NOTICE

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

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

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

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

AVIS

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

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

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

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

Two Distributed Algorithms

for Finding Matchings on a Graph

by

Yuen, Ching Ki (Simon)

A thesis submitted to the Department of Computer Science
in conformity with the requirements for
the degree of Master of Computer Science

University of Ottawa
Ottawa, Ontario, Canada

NOTICE

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

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

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

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

AVIS

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

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

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

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

ISBN 0-315-60585-5

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

Abstract

Two new distributed algorithms for finding matchings on a graph are presented. The first algorithm generates a spanning matching. A spanning matching on a graph is one with respect to which no two adjacent nodes are both free. Starting from any matching, the second algorithm proceeds in phases until a maximum matching is found. In each phase, the algorithm finds at least one augmenting path on the graph with respect to the current matching and then augments the current matching. The algorithm terminates when no more augmenting paths can be found. The main features of our algorithms are the use of multiple virtual alternating trees for finding augmenting paths without using messages for synchronization, "shrinking blossoms" and the labelling. Both algorithm are based on asynchronous networks and assume that only local information is available in each process. Our first algorithm has worst-case communication complexity O(m) and time complexity O(n); and our second algorithm has worst-case communication complexity O(mn) and time complexity $O(n^2)$, where n is the number of nodes and m is the number of edges of the graph.

Acknowledgments

I am indebted to my supervisor Professor To-Yat Cheung for providing sound advice and suggestions throughout my graduate studies. In particular, he patiently read through many revisions of this thesis. I would like to thank my parents. Their continual support made me succeed in this endeavour. I would also like to express my gratitude to my wife Miu-Chi for her unflagging patience and encouragement throughout my studies.

Thanks are given to the Natural Sciences and Engineering Research Council of Canada, Telecommunications Research Institute of Ontario and the University of Ottawa for their financial assistance.

Contents

Figures

Page

Tables

Page

Chapter 1

INTRODUCTION

1.1 Motivation and Description of the Problem of the Thesis

The dream of distributed computing in which problems are solved with their data and control processes dispersed over many places has become a reality due to the progress of network technology in the last two decades. In particular, many graph-related problems already solved in a non-distributed environment are reconsidered in a distributed environment. In the distributed environment, a graph is embedded in a network. Distributed graph algorithms for problems such as minimum spanning trees [CHAN 79] [GALL 83], max-flow [CHEU 83], depth-first-search [AWER 85c], and breadth-first-search [CHEU 83] [ZHU 87] have appeared in the literature. However, many algorithms still do not have a distributed version.

In this thesis, we propose two new distributed algorithms, one for finding a spanning matching, and another for finding a maximum-cardinality matching. A matching M on a graph is a subset of edges such that no two edges in M have a node in common. The spanning matching problem is to find a matching M such that no two adjacent nodes of the graph are both free with respect to M. The maximum matching problem is to find a matching M including as many edges as possible.

The maximum-cardinality matching problem has many applications. The stable marriage problem of operations research can be formulated as a matching problem (see [SCHN 77]). The scheduling of tasks on multiprocessor computers can be modeled as matching problem (see [FLYN 66]). The maximum matching algorithm is used as preprocess of the Chinese postman problem (see [GUAN 62]). In linear programming, the maximum matching algorithm is used to convert a polyhedral matching into a polynomial-time algorithm (see [GROT 81]).

In the following sections, we present some fundamental knowledge of distributed algorithms and some factors which affect the design of distributed algorithms. These include global and local information, control strategies, complexity measurement, synchronous and asynchronous networks and a model of distributed computation.

1.2 Fundamentals of Distributed Algorithms

Sharing the workload in solving a problem is one of the main objectives of distributed computing. A distributed algorithm permits many processes located at different sites of a network to communicate with one another. Information about the problem and the network is distributed over these processes. The data and control information has to be exchanged through messages.

Descriptions of a distributed algorithm should include the computational environment and the operational details of the processes. A computational environment includes a general model of computation and a specific operational architecture of the network. In Section 1.2.2, we describe the model for distributed computation for graph-related problems. The operational details should include lists of the local and global variables, messages and an explanation of the processes from two points of view : a global-view and a distributed-view. A global-view describes the overall coordination of the processes over the network. A distributed-view stipulates the details of how a process responds to the messages received.

In general, an initiator node starts the algorithm and/or creates the operational environment on the network. A process is used in every node for interprocess communications.

1.2.1 Factors Affecting the Design of Distributed Algorithms

To design a distributed algorithm is to plan and validate the functions, contents and transmission schedules of the messages so as to achieve the objective of the algorithm. Many factors may affect their design. We briefly explain some of them in the following.

Global and Local Information is Knowledge about the Network and Problem

Both global and local information may be available at each node for coordinating the transmission of messages. Local information is about its immediate neighbors only. Global information is knowledge beyond its immediate neighbors, such as the total number of nodes and their identities, the entire routes for transmitting messages and some sub-structures (spanning tree, binary tree, shortest path, etc.) created on the network.

The amount of knowledge available to a process has profound effects on the design of a distributed algorithm. In general, the more global information the processes know, the lower complexity an algorithm should achieve. Strictly speaking, it is unfair to compare algorithms which use different amounts of global information. Note that one of the main challenges in distributed computing is to solve problems using as little global information as possible.

Control Strategies for Distributed Algorithms

In a distributed system, the data and tasks are dispersed over many sites. A control strategy specifies how the processes co-operate with one another and which process(s) will give commands exchanging messages. In some distributed systems, after a process finishes with a sub-task, it transfers the control to another process. In some other systems, a single process keeps the control throughout the entire algorithm. The types of control in a distributed system (see [CHEU 88] for more details) can be classified as follows.

- Centralized Control (CC) : In this strategy, one of the processes acts as the single master throughout the entire algorithm. All the communication activities are triggered by the master. Distributed algorithms [CHEU 83], [SHAM 82] and [ZHU 87] use this strategy.

- Decentralized control (DC) : In this strategy, there is no master-slave relationship among the processes. Synchronization of the communication activities is achieved only through the exchange of messages. The work load is evenly distributed among the processes. This strategy is used in [CHAN 79].

- Shifting centralized control (SCC) : In this strategy, an algorithm has several phases. Each phase is under the centralized control of one of the processes until the phase is over. Control is then shifted to another process. Cheung's sorting algorithm [CHEU 89] is a good example to demonstrate the characteristic of SCC algorithms.

In general, a CC algorithm is easier to design and has lower communication costs in comparison with a DC or SCC algorithm. However, a CC algorithm creates a bottleneck in the network. This is against the objective of distributed computing. In general, therefore, it is unfair to compare the communication costs between a CC algorithm and a DC algorithm. Sometimes, two or three different strategies may be used in a distributed algorithm.

Synchronous and Asynchronous Networks for Distributed Algorithms

A synchronous network satisfies the following assumptions: All the processes are synchronized by a global clock and the messages are sent only at times of clock-pulses. Only one message can be sent out over a given link at a certain pulse. The delay of each message is at most one time unit of the global clock so that the messages must arrive before the next pulse begins. An asynchronous network does not make any time-related assumptions on the transmission of messages except that every messages sent will eventually arrive.

In this thesis, we use the term *synchronous algorithm* to mean a distributed algorithm which works on a synchronous network and the term *asynchronous algorithm* to mean a distributed algorithm which works on an asynchronous network.

Usually, asynchronous algorithms are inferior to synchronous algorithms in terms of message complexity. The reason is that, in a synchronous algorithm, by assuming that the messages are already synchronized and a lot of efforts for attaining certain functions may be ignored. Thus, it requires less messages for coordinating the transmission. However, strictly speaking, it is not legitimate to compare synchronous and asynchronous algorithms.

1.2.2 A Model of Distributed Computation for Graph-Related Problems

A model of distributed computation specifies the operational architecture of the network and uses that network as an abstraction. It includes a set of assumptions on the network and the characteristics of processes. In this thesis, we adopt a model proposed in [CHEU 83], which has been used with different variations in the literature for the design of many distributed algorithms for graph-related problems, such as parallel graph traversal [CHAN 79], maximum network flow [CHEU 83], maximum matching [WU 87], depth-first-search [AWER 84], etc. The major assumptions of this model are summarized in the following :

a) The graph is embedded in a loosely-coupled communication network. Each graph node is represented by one network node. Each graph edge is represented by one bidirectional communication channel.

b) Each communication channel is FIFO so that messages on a channel arrive in the same order in which they are sent.

c) The network is asynchronous.

d) Each node has only local information of the graph problem and the network. It has some local variables for storing the state of the distributed algorithm and knows only its adjacent neighbors.

e) The network is reliable, meaning that there are no failures in message transmission and node operations. This is a legitimate assumption for high-level protocols.

1.2.3 Complexity Measures for Distributed Algorithms

The complexity of a distributed algorithm can be measured in four ways : the total number of messages, the elapsed time, the total number of bits of messages, and the storage needed at each node.

The total number of messages required by a distributed algorithm is called its *message* or *communication* complexity.

For a synchronous algorithm, the elapsed time, i.e., the *time* complexity of an algorithm, is the total number of the pulses elapsed between the initiation and termination of the algorithm. For an asynchronous algorithm, strictly speaking, there should be no time complexity because there is no assumption on the time taken for transmitting a message. However, sometimes, just for the purpose of estimating the total time, it is assumed that each message takes a unit of time to travel on a link. Note that this assumption is not used in scheduling the messages. Therefore, the time complexity of an asynchronous algorithm is also the total number of pulses elapsed between the initiation and termination of the algorithm.

The total amount of information transmitted in the network is called the *bit* complexity.

The storage space used in each process for a distributed algorithm is called its *space* complexity. In the *memory restricted model*, the amount of storage space available at each node is bounded by a linear function of its degree of connection. Otherwise, it is called *general distributed model* (for more details see [AWER 84]). In some cases, measuring the storage space requirement of a distributed algorithm is important.

There is always a trade-off when using different complexity measures. In some cases, the increase in bit or space complexity will result in the improvement of time and/or message complexities. For example, if more space is provided in each node in the file sorting problem [CHEU 89], each process can request more records in each iteration and thus reduce the time needed for sorting the subfiles. Another example is that the more global information is available, the better complexity can be obtained. Still another example is the complexity for synchronous and asynchronous algorithms.

1.3 Contribution and Summary of the Thesis

In this thesis, we present two new asynchronous distributed algorithms for finding matchings on a graph, based on local information only. The first algorithm (DSM) generates a spanning matching. A spanning matching is one where no two adjacent nodes of the graph are

both free with respect to that matching. Starting from any matching, the second algorithm (DMM) finds a maximum matching (i.e., maximum number of matched edges) on that graph. The communication complexities of DSM and DMM are O(m) and O(mn), respectively, where m is the number of edges and n is the number of nodes of the graph. The time complexities of DSM and DMM are O(n) and $O(n^2)$, respectively. For the maximum matching problem, DMM has a better communication complexity than all existing algorithms. The time complexity of DMM is lower than [WU 87], but is higher than [SCHI 86]. As discussed in Section 1.2.1, this is not surprising because [SCHI 86] is a synchronous algorithm. Strictly speaking, it is not proper to compare it with our asynchronous algorithm DMM. Also, [SCHI 86] used global information.

The rest of the thesis is organized as follows. In Chapter 2, we review the fundamentals and the literature of both non-distributed and distributed algorithms for the maximum matching problem. In Chapter 3, we present our first distributed algorithm DSM which creates a spanning matching on a graph. In Chapter 4, we present our second distributed algorithm DMM which obtains a maximum matching by finding augmenting paths. In Chapter 5, we summarize the result of our research and compare it with other works. Also, we point out some directions in which we may improve our algorithms.

Chapter 2

FUNDAMENTALS AND REVIEW ON THE MAXIMUM MATCHING PROBLEM

In this chapter, we present the fundamentals of the maximum matching problem and briefly review the existing methods of solution for this problem in both non-distributed and distributed environments.

2.1 Fundamentals of the Maximum Matching Problem

Let G(N,E) be a graph, where N is a set of nodes and E is a set of edges. A *matching* M on G(N,E) is a subset of E such that no two elements of M have a node in common. M is a *maximum-cardinality matching* of G if it has the maximum number of edges.

Figure 2.1 illustrates the terminology to be defined below. A node *i* of G is said to be *matched* with respect to a matching M if either the edge (i,j) or the edge (j,i) is in M; otherwise, i is said to be *free*. An edge is said to be *matched* with respect to M if it is in M and *unmatched* other-

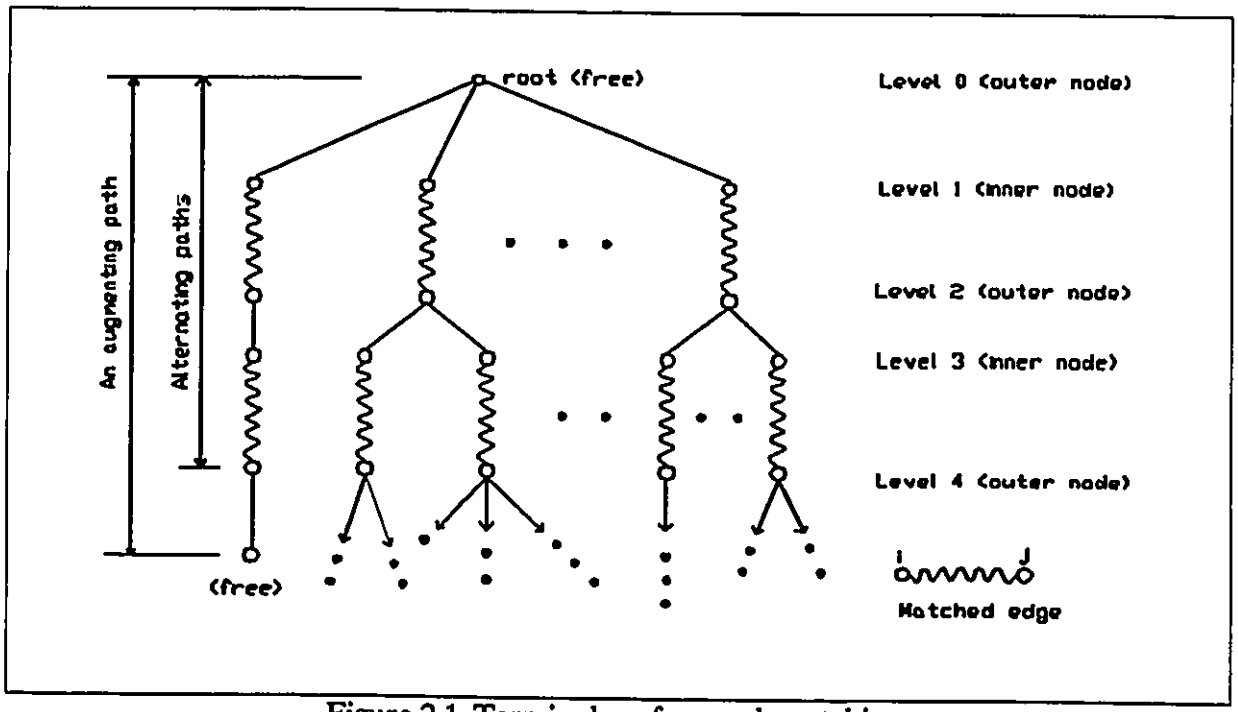


Figure 2.1. Terminology for graph matching.

wise. An *alternating path* with respect to M is a simple path in G (i.e., a path which does not include the same node twice), whose edges are alternatively matched and unmatched with respect to M. An *augmenting path* with respect to M is an alternating path whose two end nodes are both free. If G has such a path, reversing the role of its edges (i.e., those edges not in M become matched, whereas those in M become unmatched) can create another matching M' such that $|M'| = |M| + 1$, where $|M|$ denotes the cardinality of M.

An *alternating tree* with respect to M is a tree with the property that its root is free and all its paths emanating from the root are alternating. Every node of an alternating tree has either an 'inner' status or an 'outer' status. A node has an *inner* (resp., *outer*) status if the alternating path from the root to that node has an odd (resp., even or zero) number of edges. In particular, the root of an alternating tree has an 'outer' status. Figure 2.1 shows the terminology defined above.

A *blossom* is an odd-length alternating cycle. Figure 2.2a shows the blossom (c,d,e) which shrinks into a pseudo-node X shown in Figure 2.2b. A blossom may include pseudo-node(s).

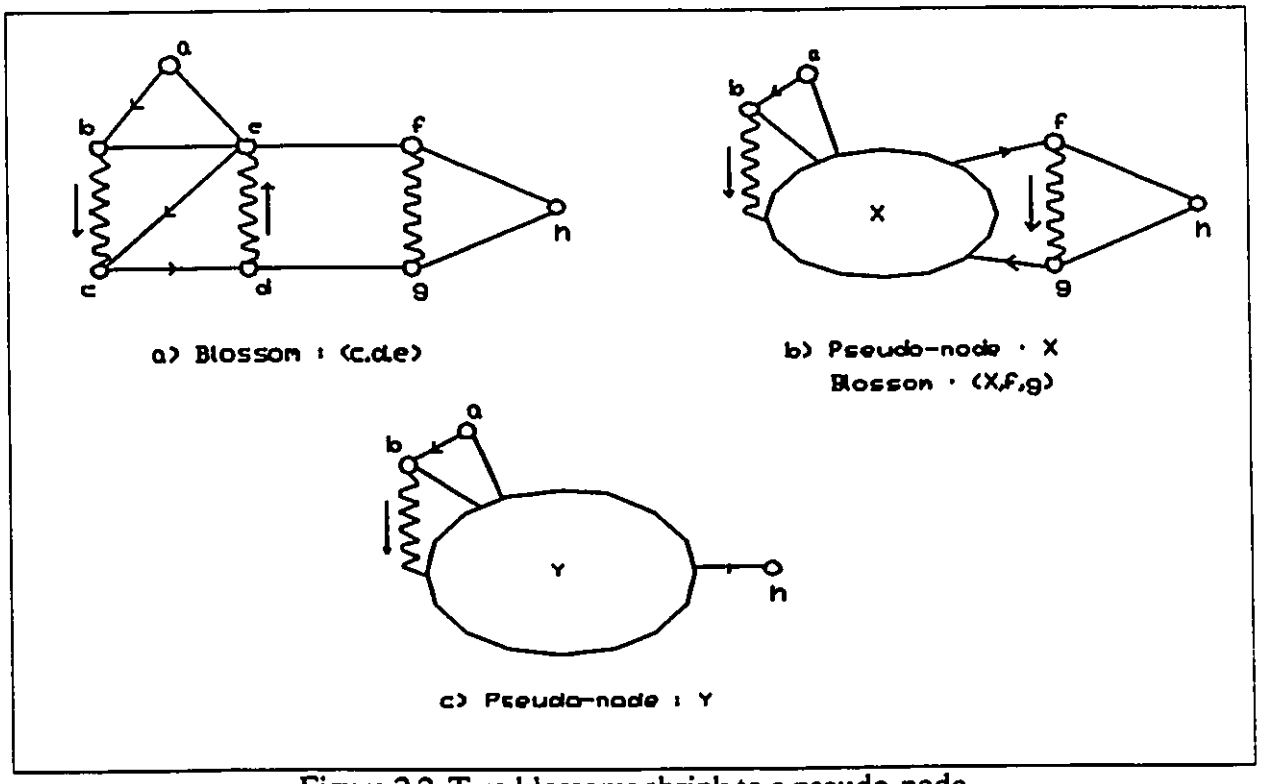


Figure 2.2. Two blossoms shrink to a pseudo-node.

Figure 2.2b shows a blossom (X,f,g) which includes the pseudo-node X and shrinks into a bigger pseudo-node Y in Figure 2.2c.

An edge (i,j) is said to be a *bridge* between two different alternating trees if i and j belong to two different alternating trees and either both i and j are at odd level (have an 'inner' status) or both i and j are at even level (have an 'outer' status). Figure 2.3 shows two bridges. Note that, in a bridge (i,j), the level of i is not necessarily equal to the level of j.

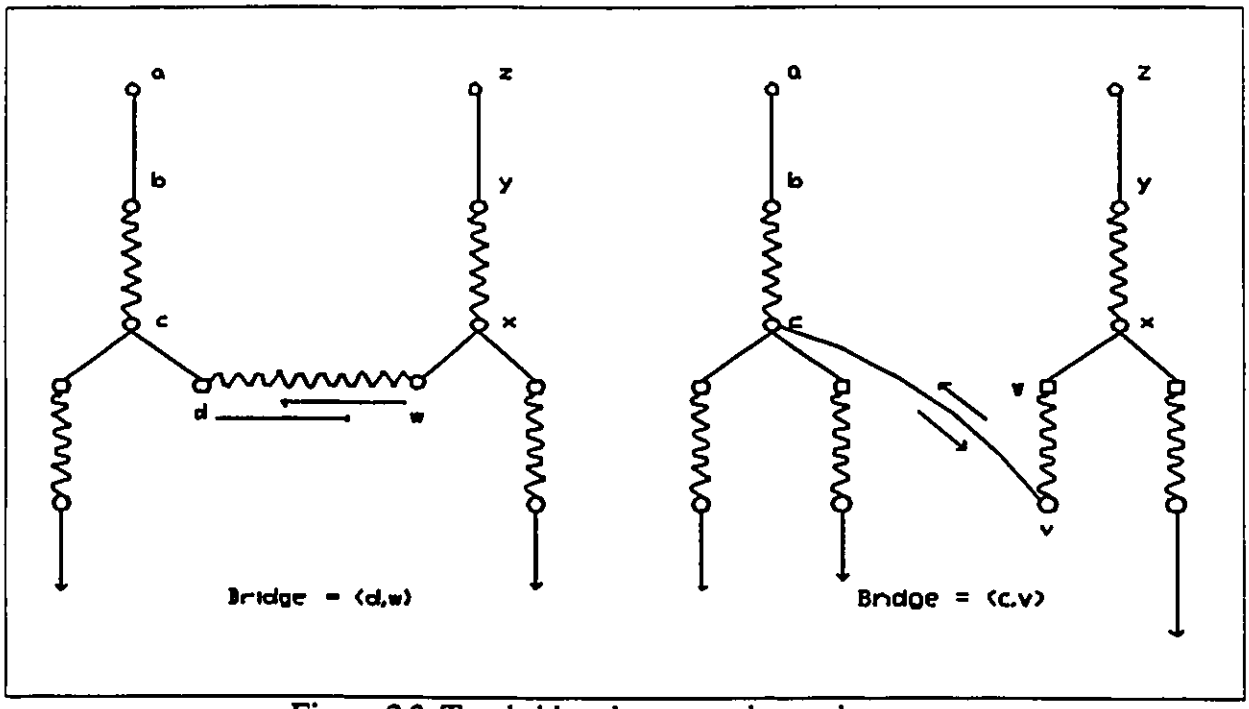


Figure 2.3. Two bridges between alternating trees.

