcost[2] =          fcost[temp[match_info[i].i].index]
    + fcost[temp[match_info[i].i].index * column+temp[match_info[i].mate].index]
        + fcost[temp[match_info[i].mate].index * column+temp[match_info[j].mate].index]
         + fcost[temp[match_info[j].mate].index * column+temp[match_info[j].i].index]
          + fcost[temp[match_info[j].i].index * column];
  if(dum_fcost[2] < min_fcost)
    min_fcost = dum_fcost[2];
}
/* 3 */
result = feasibility_check(i,j,match_info[i].mate, match_info[i].i,
                           match_info[j].i, match_info[j].mate, temp);
if(result == YES) {
  dum_fcost[3] =          fcost[temp[match_info[i].mate].index]
    + fcost[temp[match_info[i].mate].index * column+temp[match_info[i].i].index]
        + fcost[temp[match_info[i].i].index * column+temp[match_info[j].i].index]
         + fcost[temp[match_info[j].i].index * column+temp[match_info[j].mate].index]
          + fcost[temp[match_info[j].mate].index * column];
  if(dum_fcost[3] < min_fcost)
    min_fcost = dum_fcost[3];
}
/* 4 */
result = feasibility_check(i,j,match_info[i].mate, match_info[i].i,
                           match_info[j].mate, match_info[j].i, temp);
if(result == YES) {
  dum_fcost[4] =          fcost[temp[match_info[i].mate].index]
    + fcost[temp[match_info[i].mate].index * column+temp[match_info[i].i].index]
        + fcost[temp[match_info[i].i].index * column+temp[match_info[j].mate].index]
         + fcost[temp[match_info[j].mate].index * column+temp[match_info[j].i].index]
          + fcost[temp[match_info[j].i].index * column];
  if(dum_fcost[4] < min_fcost)
    min_fcost = dum_fcost[4];
}
/* 5 */
result = feasibility_check(i,j,match_info[j].mate, match_info[j].i,
                           match_info[i].mate, match_info[i].i, temp);
```

```
if (result == YES){
  dum_fcost[5] =          fcost[temp[match_info[j].mate].index]
   + fcost[temp[match_info[j].mate].index * column+temp[match_info[j].i].index]
       + fcost[temp[match_info[j].i].index * column+temp[match_info[i].mate].index]
        + fcost[temp[match_info[i].mate].index * column+temp[match_info[i].i].index]
         + fcost[temp[match_info[i].i].index * column];
  if(dum_fcost[5] < min_fcost)
   min_fcost = dum_fcost[5];
}
/* 6 */
result = feasibility_check(i,j,match_info[j].i, match_info[j].mate,
                          match_info[i].mate, match_info[i].i, temp);
if(result == YES) {
  dum_fcost[6] =          fcost[temp[match_info[j].i].index]
   + fcost[temp[match_info[j].i].index * column+temp[match_info[j].mate].index]
       + fcost[temp[match_info[j].mate].index * column+temp[match_info[i].mate].index]
        + fcost[temp[match_info[i].mate].index * column+temp[match_info[i].i].index]
         + fcost[temp[match_info[i].i].index * column];
  if(dum_fcost[6] < min_fcost)
   min_fcost = dum_fcost[6];
}
/* 7 */
result = feasibility_check(i,j,match_info[j].mate, match_info[j].i,
                          match_info[i].i, match_info[i].mate, temp);
if(result == YES) {
  dum_fcost[7] =          fcost[temp[match_info[j].mate].index]
   + fcost[temp[match_info[j].mate].index * column+temp[match_info[j].i].index]
       + fcost[temp[match_info[j].i].index * column+temp[match_info[i].i].index]
        + fcost[temp[match_info[i].i].index * column+temp[match_info[i].mate].index]
         + fcost[temp[match_info[i].mate].index * column];
  if(dum_fcost[7] < min_fcost)
   min_fcost = dum_fcost[7];
}
/* 8 */
result = feasibility_check(i,j,match_info[j].i, match_info[j].mate,
                          match_info[i].i, match_info[i].mate, temp);
if(result == YES) {
  dum_fcost[8] =          fcost[temp[match_info[j].i].index]
   + fcost[temp[match_info[j].i].index * column+temp[match_info[j].mate].index]
       + fcost[temp[match_info[j].mate].index * column+temp[match_info[i].i].index]
        + fcost[temp[match_info[i].i].index * column+temp[match_info[i].mate].index]
         + fcost[temp[match_info[i].mate].index * column];
  if(dum_fcost[8] < min_fcost)
   min_fcost = dum_fcost[8];
}

/* Result PrintOut */

if(min_fcost != dum_fcost[0]) {
  fprintf(fpo,"\n");
  fprintf(fpo,"* Routing (from %3d to %3d) : ", i,j);
  if(min_fcost == dum_fcost[1]) {
```

```
    route_info[i*(next_max_node+1)+j].strgy = 1;

    route_info[i*(next_max_node+1)+j].from = temp[match_info[i].i].index;
    route_info[i*(next_max_node+1)+j].from_pk = temp[match_info[i].i].pickup;
    route_info[i*(next_max_node+1)+j].from_dl = temp[match_info[i].i].delivery;

    route_info[i*(next_max_node+1)+j].mate_from = temp[match_info[i].mate].index;
    route_info[i*(next_max_node+1)+j].mate_from_pk = temp[match_info[i].mate].pickup;
    route_info[i*(next_max_node+1)+j].mate_from_dl = temp[match_info[i].mate].delivery;

    route_info[i*(next_max_node+1)+j].to = temp[match_info[j].i].index;
    route_info[i*(next_max_node+1)+j].to_pk = temp[match_info[j].i].pickup;
    route_info[i*(next_max_node+1)+j].to_dl = temp[match_info[j].i].delivery;

    route_info[i*(next_max_node+1)+j].mate_to = temp[match_info[j].mate].index;
    route_info[i*(next_max_node+1)+j].mate_to_pk = temp[match_info[j].mate].pickup;
    route_info[i*(next_max_node+1)+j].mate_to_dl = temp[match_info[j].mate].delivery;

    route_info[i*(next_max_node+1)+j].min_cost = min_fcost;

    fprintf(fpo," CC -> %3d -> %3d -> %3d -> %3d -> CC",
            temp[match_info[i].i].index, temp[match_info[i].mate].index,
            temp[match_info[j].i].index, temp[match_info[j].mate].index);
  }
  else if(min_fcost == dum_fcost[2]) {