2.2 Review on Algorithms for Matching a Graph in a Non-distributed Environment

In this section, we review seven existing algorithms for finding a maximum-cardinality matching on a graph in a non-distributed environment. Table 2.1 summarizes the complexity of these algorithms, where n is the number of nodes and m is the number of edges of the graph.

	Authors (year of paper)	Complexity
I	Berge (1957)	exponential
II	Edmonds (1965)	$O(n^2m)$
III	Hopcroft and Karp (1973)	$O(n^{1/2}m)$ for bipartite graphs
IV	Even and Kariv (1975)	$O(n^{2.5})$
V	Gabow (1976)	$O(n^3)$
VI	Pape and Conradt (1980)	$O(n^3)$
VII	Micali and Vazirani (1980)	$O(n^{1/2}m)$

Table 2.1. Complexity of existing non-distributed maximum-matching algorithms.

The approach adopted by all the algorithms in Table 2.1 includes two steps : (1) Find augmenting paths with respect to a current matching. (2) Augment those augmenting paths with respect to the current matching. The algorithms differ mainly in the techniques for finding these paths. In the following, we first classify the algorithms into three different categories and then describe each of them briefly.

Category 1) An algorithm generates only one alternating path in each iteration. It starts from a 'non-explored' free node and extends the alternating path until the path reaches another free node and thus forms an augmenting path. If the expansion cannot reach another free node, the starting free node is marked as 'explored'. Otherwise, the algorithm augments that augmenting path. It then starts the searching of another augmenting path from another 'non-explored' free node until all the free nodes are 'explored'.

Category 2) An algorithm generates only one alternating tree in each iteration. It starts from a 'non-explored' free node as the root of the alternating tree and extends the tree level by level until one of its alternating paths reaches another free node. If the expansion cannot reach any of the free nodes, the root is marked as 'explored'. Otherwise, the algorithm augments that augmenting path. It then starts to generate another alternating tree from another 'non-explored' free node until all the free nodes are 'explored'.

Category 3) An algorithm generates a set of alternating trees simultaneously. It starts from all the free nodes as the roots of the alternating trees and extends all these trees level by level. After the extension of each level, if two alternating paths from two different alternating trees meet each other, the algorithm augments the matching through these two alternating paths. The algorithm also tries to augment the other augmenting paths if there exist more than one augmenting paths at the same level. After the augmentation, the algorithm generates another set of alternating trees from all the free nodes. The algorithm terminates if no augmenting path is found.

The following paragraphs describe each algorithm in more details.

I) Berge [BERG 57] -

Berge was the first to propose the concept of alternating paths and derive a basic result which has become the theoretical foundation of all the matching algorithms developed afterwards. Let us first explain the notation '$\oplus$' (XOR) by Figure 2.4.

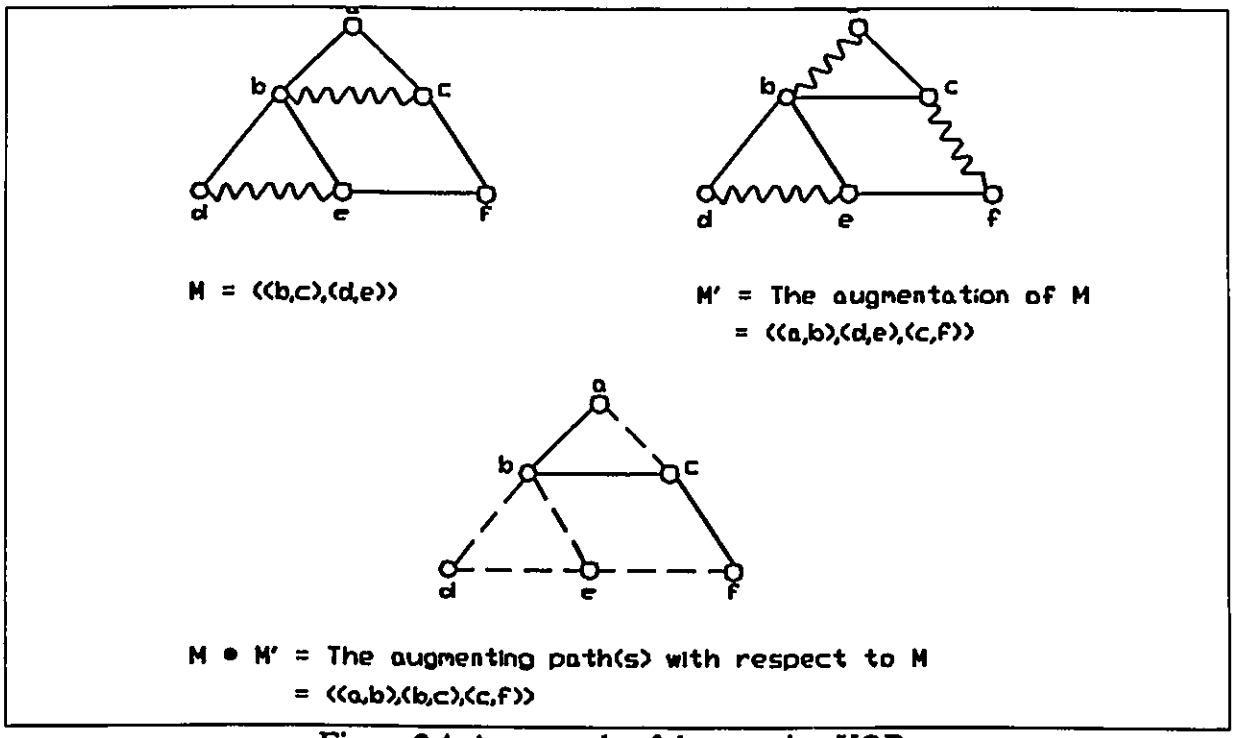


Figure 2.4. An example of the notation XOR.

Notation. Let M be a matching and M' be another matching which is obtained by augmenting M. $M \oplus M'$ denotes all the edges in the augmenting paths with respect to M, which has applied to the reversing the role of their edges during the augmentation. In Figure 2.4, $M \oplus M'$ shows the set of such edges.

Theorem 2.1 (Berge [BERG 57]). A matching M on G has maximum cardinality if and only if G has no augmenting paths with respect to M.

Proof. (only if) : Suppose that G has an augmenting path P. By changing the status of the edges of P (i.e., matched edges become unmatched, and vice versa), the cardinality of M is increased by 1. (if) : Suppose that M is not a maximum matching. Let M' be a matching with a larger cardinality (i.e., $|M'| = |M| + 1$). Consider the a set of edges Q that are in either M or M' but not in both (i.e., $Q = M \oplus M'$). Since $M \neq M'$, Q is not empty and contains at least one augmenting path for G with respect to M. ♦

Berge proposed an algorithm which uses the depth-first-search (DFS) technique to look for augmenting paths. An alternating path starts from a free node and is extended by including one edge at a time. The alternating path cannot visit the same node twice. If the alternating path cannot be extended further, it backtracks to the closest ancester and tries another branch. If an augmenting path is found, the path is augmented. If no augmenting path can be found from this free node, the algorithm starts the search from another free node. This exhaustive search continues until all the free nodes have been explored. The algorithm requires exponential time.

II) Edmonds [EDMO 65] -

Edmonds developed the following algorithm for constructing a maximum matching and claimed a complexity estimation of $O(n^4)$. However, a closer examination [WITZ 65] of the algorithm reveals that it is in fact of complexity $O(n^2m)$.

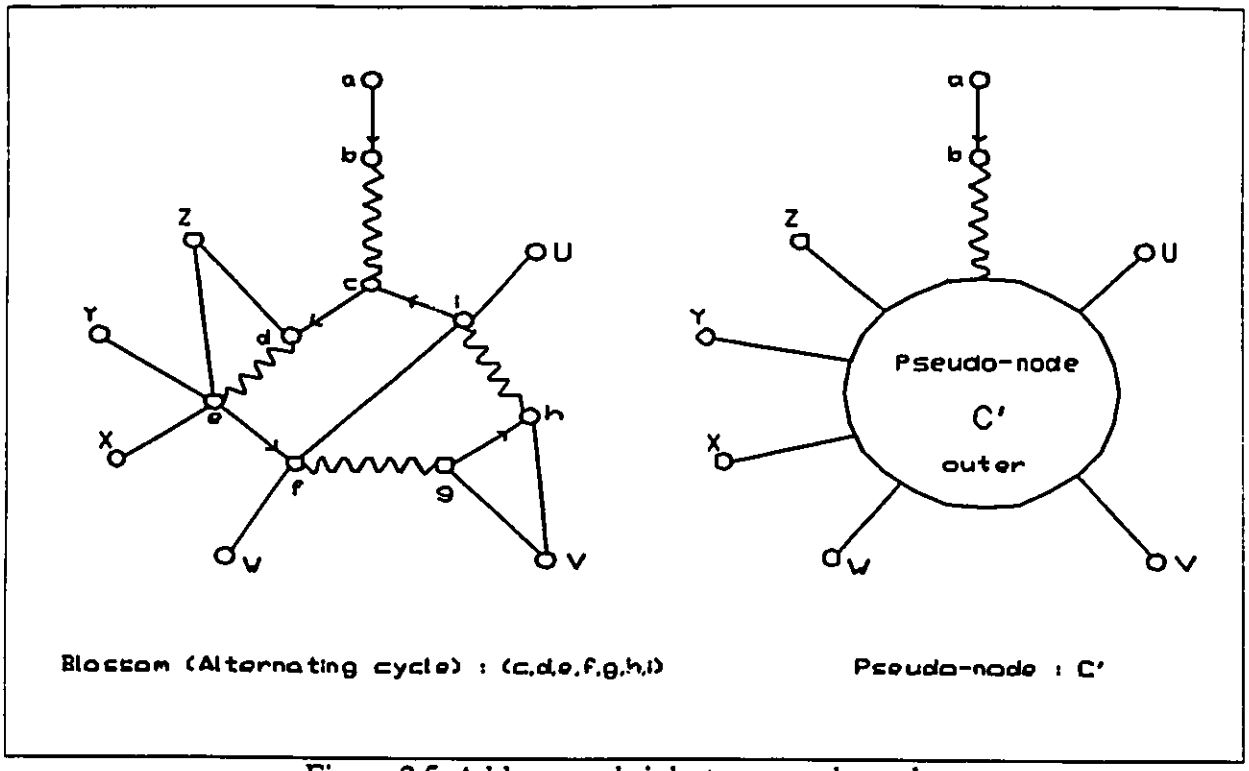


Figure 2.5. A blossom shrinks to a pseudo-node.

An alternating path P is constructed from a free node r (r is labelled as 'outer'). If P meets a free node, an augmenting path has been found. Two cases are considered when P is extended to a node i which has already been labelled. If i has been labelled as 'inner', P does not expand to i. If i has been labelled as 'outer', an odd-length alternating cycle C (explained in Section 2.1 and Figure 2.2) has been detected. Edmonds gave the name *blossom* to such a cycle and defined that all the nodes of a blossom can be conceptually considered as an 'outer' node by shrinking C to form a single *pseudo-node* in P (the pseudo-node is labelled as an 'outer'). In Figure 2.5, the blossom (c,d,e,f,g,h,i) on the left shrinks to a pseudo-node c' on the right and is labelled as an 'outer'. Node c, e, g, and i have the 'outer' status when considered in the alternating path (a,b,c,d,e,f,g,h,i); and node c, h, f, and d are considered as 'outer' in the alternating path (a,b,c,i,h,g,f,e,d). Since all the nodes in c' are 'outer', the extended P (a,b,c') can extend to all the nodes U,V,W,X,Y, and Z. To continue the extension of P, another blossom can be formed by many small blossoms. If no augmenting path can be found from the starting free node, the

algorithm starts the searching from another free node. This exhaustive searching continues until all the free nodes are explored.

III) Hopcroft and Karp [HOPC 73] -

Hopcroft and Karp proposed an algorithm for finding a maximum matching on a bipartite graph. They first defined : An augmenting path P with respect to a matching M is said to be *shortest* if P has the smallest cardinality among all the possible augmenting paths with respect to M. In the following, we first extend the notation '$\oplus$' (XOR) from Berge, then describle their result.

Notation. Let M be a matching and P be a set of edges of an augmenting path p with respect to M. $M \oplus P$ is another matching which is obtained by augmenting p with respect to M. In Figure 2.2, M is a matching ((b,c),(d,c)), p is an augmenting path (a,b,c,f), and P is the set of edges ((a,b),(b,c),(c,f)) of p. $M \oplus P$ is another matching ((a,b),(d,e),(c,f)) which is obtained by augmenting p with respect to M.

Theorem 2.2. Let M be a matching, P be a set of edges of a shortest augmenting path with respect to M and P' be another set of edges of an augmenting path with respect to $M \oplus P$. Then, $|P'| \geq |P| + |P \cap P'|$.

Proof. Let M' be a matching which is obtained by $M \oplus P$ and M" be another matching which is obtained by $M' \oplus P'$. Since M" is obtained by augmenting M (the cardinality is increased by 1 twice), so $|M"| = |M| + 2$. If M can be augmented once (the cardinality is increased by 2) to obtain M", M must contain two node-disjoint augmenting paths, say P_1 and P_2, with respect to M, which can be obtained by $M \oplus M"$. Since $M \oplus M" = P \oplus P'$, $|P \oplus P'| \geq |P_1| + |P_2|$ because P represents a shortest augmenting path (notice that $|P_1| \geq |P|$ and $|P_2| \geq |P|$). Thus, $|P \oplus P'| \geq |P_1| + |P_2| \geq 2|P|$. Also, since $|P \oplus P'| = |P| + |P'| - |P \cap P'|$, we have $|P'| \geq |P| + |P \cap P'|$. ♦

Based on this theorem, they proposed an algorithm with the following computational scheme (In the following, M_i is a matching which is obtained by the augmentation of M_{i-1}): Starting with an initial matching $M_0 = \varnothing$, the algorithm computes a sequence of matchings $\{M_0, M_1, M_2, ...\}$ and a sequence of shortest augmenting paths $\{P_0, P_1, P_2, ...\}$ with respect to M_i such that $M_{i+1} = M_i \oplus P_i$ where $i = 0,1,2,...$. It follows that, $|P_{i+1}| \geq |P_i|$. If $|P_i|$ is equal to $|P_j|$ for all i and j, P_i and P_j are node-disjoint augmenting paths.

They showed the following property for this algorithm : The number of distinct integers in the sequence $\{ |P_0|, |P_1|, ..., |P_i|, ... \}$ is less than or equal to $2 \lfloor \sqrt{s} \rfloor + 2$, where s ($s \leq n/2$) is the cardinality of the maximum-matching. Thus, the sequence $\{M_i\}_{i=1,2,...}$ can be obtained in $2 \lfloor \sqrt{s} \rfloor + 2$ phases, within each of which a maximal set of node-disjoint shortest augmenting paths is found.

IV) Even and Kariv [EVEN 75] -

Even and Kariv proposed an algorithm which uses a labelling technique for searching the augmenting paths. The algorithm first uses the breadth-first-search (BFS) method for generating a set of an alternating trees from every free node simultanously. The generation continues until two alternating paths from two different alternating trees meet each other. Then, the algorithm augments the matching through those two alternating paths. If there are more than two alternating paths have met, the algorithm then uses the depth-first-search (DFS) method to verify those alternating paths because some paths are node-joined. During the generation of the alternating trees, the algorithm will shrink every odd-length cycle and stores them into a data structure which serves to restore the augmenting paths. Every node in the odd-length cycle is labelled inside the data structure which enables the algorithm to trace the minimum augmenting path in case one of the former paths fails. This is done without searching the graph again. The algorithm works in phases (using Hopcroft and Karp's results). Each phase consists of four steps:

Step 1) The algorithm generates an alternating tree from every free node. The algorithm finds one of the minimum alternating paths leading to the root and uses it to determine the level of nodes in the tree. The information of the alternating paths are registered in another special data structure which makes it restorable later on. At the end, the algorithm knows the level of each node and the length of any minimum augmenting path in the graph.

Step 2) The algorithm generates another reduced graph in which each node lies on a minimum augmenting path of the original graph, and the place of the node on this graph is identical to its level. Thus, every maximal set of disjoint augmenting paths of the reduced graph corresponds to a maximal set of disjoint and minimum augmenting paths of the original graph.

Step 3) For every overlaped edge (from two different alternating trees), the algorithm searches the reduced graph by DFS technique in order to find an augmenting path leading to the free nodes. The augmenting paths are registered in another data structure.

Step 4) The algorithm restores the augmenting paths found in the third step. The algorithm traces the path as it appeared in the reduced graph and then traces the path in the original graph from which the path of the reduced graph was derived. The algorithm then augments the augmenting path.

Even and Kariv showed that the augmentation is along the shortest augmenting paths and the complexity is $O(n^{2.5})$. However, the algorithm is extremely complicated, and its storage requirement is very high. Thus, their algorithm is not practical. It is primarily for theoretical interest.

V) Gabow [GABO 76] -

Gabow described the following blossom handling technique : Since the blossoms are disjoint, the total size of all alternating trees at any moment is O(n). When the algorithm generates a new blossom, it does not rename the edges (edges retain their original names). In order to find out quickly to which blossom a given node belongs, the algorithm maintains a membership array in a special data structure. When a blossom B is shrunk, the algorithm puts the nodes of B into a queue Q; so that it can later scan those nodes instead of scanning the new node B. If the other nodes of B have already been inserted into Q, the algorithm does not delete them from Q. When the algorithm considers an edge in a cycle, it ignores the cycle if both endpoints are in the same blossom. Thus, a stage takes $O(n^2)$ time and the total complexity is $O(n^3)$ time.

VI) Pape and Conradt [PAPE 80] -

Pape and Conradt's algorithm has two parts. The first part randomly generates an initial matching. The second part searches for augmenting paths by constructing an alternating tree with respect to the initial matching. The algorithm starts from a free node as the root of an alternating tree. A node of an alternating tree cannot be passed through twice by an alternating path. However, a node can be passed through by any number of different alternating paths. If an augmenting path has been found, the current matching is augmented with that augmenting path. If no augmenting path has been found, the free node is marked as 'explored'. The algorithm continues to assign every free node as the root for searching augmenting paths until all the free nodes are marked, then the algorithm terminates.

It turns out that in a randomly generated, undirected, connected graph, the performance of this algorithm is very fast. The primary reason for this good performance is that the initial matching is often close to being a maximum matching, and therefore only a few iterations of the second part are performed. The time complexity depends on the structure of the graph. In the

worst case, it takes $O(n^3)$ time for finding a maximum matching on a general graph. Extensive timing studies performed by Derigs (appear in [DERI 80]) show that this algorithm is indeed very fast, almost the fastest among all known matching algorithm for all practical purposes.

VII) Micali and Vazirani [MICA 80] -

Micali and Vazirani developed an algorithm with the lowest complexity. They showed that their method finds a maximum matching in $O(n^{1/2}m)$ time. The algorithm proceeds in phases. In each phase, the algorithm constructs a set of alternating trees from every free node using the labelling technique and detects a maximal set of node-disjoint minimum-length augmenting paths. They used the result of Hopcroft to show that only $O(n^{1/2})$ such phases are needed.

In each phase, the alternating trees are grown together level by level. If two adjacent nodes are both assigned by two alternating trees (once as an even-level and another as an odd-level), a *bridge* is found. Two alternating paths are connected through such a bridge. Thus, an augmenting path may be formed through such a bridge to two free nodes. As soon as a bridge is found, the algorithm stops the growth of the alternating trees and searches for an augmenting path from that bridge. If no augmenting path is found, the extension process is resumed. If an augmenting path is found, the path is augmented. The algorithm starts another phase after having augmented all the augmenting paths. Since the algorithm executes a phase in O(m) time, it finds a maximum matching in $O(n^{1/2}m)$ time.

2.3 Review on Algorithms for Matching a Graph in a Distributed Environment

In this section, we review three existing distributed algorithms for finding a maximum-cardinality matching on a graph. Since, the first paper [SHAM 82] does not describe the algorithm at all, we will only describle the other two. Table 2.2 summarizes their complexities, where n is the number of nodes and m is the number of edges of the graph.

	Authors (year of paper)	Communication Complexity	Time Complexity
I	Shamir and Upfal (1982)	not derived	$O(\log^2 n)$ (Note 1)
II	Schieber and Moran (1986)	$O(n^2 m)$ and $O(n\ m \log n)$	$O(n \log n)$ (Note 2)
III	Wu (1987)	$O(n^{5/2})$	$O(n^{5/2})$

Table 2.2 Complexity of existing distributed maximum-matching algorithms

Notes : 1. For synchronous networks.

2. Not including complexity for synchronization.

I) Shamir and Upfal [SHAM 82] -

Shamir and Upfal's paper does not include any description of their algorithm. For example, there is no detail as to how to handle a blossom and choose the augmentation from two node-jointed augmenting paths. No communication complexity has been obtained. Instead, they prove that the maximum matching problem can be solved in $O(\log^2 n)$ time. Since so little detail is given, it is doubtful whether their result is correct or not.

II) Schieber and Moran [SCHI 86] -

Schieber and Moran assume that the network is asynchronous and that it is synchronized by a *synchronizer* [AWER 84]. They use [MICA 80]'s non-distributed algorithm as their algorithm's searching routine and modify it to fit into the distributed environment.

Their algorithm has two stages : search and update, and proceeds in phases. In each phase, it constructs a set of alternating trees each from a free node and detects a single minimum-length augmenting path. In the i^{th} phase, the search routine searches for an augmenting path whose length is bounded by l_i. At the beginning of phase i, each free node assigns itself as the root of an alternating tree. Each node in the alternating tree has a label which is the distance between the root and itself. The algorithm continues to send a pulse to every node in the net-

work. In pulse t, each one of the nodes whose label is t will advance the alternating path to the next level. During the extension of the alternating trees, the leap nodes of the alternating trees only send a forward labelling to its neighbors. In the forward labelling, the algorithm does not allow two visits to a node, so that the alternating paths are simple paths. However, some of the nodes which should be labelled are not labelled in the forward labelling. The algorithm labels these nodes in the backward labelling.

When an evenly (resp., oddly) labelled node receives a forward labelling in an oddly (resp., evenly) pulse, a *bridge* is formed. The evenly (resp., oddly) labelled node replaces the forward label by the backward label to continue the searching. The search continues until the backward labelling reaches another free node. The backtracking process will stop, if the alternating path is not simple or two backward labellings reach the same node. This technique allows to search an augmenting path without backtrack the blossoms.

Once a set of minimum-length augmenting paths are found, the search routine terminates. The update routine augments those augmenting paths serially (one path at a time). Thus, the routine only augments all the node-disjoint augmenting paths. After the updating, another iteration starts. If the search length of the alternating paths exceeds $n/2$, the algorithm terminates and the matching is maximum.

Their observation is that, unlike the non-distributed algorithm, a search for a single minimum-length augmenting path can be computed in a distributed environment faster than a search for a maximal set of minimum-length augmenting paths. The non-distributed algorithm needs O(m) time to find either one augmenting path or a maximal set of such paths. In the distributed environment, however, the search for augmenting paths can be made in parallel. Hence, an augmenting path of length l can be found in O(l) time. On the other hand, O(n) time would be required to find a maximal set of such paths. Based on this observation, they showed that O(n)

phases for finding one minimum-length augmenting path are faster than $O(n^{1/2})$ phases for finding a maximal set of such augmenting paths.

Their algorithm has time complexity O(n log n) which does not include the complexity of the synchronizer. The communication complexity is $O(n^2m)$, for the restricted-memory model where only a constant amount of storage is available at every node of the network, and O(n m log n) for the unrestricted-memory model.

III) Wu [WU 87] -

Wu's algorithm is asynchronous. The algorithm is based on the non-distributed matching algorithm from [MICA 80].

Before the algorithm starts, a minimum spanning tree (MST) is assumed on the graph. The root of the MST is assigned as the leader of the graph. The following only describes the synchronization technique of his algorithm. During the execution, the leader synchronizes all the operations by broadcasting a message to all the nodes through the MST. The leader will not perform the next step until all the nodes have sent back a return message. The operational details follows. Before a free node expands its alternating tree, it must receive the broadcasting message from the leader. After the free node finishes the expansion of a single level of its alternating tree, it must return a message to the leader. Thus, the leader of the MST can control the level of searching of all the alternating trees.

His algorithm also has two stages : search and update, and proceeds in phases. In each phase, every free node starts generating its own alternating tree to search for a bridge. If a bridge (i.e., an edge whose two nodes are the leaves of two different alternating trees) is found, the algorithm stops the phase and trys to find an augmenting path with a bridge. If an augmenting path is found, the algorithm increases the matching. Otherwise, it resume the phase. If any augment-

ing path is found during the current phase, a maximal set of node-disjoint minimum length augmenting paths is found, and it proceeds to the next phase.

In order to search for a bridge, each phase generates the alternating trees level by level. At level *i*, if a bridge is discovered (two leap nodes send a expand message to each other), the generation stops and it performs DFS from the nodes of the bridge to find an augmenting path. If no alternating path is found which contains a bridge, the generation proceeds to the next level. If an augmenting path is found, it updates the augmenting paths and begins a new phase.

The aigorithm continues to increase the matching until a maximum matching is obtained. It recognizes that the matching is maximum and halts when the level reaches a level which has no node. Only $O(n^{1/2})$ phases suffice to find a maximum matching. It has communication complexity $O(n^{5/2})$. It searches for augmenting paths in parallel but augments those augmenting paths in sequence. Thus, the time complexity of the algorithm can be as large as the message complexity, $O(n^{5/2})$.

Chapter 3

A DISTRIBUTED ALGORITHM FOR GENERATING A SPANNING MATCHING ON A GRAPH

In this chapter, we present a new distributed algorithm, called DSM, for finding a spanning matching on an undirected connected graph G(N,E). Remember that a matching M of a graph G is said to be *spanning* if no two adjacent nodes of G are both free with respect to M. DSM has communication complexity O(m) and time complexity O(n), where m is the number of edges and n is the number of nodes. We know of no other distributed algorithms in the literature for the same purpose.

Algorithm DSM

Given : An undirected connected graph G whose node identities form an ordered set and a designated node where the algorithm starts. The environment is based on the computational model described in Section 1.2.2.

Outcome : A spanning matching M on G, with each node in M being marked as 'matched' and storing the identity of its mate. The designated node d stores the identities of all the free nodes but has no knowledge of how the free nodes are interconnected in G.

Procedure : The procedure of DSM is described in three sections as follows : Section 3.1 describes the messages, parameters and functions used in DSM. Section 3.2 presents a global-view of DSM. Section 3.3 describes the distributed-view of DSM.

An example is given in Section 3.4. Section 3.5 shows the correctness and complexity analysis of DSM.

3.1 Messages, Parameters and Functions Used in Algorithm DSM

Messages sent by node k and received by node i :

In the following, M denotes the partial matching under expansion.

ECHO-A< k > : Here, A indicates 'Accept'. This message informs i that, in response to a MATE sent previously by i, k accepts the request to become a mate candidate of i. On receiving this message, i assigns the status 'outer' to k by sending an OUTER to k and the status 'inner' to the other 'not-tried-yet' neighbors (if any) by sending an INNER to each of them.

ECHO-F< k > : Here, F indicates 'Free'. This message informs i that, in response to a MATE sent previously by i, k rejects the request to become a mate candidate of i. However, k is still free and i can send a MATE to k later. Upon receiving this message, i sends a MATE to another 'not-tried-yet' neighbor (if any).

ECHO-S< k, Nodes > : Here, S indicates 'Stop' and Nodes is a set of free nodes. This message is sent by k according to two cases :

Case 1. In response to a MATE sent previously by i when i is in the inner status ('Trying to get a mate' state), this message informs i that k rejects the request to become a mate candidate of i and tells i to stop sending any more messages to k. k is marked as a 'tried' neighbor. i then sends a MATE to another 'not-tried-yet' neighbor (if any). If there are no more 'not-tried-yet' neighbors, i changes its status to 'free' and sends an ECHO-S to its parent.

Case 2. In response to either an INNER or an OUTER sent previously by i when i is in the 'Echoing' state, this message informs i that the expansion of M through k has been blocked (completed). After having received an ECHO-S from each of its neighbors, i changes its own status to 'matched' and sends an ECHO-S to its parent.

INIT: This message requests i to initiate Algorithm DSM.

INNER < k > : This message requests i to assume the status 'inner'. On receiving INNER, i reacts in four possible ways :

Case 1 (i is 'free') : i marks k as its parent. If i has no neighbor, i sends an ECHO-S to k. Otherwise, i assigns an 'inner' status to itself and tries to get a mate.

Case 2 (i already has a mate) : i rejects k's request by returning an ECHO-S to k.

Case 3 (i has an 'inner' status without a mate and the INNER is not sent by its mate nominator) : i rejects the request by returning an ECHO-S to k.

Case 4 (i has an 'inner' status without a mate and INNER is sent by its mate nominator) : i excludes k as a mate candidate and tries to get a mate from another neighbor.

MATE < k, Single > : This message requests i to be the mate candidate of k and Single is a boolean variable. Single has a value true if i is the only not-matched-yet neighbor of k; and false otherwise. On receiving MATE, i reacts in three possible ways :

Case 1 (i has a 'free' status) : i marks k as its parent and mate nominator. i accepts the request by returning an ECHO-A to k and waits for either an INNER or an OUTER from k.

Case 2 (i already has a mate) : i rejects the request by returning an ECHO-S to k.

Case 3 (i has a 'inner' status without a mate and Single has value false) : If the identity of i is bigger than the identity of k (assuming that the identities of the nodes form an ordered set) and k is not the mate candidate of i, i rejects the request by returning an ECHO-F to k. If the identity of i is smaller than the identity of k or k is the mate candidate of i, k is eligible as the mate nominator of i. If i already has a mate nominator, i rejects the request by returning an ECHO-F to k. If i has no such nominator, i accepts the request by returning an ECHO-A to k and marks k as its mate nominator. i then waits for either an INNER or an OUTER from k.

Case 4 (i has a 'inner' status without a mate and Single has value true) : i assumes that its identity is smaller than the identity of k and perform the above *Case 3*.

OUTER< k > : This message from the mate nominator k requests i to join M as an 'outer' node, the mate of k. On receiving this message, i assigns itself an 'outer' status. i changes its status to 'matched' and sends an ECHO-S to k if i has no neighbor; and sends an INNER to each neighbor otherwise.

Local variables used in the process at node i :

Candidate : The mate candidate of i.

Echoed(j) : This logical variable has value 'false' if i has sent to node j either an INNER or an OUTER message but has not received an ECHO-S from j. If this logical variable has value 'true', i has received an ECHO-S from j or i has not sent any message to j.

Found : This logical variable has value 'true' if i has found a neighbor who is willing to join M as the mate candidate of i. If this logical variable has value 'false', i continues to search for its mate.

Free : The identity of a set of free nodes. At the end of DSM, the starting node stores the identities of all the free nodes in Free.

Mate : The matching mate of i.

Neighbor : The set of adjacent nodes (in G) of i.

Nominator : The neighbor which nominates i (by sending a MATE message) as its mate candidate. At the beginning of DSM, Nominator is set to '0'.

Parent : The parent of i in M. The parent of the starting node is itself.

Sendto : The set of neighbors to which i sends either a MATE, an INNER or an OUTER message.

Status : Three cases :

a) Status has value 'free' if either i does not belong to M or i has been assigned as an 'inner' node but i has not been able to find a mate.

b) Status has value 'inner' if i is an inner node in M. At the end of DSM, 'inner' is changed to 'free' if i has no mate; and to 'matched' otherwise.

c) Status has value 'outer' if i is an outer node in M. At the end of DSM, 'outer' is changed to 'matched'.

The following variables are used only in Algorithm DMM (to be described in Chapter 4) but are initialized within DSM as follows. ST-Start to 'false'; Con-Exp to 'false'; Start-Node to '0'; ST-Echoed(j) to 'true' for all j in ST-Child; $Echoed_j(x)$ to 'true' for all x in Neighbor and $I\text{-}Path_j$ to $\varnothing$ and $O\text{-}Path_j$ to $\varnothing$ where $j = 1, 2$.

Function RESET(C.Status) used in the process :

This function initializes a set of variables as follows. They are used in Algorithm DMM only.

Status to 'C.Status'; Con-Change to 'false'; Con-Exp to 'false'; $Echoed_j(x)$ to 'true' for all x in Neighbor; $I\text{-}Path_j$ and $O\text{-}Path_j$ to $\varnothing$, where $j = 1, 2$; Start-Node to '0'; ST-Echoed(j) to 'true' for all j in ST-Child; ST-Start to 'false'.

3.2 Global-view Description of Algorithm DSM

Procedure : The spanning matching M is generated in the following way :

DSM is similar to an asynchronous distributed algorithm for pure graph traversal [CHAN 82]. Initially, M is empty, i.e., every node of G has a 'free' status. M continues to expand by adding the matched edges until it becomes a spanning matching. A designated node d starts DSM by assigning to itself the 'inner' status. In general, every node may be in a 'free', 'inner' or 'outer' status with respect to M. An 'outer' node is always matched. An 'inner' node is matched only if it has got a mate. An 'inner' node without a mate will try to get one by sending a MATE to its neighbors. (See Section 2.1 for more details of these terms.)

Note that DSM is an asynchronous algorithm. Every nodes acts independently and simultaneously, according to its status and the message it receives. Briefly (not a complete description), M expands in the following way.

Case 1 (node i has status 'free'). When free, i remains idle until it receives a message. If i receives an INNER (resp., OUTER), it changes its own status to 'inner' (resp., 'outer'). If i receives a MATE nomination, it may accept or reject the request.

Case 2 (node i has status 'inner'). i nominates k, one of its neighbors, as its mate candidate by sending a MATE to k. If k accepts the nomination (by returning an ECHO-A), (i,k) becomes matched. i then extends to its other neighbors by sending an INNER to each of them. If k rejects the nomination (by returning either an ECHO-S or an ECHO-F), i nominates another neighbor and this step will be repeated. When all of its neighbors have returned an ECHO-S, if i has found a mate, its status is changed to 'matched'; otherwise, it is changed to 'free'. Then, i sends an ECHO-S to its parent.

Case 3 (node i has status 'outer'). Once in this status, i extends M by sending an INNER to all its neighbors except its mate. When each of these neighbors has returned an ECHO-S, i changes its status to 'matched' and sends an ECHO-S to its parent.

DSM terminates when the designated node d receives the ECHO-S messages from all its neighbors. The ECHO-S message carries the identities of the free nodes to d. Hence, d knows the identities of all the free nodes in M. At termination of DSM, G contains a spanning matching M and the status of each node is either 'free' or 'matched'. It is impossible for two neighboring nodes to be both free.

The above description is oversimplified. In fact, the following two problems may arise:

Mating Problem I: Consider three not-matched-yet 'inner' nodes x_1, x_2, and x_3. In the process of nominating and accepting mates, the following situation may arise: x_2 has sent a MATE to x_3 (nominating x_3 as its mate) and x_3 has accepted the nomination by returning an ECHO-A. But, before receiving this response from x_3, x_2 receives a MATE from x_1 and thus responds with an ECHO-A. If the ECHO-A is considered as a confirmation of matching, x_2 will become the mate of both x_1 and x_3.

To avoid such an error, matching has to go through the following confirmation stage in addition to the nomination and acceptance stages mentioned above:

Confirmation stage: When x_2 receives an ECHO-A from x_3, x_2 will respond as follows:

If x_2 has not sent any ECHO-A to other nodes, x_2 sends an OUTER to x_3 and the edge (x_2,x_3) becomes matched.

If x_2 has sent an ECHO-A to x_1, say, x_2 will wait for a response from x_1. If the response is an OUTER (i.e., acceptance is confirmed), the edge (x_1,x_2) becomes matched and x_2 will reject x_3's acceptance by sending an INNER. If the response is an INNER (i.e., acceptance is rejected), x_2 will confirm x_3's acceptance by sending an OUTER and the edge (x_2,x_3) becomes matched.

Mating Problem II: Suppose S is a pure loop (i.e., $x_1 = x_n$ and it is impossible to branch out from the loop to form another alternating path). The three-stage augmentation method described above will create a loop if the following phenomenon arises : Every x_i sends a MATE request to its neighbor x_{i+1} and, before receiving any response message from x_{i+1}, receives a MATE request from x_{i-1}. (See Figure 3.1a.) Since x_i has not accepted any mate request, it ac-

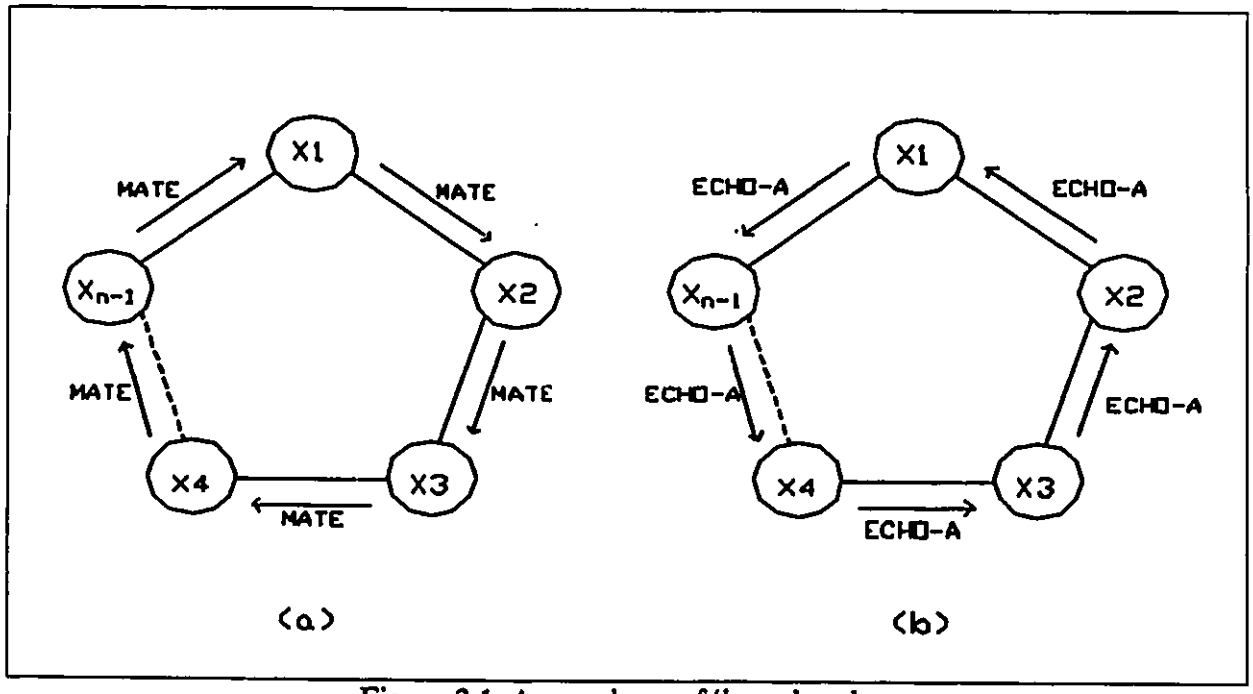


Figure 3.1. A pure loop of 'inner' nodes.

cepts the mate request and returns an ECHO-A to x_{i-1}. (See Figure 3.1b.) Hence, no x_i will become matched because every x_i is waiting for the confirmation from x_{i-1}.

This problem can be solved as follows : On receiving a MATE request from node y, a node x will respond with ECHO-A if ident(y) > ident(x) and with ECHO-F if ident(y) < ident(x). This will split S into several chains of the form $C = \{x_j\}, j = 1, 2, \ldots m$, where $x_1 \neq x_m$, ident(x_j) > ident(x_{j+1}), and x_1 has not sent ECHO-A. Each chain can be solved as a case of Mating Problem I. As a result, x_1 can confirm x_2's acceptance (by message OUTER) and becomes matched with x_2.

3.3 Distributed-view Description of Algorithm DSM

A distributed-view description includes a detailed explanation of the movements of the messages in the form of a pseudo-code.

Pseudo-code of Algorithm DSM

Note : At termination, the identities of those nodes which remain free are stored in the variable Free of the designated node.

Initialization :

Each node initializes its variables as follows : Status to 'free', Nominator to '0' and Echoed(x) to 'true' for every x in Neighbor. To start DSM, message INIT is sent to a designated node for initiation.