    route_info[i*(next_max_node+1)+j].strgy = 2;

    route_info[i*(next_max_node+1)+j].from = temp[match_info[i].i].index;
    route_info[i*(next_max_node+1)+j].from_pk = temp[match_info[i].i].pickup;
    route_info[i*(next_max_node+1)+j].from_dl = temp[match_info[i].i].delivery;

    route_info[i*(next_max_node+1)+j].mate_from = temp[match_info[i].mate].index;
    route_info[i*(next_max_node+1)+j].mate_from_pk = temp[match_info[i].mate].pickup;
    route_info[i*(next_max_node+1)+j].mate_from_dl = temp[match_info[i].mate].delivery;

    route_info[i*(next_max_node+1)+j].to = temp[match_info[j].mate].index;
    route_info[i*(next_max_node+1)+j].to_pk = temp[match_info[j].mate].pickup;
    route_info[i*(next_max_node+1)+j].to_dl = temp[match_info[j].mate].delivery;

    route_info[i*(next_max_node+1)+j].mate_to = temp[match_info[j].i].index;
    route_info[i*(next_max_node+1)+j].mate_to_pk = temp[match_info[j].i].pickup;
    route_info[i*(next_max_node+1)+j].mate_to_dl = temp[match_info[j].i].delivery;

    route_info[i*(next_max_node+1)+j].min_cost = min_fcost;

    fprintf(fpo," CC -> %3d -> %3d -> %3d -> %3d -> CC",
            temp[match_info[i].i].index, temp[match_info[i].mate].index,
            temp[match_info[j].mate].index, temp[match_info[j].i].index);
  }
  else if(min_fcost == dum_fcost[3]){

    route_info[i*(next_max_node+1)+j].strgy = 3;
```

```
  route_info[i*(next_max_node+1)+j].from = temp[match_info[i].mate].index;
  route_info[i*(next_max_node+1)+j].from_pk = temp[match_info[i].mate].pickup;
  route_info[i*(next_max_node+1)+j].from_dl = temp[match_info[i].mate].delivery;

  route_info[i*(next_max_node+1)+j].mate_from = temp[match_info[i].i].index;
  route_info[i*(next_max_node+1)+j].mate_from_pk = temp[match_info[i].i].pickup;
  route_info[i*(next_max_node+1)+j].mate_from_dl = temp[match_info[i].i].delivery;

  route_info[i*(next_max_node+1)+j].to = temp[match_info[j].i].index;
  route_info[i*(next_max_node+1)+j].to_pk = temp[match_info[j].i].pickup;
  route_info[i*(next_max_node+1)+j].to_dl = temp[match_info[j].i].delivery;

  route_info[i*(next_max_node+1)+j].mate_to = temp[match_info[j].mate].index;
  route_info[i*(next_max_node+1)+j].mate_to_pk = temp[match_info[j].mate].pickup;
  route_info[i*(next_max_node+1)+j].mate_to_dl = temp[match_info[j].mate].delivery;

  route_info[i*(next_max_node+1)+j].min_cost = min_fcost;

  fprintf(fpo," CC -> %3d -> %3d -> %3d -> %3d -> CC",
          temp[match_info[i].mate].index, temp[match_info[i].i].index,
          temp[match_info[j].i].index, temp[match_info[j].mate].index);
}
else if(min_fcost == dum_fcost[4]){

  route_info[i*(next_max_node+1)+j].strgy = 4;

  route_info[i*(next_max_node+1)+j].from = temp[match_info[i].mate].index;
  route_info[i*(next_max_node+1)+j].from_pk = temp[match_info[i].mate].pickup;
  route_info[i*(next_max_node+1)+j].from_dl = temp[match_info[i].mate].delivery;

  route_info[i*(next_max_node+1)+j].mate_from = temp[match_info[i].i].index;
  route_info[i*(next_max_node+1)+j].mate_from_pk = temp[match_info[i].i].pickup;
  route_info[i*(next_max_node+1)+j].mate_from_dl = temp[match_info[i].i].delivery;

  route_info[i*(next_max_node+1)+j].to = temp[match_info[j].mate].index;
  route_info[i*(next_max_node+1)+j].to_pk = temp[match_info[j].mate].pickup;
  route_info[i*(next_max_node+1)+j].to_dl = temp[match_info[j].mate].delivery;

  route_info[i*(next_max_node+1)+j].mate_to = temp[match_info[j].i].index;
  route_info[i*(next_max_node+1)+j].mate_to_pk = temp[match_info[j].i].pickup;
  route_info[i*(next_max_node+1)+j].mate_to_dl = temp[match_info[j].i].delivery;

  route_info[i*(next_max_node+1)+j].min_cost = min_fcost;

  fprintf(fpo," CC -> %3d -> %3d -> %3d -> %3d -> CC",
          temp[match_info[i].mate].index, temp[match_info[i].i].index,
          temp[match_info[j].mate].index, temp[match_info[j].i].index);
}
else if(min_fcost == dum_fcost[5]){

  route_info[i*(next_max_node+1)+j].strgy = 5;
```

```
    route_info[i*(next_max_node+1)+j].from = temp[match_info[j].mate].index;
    route_info[i*(next_max_node+1)+j].from_pk = temp[match_info[j].mate].pickup;
    route_info[i*(next_max_node+1)+j].from_dl = temp[match_info[j].mate].delivery;

    route_info[i*(next_max_node+1)+j].mate_from = temp[match_info[j].i].index;
    route_info[i*(next_max_node+1)+j].mate_from_pk = temp[match_info[j].i].pickup;
    route_info[i*(next_max_node+1)+j].mate_from_dl = temp[match_info[j].i].delivery;

    route_info[i*(next_max_node+1)+j].to = temp[match_info[i].mate].index;
    route_info[i*(next_max_node+1)+j].to_pk = temp[match_info[i].mate].pickup;
    route_info[i*(next_max_node+1)+j].to_dl = temp[match_info[i].mate].delivery;

    route_info[i*(next_max_node+1)+j].mate_to = temp[match_info[i].i].index;
    route_info[i*(next_max_node+1)+j].mate_to_pk = temp[match_info[i].i].pickup;
    route_info[i*(next_max_node+1)+j].mate_to_dl = temp[match_info[i].i].delivery;

    route_info[i*(next_max_node+1)+j].min_cost = min_fcost;