The process at node i :

```
{* On receiving message INIT *}
Sendto := Neighbor;
if Sendto = ∅ → Completed;
[] Sendto ≠ ∅ → Status := 'inner'; Mate := '0'; Parent := i; Found := 'false'; Free := ∅;
                Candidate := an arbitrary element of Sendto;
                send MATE< i, false > to Candidate;                          {* 1 *}
fi

{* On receiving message MATE< k, Single > *}
if Status = 'outer' →   send ECHO-S< i, Free > to k;                         {* 2 *}
[] Status = 'inner' →   if Mate ≠ '0' → send ECHO-S< i, Free > to k;          {* 3 *}
                        [] Mate = '0' →
```

```
                    if i > k and Candidate ≠ k and Single = 'false' →
                                send ECHO-F< i > to k;                                  {* 4 *}
                    [] i < k or  Candidate = k  or  Single = 'true' →
                                if Nominator ≠ '0' →  send ECHO-F< i > to k;            {* 5 *}
                                [] Nominator = '0' →  Nominator := k;
                                                      send ECHO-A< i > to k;            {* 6 *}
                                fi
                    fi
                fi
[] Status = 'free' →  if Nominator ≠ '0' →  send ECHO-F< i > to k;                      {* 7 *}
                      [] Nominator = '0' →  Nominator := k; Parent := k; Free := ∅;
                                            send ECHO-A< i > to k;                      {* 8 *}
                      fi
fi

{* On receiving message OUTER< k > *}
Mate := k; Candidate := '0';
if Status = 'inner' → Sendto := Sendto - { k }; send ECHO-S< i, Free > to k;           {* 9 *}
[] Status = 'free'  → Status := 'outer'; Sendto := Neighbor - Parent;                  {* 10 *}
fi
if Sendto = ∅ → RESET( 'matched' );   send ECHO-S< i, Free > to Parent;                {* 11 *}
[] Sendto ≠ ∅ → for each x in Sendto do
                  [ Echoed( x ) := 'false'; send INNER< i > to x ]                     {* 12 *}
fi

{* On receiving message INNER< k > *}
if Status = 'outer' → send ECHO-S< i, Free > to k;                                     {* 13 *}
[] Status = 'inner' → Sendto := Sendto - { k }; send ECHO-S< i, Free > to k;           {* 14 *}
    if Nominator = k →  if Found = 'true' →
                            Mate := Candidate; Candidate := '0';
                            for each x in Sendto do
                            [   Echoed( x ) := 'false';
                                if x = Mate → send OUTER< i > to x;                    {* 15 *}
                                [] x ≠ Mate → send INNER< i > to x;                    {* 16 *}
                                fi   ]
                        [] Found = 'false' →
                            Nominator := '0';
                            if Sendto = ∅ → RESET( 'free' ); Free := Free ∪ {i};
                                            send ECHO-S< i, Free > to Parent;          {* 17 *}
                            [] Sendto ≠ ∅ → Candidate := an arbitrary element of Sendto;
                                            if Sendto - Candidate = ∅ →
                                              send MATE< i, true > to Candidate;       {* 18 *}
                                            [] Sendto - Candidate ≠ ∅ →
                                              send MATE< i, false > to Candidate;      {* 19 *}
                                            fi
```

```
                                        fi
                                fi
    [] Nominator ≠ k →    Skip                                                    {* 20 *}
    fi
[] Status = 'free' →
    if Nominator ≠ '0' and Nominator ≠ k → send ECHO-S< i, Free > to k;           {* 21 *}
    [] Nominator = '0'  or  Nominator = k →
                        Status := 'inner'; Mate := '0'; Parent := k; Free := ∅;
                        Nominator := '0'; Found := 'false'; Sendto := Neighbor - Parent;
                        if Sendto = ∅ → RESET( 'free' ); Free := Free ∪ {i};
                                        send ECHO-S< i, Free > to Parent;       {* 22 *}
                        [] Sendto ≠ ∅ → Candidate := an arbitrary element of Sendto;
                                        if Sendto - Candidate = ∅ →
                                            send MATE< i, true > to Candidate;  {* 23 *}
                                        [] Sendto - Candidate ≠ ∅ →
                                            send MATE< i, false > to Candidate; {* 24 *}
                                        fi
                        fi
    fi
fi

{* On receiving message ECHO-A< k > *}
Found := 'true';
if Nominator ≠ '0' and Nominator ≠ k → Skip;                                    {* 25 *}
[] Nominator = '0' →  Mate := Candidate; Candidate := '0';
                      for each x in Sendto do
                      [   Echoed( x ) := 'false';
                          if x = Mate → send OUTER< i > to x;                   {* 26 *}
                          [] x ≠ Mate → send INNER< i > to x;                   {* 27 *}
                          fi    ]
[] Nominator = k →    Mate := k; Candidate := '0'; Sendto := Sendto - { k };
                      if Sendto = ∅ → RESET( 'matched' );
                                      send ECHO-S< i, Free > to Parent;         {* 28 *}
                      [] Sendto ≠ ∅ → for each x in Sendto do
                                      [ Echoed( x ) := 'false'; send INNER< i > to x ]   {* 29 *}
                      fi
fi

{* On receiving message ECHO-F< k > *}
if Nominator ≠ '0' →  Skip;                                                     {* 30 *}
[] Nominator = '0' →  Candidate := an arbitrary element of Sendto;
                      if Sendto - Candidate = ∅ → send MATE< i, true > to Candidate;   {* 31 *}
                      [] Sendto - Candidate ≠ ∅ → send MATE< i, false > to Candidate;  {* 32 *}
                      fi
fi
```

```
{* On receiving message ECHO-S< k, Nodes > *}
Sendto := Sendto - { k }; Echoed( k ) := 'true'; Free := Free ∪ Nodes;
if Candidate = k →    if Nominator ≠ '0' → Skip;                                  {* 33 *}
                      [] Nominator = '0' →
                           if Sendto = ∅ → RESET( 'free' ); Free := Free ∪ {i};
                                           send ECHO-S< i, Free > to Parent;      {* 34 *}
                           [] Sendto ≠ ∅ → Candidate := any element of Sendto;
                                           if Sendto - Candidate = ∅ →
                                                send MATE< i, true > to Candidate; {* 35 *}
                                           [] Sendto - Candidate ≠ ∅ →
                                                send MATE< i, false > to Candidate;{* 36 *}
                                           fi
                           fi
                      fi
[] Candidate ≠ k →    if Not Echoed( j ) for some j in Neighbor → Skip;            {* 37 *}
                      [] Echoed( j ) for all j in Neighbor → RESET( 'matched' );
                                 if Parent = i → Completed;                       {* 38 *}
                                 [] Parent ≠ i → send ECHO-S< i, Free > to Parent; {* 39 *}
                                 fi
                      fi
fi
```

Special situation :

Consider that two not-matched-yet 'inner' nodes x and y where the ident(x) > ident(y), y is the last member in x's ordered neighbor set, and x is the centre of a star network (x connects to all the nodes in the network). y has only two neighbors, its own parent and x. It may happen that y sends a mate request (by message MATE) to x. x has nominated another node. Since the ident(x) > ident(y), x will return a not-acceptance (by message ECHO-F) to y. y will repeat to nominate x because y has only one not-matched-yet neighbor x. x will keep on returning 'not-acceptance' to y. Eventually, if x cannot get a mate from its other neighbors, x and y nominate each other and become a matched pair. Though, this problem does not cause any error, it causes a lot of overhead messages.

This problem is solved as follows : y sends a mate request with the value true in the parameter Single (by message MATE(y,true)) to x because y has only one not-matched-yet neighbor x. x

assumes that ident(x) < ident(y) and continues the Mating process. x returns an acceptance (by message ECHO-A) to y if x has not committed with another node; x returns a not-acceptance (by message ECHO-F) to y, otherwise.

3.4 An Example

In this section, we give an example on applying DSM to find a spanning matching on a graph. In particular, it shows how a loop of inner nodes having received MATE messages (Node a sends MATE<a,false> to Node c; c sends MATE<c,false> to Node b; b sends MATE<b,false> to a) is resolved (see Events 9, 10, and 11).

Figure 3.2 shows the initial undirected graph without any matching. Figure 3.3 shows the events and the messages at an intermediate stage. Figure 3.4 shows the spanning matching obtained at the end. Table 3.1 lists the sequence of events. At the beginning, every node initializes its local variables as follows : Status = 'free'; Nominator = '0'; Echoed(j) = 'true', for every j in Neighbor.

Note : In Table 3.1, each column describes the occurrence of one event. The event numbers indicate a temporal order in which events occur at the same node, but not necessarily for events at different nodes. Also, 'reset local variables' means 'execute the function RESET'; 'α/β/γ' means 'α or β or γ'; and 'α,β,γ' means 'α and β and γ'.

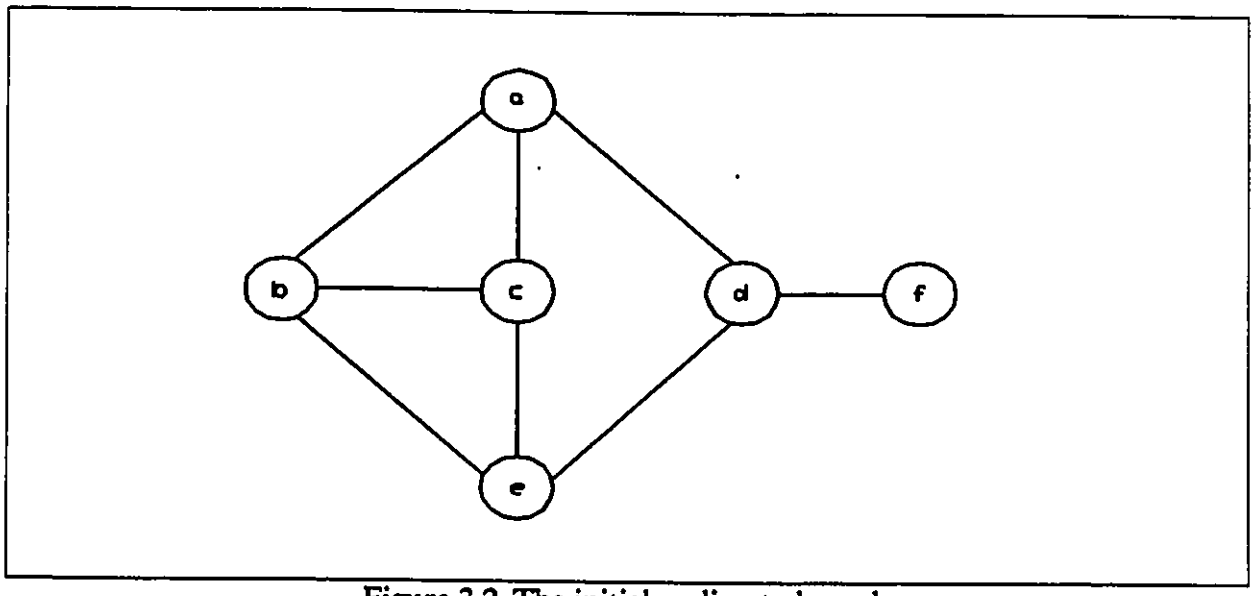


Figure 3.2. The initial undirected graph.

A	Event number	1	2	3
B	Node where event occurs	e (Designated node)	d	e
C	Message received	INIT	MATE<e,false>	ECHO-A<d>
D	Values of decision variables just before receiving message		Status = 'free' Nominator = '0'	Nominator = '0'
E	Actions and conditions after receiving the last (if more than one) possible input message as listed in C. {* x *} refers to the statement number in Algorithm DSM	Sendto = {d,c,b} Status := 'inner' Mate := '0' Parent := e Found := 'false' Free := ∅ Candidate := d send MATE<e,false> to d {*1*}	Nominator := e Parent := e Free := ∅ send ECHO-A<d> to e {*8*}	Found := 'true' Mate := d Candidate := '0' Echoed(d,c,b) := 'false' send OUTER<e> to d {*26*} send INNER<e> to c,b {*27*}
F	Next event number (s)	2	3	4, 6, 7

A	4	5	6	7
B	d	a	c	b
C	OUTER<e>	INNER<d>	INNER<e>	INNER<e>
D	Status = 'free'	Status = 'free' Nominator = '0'	Status = 'free' Nominator = '0'	Status = 'free' Nominator = '0'
E	Mate := e Candidate := '0' Status := 'outer' Sendto := {a,f} {*10*} Echoed(a,f) := 'false' send INNER<d> to a,f {*12*}	Status := 'inner' Mate := '0' Parent := d Free := ∅ Nominator := '0' Found := 'false' Sendto := {c,b} Candidate := c send MATE<a,false> to c {*24*}	Status := 'inner' Mate := '0' Parent := e Free := ∅ Nominator := '0' Found := 'false' Sendto := {a,b} Candidate := b send MATE<c,false> to b {*24*}	Status := 'inner' Mate := '0' Parent := e Free := ∅ Nominator := '0' Found := 'false' Sendto := {a,c} Candidate := a send MATE<b,false> to a {*24*}
F	5, 8	9	10	11

Table 3.1.

A	8	9	10	11
B	f	c	b	a
C	INNER < d >	MATE < a,false >	MATE < c,false >	MATE < b,false >
D	Status = 'free' Nominator = '0'	Status = 'inner' Mate = '0' c > a and Candidate = b ≠ a	Status = 'inner' Mate = '0' b < c Nominator = '0'	Status = 'inner' Mate = '0' a < b Nominator = '0'
E	Status : = 'inner' Mate : = '0' Parent : = d Free : = Ø Status : = 'free' Nominator : = '0' Found : = false Sendto : = Ø reset local variables Free : = Free ∪ {f} send ECHO-S < f,{f} > to d {*22*}	send ECHO-F < c > to a {*4*}	Nominator : = c send ECHO-A < b > to c {*6*}	Nominator : = b send ECHO-A < a > to b {*6*}
F	17	12	13	15

A	12	13	14	15
B	a	c	b	b
C	ECHO-F < c >	ECHO-A < b >	OUTER < c >	ECHO-A < a >
D	Nominator = b ≠ '0'	Nominator = '0'	Status = 'inner'	Nominator = c ≠ '0' Nominator = c ≠ a
E	Skip {*30*}	Found : = 'true' Mate : = b Candidate : = '0' Echoed(a,b) : = 'false' send OUTER < c > to b {*26*} send INNER < c > to a {*27*}	Mate : = c Candidate : = '0' Sendto : = {a} send ECHO-S < b,Ø > to c {*9*} Echoed(a) : = 'false' send INNER < b > to a {*12*}	Found : = 'true' Skip {*25*}
F	End of branch	14, 16	18, 16	End of branch

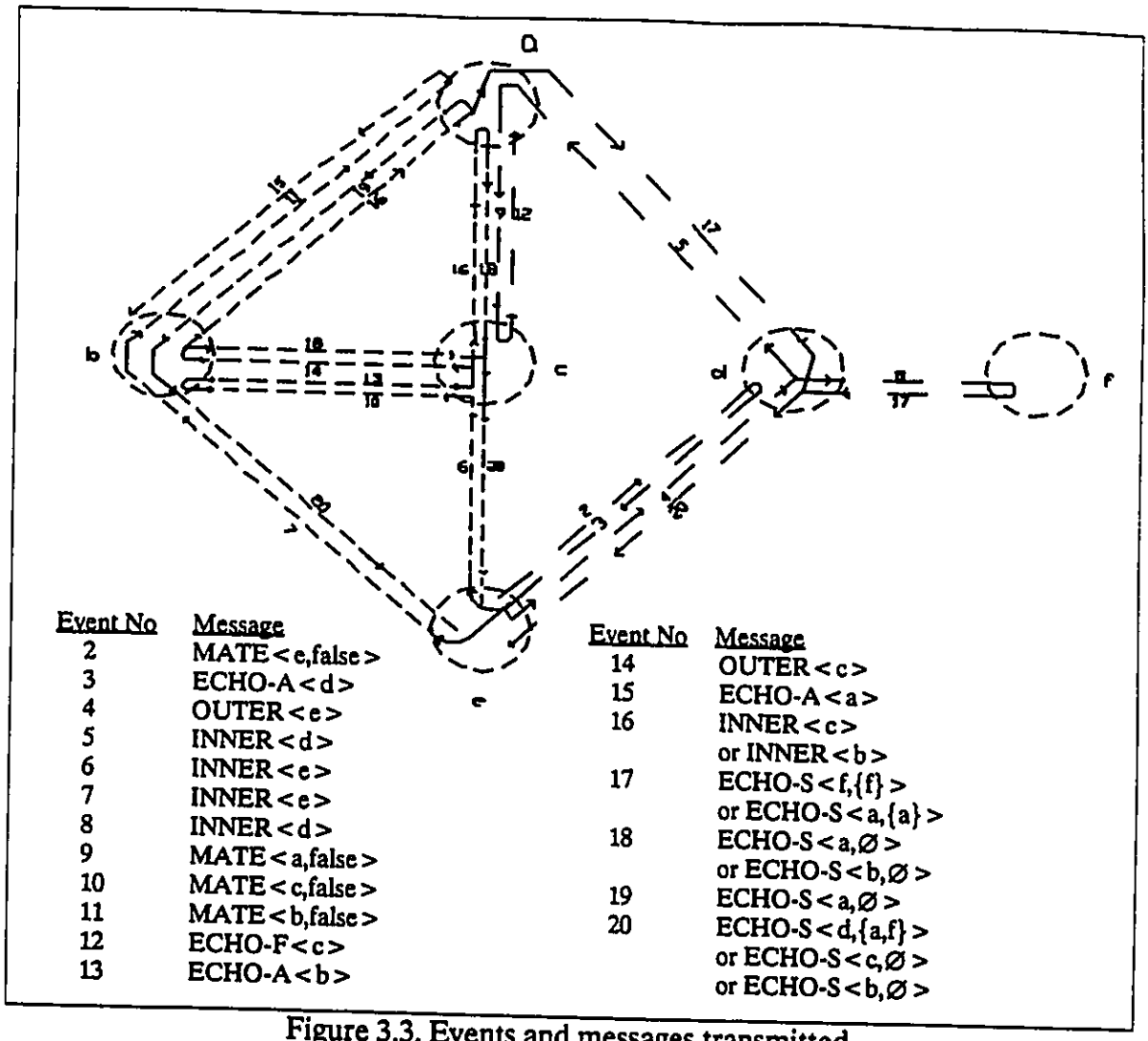
Table 3.1. (Continued)

Assumptions : For event 16, message INNER < c > arrives at node a eariler than message INNER < b >. For events 17, 18 and 20, the messages do not arrive at the same time. The steps show only the results after that node receives all the messages.

A	16	17	18	19
B	a	d	c	b
C	INNER<c>/ INNER<b>	ECHO-S<f,{f}>/ ECHO-S<a,{a}>	ECHO-S<a,∅>/ ECHO-S<b,∅>	ECHO-S<a,∅>
D	Status = 'inner' Nominator = b ≠ c Found = 'false'	Candidate = '0' ≠ f/a Parent ≠ d	Candidate = '0' ≠ a/b Parent ≠ c	Candidate = '0' ≠ a Parent ≠ b
E	Sendto := ∅ send ECHO-S<a,∅> to c/b {*14*} Nominator := '0' Status := 'free' reset local variables Free := Free ∪ {a} send ECHO-S<a,{a}> to d {*17*}	Sendto := ∅ Echoed(f/a) := 'true' Free := { a, f } Status := 'matched' reset local variable send ECHO-S <d,{a,f}> to e {*39*}	Sendto := ∅ Echoed(a/b) := 'true' Free := ∅ Status := 'matched' reset local variable send ECHO-S<c,∅> to e {*39*}	Sendto := ∅ Echoed(a) := 'true' Free := ∅ Status := 'matched' reset local variables send ECHO-S<b,∅> to e {*39*}
F	18/ 19, 17	20	20	20

A	20
B	e
C	ECHO-S<d,{a,f}>/ ECHO-S<c,∅>/ ECHO-S<b,∅>
D	Candidate = '0' ≠ d/c/b Parent = e
E	Sendto := ∅ Echoed(d/c/b) := 'true' Free := { a, f } Skip/(Status := 'matched' reset local variables Completed) {*38*}
F	Skip / Termination

Table 3.1. (Continued)



Event No	Message
2	MATE<e,false>
3	ECHO-A<d>
4	OUTER<e>
5	INNER<d>
6	INNER<e>
7	INNER<e>
8	INNER<d>
9	MATE<a,false>
10	MATE<c,false>
11	MATE<b,false>
12	ECHO-F<c>
13	ECHO-A<b>

Event No	Message
14	OUTER<c>
15	ECHO-A<a>
16	INNER<c> or INNER<b>
17	ECHO-S<f,{f}> or ECHO-S<a,{a}>
18	ECHO-S<a,∅> or ECHO-S<b,∅>
19	ECHO-S<a,∅>
20	ECHO-S<d,{a,f}> or ECHO-S<c,∅> or ECHO-S<b,∅>

Figure 3.3. Events and messages transmitted.

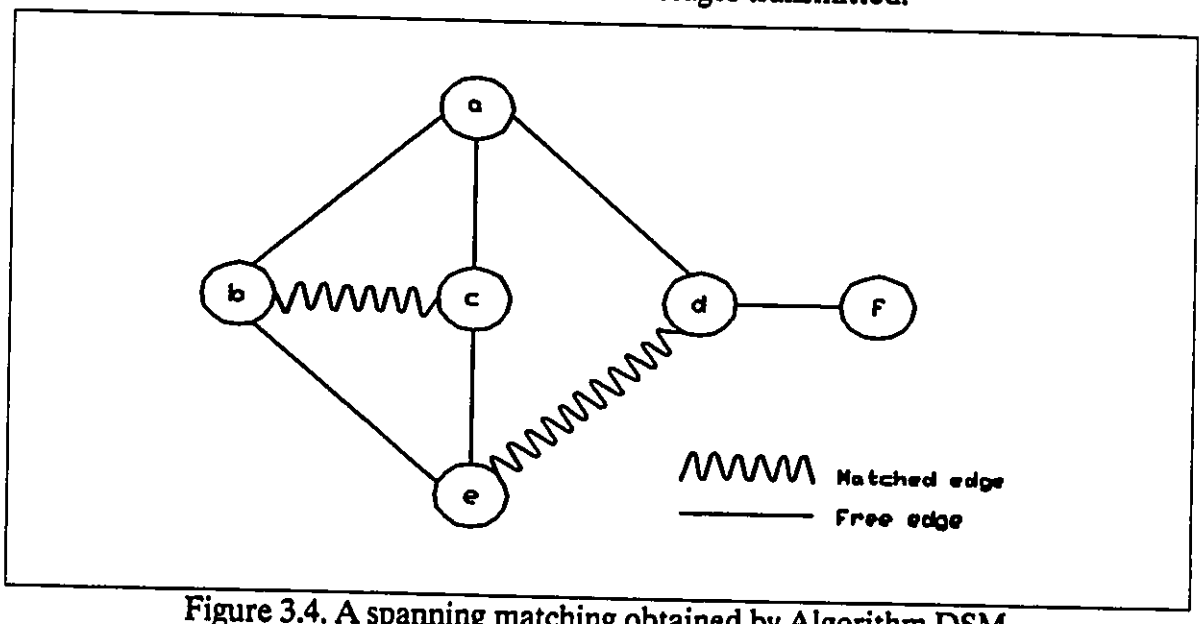


Figure 3.4. A spanning matching obtained by Algorithm DSM.

3.5 Correctness and Complexity of DSM

Theorem 3.1. Algorithm DSM correctly generates a spanning matching on a connected undirected graph.

Proof. (In the following, (* x1, x2, ... *) refers to statements x1, x2, ... in Algorithm DSM.) To prove the correctness of Algorithm DSM, we have to show three facts :

a) A matching M has been generated, i.e., no node becomes the mate of two neighbors. To prove this, we have to make sure that the Mating problem I described in Section 3.1 does not occur. In fact, three stages proposed in that section for solving this problem are achieved by a combined use of the messages MATE, ECHO-A, ECHO-F, INNER and OUTER.

b) The generated matching M is spanning, i.e., no two neighbors are both free.

First, it is obvious that every node of graph G will be reached and become either 'inner' or 'outer' at least once. If it becomes outer, it becomes matched forever. When it becomes inner, it issues a MATE (* 23/ 24 *) to all its neighbors. It follows that a matched edge should have been created with one of these free neighbors.

c) The algorithm will terminate.

To prove this, we have to make sure that the Mating problem II described in Section 3.1 does not occur. In fact, a tie in a loop of mating nominations is broken by imposing a priority on the candidates' identities (* 4, 5, 6 *). At the end, if a node has no non-parent neighbor or has received an ECHO-S from all its non-parent neighbors, it sends an ECHO-S (* 34, 39 *) to its parent. The starting node eventually receives ECHO-S from all of its neighbors. Then DSM terminates. ♦

Theorem 3.2. The communication complexity of Algorithm DSM is O(m), where m is the number of edges of the graph.

Proof. We shall show that, along each edge (x,y), at most 4 messages are transmitted starting from x. Depending on the status of x and y, four cases can be distinguished (Note that, since a free node never initiates any message, x cannot be free) :

a) (x - inner; y - free) Consider two sub-cases:

i. (x already has a mate) : x sends an INNER to y and y eventually returns an ECHO-S to x.

ii. (x does not have a mate) : x sends a MATE to y and y immediately returns an ECHO-A to x. Then, x sends either an INNER or an OUTER to y and y eventually returns an ECHO-S to x.

b) (x - inner; y - inner) Consider four sub-cases:

i. (both x and y have no mate) : x sends a MATE to y and y immediately returns an ECHO-A to x. Then, x sends an OUTER to y and y immediately returns an ECHO-S to x.

ii. (x has a mate but y has no mate) : x sends an INNER to y and y eventually returns an ECHO-S to x.

iii. (x has no mate but y has a mate) : x sends a MATE to y and y immediately returns an ECHO-S to x.

iv. (x and y both have a mate) : x sends an INNER to y and y immediately returns an ECHO-S to x.

c) (x - inner; y - outer) : x sends a MATE or an INNER to y and y immediately returns an ECHO-S to x.

d) (x - outer; y - any status) : x sends an INNER to y and y (eventually or immediately) returns an ECHO-S to x.

Thus, the total number of messages transmitted along each edge (including both directions) is smaller than or equal to 8. ♦

Theorem 3.3. The time complexity of Algorithm DSM is O(n), where n is the number of nodes of the graph.

Proof. For time complexity, the worst case occurs if the network is a linear structure. On each matched edge, 4 messages (MATE, ECHO-A, OUTER and ECHO-S) have been exchanged; and, on each free edge, 2 messages (INNER and ECHO-S) have been exchanged. Thus, the time complexity of DSM is bounded by O(n). ♦

Chapter 4

A DISTRIBUTED ALGORITHM FOR FINDING A MAXIMUM MATCHING ON A GRAPH

In this chapter, we present a new distributed algorithm, called DMM, for finding a maximum-cardinality matching on a graph over an asynchronous network. DMM is simple in comparison with other existing distributed algorithms because it does not have to shrink the blossoms and trace back the alternating paths and bridges. Instead, it makes use of the simple pure traversal technique [CHAN 79, CHAN 82] for searching augmenting paths and the idea of acceptance and commitment in the augmentation process. DMM has communication complexity O(nm) and time complexity $O(n^2)$, where m is the number of edges and n is the number of nodes.

It is assumed that, before DMM is initiated, the graph already has a matching (possibly the spanning matching obtained by Algorithm DSM of Chapter 3) and a spanning tree ST (possibly obtained by using the distributed algorithms of Zhu and Cheung [Zhu 87], Gallagher [GALL 83] or Chang [CHAN 79]). Note that, with a slight modification, DSM can simultaneously generate a minimum spanning tree (MST) on the graph.

Algorithm DMM

Given : A graph with any matching and a spanning tree ST. The root of ST contains the identities of all the free nodes of the matched graph. The environment is based on the computational model described in Section 1.2.2.

Outcome : A maximum matching represented in the following way : The status of each node is either 'free' or 'matched'. Each matched node stores the identity of its mate.

Procedure : The procedure of DMM is described in four sections as follows : Section 4.1 describes the basic ideas and terminology. Section 4.2 describes the messages, parameters and

functions which are used in DMM. Section 4.3 gives a global-view of DMM. Section 4.4 describes the distributed-view of DMM.

An example is given in Section 4.5. Section 4.6 shows the correctness and complexity analysis of DMM.

4.1 Virtual Alternating Trees

We have to extend the concept of an alternating tree described in Section 2.1 to a *virtual alternating tree* (VAT). A VAT starts from a free node as its root. Like an alternating tree, VAT consists of a set of alternating paths starting at the root. 'Virtual' here refers to the fact that some of the nodes of the tree, though logically different, are physically the same. This means that its paths may cross one another. However, the following rules must be satisfied in a VAT:

Rule 1. Free nodes can occur only at the end of an alternating path.

Rule 2. A node cannot be included more than once in the same alternating path.

Rule 3. A node may be included in at most two alternating paths of the same VAT, as an inner node in one and as an outer node in the other.

Figure 4.1 shows a VAT. The alternating path (r,a3,b3,c3,d3) cannot be extended to a3 or b3, because a3 and b3 are in the same alternating path (Rule 2). Also, c2, as an inner node, cannot be included in both (r,a2,b2,c2,d2,...) and (r,a1,b1) (Rule 3). However, d1 can be included in (r,a1,b1,c1,d1,d2,c2,b2,a2) as an 'outer' node and in (r,a2,b2,c2,d2,d1,c1,b1,a1) as an 'inner' node.

According to the above rules, it may happen that, if only one VAT is allowed on a graph, then when expanding its alternating paths from the root, all of them may be blocked before reaching any free node. As a consequence, an existing augmenting path will not be discovered. This problem can be overcome if several VAT's are allowed to be generated and the nodes are

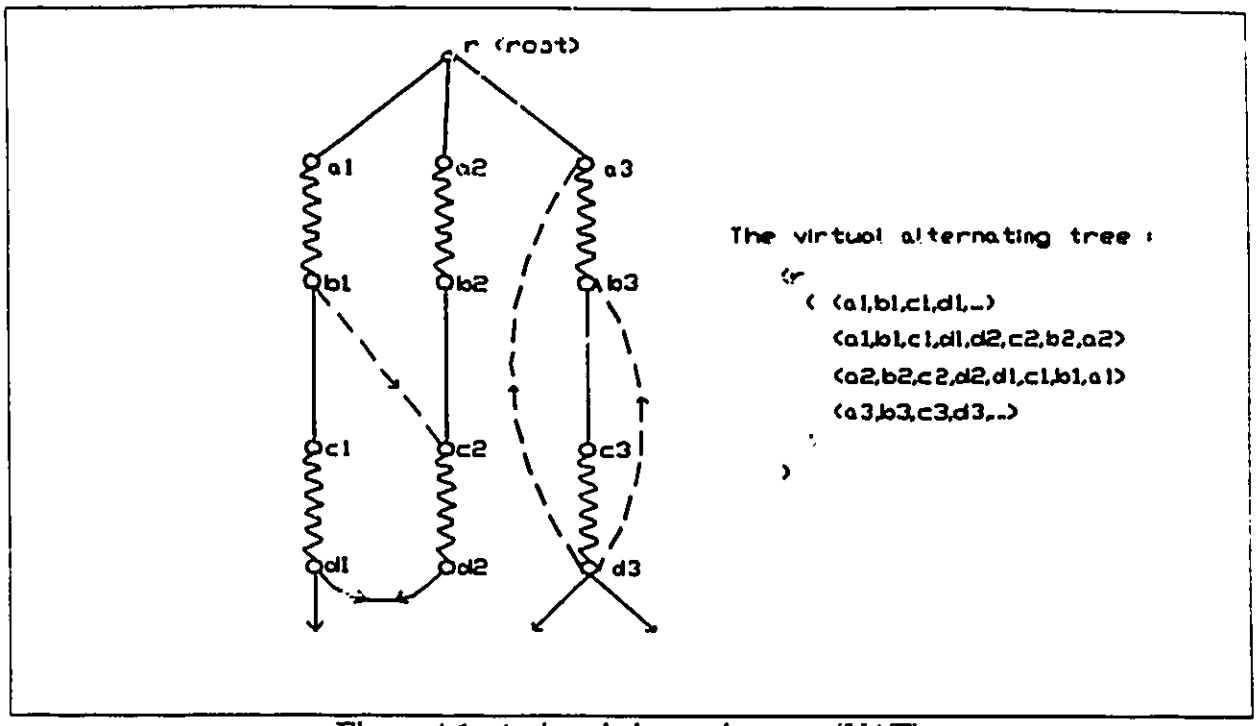


Figure 4.1. A virtual alternating tree (VAT).

shared (physically but not logically) by different VAT's. The objective is to guarantee that, if an augmenting path cannot be found through one VAT, it will be discovered through another. Figure 4.2 shows such an example. The graph has an augmenting path p = (r,a,b,c,d,e,f,g,h,i). However, the search based on the VAT rooted at the free node r fails to reach the free node i, because 'extension of' the alternating path (r,a,b) is blocked at c by a previously generated alternating path (r,h,g,c,d,...) (Rule 3), the alternating path (r,h,g,c,d,e,f) is blocked at g (Rule 2), and the alternating path (r,h,g,f,e,d,c,b,a) is blocked at r (Rule 2). However, another VAT containing the alternating path starting at i finds the augmenting path (i,h,g,f,e,d,c,b,a,r) in the reverse order of p.

Figure 4.3 shows that some nodes are included in several paths of different VAT's. For example, node i is used in all three alternating paths : (a,c,d,i,k), (f,b,e,i,k), and (g,j,h,i,k).

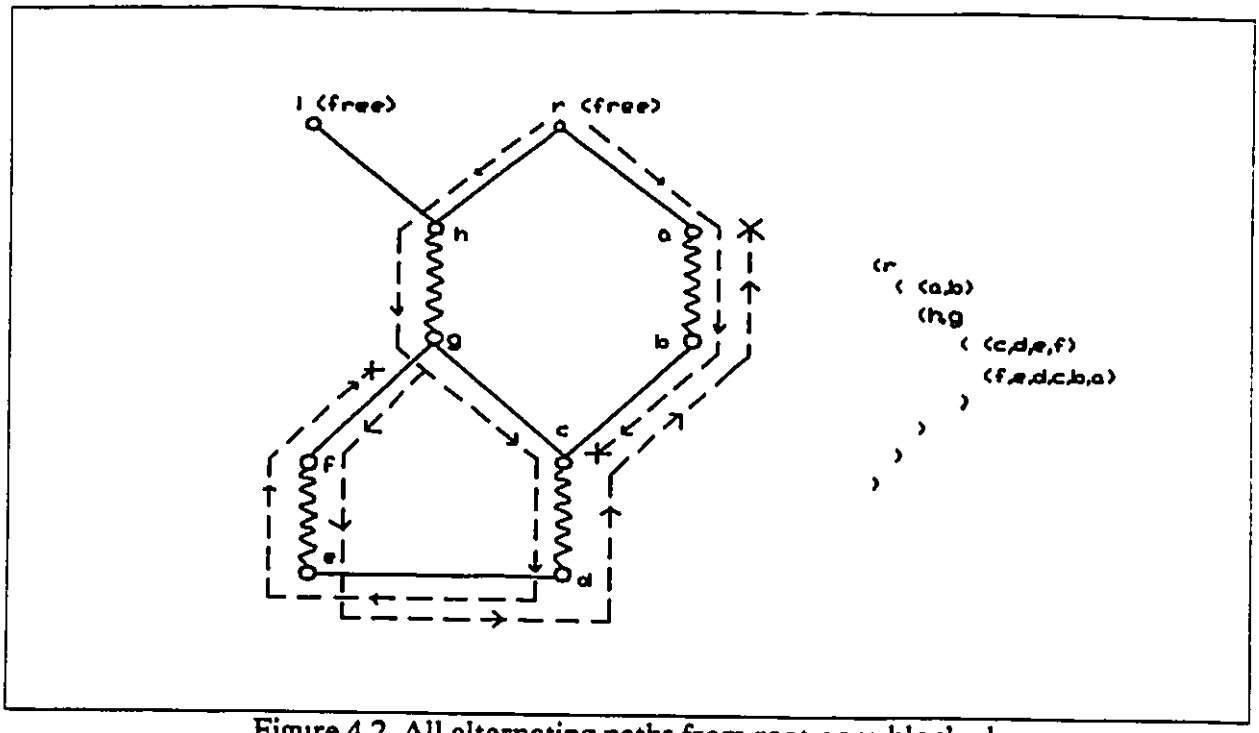


Figure 4.2. All alternating paths from root r are blocked.

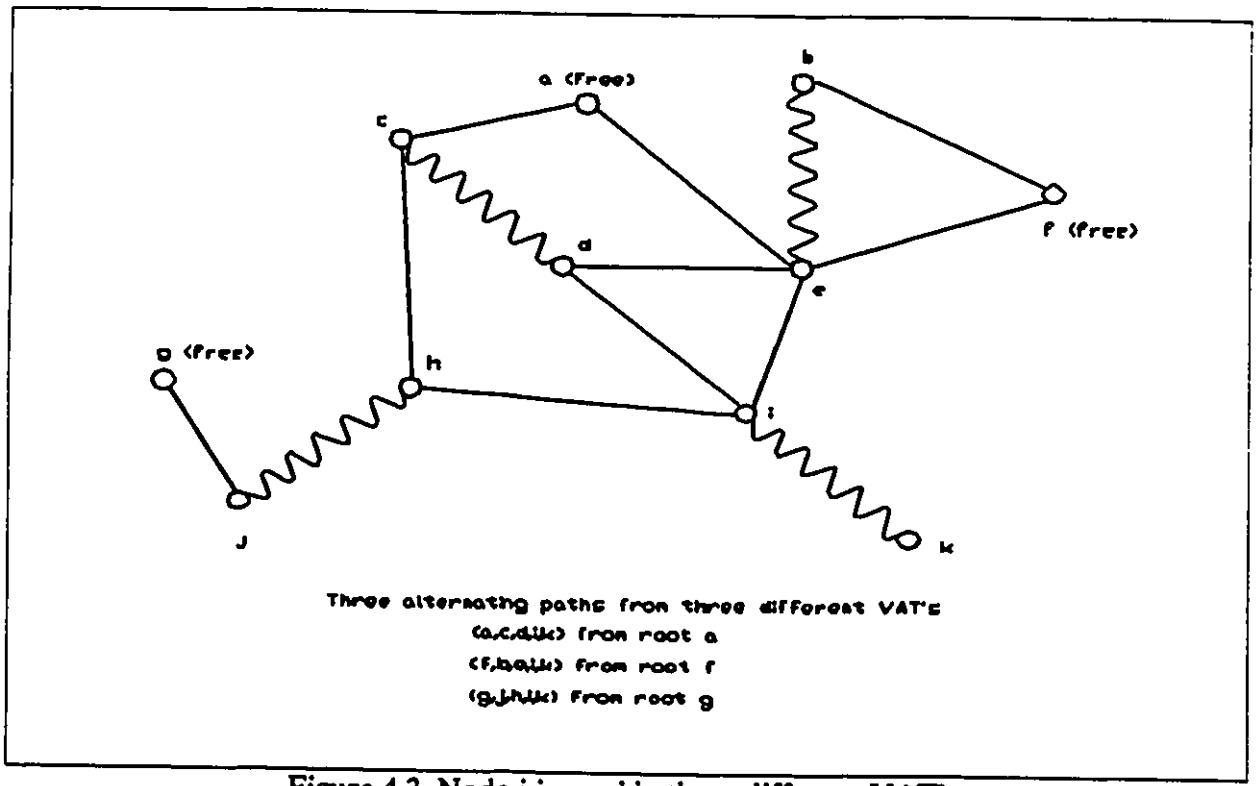


Figure 4.3. Node i is used in three different VAT's.

However, if a node allows too many alternating paths to pass through, the total number of paths may become explosive and a huge number of messages will be transmitted. To alleviate this problem, we impose the following additional rule on generation of VAT's :

Rule 4. A node can be involved as an 'inner' node in at most two alternating paths from two different VAT's.

4.2 Messages, Parameters and Functions Used in Algorithm DMM

Messages sent by node k and received by node i :

AUGFOUND < k, Path > : AUGFOUND informs i that 'AUGmenting path 'Path' has been FOUND'. This message is sent from the child k of i in the Path, in response to an EXPAND sent by i. i may receive an AUGFOUND in the following cases:

Case 1 (i has a 'matched' status) : i may receive this AUGFOUND in three sub-cases:

case 1.1 (k is the mate of i) : i sends this message to the parent of i in the Path.

case 1.2 (k is not the mate of i and i has not received an AUGFOUND before) : i sends this message to its mate.

case 1.3 (k is not the mate of i and i has received an AUGFOUND before) : i ignores this AUGFOUND.

Case 2 (i has a 'free' status) : i may receive this AUGFOUND in two sub-cases:

case 2.1 (i has not received any EXPAND and has not sent out an AUGFOUND) : i may receive this AUGFOUND in two sub-cases :

case 2.1.1 (i has not received an AUGFOUND before) : i sends a CHANGE-REQ message back to k for requesting the augmentation. k forwards this message to the other end of the Path.

case 2.1.2 (i has received an AUGFOUND before) : i ignores this AUGFOUND.

case 2.2 (i has received an EXPAND and has sent out an AUGFOUND) : i may receive this AUGFOUND in two sub-cases :

case 2.2.1 (i has sent out a CHANGE-REQ in response to an AUGFOUND before): i ignores this AUGFOUND.

case 2.2.2 (i has not sent out a CHANGE-REQ before) : i compares the identity of the free node x in Path with the identity of the other free node y which i has committed (by an AUGFOUND). If ident(x) > ident(y), i sends a CHANGE-REQ to k for requesting the augmentation of the Path. Otherwise, i ignores this AUGFOUND.

CHANGE< k, Path > : Path is the incoming augmenting path. In response to a CHANGE-REQ sent by i, this message confirms that the augmentation of matching from k to the end of Path was completed. i may receive a CHANGE in two cases:

Case 1 (i has a 'matched' status) : i changes its mate to the neighbor of i which is not the recent mate of i in Path and sends this message to the parent of i in Path.

Case 2 (i has a 'free' status) : i becomes a 'matched' node and assigns k as its mate. i then convergecasts an ECHO-G message to its ST-Parent to indicate the end of searching and augmentation.

CHANGE-REQ< k, Path > : 'REQ' means 'REQuest' and Path is the incoming augmenting path. This message requests i to augment Path. i should forward the same message to the child of i in the Path. i may receive a CHANGE-REQ in four cases:

Case 1 (i has a 'matched' status and i has not received any CHANGE-REQ before): i forwards the same message to the child of i in the Path.

Case 2 (i has a 'matched' status and i has received a CHANGE-REQ before) : i rejects the request by sending a NO-CHANGE to k.

Case 3 (i has a 'free' status and i has not sent out a CHANGE-REQ before) : i becomes a 'matched' node and assigns k as its mate. i then sends a CHANGE to k for respond-

ing to this message and confirms the augmentation. i then convergecasts an ECHO-G message to its ST-Parent to indicate the end of searching and augmentation.

Case 4 (i has a 'free' status and i has sent out a CHANGE-REQ for another augmenting path before) : i sends a NO-CHANGE to k for rejecting the request of augmentation.

ECHO-G< k, Matched > : Here, G stands for 'GENALT' and Matched is a set of nodes which have changed their status from 'free' to 'matched' in the current phase. This message informs i that, in response to a GENALT sent by i, the broadcasting through k has been completed. After i receives this message, it waits for an ECHO-G from all its ST-Child. Once it receives all the ECHO-G's, i sends an ECHO-G to its ST-Parent. Eventually, the root of ST will receive all its ECHO-Gs, it will then decide to terminate DMM or to start another phase by broadcasting a STARTPHASE message along ST.

ECHO-S< k, Path > : Here, S means 'Stop' and Path is an alternating path. This message informs i that, in response to an EXPAND sent by i, k rejects the request to join that VAT and i should stop sending any message to k. i may receive an ECHO-S in three cases:

Case 1 (k is the mate of i) : i deletes its identity from the Path and sends the same message to the parent of i in the Path.

Case 2 (k is not the mate of i and i has not received an AUGFOUND message before) : i may receive this ECHO-S in two sub-cases:

case 2.1 (i has a 'matched' status) : i waits for an ECHO-S from all its children. Once it receives all the ECHO-S's, i sends an ECHO-S to its mate.

case 2.2 (i has a 'free' status) : i waits for an ECHO-S from all its children. Once it receives all the ECHO-S's, i then convergecasts an ECHO-G to its ST-Parent to indicate the end of searching.

Case 3 (k is not the mate of i and i has received an AUGFOUND message before) : i ignores this ECHO-S.

EXPAND < k, Path > : Path is an alternating path. This message expands Path by requesting i to be added into Path. i may receive an EXPAND in the following cases:

Case 1 (i has committed with another augmenting path) : i sends an ECHO-S to k for rejecting to join that VAT.