    fprintf(fpo," CC -> %3d -> %3d -> %3d -> %3d -> CC",
            temp[match_info[j].mate].index, temp[match_info[j].i].index,
            temp[match_info[i].mate].index, temp[match_info[i].i].index);
  }
  else if(min_fcost == dum_fcost[6]){

    route_info[i*(next_max_node+1)+j].strgy = 6;

    route_info[i*(next_max_node+1)+j].from = temp[match_info[j].i].index;
    route_info[i*(next_max_node+1)+j].from_pk = temp[match_info[j].i].pickup;
    route_info[i*(next_max_node+1)+j].from_dl = temp[match_info[j].i].delivery;

    route_info[i*(next_max_node+1)+j].mate_from = temp[match_info[j].mate].index;
    route_info[i*(next_max_node+1)+j].mate_from_pk = temp[match_info[j].mate].pickup;
    route_info[i*(next_max_node+1)+j].mate_from_dl = temp[match_info[j].mate].delivery;

    route_info[i*(next_max_node+1)+j].to = temp[match_info[i].mate].index;
    route_info[i*(next_max_node+1)+j].to_pk = temp[match_info[i].mate].pickup;
    route_info[i*(next_max_node+1)+j].to_dl = temp[match_info[i].mate].delivery;

    route_info[i*(next_max_node+1)+j].mate_to = temp[match_info[i].i].index;
    route_info[i*(next_max_node+1)+j].mate_to_pk = temp[match_info[i].i].pickup;
    route_info[i*(next_max_node+1)+j].mate_to_dl = temp[match_info[i].i].delivery;

    route_info[i*(next_max_node+1)+j].min_cost = min_fcost;

    fprintf(fpo," CC -> %3d -> %3d -> %3d -> %3d -> CC",
            temp[match_info[j].i].index, temp[match_info[j].mate].index,
            temp[match_info[i].mate].index, temp[match_info[i].i].index);
  }
  else if(min_fcost == dum_fcost[7]){

    route_info[i*(next_max_node+1)+j].strgy = 7;

    route_info[i*(next_max_node+1)+j].from = temp[match_info[j].mate].index;
```

```
    route_info[i*(next_max_node+1)+j].from_pk = temp[match_info[j].mate].pickup;
    route_info[i*(next_max_node+1)+j].from_dl = temp[match_info[j].mate].delivery;

    route_info[i*(next_max_node+1)+j].mate_from = temp[match_info[j].i].index;
    route_info[i*(next_max_node+1)+j].mate_from_pk = temp[match_info[j].i].pickup;
    route_info[i*(next_max_node+1)+j].mate_from_dl = temp[match_info[j].i].delivery;

    route_info[i*(next_max_node+1)+j].to = temp[match_info[i].i].index;
    route_info[i*(next_max_node+1)+j].to_pk = temp[match_info[i].i].pickup;
    route_info[i*(next_max_node+1)+j].to_dl = temp[match_info[i].i].delivery;

    route_info[i*(next_max_node+1)+j].mate_to = temp[match_info[i].mate].index;
    route_info[i*(next_max_node+1)+j].mate_to_pk = temp[match_info[i].mate].pickup;
    route_info[i*(next_max_node+1)+j].mate_to_dl = temp[match_info[i].mate].delivery;

    route_info[i*(next_max_node+1)+j].min_cost = min_fcost;

    fprintf(fpo," CC -> %3d -> %3d -> %3d -> %3d -> CC",
            temp[match_info[j].mate].index, temp[match_info[j].i].index,
            temp[match_info[i].i].index, temp[match_info[i].mate].index);
   }
   else if(min_fcost == dum_fcost[8]){

    route_info[i*(next_max_node+1)+j].strgy = 8;

    route_info[i*(next_max_node+1)+j].from = temp[match_info[j].i].index;
    route_info[i*(next_max_node+1)+j].from_pk = temp[match_info[j].i].pickup;
    route_info[i*(next_max_node+1)+j].from_dl = temp[match_info[j].i].delivery;

    route_info[i*(next_max_node+1)+j].mate_from = temp[match_info[j].mate].index;
    route_info[i*(next_max_node+1)+j].mate_from_pk = temp[match_info[j].mate].pickup;
    route_info[i*(next_max_node+1)+j].mate_from_dl = temp[match_info[j].mate].delivery;

    route_info[i*(next_max_node+1)+j].to = temp[match_info[i].i].index;
    route_info[i*(next_max_node+1)+j].to_pk = temp[match_info[i].i].pickup;
    route_info[i*(next_max_node+1)+j].to_dl = temp[match_info[i].i].delivery;

    route_info[i*(next_max_node+1)+j].mate_to = temp[match_info[i].mate].index;
    route_info[i*(next_max_node+1)+j].mate_to_pk = temp[match_info[i].mate].pickup;
    route_info[i*(next_max_node+1)+j].mate_to_dl = temp[match_info[i].mate].delivery;

    route_info[i*(next_max_node+1)+j].min_cost = min_fcost;

    fprintf(fpo," CC -> %3d -> %3d -> %3d -> %3d -> CC",
            temp[match_info[j].i].index, temp[match_info[j].mate].index,
            temp[match_info[i].i].index, temp[match_info[i].mate].index);
   }
   fprintf(fpo," (%8d)", min_fcost);
  }
  else {
   route_info[i*(next_max_node+1)+j].strgy = 0;
  }
  return(min_fcost);
```

```
}

/*****************************************************************************
 Route Legality Check
        example;
                call feasibility_check(i,j,match_info[i].i, match_info[i].mate,
                                       match_info[j].i, match_info[j].mate, temp);
*****************************************************************************/
int feasibility_check(int i,int j,int one,int two,int three,int four,Terminal *temp)
{
 int     pickup[2], delivery[2];
 int     l, m, k;
 int     nonzero;
 int  feasible;

 nonzero = 0;
 feasible = YES;
 if(i == j) {
  feasible = NO;
  return(feasible);
 }
 if(temp[one].delivery+temp[two].delivery+temp[three].delivery+temp[four].delivery > 2){
  feasible = NO;
  return(feasible);
 }
 if(temp[one].pickup+temp[two].pickup+temp[three].pickup+temp[four].pickup > 2){
  feasible = NO;
  return(feasible);
 }
 /* pickup[] and delivery[] initialization */
 pickup[0] = pickup[1] = 0;
 delivery[0] = delivery[1] = 0;
 if(match_info[i].delivery + match_info[j].delivery == 2)
  delivery[0] = delivery[1] = 1;
 else if(match_info[i].delivery + match_info[j].delivery == 1)
  delivery[0] = 1;
 /* end of initialization */

 for(k=0; k < 2; k++) {
  if(pickup[k] != 0)   nonzero++;
  if(delivery[k] != 0) nonzero++;
 }

 if(temp[one].pickup == 0) {
  if(temp[one].delivery == 1) nonzero--;
  else if(temp[one].delivery == 2) nonzero -= 2;
 }
 else if(temp[one].pickup == 1) {
  nonzero++;
  if(temp[one].delivery == 1) nonzero--;
  else if(temp[one].delivery == 2) nonzero -= 2;
 }
 else if(temp[one].pickup == 2) {
```