Case 2 (i has not committed with any augmenting path) : i may receive this EXPAND in three sub-cases:

case 2.1 (i has a 'free' status) : i sends an AUGFOUND to inform k that an augmenting path is found.

case 2.2 (i has a 'matched' status and k is not its mate) : i only accepts the first two EXPANDs which come from two different VATs. These messages assign i as an 'inner' node. i appends itself into these VATs' alternating path Path and forwards this message to its mate. If there is any subsequent EXPAND, i sends an ECHO-S to the sender for rejecting to join those VAT.

case 2.3 (i has a status 'matched' and k is the mate of i) : i may add itself into that VAT in two sub-cases:

case 2.3.1 (i has been assigned as an inner node by an alternating path p) : i appends its identity to Path and sends an EXPAND to the parent of i in p for expanding the alternating path Path.

case 2.3.2 (i has not been assigned as an inner node) : i appends its identity to Path and sends an EXPAND to all its neighbors except the ancestors of i in the Path for expanding Path.

GENALT < k > : GENALT informs i to 'GENerate ALternating Trees'. This message is sent from the ST-Parent k and requests i to forward the same message to all its ST-Child. Then, if i has a 'matched' status, i does nothing else. If i has a 'free' status, i initiates

a search for augmenting paths. i first assigns itself as the root of a VAT and then sends an EXPAND to all its neighbors.

NO-CHANGE< k, Path > : Path is an augmenting path. This message is sent from the child k of i in Path, in response to a CHANGE-REQ sent by i, and rejects the augmentation of Path. i may receive NO-CHANGE in two cases:

Case 1 (i has a 'matched' status) : i sends the same message to the parent of i in Path.

Case 2 (i has a 'free' status) : i convergecasts an ECHO-G to its ST-Parent to indicate the end of searching.

READY< k > : This message informs i that, in response to a STARTPHASE sent by i, the broadcasting through k has completed. i waits for a READY from all its ST-Child. Once it receives all the READYs, i convergecasts a READY to its ST-Parent. Eventually, the root of ST will receive all its READYs, it then broadcasts a GENALT along ST to every node of G for starting the searching of augmenting paths.

STARTPHASE< k > : This message is sent from the ST-Parent k and requests i to initiate or reset its local variables. If i has some ST-Child's, i forwards the message to them. Otherwise, i sends a READY to k.

Local variables used in the process :

AugFound: This is a logical variable with value 'true' if i has received an AUGFOUND.

Changed: This variable contains a set of matched nodes which has its status changed from 'free' to 'matched' during the current phase.

Con-Change: This is a logical variable with value 'true' if i has received a CHANGE-REQ.

Con-Path: This variable contains an augmenting path if i has received a CHANGE-REQ and has committed the augmentation with the augmenting path.

Echoedj(x): This is a set of logical variables with value 'false' if i has sent to node x an EXPAND but has not received an ECHO-S for it. j has only two possible values, either 1 or 2 which indicates two different alternating paths.

Free: This variable contains a set of free nodes and only resides in the root (ST-root) of the graph.

I-Pathj: This variable contains the two alternating paths which start from two free nodes and end at i as an 'inner' node. j has only two possible values, either 1 or 2 which indicates two different alternating paths.

Mate: This variable contains matching mate of i.

Neighbor: The set of adjacent nodes of i.

O-Pathj: This variable contains the two alternating paths which start from two free nodes and end at i as an 'outer' node. j has only two possible values, either 1 or 2 which indicates two different alternating paths.

Sendto: This is a temporary variable. It is used only for forwarding an EXPAND to the neighbors of i.

Start-Node: This variable contains the identity of the root. The root has its own identity in this variable; the other nodes has value 'null' in this variable.

Status: This variable denotes the status of i as follows :

a) Status has value 'free' if i has no mate.

b) Status has value 'matched' if i has a mate.

ST-Child: This variable contains a set of children of i in ST.

ST-Echoed(j): This is a set of logical variables with value 'false' if i has sent to node j a GENALT but has not received an ECHO-G from j.

ST-Parent: This variable contains the parent of i in ST.

ST-Start: T .is is a logical variable with value 'true' if i has received a GENALT from its ST-Parent.

TemPath: This variable contains an augmenting path if i has received an AUGFOUND and i has confirmed the expansion with this temporary augmenting path.

Functions used by Algorithm DMM :

Top(Path) returns the identity of the first element of Path.

Bottom(Path) returns the identity of the last element of Path.

4.3 Global-view Description of Algorithm DMM

Procedure : Based on any matching and a spanning tree ST, DMM proceeds in phases until a maximum matching is found. In each phase, at least one augmenting path is augmented by the following steps:

Step I. Initializes each node.

Step II. Generate VAT's from all the free nodes and find one augmenting path (if any) from each VAT.

Step III. Augment the matching along the discovered augmenting paths.

Step IV. Inform the root of ST to end the phase.

Details of these steps are given below:

Step I. Initializes each node.

To initiate a phase, the root of ST broadcasts the STARTPHASE message to every node of G along the edges of ST. On receiving STARTPHASE, a node re-initializes its local variables. If it has ST-Child, it continues the broadcasting. If it has no ST-Child or has received a READY

message from each of its ST-Child, it convergecasts a READY message to its ST-Parent. Eventually, when the root receives the READY messages from all its ST-Child, it is ready to start Step II in the search for augmenting paths.

Step II. Generate VAT's from all the free nodes and find one augmenting path (if any) from each VAT.

In order to search for augmenting paths, the root of ST broadcasts a GENALT message to every node of G along the edges of ST. GENALT triggers each free node to generate a virtual alternating tree (VAT) as follows :

A free node r assigns itself as the root of VAT and sends an EXPAND $<r,\{r\}>$ to each of its neighbors. In general, suppose node x receives an EXPAND $<y,p>$ from node y, where p is the alternating path. Three conditions should be considered: (1) If x has a 'free' status, an augmenting path is found. x extends p by appending itself to it and sends an AUGFOUND $<x,p>$ (indicating 'AUGmenting path FOUND') to y. (2) If x has a 'matched' status and cannot be passed through (according to the four rules stated in Section 4.1), it sends an ECHO-S $<x>$ to y to 'stop' the expansion. (3) If x has a 'matched' status and can be passed through, p is extended and x forwards the EXPAND $<x,p>$ to all its neighbors except y. When a node receives an ECHO-S from all its VAT-children, it sends an ECHO-S to its VAT-parent to indicate that expansion through this node is finished.

Even when an augmenting path $p=(r,...,i,...,x)$ is found at a free node x, the augmentation may not be feasible because another augmenting path $q=(r,...,i,...,y)$ may be found at another free node y, such that p and q meet at an outer node i (in particular, i may be r). Obviously, p and q cannot be both augmented. Figure 4.4a shows such an example. To solve this problem, our algorithm requires that, for each VAT, only one such path is selected for augmentation. Hence, at i, only the first AUGFOUND is forwarded and all the subsequent AUGFOUNDs will be ignored. Figure 4.4b shows that i forwards only the first AUGFOUND(i,(r,...,i,...,x)) to its parent and ignores the subsequent AUGFOUND's.

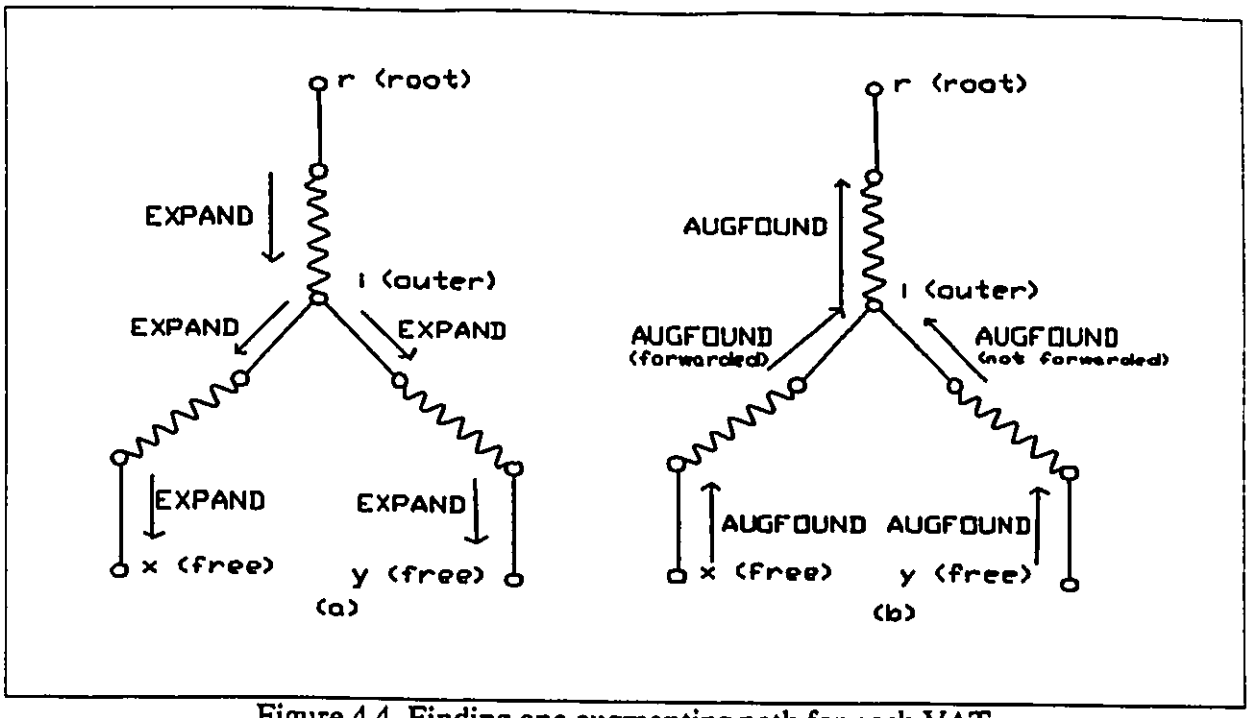


Figure 4.4. Finding one augmenting path for each VAT.

Step III. Augment the matching along the discovered augmenting paths.

In Step II, though only one augmenting path is obtained from each VAT, several augmenting paths may have been obtained from different VAT's. However, not all of them can always be augmented because some of them may be node-joint. In Step III, the main issue is to determine which of these augmenting paths are *feasible*, i.e., 'really' augmentable.

Let us first describe the general process of augmentation. Briefly, every augmenting path p = (r,...,i,j,...e) tries to augment itself in two stages: *Request* and *Commit*. In the Request stage, the root r of p 'requests to become feasible' by sending CHANGE-REQ < ,p > down the path. If this request can eventually reach the end free node e, e will decide whether or not the augmentation can be done. If so, it will initiate the Commit stage by returning CHANGE along p (in reverse direction). The entire path is then committed and will be augmented. Otherwise, it will return NO-CHANGE.

According to the above process, therefore, there may be several CHANGE-REQ's reaching a node, say i. Since every node can belong to at most one feasible augmenting path (due to the nature of matching), only one of these requests can go through and the rest have to be abolished. This is achieved in Algorithm DMM as follows :

Basic strategy (actions to be taken by node i)

i. If the CHANGE-REQ's are for augmenting paths whose end-points are different, the first request will continue and the subsequent ones will be rejected.

ii. If the CHANGE-REQ's are for augmenting paths whose end-points are the same (there may be at most two such paths rooted at the opposite end-points), the one whose root has a higher identity will continue and the other one will be ignored.

Details of how to carry out the above strategy

In the following, p, q denote different augmenting paths and p' denotes the same augmenting path as p but in the reverse order.

When a node i receives CHANGE-REQ< ,p >, it responds according to the following cases:

a) i is a 'free' node (i.e., one of the end-points of p) :

If i has never sent any CHANGE-REQ before, i issues CHANGE<i,p> along p'.

If i has sent any CHANGE-REQ<i,q> before, i issues NO-CHANGE<i,p> along p'. For example, suppose Figure 4.5 has two augmenting paths p=(f,h,g,e) and q=(e,c,d,b). e has sent CHANGE-REQ<c,q> to c and receives CHANGE-REQ<g,p> from g. e rejects the request by sending NO-CHANGE<e,p> to g.

If i has sent any CHANGE-REQ<i,p'> before, i issues CHANGE<i,p> along p' if ident(root(p)) > ident(i) and ignores CHANGE-REQ< ,p> otherwise. For example, suppose Figure 4.5 has two augmenting path p=(f,h,g,e) and p'=(e,g,h,f). e has sent CHANGE-REQ<e,p'> to g and receives CHANGE-REQ<g,p>

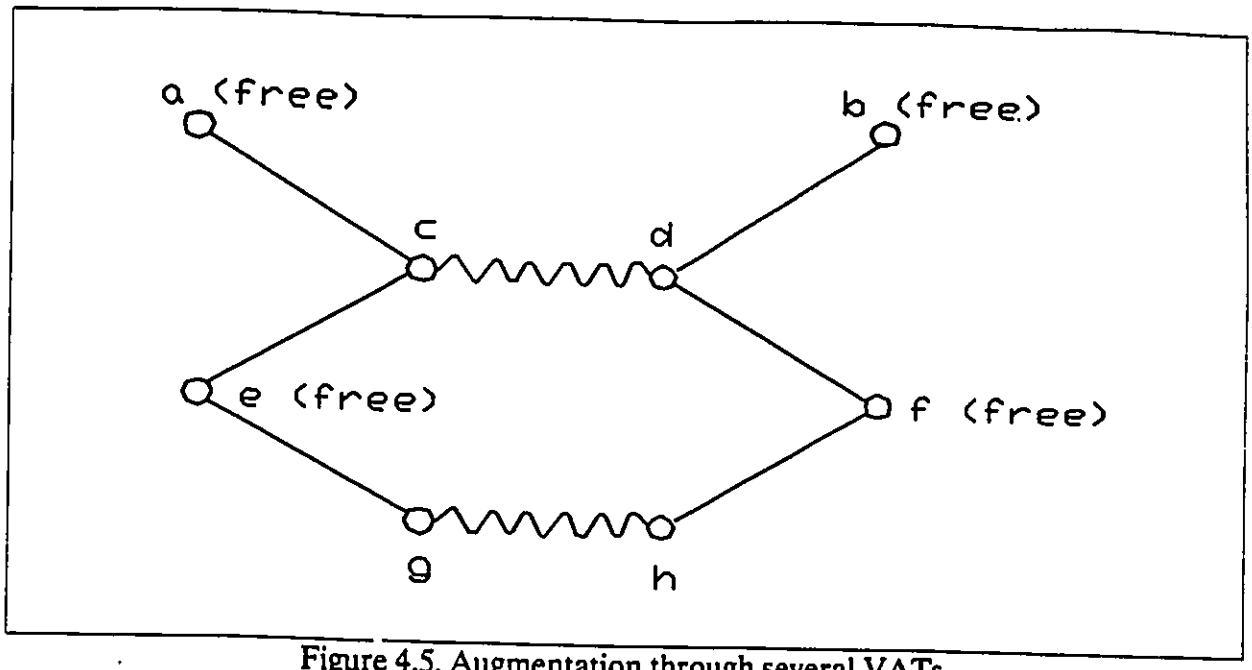


Figure 4.5. Augmentation through several VATs.

from g. Since ident(f) > ident(e), e accepts the request by sending CHANGE<e,p> to g. As another example, f has sent CHANGE-REQ<f,p> to h and receives CHANGE-REQ<h,p'> from h. Since ident(f) > ident(e), f ignores CHANGE-REQ<h,p'>.

b) i is a 'matched' node (i.e., one of the intermediate-points of p) :

If i has never sent any CHANGE-REQ before, i issues CHANGE-REQ< ,p> along p.

If i has sent any CHANGE-REQ<i,q> to k before and the CHANGE-REQ is sent by j, i issues NO-CHANGE<i,p> along p'. For example, suppose Figure 4.5 has two augmenting paths p = (a,c,d,b) and q = (e,c,d,f). c has sent CHANGE-REQ<c,q> to d and receives CHANGE-REQ<a,p> from a. c rejects the request by sending NO-CHANGE<c,p> to a. As another example, d has sent CHANGE-REQ<d,q> to f and receives CHANGE-REQ<b,p'> from b. d rejects the request by sending NO-CHANGE<d,p'> to b.

If i has sent any CHANGE-REQ<i,q> to j before and the CHANGE-REQ is sent by j, i issues CHANGE-REQ<i,p> along p if ident((root(p)) > ident(root(q)) and sends NO-CHANGE<i,p> along p' otherwise. For example, suppose Figure 4.5 has two augmenting path p=(f,d,c,a) and q=(e,c,d,b). c has sent CHANGE-REQ<c,q> to d and receives CHANGE-REQ<d,p> from d. Since ident(f) > ident(e), c sends CHANGE-REQ<c,p> to a. On the other side, d has sent CHANGE-REQ<d,p> to c and receives CHANGE-REQ<c,q> from c. Since ident(e) < ident(f), d sends NO-CHANGE<d,q> to c.

If i has sent any CHANGE-REQ<i,p'> before, i issues CHANGE-REQ<i,p> along p if ident(root(p)) > ident(root(p')) and ignores CHANGE-REQ< ,p> otherwise. For example, suppose Figure 4.5 has two augmenting paths p=(f,h,g,e) and p'=(e,g,h,f). g has sent CHANGE-REQ<g,p'> to h and receives CHANGE-REQ<h,p> from h. Since ident(f) > ident(e), g sends CHANGE-REQ<g,p> to e. On the other side, h has sent CHANGE-REQ<h,p'> to g and receives CHANGE-REQ<g,p> from g. Since ident(f) > ident(e), h ignores CHANGE-REQ<g,p>.

When a node receives either CHANGE or NO-CHANGE, it forwards the same message to the root along the augmenting path. For these nodes to forward a CHANGE, they also change their mate to their neighbor in that augmenting path. Eventually, the root will receive either a CHANGE or a NO-CHANGE, and finishes the augmentation by sending an ECHO-G to its ST-Parent.

Step IV. Inform the root of ST to end a phase.

To indicate the end of Step III, every node convergecasts an ECHO-G to its ST-Parent. When a node receives an ECHO-G from all its ST-Child, it sends an ECHO-G to its ST-Parent. After all the nodes convergecast in the current phase, the ST-root decides whether to start

another phase or not. If no augmenting path is found in the current phase or the number of free nodes remaining on the graph is less than two, the algorithm terminates and the matching is maximum [BERG 57]. Otherwise, the ST-root initiates another phase by broadcasting a STARTPHASE along ST to every node of G.

4.4 Distributed-view Description of Algorithm DMM

A distributed-view description includes a detailed explanation of the movements of the messages in the form of a pseudo-code.

Pseudo-code of Algorithm DMM

Initiation :

The initial spanning matching is obtained by Algorithm DSM, no preparation is required. Before executing DMM, a distributed algorithm creates a spanning tree ST. The algorithm starts at the root of ST.