```
  nonzero += 2;
  if(temp[one].delivery == 1) nonzero--;
  else if(temp[one].delivery == 2) nonzero -= 2;
 }

 if(nonzero > 2) {
  feasible = NO;
  return(feasible);
 }

 if(temp[two].pickup == 0) {
  if(temp[two].delivery == 1) nonzero--;
  else if(temp[two].delivery == 2) nonzero -= 2;
 }
 else if(temp[two].pickup == 1) {
  nonzero++;
  if(temp[two].delivery == 1) nonzero--;
  else if(temp[two].delivery == 2) nonzero -= 2;
 }
 else if(temp[two].pickup == 2) {
  nonzero += 2;
  if(temp[two].delivery == 1) nonzero--;
  else if(temp[two].delivery == 2) nonzero -= 2;
 }

 if(nonzero > 2) {
  feasible = NO;
  return(feasible);
 }

 if(temp[three].pickup == 0) {
  if(temp[three].delivery == 1) nonzero--;
  else if(temp[three].delivery == 2) nonzero -= 2;
 }
 else if(temp[three].pickup == 1) {
  nonzero++;
  if(temp[three].delivery == 1) nonzero--;
  else if(temp[three].delivery == 2) nonzero -= 2;
 }
 else if(temp[three].pickup == 2) {
  nonzero += 2;
  if(temp[three].delivery == 1) nonzero--;
  else if(temp[three].delivery == 2) nonzero -= 2;
 }
 if(nonzero > 2) {
  feasible = NO;
  return(feasible);
 }

 if(temp[four].pickup == 0) {
  if(temp[four].delivery == 1) nonzero--;
  else if(temp[four].delivery == 2) nonzero -= 2;
 }
```

```
 else if(temp[four].pickup == 1) {
  nonzero++;
  if(temp[four].delivery == 1) nonzero--;
  else if(temp[four].delivery == 2) nonzero -= 2;
 }
 else if(temp[four].pickup == 2) {
  nonzero += 2;
  if(temp[four].delivery == 1) nonzero--;
  else if(temp[four].delivery == 2) nonzero -= 2;
 }
 if(nonzero > 2) {
  feasible = NO;
  return(feasible);
 }

 return(feasible);
}

/*****************************************************************************************
 PREPROCESSING FUNCTION :  1st node split processing
*****************************************************************************************/
Terminal *preprocessing(char *filename_1)
{
 FILE     *fpi_1;
 int      MAX_SD;
 int         i,j;
 int      node, pickup, delivery, dummy;
 Terminal *temp;

 fpi_1 = fopen(filename_1, "r");
 if(!fpi_1) {
  printf("\nERROR. I can't find the filename of %s", filename_1);
  exit(1);
 }

 fscanf(fpi_1, "%d %d", &max_node, &MAX_SD);

 temp = CALLOC(Terminal, (2*(max_node)*(MAX_SD/2+1)+1));
 if (!temp) {
  printf("\nERROR. Out of Memory in temp.");
  exit(1);
 }


 max_split   = max_node+1;
 dum_tractor = CALLOC(int, max_node+1);

 if (!dum_tractor) {
  printf("\nERROR. Out of Memory in dum_tractor.");
  exit(1);
 }

 for (i=0; (fgetc(fpi_1) != EOF)&& (i < max_node); i++){
```

{

```
fscanf(fpi_1, "%d %d %d", &node, &pickup, &delivery);
temp[node].num = node;
temp[node].index = node;
temp[node].pickup = pickup;
temp[node].delivery = delivery;
if((temp[node].pickup >= 2) &&(temp[node].delivery >= 2)){
  while ((temp[node].pickup >= 2)&&(temp[node].delivery >= 2)) {
        temp[node].pickup -= 2;
        temp[node].delivery -= 2;
        dum_tractor[node] ++;
  }
}
/* node split */
if(temp[node].pickup > 2){
  dummy = temp[node].pickup;
  temp[node].pickup = 2;
  dummy -= 2;
  for ( ; dummy > 0 ; max_split++) {
        if(dummy >= 2) {
          temp[max_split].num    = max_split;
          temp[max_split].index  = node;
          temp[max_split].pickup = 2;
          temp[max_split].delivery = 0;
        }
        else {
          temp[max_split].num    = max_split;
          temp[max_split].index  = node;
          temp[max_split].pickup = dummy;
          temp[max_split].delivery = 0;
        }
        dummy -= 2;
  }
}

if(temp[node].delivery > 2){
  dummy = temp[node].delivery;
  temp[node].delivery = 2;
  dummy -= 2;
  for ( ; dummy > 0 ; max_split++) {
        if(dummy >= 2) {
          temp[max_split].num    = max_split;
          temp[max_split].index  = node;
          temp[max_split].delivery = 2;
          temp[max_split].pickup = 0;
        }
        else {
          temp[max_split].num    = max_split;
          temp[max_split].index  = node;
          temp[max_split].delivery = dummy;
          temp[max_split].pickup = 0;
        }
        dummy -= 2;
  }
```

```
   }
  }
  max_split--;

  fclose(fpi_1);
  return(temp);
}

/****************************************************************************************
  MATCHING_NETWORK CREATION : 2nd dummy node creation and arc connection.
****************************************************************************************/
void matching_network_creation(Terminal *temp, char *filename_2, char *net_file)
{
  int    i,j,k;
  int    *ndegree, nedges, dum_degree;
  FILE   *fpi_2, *fpo;
  static  int  step = 0;

  step++;

  /* read cost matrix */
  fpi_2 = fopen(filename_2, "r");
  if(!fpi_2) {
    printf("\nERROR. I can't find the filename of %s", filename_2);
    exit(1);
  }

  /* Initial Matching Cost construction */

  if (step == 1) {

    UPPER  = 0;
    COLUMN = column = max_node+1;

    fcost = CALLOC(int, COLUMN*COLUMN);
    if (!fcost) {
      printf("\nERROR. Out of Memory in fcost.");
      exit(1);
    }
    for (i=0; i < COLUMN; i++)
      for (j=0; j < COLUMN; j++)
          fscanf(fpi_2, "%d", &fcost[i*COLUMN + j]);

    fclose(fpi_2);

    cost = CALLOC(int, (max_node+1) * (max_node+1));
    if (!cost) {
      printf("\nERROR. Out of Memory in cost.");
      exit(1);
    }

    for(i=0; i <= max_node; i++)
      for(j=0; j <= max_node; j++)
```

```
        cost[i*column + j] = fcost[i*COLUMN + j];


  for (i=0; i <= max_node; i++)
   for (j=0; j <= max_node; j++)
        if ((i*j != 0)&&(i!=j))
         cost[i*column + j] += (fcost[j] + fcost[i*column]);

  for (i=0; i <= max_node; i++) {
   for (j=0; j <= max_node; j++) {
        if ((i*j == 0)&&(i!=j)) {
         cost[i*column + j] = 2 * fcost[i*column + j];
        }
        if(UPPER < cost[i*column+j]) {
         UPPER = cost[i*column+j];
        }
   }
  }
 }

 /* Re-Matching Cost construction */
 else if (step > 1) {
  cost = CALLOC(int, (max_node+1) * (max_node+1));
  if (!cost) {
   printf("\nERROR. Out of Memory in cost.");
   exit(1);
  }

  UPPER  = 0;
  column = max_node+1;

  for (i=0; i <= max_node; i++)
   for (j=0; j <= max_node; j++)
        fscanf(fpi_2, "%d", &cost[i*column + j]);

  for (i=0; i <= max_node; i++) {
   for (j=0; j <= max_node; j++) {
        if(UPPER < cost[i*column+j]) {
         UPPER = cost[i*column+j];
        }
   }
  }
 }

 UPPER += 10;

 /* matching network printout */
 fpo = fopen(net_file, "w");
 if(fpo == NULL) {
  printf("\nERROR. Cann't write file : %s. No Memory is left.", net_file);
  exit(1);
 }
```

```
ndegree = CALLOC(int, (2*max_split)+1);
k = 0;
nedges = 0;

Loop:

k++;

for (i = 1; i <= max_split; i++) {
  if (k == 2) {
    fprintf(fpo, "\n%8d %8d       0       0", ndegree[i]+1, temp[i].num);
    if((temp[i].pickup == 0) && (temp[i].delivery == 0))
          fprintf(fpo, "\n%8d       0 ",i+max_split);
    else
          fprintf(fpo, "\n%8d %8d", i+max_split, cost[temp[i].index*column]);
  }
  for (j=1; j <= max_split; j++) {
    if (temp[i].pickup == 2) {
        if(temp[i].delivery == 1) {
          if ((temp[j].pickup == 0)&&(temp[j].delivery == 1)) {
            if (i!=j) {
              if (k==1) {
                  ndegree[i]++;
              }
              else if (k == 2)
                  fprintf(fpo, "\n%8d %8d", temp[j].num,
                          cost[temp[i].index * column + temp[j].index]);
            }
          }
        }
        else if(temp[i].delivery == 0) {
          if(temp[j].pickup == 0) {
            if ((temp[j].delivery == 1) || (temp[j].delivery == 2)){
              if (i!=j) {
                  if (k==1) {
                    ndegree[i]++;
                  }
                  else if (k == 2)
                    fprintf(fpo, "\n%8d %8d", temp[j].num,
                            cost[temp[i].index * column + temp[j].index]);
              }
            }
          }
        }
    }

    else if (temp[i].pickup == 1) {
        if (temp[i].delivery == 2) {
          if ((temp[j].pickup == 1)&&(temp[j].delivery == 0)) {
            if (i!=j) {
              if (k==1) {
                  ndegree[i]++;
              }
```

```
        else if (k == 2)
            fprintf(fpo, "\n%8d %8d", temp[j].num,
                    cost[temp[i].index * column + temp[j].index]);
       }
      }
     }
    else if (temp[i].delivery == 1) {
     if (temp[j].pickup == 1) {
      if (temp[j].delivery != 2) {
       if (i!=j) {
            if (k==1) {
             ndegree[i]++;
            }
            else if (k == 2)
             fprintf(fpo, "\n%8d %8d", temp[j].num,
                    cost[temp[i].index * column + temp[j].index]);
       }
      }
     }
     else if (temp[j].pickup == 0) {
      if (temp[j].delivery == 1) {
       if (i!=j) {
            if (k==1) {
             ndegree[i]++;
            }
            else if (k == 2)
             fprintf(fpo, "\n%8d %8d", temp[j].num,
                    cost[temp[i].index * column + temp[j].index]);
       }
      }
     }
    }
    else if (temp[i].delivery == 0) {
     if (temp[j].pickup == 1) {
      if (i!=j) {
       if (k==1) {
            ndegree[i]++;
       }
       else if (k == 2)
            fprintf(fpo, "\n%8d %8d", temp[j].num,
                    cost[temp[i].index * column + temp[j].index]);
      }
     }
     else if (temp[j].pickup == 0) {
      if (temp[j].delivery != 0) {
       if (i!=j) {
            if (k==1) {
             ndegree[i]++;
            }
            else if (k == 2)
             fprintf(fpo, "\n%8d %8d", temp[j].num,
                    cost[temp[i].index * column + temp[j].index]);
       }
```