The starting node performs as follows :

```
Start-Node := its own identity; Changed := ∅; ST-Start := 'true';
for each x in ST-Child do [ ST-Echoed( x ) := 'false'; send GENALT< i > to x ]        {*1*}
```

The process at node i

```
{* On receiving message GENALT< k > *}
Changed := ∅; ST-Start := 'true';
if ST-Child = ∅ →
      if Status = 'free' → Skip;                                                     {*2*}
      [] Status = 'matched' → send ECHO-G< i, ∅ > to ST-Parent;                       {*3*}
      fi
[] ST-Child ≠ ∅ →
      for each x in ST-Child do [ ST-Echoed( x ) := 'false'; send GENALT< i > to x ] {*4*}
fi
if Status = 'matched' → Skip;                                                        {*5*}
[] Status = 'free' →
      if AugFound = 'true' →      Skip;                                            {*6*}
      [] AugFound = 'false' → O-Path1 := i;
            for each x in Neighbor do [ Echoed1( x ) := 'false'; send EXPAND< i, O-Path1 > to x ]   {*7*}
      fi
. fi
```

```
{* On receiving message ECHO-G< k, Matched > *}
ST-Echoed( k ) : = 'true'; Changed : = Changed ∪ Matched;
if Not ST-Echoed( j ) for some j in ST-Child → Skip;                                    {*8*}
[] ST-Echoed( j ) for all j in ST-Child →
      if i = Start-Node →
            Free : = Free - Changed;
            if |Free| < 2  or  |Changed| = 0 → Completed;                               {*9*}
            [] |Free| ≥ 2 and |Changed| > 0 →
                  for each x in ST-Child do
                  [ ST-Echoed( j ) : = 'false'; send STARTPHASE< i > to x ]              {*10*}
            fi
      [] i ≠ Start-Node →
            if Status = 'matched' → send ECHO-G< i, Changed > to ST-Parent;            {*11*}
            [] Status = 'free' →
                  if Not Echoed1( j ) for some j in Neighbor → Skip;                    {*12*}
                  [] Echoed1( j ) for all j in Neighbor → send ECHO-G< i, Changed > to ST-Parent;  {*13*}
                  fi
            fi
      fi
fi

{* On receiving message EXPAND< k, Path > *}
if Con-Change = 'true' or AugFound = 'true' → send ECHO-S< i, Path > to k;              {*14*}
[] Con-Change = 'false' and AugFound = 'false' →
      if k = Mate →
            store Path into one of O-Pathj where j = 1, 2 and O-Pathj = ∅;
            TempBoolean : = 'false';
            for each p in { I-Pathk | k = 1, 2 } do
            [     if p = ∅  or  Top( p ) = Top( Path ) → Skip;                        {*15*}
                  [] p ≠ ∅  and Top( p ) ≠ Top( Path ) → TempBoolean : = 'true';
                        x : = the neighbor of i in p;
                        Echoedj( x ) : = 'false'; send EXPAND< i, Path.i > to x;     {*16*}
                  fi
            ]
            if TempBoolean = 'true' → Skip;                                          {*17*}
            [] TempBoolean = 'false' → Sendto : = Neighbor - k - Path;
                  if Sendto = ∅ → send ECHO-S< i, Path > to k;                        {*18*}
                  [] Sendto ≠ ∅ →
                        for each x in Sendto do
                        [ Echoedj( x ) : = 'false'; send EXPAND< i, Path.i > to x ]   {*19*}
                  fi
            fi
      [] k ≠ Mate →
            if Status = 'free' →
                  if AugFound = 'true' → send ECHO-S< i, i'ath > to k;                {*20*}
```

```
            [] AugFound = 'false' → AugFound := 'true'; TemPath := Path.i;
                send AUGFOUND< i, TemPath > to k;                                    {*21*}
            fi
        [] Status = 'matched' →
            if ( I-Path1 ≠ ∅ and I-Path2 ≠ ∅ ) or Top( Path ) = Top( I-Path1 or I-Path2 ) →
                send ECHO-S< i, Path > to k;                                         {*22*}
            [] ( I-Path1 = ∅ or I-Path2 = ∅ ) and Top( Path ) ≠ Top( I-Path1 and I-Path2 ) →
                store Path into one of I-Pathj | j = 1,2 where I-Pathj = ∅;
                send EXPAND< i, Path.i > to Mate;                                    {*23*}
            fi
        fi
    fi
fi

{* On receiving message ECHO-S< k, Path > *}
if k = Mate →
    send ECHO-S< i, Path - i > to the neighbor of i in Path;                         {*24*}
[] k ≠ Mate →
    Echoedj( k ) := 'true' where j = 1,2 and O-Pathj = Path;
    if AugFound = 'true' → Skip;                                                     {*25*}
    [] AugFound = 'false' →
        if Not Echoed1( j ) or Not Echoed2( j ) for some j in Neighbor → Skip;       {*26*}
        [] Echoed1( j ) and Echoed2( j ) for all j in Neighbor →
            if Status = 'matched' →
                send ECHO-S< i, O-Pathj | j = 1,2 > to Mate where O-Pathj ≠ ∅;    {*27*}
            [] Status = 'free' →
                if Not ST-Echoed( j ) for some j in ST-Child → Skip;              {*28*}
                [] ST-Echoed( j ) for all j in ST-Child →
                    send ECHO-G< i, Changed > to ST-Parent;                       {*29*}
                fi
            fi
        fi
    fi
fi

{* On receiving message AUGFOUND< k, Path > *}
if k = Mate →
    send AUGFOUND< i, Path > to the neighbor of i in Path - k;
    send ECHO-S< i, O-Pathj | j = 1,2 > to the neighbor of i in O-Pathj - k where O-Pathj ≠ Path;   {*30*}
[] k ≠ Mate →
    if AugFound = 'true' →
        if Status = 'matched' → Skip;                                                {*31*}
        [] Status = 'free' →
            if Path ≠ TemPath →
                if Bottom( Path ) > Top( TemPath ) → Con-Change := 'true'; Con-Path := Path;
```

```
                    send CHANGE-REQ< i, Path > to k;                                {*32*}
                [] Bottom( Path ) < Top( TemPath ) → Skip;                          {*33*}
                fi
            [] Path = TemPath →
                if i < Top( Path ) → Skip;                                          {*34*}
                [] i > Top( Path ) → send CHANGE-REQ< i, Path > to k;                {*35*}
                fi
            fi
        fi
    [] AugFound = 'false' → AugFound := 'true'; TemPath := Path;
        if Status = 'free' → send CHANGE-REQ< i, Path > to k;                        {*36*}
        [] Status = 'matched' → send AUGFOUND< i, Path > to Mate;                    {*37*}
        fi
    fi
fi

{* On receiving message CHANGE-REQ< k, Path > *}
if Con-Change = 'true' →
    if Path ≠ Con-Path →
        send NO-CHANGE< i, Path > to k;                                             {*38*}
    [] Path = Con-Path →
        if Top( Path ) < Top( Con-Path ) → Skip;                                    {*39*}
        [] Top( Path ) ≥ Top( Con-Path ) →
            if Status = 'free' → Status := 'matched'; Mate := the neighbor of i in Path;
                send CHANGE< i, Path > to k;
                if Not ST-Echoed( j ) for some j in ST-Child → Skip;                 {*40*}
                [] ST-Echoed( j ) for all j in ST-Child →
                    send ECHO-G< i, Changed > to ST-Parent;                         {*41*}
                fi
            [] Status = 'matched' →
                send CHANGE-REQ< i, Path > to the neighbor of i in Path - k;        {*42*}
            fi
        fi
    fi
[] Con-Change = 'false' → Con-Change = 'true'; Con-Path := Path;
    if Status = 'free' →
        Status := 'matched'; Mate := the neighbor of i in Path; Changed = Changed + { i };
        send CHANGE< i, Path > to k;
        if Not ST-Echoed( j ) for some j in ST-Child → Skip;                         {*43*}
        [] ST-Echoed( j ) for all j in ST-Child →
            send ECHO-G< i, Changed > to ST-Parent;                                 {*44*}
        fi
    [] Status = 'matched' → send CHANGE-REQ< i, Path > to the neighbor of i in Path - k;  {*45*}
    fi
fi
```

```
{* On receiving message CHANGE< k, Path > *}
Mate := neighbor of i in Path - Mate;
if Status = 'free' →
      Status := 'matched'; Changed := Changed + Top( Path ) + Bottem( Path );
      if Not ST-Echoed( j ) for some j in ST-Child → Skip;                          {*46*}
      [] ST-Echoed( j ) for all j in ST-Child →
            send ECHO-G< i, Changed > to ST-Parent;                                {*47*}
      fi
[] Status = 'matched' →
      send CHANGE< i, Path > to the neighbor of i in Path - k;                     {*48*}
fi

{* On receiving message NO-CHANGE< k, Path > *}
if Status = 'matched' → send NO-CHANGE< i, Path > to the neighbor of i in Path - k;   {*49*}
[] Status = 'free' →
      if Not ST-Echoed( j ) for some j in ST-Child → Skip;                          {*50*}
      [] ST-Echoed( j ) for all j in ST-Child →
            send ECHO-G< i, Changed > to ST-Parent;                                {*51*}
      fi
fi

{* On receiving message STARTPHASE< k > *}
ST-Start := 'false'; AugFound := 'false'; Con-Change := 'false';
ST-Echoed( j ) := 'true' for all j in ST-Child;
Echoed_j( x ) := 'true' for all x in Neighbor where j = 1, 2;
I-Path_j := ∅ and O-Path_j := ∅ where j = 1, 2;
if ST-Child = ∅ →     send READY< i > to k;                                          {*52*}
[] ST-Child ≠ ∅ →     for each x in ST-Child do
                      [ ST-Echoed( x ) := 'false'; send STARTPHASE< i > to x ]      {*53*}
fi

{* On receiving message READY< k > *}
ST-Echoed( k ) := 'true';
if Not ST-Echoed( j ) for some j in ST-Child → Skip;                                {*54*}
[] ST-Echoed( j ) for all j in ST-Child →
      if i = Start-Node → Changed := ∅; ST-Start := 'true'; AugFound := 'false'; Con-Change := 'false';
            Echoed_j( x ) := 'true' for all x in Neighbor where j = 1, 2;
            I-Path_j := ∅ and O-Path_j := ∅ where j = 1, 2;
            for each x in ST-Child do
            [ ST-Echoed( x ) := 'false'; send GENALT< i > to x ]                    {*55*}
      [] i ≠ Start-Node → send READY< i > to ST-Parent;                            {*56*}
      fi
fi
```

A special situation :

Algorithm DMM must handle the following situation due to the asynchronous nature of the network.

It may happen that more than one 'free' node (x_1, x_2, ..., x_n) starts its own VAT. After x_2 receives an EXPAND from x_1, x_2 confirms the augmenting path by returning an AUGFOUND to x_1. The above situation may be generalized to a sequence: x_{i+1} receives an EXPAND from x_i, and then confirms the augmenting path by returning an AUGFOUND to x_i. x_1 receives an EXPAND from x_n, and then confirms the augmenting path by returning an AUGFOUND to x_n. Such so that a pure loop is formed. Since every x_i can only commit with one augmenting path, all the free nodes cannot been augmented.

This problem is solved in DMM as follows : Each free node only accepts the augmenting path from a free node which has a higher node identity than the other. Thus, the pure loop is then split into one or several chains and only one augmenting path is augmented in each chain.

4.5 An Example

This simple example demonstrates how Algorithm DMM works and handles the special situation with loops (see the explanation at the previous section). Figure 4.6a shows the graph with an initial spanning matching (possibly obtained by Algorithm DSM). Figure 4.6b shows a spanning tree ST (possibly obtained by using [CHAN 79]) on the graph. Node g, the root of ST, is the designated node. Each node initializes its local variables as follows : ST-Start to 'false'; AugFound to 'false'; Start-Node to '0'; ST-Echoed(j) to 'true' for all j in ST-Child; $\text{Echoed}_j(x)$ to 'true' for all x in Neighbor and I-Path_j to $\varnothing$ and O-Path_j to $\varnothing$ where $j = 1, 2$. Figure 4.7 shows how the ST-root g broadcasts the GENALT messages through ST.

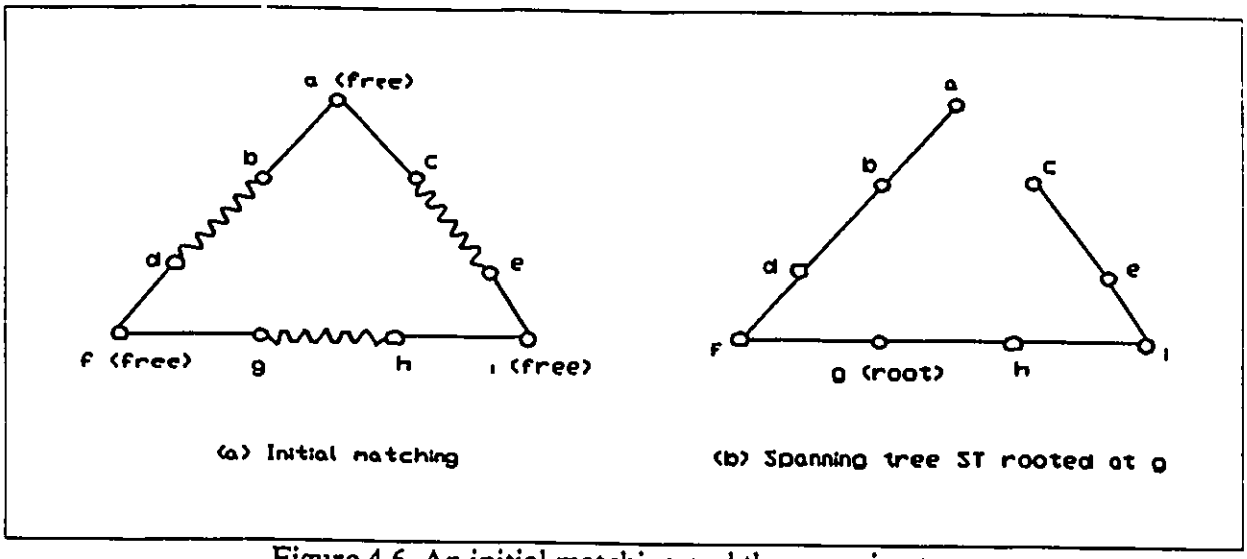


Figure 4.6. An initial matching and the spanning tree.

Note : In Table 4.1, each column describes the occurrence of one event. The event numbers indicate a temporal order in which events occur at the same node, but not necessarily for events at different nodes. Also, '$\alpha/\beta/\gamma$' means 'α or β or γ'; 'α,β,γ' means 'α and β and γ'; '$\alpha;\beta$' means 'α then β'; 'Top(Path)' means 'the first element of Path'; 'Bottom(Path)' means 'the last element of Path'; 'T' stands for 'true'; and 'F' stands for 'false'. 'EXP' is an abbreviation for 'EXPAND'.

A	Event number	1	2	3
B	Node where event occurs	g (ST-root)	h	f
C	Message received		GENALT<g>	GENALT<g>
D	Values of decision variables just before receiving message	Free = {a,f,i}	ST-Child = {i} Status = 'matched'	ST-Child = {d} Status = 'free' AugFound = 'F'
E	Actions and conditions after receiving the last (if more than one) possible input message at listed in C. {* x *} refers to the statement number in Algorithm DMM	Start-Node : = g Changed : = ∅ ST-Start : = 'T' ST-Echoed(f,h) : = 'F' send GENALT<g> to f,h {*1*}	Changed : = ∅ ST-Start : = 'T' ST-Echoed(i) : = 'F' send GENALT<h> to i {*4*} Skip {*5*}	Changed : = ∅ ST-Start : = 'T' ST-Echoed(d) = 'F' send GENALT<f> to d {*4*} O-Path$_1$: = f Echoed$_1$(d,g) : = 'F' send EXP<f,f> to d,g {*7*}
F	Next event number (s)	2, 3	4	5, 11,16

Table 4.1.

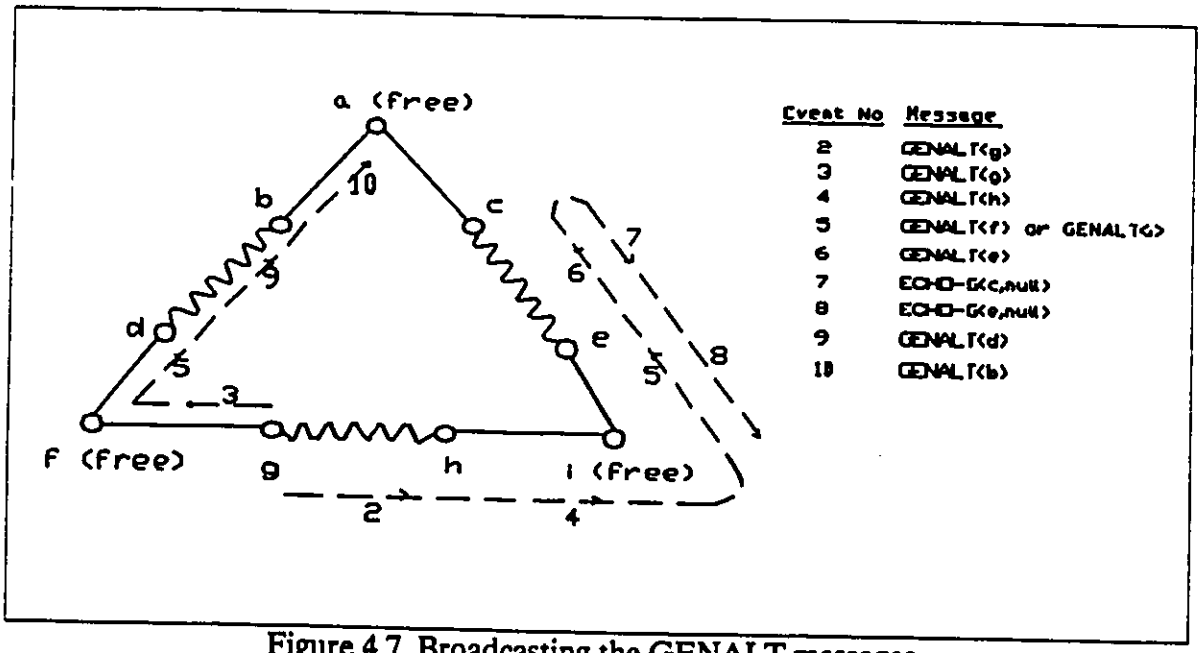


Figure 4.7. Broadcasting the GENALT messages.

A	4	5	6	7
B	i	d/e	c	e
C	GENALT<h>	GENALT<f> / GENALT<i>	GENALT<e>	ECHO-G<c,∅>
D	ST-Child = {e} Status = 'free' AugFound = 'F'	ST-Child = {b} / ST-Child = {c} Status = 'matched'	ST-Child = ∅ Status = 'matched'	ST-Echoed(c) = 'F' i ≠ Start-Node Status = 'matched'
E	Changed := ∅ ST-Start := 'T' ST-Echoed(e) = 'F' send GENALT<i> to e {*4*} O-Path$_1$:= i Echoed$_1$(e,h) := 'F' send EXP<i,i> to e,h {*7*}	Changed := ∅ ST-Start := 'T' ST-Echoed(b) := 'F' send GENALT<d> to b {*4*} Skip / {*5*} ST-Echoed(c) := 'F' send GENALT<e> to c {*4*} Skip {*5*}	Changed := ∅ ST-Start := 'T' send ECHO-G<c,∅> to e {*3*} Skip {*5*}	ST-Echoed(c) := 'T' Changed := ∅ send ECHO-G<e,∅> to i {*11*}
F	5, 16,11	9, 6	7	8

A	8	9	10	11
B	i	b	a	c/d/h
C	ECHO-G<e,∅>	GENALT<d>	GENALT<b>	EXP<a,a> / EXP<f,f> / EXP<i,i>
D	ST-Echoed(i) = 'F' i ≠ Start-Node Status = 'free' Echoed$_1$(e,h) = 'F'	ST-Child = {a} Status = 'matched'	ST-Child = ∅ Status = 'free' AugFound = 'F'	Con-Change = 'F' AugFound = 'F' a,f,i ≠ Mate Status = 'matched' I-Path$_{1,2}$ = ∅ Top(a/f/i) ≠ Top(∅)
E	ST-Echoed(i) := 'T' Changed := ∅ Skip {*12*}	Changed := ∅ ST-Start := 'T' ST-Echoed(a) := 'F' serd GENALT<b> to a {*4*} Skip {*5*}	Changed := ∅ ST-Start := 'T' Skip {*2*} O-Path$_1$:= a Echoed$_1$(b,c) := 'F' send EXP<a,a> to b,c {*7*}	I-Path$_1$:= a send EXP<c,a.c> to e / {*23*} I-Path$_1$:= f send EXP<d,f.d> to b / {*23*} I-Path$_1$:= i send EXP<h,i.h> to g {*23*}
F	waiting	10	16,11	12,13,14

Table 4.1. (Continued)

A	12	13	14	15
B	e	b	g	i/a/f
C	EXP<c,a.c>	EXP<d,f.d>	EXP<h,i.h>	EXP<e,a.c.e> / EXP<b,f.d.b> / EXP<g,i.h.g>
D	Con-Change = 'F' AugFound = 'F' c = Mate I-Path$_{1,2}$ = ∅ O-Path$_{1,2}$ = ∅	Con-Change = 'F' AugFound = 'F' d = Mate I-Path$_{1,2}$ = ∅ O-Path$_{1,2}$ = ∅	Con-Change = 'F' AugFound = 'F' h = Mate I-Path$_{1,2}$ = ∅ O-Path$_{1,2}$ = ∅	Con-Change = 'F' AugFound = 'F' e,b,g ≠ Mate Status = 'free'
E	O-Path$_1$ = a.c Skip {*15*} Sendto := i Echoed$_1$(i) := 'F' send EXP<e,a.c.e> to i {*19*}	O-Path$_1$ = f.d Skip {*15*} Sendto := a Echoed$_1$(a) := 'F' send EXP<b,f.d.b> to a {*19*}	O-Path$_1$:= i.h Skip {*15*} Sendto := f Echoed$_1$(f) := 'F' send EXP<g,i.h.g> to f {*19*}	AugFound := 'T' TemPath := a.c.e.i send AUGFOUND <i,a.c.e.i> to e/ {*21*} TemPath := f.d.b.a send AUGFOUND <a,f.d.b.a> to b/ {*21*} TemPath := i.h.g.f send AUGFOUND <f,i.h.g.f> to g {*21*}
F	15	15	15	17,17,17

Table 4.1. (Continued)

Figure 4.8 shows three alternating trees extend in the graph and each of them has two alternating paths. We assume that three alternating paths in Figure 4.8a are expanded faster than the other three alternating paths in Figure 4.8b, such that a pure loop is created (see the explaination of special situations at the end of this chapter).

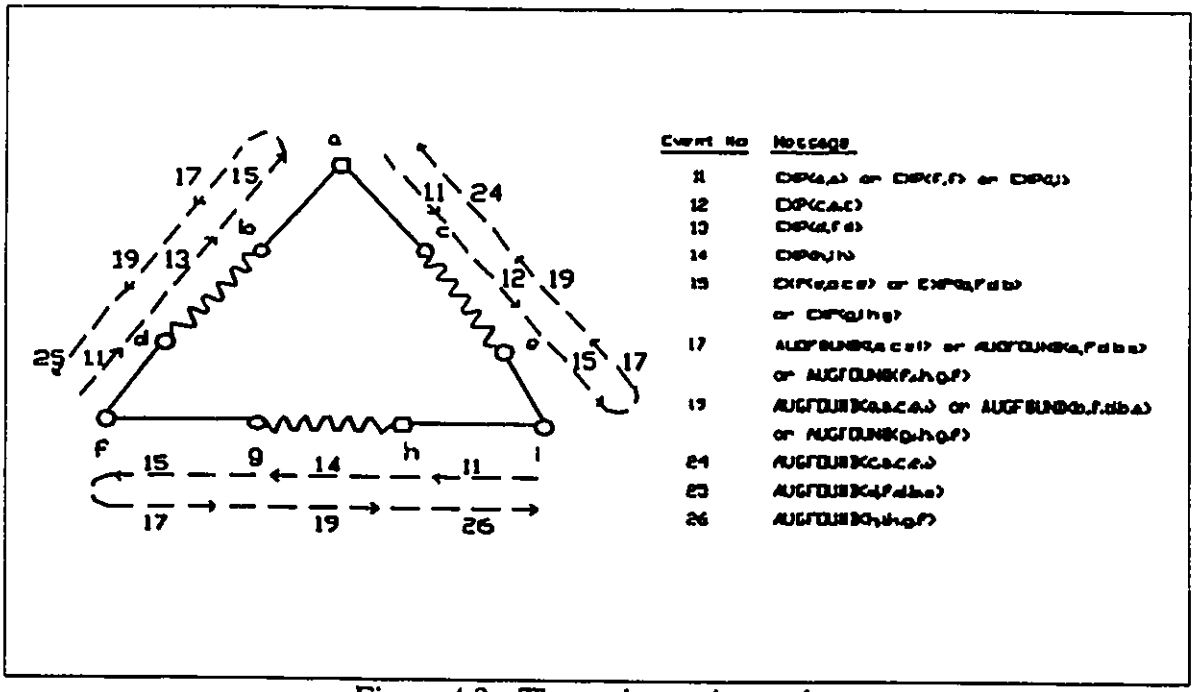


Figure 4.8a. Three alternating paths.