```
            }
          }
        }
    }

    else if (temp[i].pickup == 0) {
          if (temp[i].delivery == 1) {
            if (temp[j].pickup == 0) {
              if (temp[j].delivery == 1) {
                if (i!=j) {
                      if (k==1) {
                        ndegree[i]++;
                      }
                      else if (k == 2)
                        fprintf(fpo, "\n%8d %8d", temp[j].num,
                                    cost[temp[i].index * column + temp[j].index]);
                }
              }
            }
            else if ((temp[j].pickup == 1) || (temp[j].pickup == 2)) {
              if (temp[j].delivery != 2) {
                if (i!=j) {
                      if (k==1) {
                        ndegree[i]++;
                      }
                      else if (k == 2)
                        fprintf(fpo, "\n%8d %8d", temp[j].num,
                                    cost[temp[i] index * column + temp[j].index]);
                }
              }
            }
          }
          else if (temp[i].delivery == 2) {
            if (temp[j].delivery == 0) {
              if ((temp[j].pickup == 1) || (temp[j].pickup == 2)) {
                if (i!=j) {
                      if (k==1) {
                        ndegree[i]++;
                      }
                      else if (k == 2)
                        fprintf(fpo, "\n%8d %8d", temp[j].num,
                                    cost[temp[i].index * column + temp[j].index]);
                }
              }
            }
          }
    }
  }
}

if ( k == 1) {
  for (j=1; j<= max_split; j++){
    nedges += ndegree[j];
```

```
    }
    nedges = nedges / 2 + (max_split * (max_split-1) / 2) + max_split;
    fprintf(fpo. "%8d %8d       U       1  %10d", 2*max_split, nedges, UPPER);
    goto Loop;
  }
  if (k == 2) {                                              /* dummy node creation */
    dum_degree = max_split;
    for (i=1; i <= max_split; i++) {
      fprintf(fpo, "\n%8d %8d     0       0", dum_degree, max_split+i);
      if((temp[i].pickup == 0) && (temp[i].delivery == 0))
          fprintf(fpo, "\n%8d      0", i);
      else
          fprintf(fpo, "\n%8d %8d", i, cost[temp[i].index]);
      for(j=1; j <= max_split; j++) {
          if (i != j)
            fprintf(fpo, "\n%8d      0", max_split+j);
      }
    }
    fprintf(fpo. "\n");
  }
  rewind(fpo);
  fclose(fpo);
}

/****************************************************************************************
  PRINT_OUT ELAPSED EXECUTION TIME
****************************************************************************************/
void elapsed_time_print(FILE *fpo,time_t first,time_t second,time_t third,time_t fourth,
                        time_t fifth,time_t sixth,time_t seventh)
{
  printf("\n Total Execution Time : %.0f seconds",difftime(seventh,first));
  fprintf(fpo,"\n\n=====================================================");
  fprintf(fpo,"\nTotal Execution Time : %.0f seconds\n",difftime(seventh,first));
  fprintf(fpo,"\n  PREPROCESSING       : %.0f seconds",difftime(second,first));
  fprintf(fpo,"\n  MATCH run time      : %.0f seconds",difftime(third,second));
  fprintf(fpo,"\n  RE-MATCH network creation : %.0f seconds",difftime(fifth,fourth));
  fprintf(fpo,"\n  RE-MATCH run time        : %.0f seconds",difftime(sixth,fifth));
  fprintf(fpo,"\n  SEQUENCE ROUTES     : %.0f seconds",difftime(seventh,sixth));
}
```

Appendix 2 IGLH_MATCH.C Input and Output Files

The IGLH_MATCH program, implemented in the C programming language, requires two input files - one file storing pickup and delivery information at EOL terminals (INPUT-1.IN) and the other containing the cost matrix (INPUT-2.MAT). The input files for example I are displayed below (Figure A-1).

INPUT-1.IN for Example I

```
4 3
1 2 3
2 3 3
3 3 1
4 2 2
```

INPUT-2.MAT for Example I

```
 0 37 50  8 26
37  0 15 30 22
50 15  0 44 31
 8 30 44  0 20
26 22 31 20  0
```

Figure A-1. IGLH_MATCH Input File Format

The first row of INPUT-1.IN reports both the total number of EOL terminals within the study area and the maximum pickup or delivery at any EOL. Subsequent rows give the EOL terminal number, the number of pickups, and the number of deliveries. Using the maximum pickup or delivery at any EOL, the maximum number of copy nodes that must be created in advance are determined. INPUT-2.MAT stores link cost c_{ij} for all (i,j). Given these two input files, the output of Preprocess (Step 1 of IGLH_MATCH) is as follows.

1. IGLH_MATCH STEP 1 OUTPUT

NUM	Index	Supply	Demand	Tractor(PRE)
1	1	0	1	1
2	2	1	1	1
3	3	2	1	0
4	4	0	0	1
5	3	1	0	0

Figure A-2. Preprocess (Step 1 of IGLH_MATCH) Output for Example I

The first column (NUM) of Figure A-2 presents node numbers, while the second column (INDEX) gives original EOL terminal numbers. Hence, node number 5 corresponds to EOL terminal 3. The next two columns of Figure A-2 show the pickups and deliveries after Preprocess. This is based on the maximal sets described in Section 2.2.1. The fourth "Tractor(PRE)" column shows the required number of tractors needed to transport loaded directs that are fixed in Preprocess. So a total of three tractors are needed, one for each EOL terminal 1, 2 and 4. All tractors will transport two deliveries and two pickups. After Preprocess, the total number of nodes may be as many as $N\{\lceil MAX(supply/2, delivery/2)\rceil\}$, where N is the total number of EOL terminals in the network.

Two input files then generate one intermediate input file (i.e., NETFILE1.TMP) that is used in IGLH_MATCH Step 2 (weighted nonbipartite matching problem). The NETFILE1.TMP format is shown in Figure A-3. The first row of NETFILE1.TMP gives the total number of nodes (including copy nodes n'); the total number of undirected links in the network; an "U" (indicating that this is a undirected graph); a value of one if the problem to be solved is the minimum weighted matching problem and, zero otherwise; and an upper bound on route cost (i.e., $\{W \mid W - c_{ij} > 0 \ \forall \ (i,j)\}$). The total number of nodes, including copy nodes n', may be as great as $2N\{\lceil MAX(supply/2, delivery/2)\rceil\}$. Subsequent rows in the NETFILE1.TMP file store information about each node. For example, the second to the sixth rows contain information about EOL terminal 1.

```
10   19   U   1   117
 4    1   0   0
 6   74
 2  102
 3   75
 5   75
 3    2   0   0
 7  100
 1  102
 5  102
 2    3   0   0
 8   16
 1   75
 1    4   0   0
 9    0
 3    5   0   0
10   16
 1   75
 2  102
 5    6   0   0
 1   74
 7    0
 8    0
 9    0
10    0
 5    7   0   0
 2  100
 6    0
 8    0
 9    0
10    0
 5    8   0   0
 3   16
 6    0
 7    0
 9    0
10    0
 5    9   0   0
 4    0
 6    0
 7    0
 8    0
10    0
 5   10   0   0
 5   16
 6    0
 7    0
 8    0
 9    0
```

Figure A-3. Intermediate Input File for *Match* (NETFILE1.TMP)

The second row stores the number of degrees and the node number in sequential order. The third and fourth item (i.e., 0, 0) serve as a delimiter to highlight information about an EOL terminal. The third to sixth row contains *to-node* information and the corresponding route costs (not the simple link cost c_{ij}). For example, the third row indicates that the via containing EOL terminal 1 and EOL terminal 2 has an associated cost of 74. From the example of NETFILE1.TMP, Example I has ten nodes (including six copy nodes). Figure A-4 shows the *MATCH* network (with some infinite cost links elimiated for clarity) for the second step of IGLH_MATCH. Originally, a route cost from EOL 4 to CC was 52; however, after PREPROCESS, EOL 4 has no pickups or deliveries and hence, EOL terminal 4 need not be included in Step 2 of IGLH_MATCH. However, so as not to lose information, the node is included in the *Match* network.

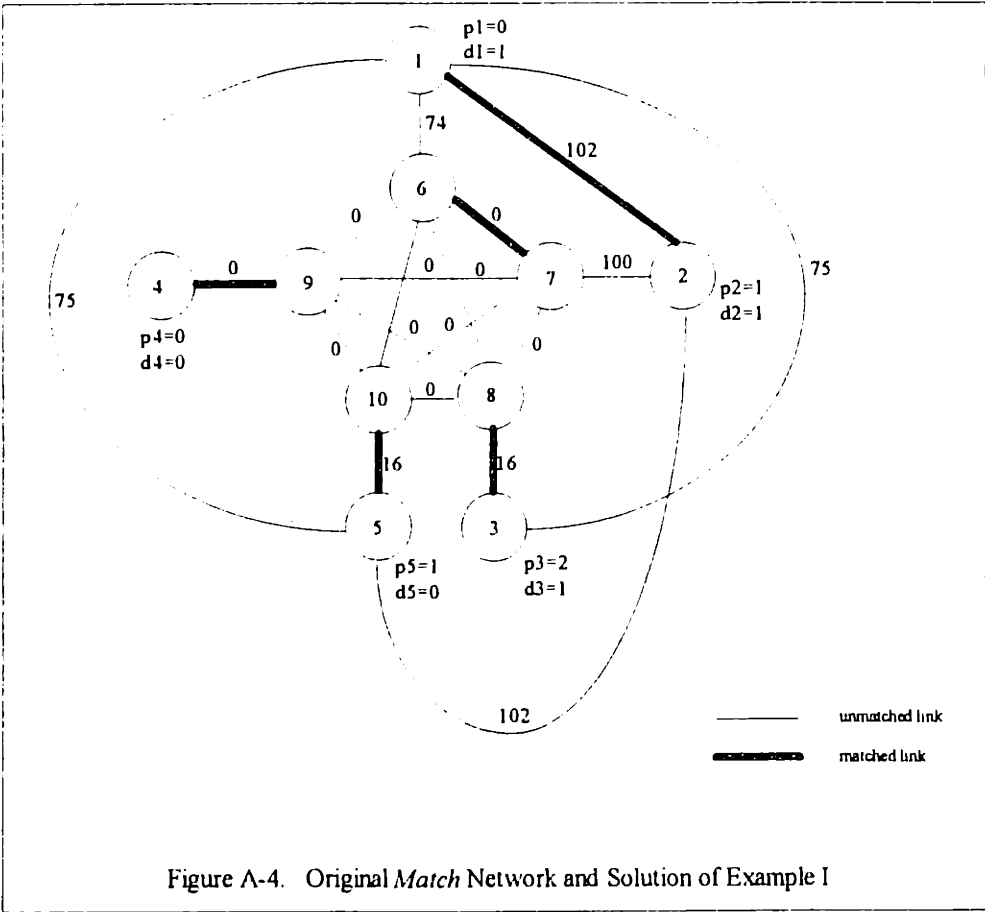


Figure A-4. Original *Match* Network and Solution of Example I

Figure A-5 is the *Match* output. Nodes 1, 3, and 5 are matched with nodes 2, 8 and 10, respectively. Numbers within brackets represent original EOL terminal numbers (INDEX), with a value of zero representing the CC. Total tractor mileage costs as well as the number of tractors needed are as follows: 360 is the value of the *Match* solution; among that 226 resulted from preprocessing and 134 from matching; six tractors are required to service the example I network- three to service Preprocess routes and the others to service *matched* routes.

```
2. IGLH_MATCH STEP 2 MATCHED NODES (INITIAL)
           FROM                TO
  1.     1 [    1]   <-->     2 [    2]
  2.     3 [    3]   <-->     8 [    0]
  3.     5 [    3]   <-->    10 [    0]

3. IGLH_MATCH INITIAL SOLUTION
      FROM         TO   TRACTORS  COSTS
  1.   0   <-->   1       1         74
  2.   0   <-->   2       1        100
  3.   0   <-->   3       2         32
  4.   0   <-->   4       1         52
  5.   1   <-->   2       1        102
=====================================================
   TOTAL TRACTOR COST   =        360
     PRE -PROCESSED COST =       226
     MATCHED COST        =       134
```

Figure A-5. IGLH_MATCH Step 2(*Match*) Output

Based on the *MATCH* solution, the *Re-Match* network is constructed (Figure A-6). There are three nodes in the *Re-Match* network: node 1 is created by adding the via containing EOL terminal 1 and 2, whereas nodes 2 and 3 represents the matched directs to EOL terminals 3 and 5, respectively. Each column of Figure A-6 is as follows:

NUM	:	RE-MATCH network node number
[I]	:	match node i
[MATE]	:	match node j (matched with i)

PICK	:	pickups of re-match network node
DEL	:	deliveries of re-match network node
[I].PIC	:	pickups of match network node i
[I].DEL	:	deliveries of match network node i
[M].PIC	:	pickups of match network node j (matched with node i)
[M].DEL	:	deliveries of match network node j (matched with node i)

Additionally, all possible legal routes are displayed with their associated costs.

4. REMATCHING NETWORK CONSTRUCTION

NUM	[I]	[MATE]	PICK	DEL	[I].PIC	[I].DEL	[M].PIC	[M].DEL
1	1	2	1	2	0	1	1	1
2	3	0	2	1	2	1	0	0
3	5	0	1	0	1	0	0	0

```
* Routing (from 0 to 1) : CC -> 1 -> 2 -> CC        ( 102)
* Routing (from 0 to 2) : CC -> 3 -> CC             (  16 )
* Routing (from 0 to 3) : CC -> 3 -> CC             (  16 )
* Routing (from 1 to 3) : CC -> 2 -> 1 -> 3 -> CC   ( 103)
```

Figure A-6. *Re-Match* Network Information

Given the *Re-Match* network, the procedure for *Match* is repeated. The *Re-Match* network for Example I is shown in Figure 4. Preprocess need not be repeated since all nodes in the *Re-Match* network contain at most two pickups and two deliveries. The following two figures (Figure A-7 and Figure A-8) show the intermediate input file for *Re-Match* and the *Re-Match* output. The *Re-Match* procedure reduces the solution from a value of 360 to 345 by constructing a 3-via containing EOL terminals 2, 1 and 3.

Figure A-9 shows the *Sequence Routes* (IGLH_MATCH Step 3) output. We see that five tractors are needed to successfully execute daily operations and one empty trailer is needed at the CC to achieve balance. Three of the five tractors perform loaded directs to EOL terminal 1, 2 and 4. Another tractor performs a direct trip to EOL terminal 3 satisfying two pickups and one delivery. The last tractor performs the 3-via visiting EOL terminals 2, 1, and 3. The total execution time for IGLH Steps 1, 2 and 3 (i.e., 0 seconds) is displayed in Figure A-9.

```
6    16
1   103
3     4    0    0
1   102
5     0
6     0
3     5    0    0
2    16
4     0
6     0
3     6    0    0
3    16
4     0
5     0
```

Figure A-7. Intermediate Input File for *Re-Match* (NETFILE2.TMP)

```
5. IGLH_MATCH STEP 1 OUTPUT (RE-MATCH NETWORK)

   NUM   Index  Supply  Demand  Tractor(PRE)
    1      1      1       2       0
    2      2      2       1       0
    3      3      1       0       0

6. IGLH_MATCH STEP 2 MATCHED NODES (RE-MATCH NETWORK)

          FROM                TO
  1.    1 [    1]  <-->    3 [    3]
  2.    2 [    2]  <-->    5 [    0]

7. IGLH_RE-MATCH SOLUTION

     FROM        TO   TRACTORS  COSTS
  1.  0   <-->   2       1        16
  2.  1   <-->   3       1       103
=================================================

  TOTAL TRACTOR COST  =          345
  PRE -PROCESSED COST =          226
  POST-PROCESSED COST =          119

This program has been successfully executed.
        Previous matching solution was :  360
        Current matching solution is   :  345
```

Figure A-8. *Re-Match* Output

```
                         ******************************
                          PREPROCESSED ROUTE MAP
                         ******************************
                      route cost /  tractors     total
                         tractor     needed   tractor costs
1. CC -> 1(2,2) -> CC     74(UC)      1(TR)     74(TC)
2. CC -> 2(2,2) -> CC    100(UC)      1(TR)    100(TC)
3. CC -> 4(2,2) -> CC     52(UC)      1(TR)     52(TC)

PreProcessed Trip Cost :    226.

                            **********************
                             MATCHED ROUTE MAP
                            **********************
1. CC->  3( 2, 1)-> CC                                   16     : Deficit route

2. CC->  2( 1, 1)->  1( 0, 1)->  3( 1, 0)-> CC          103     : Balanced Route


MATCHING_Pickup =    4,   MATCHING_Delivery =    3.
MATCHING_Empties =  -1,   MATCHING_Cost     =   119
A Deficit MATCHING Network


                  ******************************************
                        SEQUENCE ROUTES SUMMARY
                  ******************************************
* Deficit Network : 1 empties must be reserved at CC.

1.  Sroute[0] + Droute[1]
                           *************************
                             IGLH_MATCH SUMMARY
                           *************************
  Preprocessed Cost :      226
  Matched Trip Cost :      119
  Total Tractor Cost :     345

==================================================
     Total Execution Time               : 0 seconds

     PREPROCESSING                      : 0 seconds
     MATCH                              : 0 seconds
     RE-MATCH network creation          : 0 seconds
     RE-MATCH run time                  : 0 seconds
     RE-MATCH result printing           : 0 seconds
     SEQUENCE ROUTES                    : 0 seconds
```

Figure A-9. IGLH_MATCH Step 3(*Sequence Routes*) Output of Example I

In Figure A-9, Sroute[*r*] denotes for route *r*, the total number of deliveries in excess of pickups, and Droute[*r*] denotes the reverse. As shown in Section 2.3, all balanced routes (i.e., the three preprocessed routes and the first matched route) meet empty trailer balancing requirements. The last route (i.e., the second matched route), however, requires that one empty trailer be picked up at the CC before the route is begun.

Figure A-10 shows a typical output of IGLH_MATCH. In this example, without IGLH_MATCH Step 3, one empty trailer is necessary at the start of the first matched route. However, since the second matched route delivers one empty trailer to the CC, the appropriate sequencing is for route 2 to be followed by route 1. Hence, empty trailer balancing is achieved with no empty trailer inventory at the CC.

```
                         ******************************
                           PREPROCESSED ROUTE MAP
                         ******************************
                         route cost /  tractors    total
                          tractor      needed   tractor costs
1. CC -> 2(2,2) -> CC     220(UC)       1(TR)     220(TC)

PreProcessed Trip Cost :    220.

                              **********************
                              MATCHED ROUTE MAP
                              **********************
1. CC->  1( 2, 1)-> CC                               44       : Deficit route

2. CC->  3( 1, 2)-> CC                              158       : Surplus Route


MATCHING_Pickup =    3.   MATCHING_Delivery =     3.
MATCHING_Empties =   0.    MATCHING_Cost    =     202
A Balanced MATCHING Network


                   ******************************************
                          SEQUENCE ROUTES SUMMARY
                   ******************************************

* Balanced Network : No additional empties are required at CC.

1.  Sroute[2] + Droute[1]

                            *************************
                              IGLH_MATCH SUMMARY
                            *************************

  Preprocessed Cost  :    220
  Matched Trip Cost  :    202
  Total Tractor Cost  :   422

====================================================================
     Total Execution Time              : 0 seconds

     PREPROCESSING                     : 0 seconds
     MATCH  run time                   : 0 seconds
     RE-MATCH network creation         : 0 seconds
     RE-MATCH run time                 : 0 seconds
     RE-MATCH result printing          : 0 seconds
     SEQUENCE ROUTES                   : 0 seconds
```

Figure A-10. Another Example of IGLH_MATCH Step 3 Output