A	16	17	18	19
B	e/b/g	e/b/g	c/d/h	c/d/h
C	EXP<i,i> / EXP<a,a> / EXP<f,f>	AUGFOUND <i,a.c.e.i> / AUGFOUND <a,f.d.b.a> / AUGFOUND <f,i.h.g.f>	EXP<e,i.e> / EXP<b,a.b> / EXP<g,f.g>	AUGFOUND <e,a.c.e.i> / AUGFOUND <b,f.d.b.a> / AUGFOUND <g,i.h.g.f>
D	Con-Change = 'F' AugFound = 'F' i,a,f ≠ Mate Status = 'matched' I-Path$_{1,2}$ = ∅ Top(i/a/f) ≠ Top(∅)	i,a,f ≠ Mate AugFound = 'F' Status = 'matched'	Con-Change = 'F' AugFound = 'F' e,b,g = Mate I-Path$_1$ = a / f / i O-Path$_{1,2}$ = ∅	e,b,g = Mate
E	I-Path$_1$ = i send EXP<e,i.e> to c / {*23*} I-Path$_1$ = a send EXP<b,a.b> to d / {*23*} I-Path$_1$ = f send EXP<g,f.g> to h {*23*}	AugFound := 'T' TemPath := a.c.e.i send AUGFOUND <e,a.c.e.i> to c / {*37*} TemPath := f.d.b.a send AUGFOUND <b,f.d.b.a> to d / {*37*} TemPath := i.h.g.f send AUGFOUND <g,i.h.g.f> to h {*37*}	O-Path$_1$:= i.e x := a Echoed$_1$(a) := 'F' send EXP <c,i.e.c> to a / {*16*} O-Path$_1$:= a.b x := f Echoed$_1$(f) := 'F' send EXP <d,a.b.d> to f / {*16*} O-Path$_1$:= f.g x := i Echoed$_1$(i) := 'F' send EXP <h,f.g.h> to i {*16*} ; Skip {*17*}	send AUGFOUND <c,a.c.e.i> to a {*30*} / send AUGFOUND <d,f.d.b.a> to f {*30*} / send AUGFOUND <h,i.h.g.f> to i {*30*}
F	18,18,18	19,19,19	20,20,20	24,25,26

Table 4.1. (Continued)

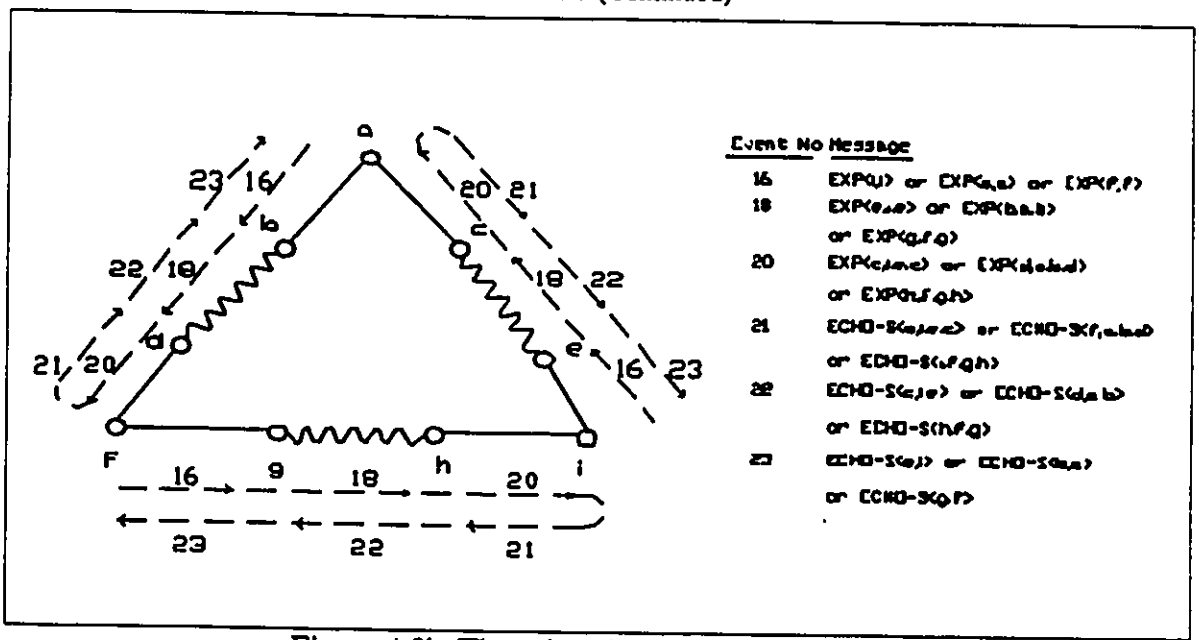


Figure 4.8b. The other three alternating paths.

A	20	21	22
B	a/f/i	c/d/h	e/b/g
C	EXP<c,i.e.c> / EXP<d,a.b.d> / EXP<h,f.g.h>	ECHO-S<a,i.e.c> / ECHO-S<f,a.b.d> / ECHO-S<i,f.g.h>	ECHO-S<c,i.e> / ECHO-S<d,a.b> / ECHO-S<h,f.g>
D	AugFound = 'T'	a,f,i ≠ Mate Echoed$_1$(a/f/i) = 'F' AugFound = 'F' Status = 'matched'	c,d,h = Mate
E	send ECHO-S<a,i.e.c> to c / {*14*} send ECHO-S<f,a.b.d> to d / {*14*} send ECHO-S<i,f.g.h> to h {*14*}	Path := i.e / a.b / f.g Echoed1(a/f/i) := 'T' send ECHO-S<c,i.e> to e / {*27*} send ECHO<d,a.b> to b / {*27*} send ECHO-S<h,f.g> to g {*27*}	Path := i / a / f send ECHO-S<e,i> to i / {*24*} send ECHO-S<b,a> to a / {*24*} send ECHO-S<g,f> to f {*24*}
F	21,21,21	22, 22,22	23,23,23

A	23	24	25	26
B	i/a/f	a	f	i
C	ECHO-S<e,i> / ECHO-S<b,a> / ECHO-S<g,f>	AUGFOUND <c,a.c.e.i>	AUGFOUND <d,f.d.b.a>	AUGFOUND <h,i.h.g.f>
D	e,b,g ≠ Mate AugFound = 'F' Echoed$_1$(e.h / b,c / d,g) = 'F'	c ≠ Mate AugFound = 'T' Status = 'free' a.c.e.i ≠ f.d.b.a Bottom(i) > Top(f)	d ≠ Mate AugFound = 'T' Status = 'free' f.d.b.a ≠ i.h.g.f Bottom(a) < Top(i)	h ≠ Mate AugFound = 'T' Status = 'free' i.h.g.f ≠ a.c.e.i Bottom(f) > Top(a)
E	Path := ∅ / ∅ / ∅ Echoed$_1$(e/b/g) := 'T' Skip {*26*}	Con-Change := 'T' Con-Path := a.c.e.i send CHANGE-REQ <a,a.c.e.i> to c {*32*}	Skip {*33*}	Con-Change := 'T' Con-Path := i.h.g.f send CHANGE-REQ <i,i.h.g.f> to h {*32*}
F	waiting	27	end of branch	27

Table 4.1. (Continued)

A	27	28	29	30
B	c/h	e/g	i	f
C	CHANGE-REQ <a,a.c.e.i> / CHANGE-REQ <i,i.h.g.f>	CHANGE-REQ <c,a.c.e.i> / CHANGE-REQ <h,i.h.g.f>	CHANGE-REQ <e,a.c.e.i>	CHANGE-REQ <g,i.h.g.f>
D	Con-Change = 'F' Status = 'matched'	Con-Change = 'F' Status = 'matched'	Con-Change = 'T' a.c.e.i ≠ i.h.g.f	Con-Change = 'T' Path = Con-Path Top(Path) = Top(Con-Path) Status = 'free' ST-Echoed(d) = 'F'
E	Con-Change := 'T' Con-Path := a.c.e.i send CHANGE-REQ <c,a.c.e.i> to e / {*45*} Con-Path := i.h.g.f send CHANGE-REQ <h,i.h.g.f> to g {*45*}	Con-Change := 'T' Con-Path := a.c.e.i send CHANGE-REQ <e,a.c.e.i> to i / {*45*} Con-Path := i.h.g.f send CHANGE-REQ <g,i.h.g.f> to f {*45*}	send NO-CHANGE <i,a.c.e.i> to e {*38*}	Status := 'matched' Mate := g send CHANGE <f,i.h.g.f> to g Skip {*40*}
F	28,28	29,30	31	34

Table 4.1. (Continued)

After a pure loop is split into a chain, Figure 4.9 shows these two augmenting paths in the chain. The free nodes a and i try to augment the matching. However, only one augmenting path (i,h,g,f) is augmented.

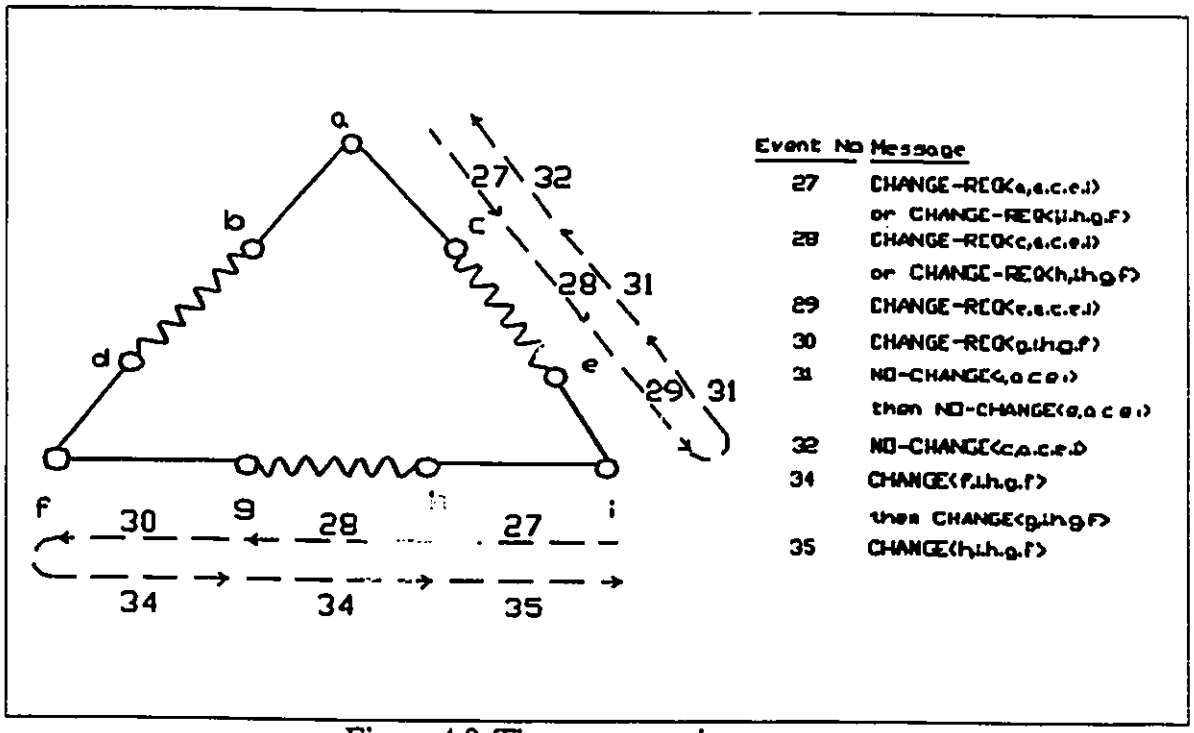


Figure 4.9. The augmentation process.

A	31	32	33	34
B	e;c	a	b;d	g;h
C	NO-CHANGE <i,a.c.e.i> ; NO-CHANGE <e,a.c.e.i>	NO-CHANGE <c,a.c.e.i>	ECHO-G<a,∅> ; ECHO-G<b,∅>	CHANGE<f,i.h.g.f> ; CHANGE<g,i.h.g.f>
D	Status = 'matched'	Status = 'free' ST-Echoed(c) = 'T'	ST-Echoed(a) = 'F'; ST-Echoed(b) = 'F' b,d ≠ Start-Node Status = 'matched'	Status = 'matched'
E	send NO-CHANGE <i,a.c.e.i> to c ; {*49*} send NO-CHANGE <c,a.c.e.i> to a {*49*}	send ECHO-G<a,∅> to b {*51*}	ST-Echoed(a) = 'T' Changed := ∅ send ECHO-G<b,∅> to d ; {*11*} ST-Echoed(b) = 'T' Changed := ∅ send ECHO-G<d,∅> to f {*11*}	Mate := f send CHANGE <g,i.h.g.f> to h ; {*48*} Mate := i send CHANGE <h,i.h.g.f> to i {*48*}
F	31,32	33	33,36	34,35

A	35	36	37	38
B	i	f	h	g
C	CHANGE<h,i.h.g.f>	ECHO-G<d,∅>	ECHO-G<i,{i,f}>	ECHO-G<f,∅> / ECHO-G<h,{i,f}>
D	Status = 'free' ST-Echoed(e) = 'T'	ST-Echoed(d) = 'F' f ≠ Start-Node Status = 'matched'	ST-Echoed(i) = 'F' h ≠ Start-Node Status = 'matched'	ST-Echoed(f,h) = 'F' g = Start-Node Free = {a,i,f}
E	Mate := h Status := 'matched' Changed := {i,f} send ECHO-G<i,{i,f}> to h {*47*}	ST-Echoed(d) = 'T' Changed := ∅ send ECHO-G<f,∅> to g {*11*}	ST-Echoed(i) := 'T' Changed := {i,f} send ECHO-G <h,{i,f}> to g {*11*}	ST-Echoed(f/h) := 'T' Changed := {i,f} Free := {a} Completed {*9*}
F	37	38	38	The End

Table 4.1. (Continued)

Figure 4.10a shows the matching after the augmenting process. Figure 4.10b shows how the nodes convergcast to the ST-root g.

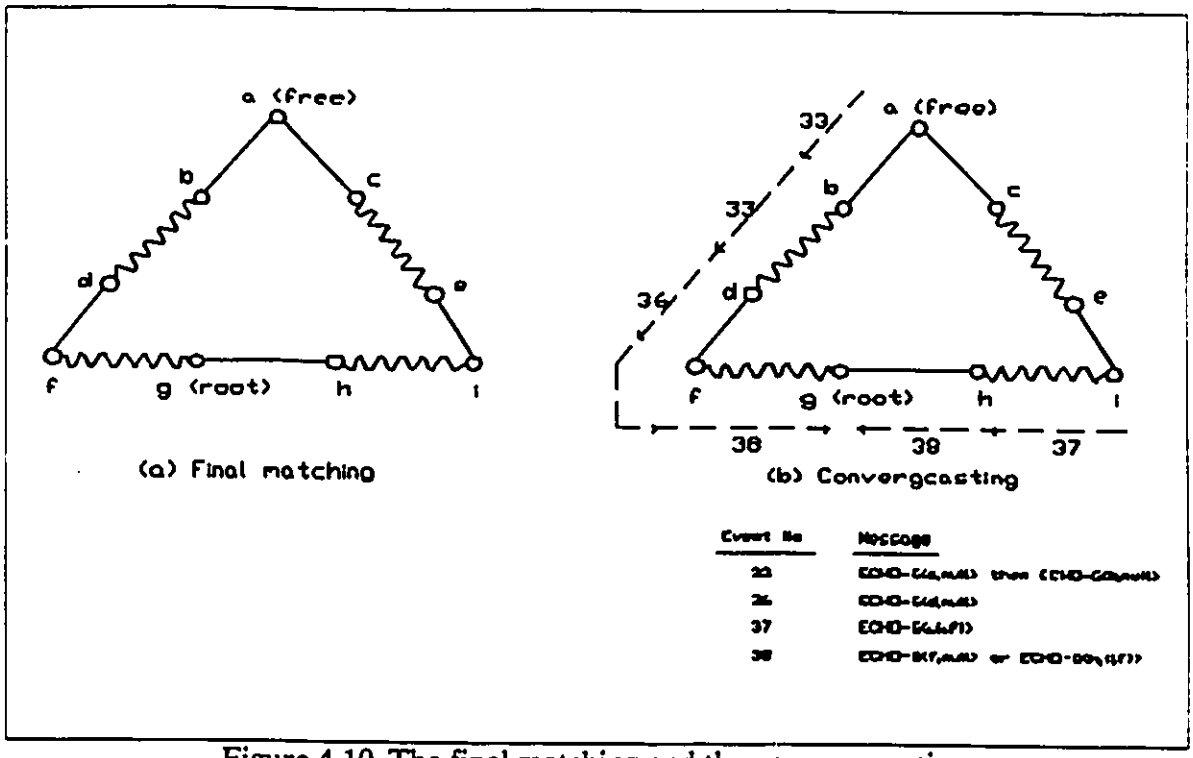


Figure 4.10. The final matching and the convergecasting.

4.4 Correctness and Complexity of DMM

To prove the correctness of Algorithm DMM, we have to show that DMM changes at least two nodes from 'free' to 'matched' in each phase, as stated in the following lemma :

Lemma 4.1. In each phase of Algorithm DMM, if there is any augmenting path, at least one of them will be discovered (and thus augmented).

Proof. (Without loss of generality, we can base our proof on Figure 4.11.)

Consider two VATs: VAT_1 consists of $p_1 = (r,a,b)$ and $q_1 = (r,f,e,c,d)$ and VAT_2 consists of $p_2 = (s,p,q)$ and $q_2 = (s,f,e,d,c)$. Suppose there exists an augmenting path $P = (r,a,b,c,d,e,f,s)$. We shall show that suppose P is not discovered in VAT_1, it will be discovered in VAT_2. Note that, in DMM, every VAT is extended to the maximum.

In VAT_1, in order to reach s, p_1 is blocked at the inner node c by q_1 and q_1 is blocked at e by itself. Since c is a matched node, it is always reachable in VAT_2 by an alternating path, say, q_2. Since P is alternating, q_2 will have no common nodes with at least one of those alternating path of VAT_1 (say, p_1) which are blocked at c. Hence, in VAT_2, q_2 can be extended further by DMM to include (b,a,r), the reverse of p_1, to form an augmenting path. This augmenting path

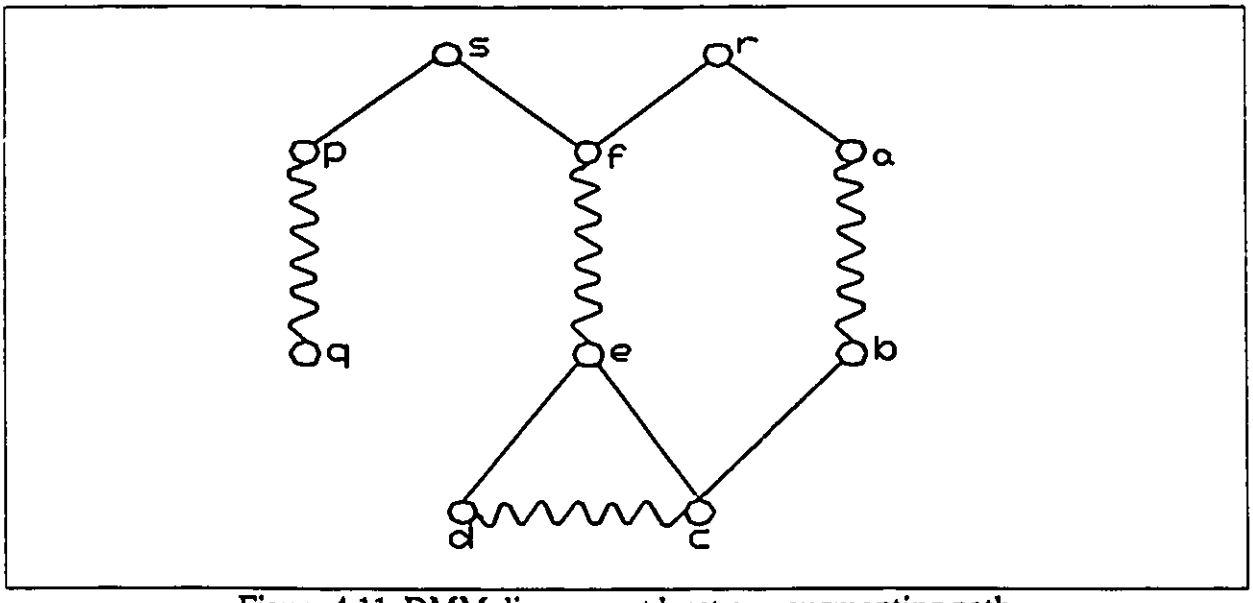


Figure 4.11. DMM discovers at least one augmenting path.

will not be blocked by any other alternating paths of VAT_2 (because of Rule 3 in Section 4.1). This means that, VAT_2, when extended to the maximum, discovers an augmenting path. ♦

Lemma 4.2. Algorithm DMM finds a maximum matching in at most n/2 phases, where n is the number of nodes of the graph.

Proof. Since at least one augmenting path is found in each phase and each such path changes the status of two nodes from 'free' to 'matched', at most n/2 phases are required. ♦

Theorem 4.1. The communication complexity of Algorithm DMM is O(mn), where n is the number of nodes and m is the number of edges of the graph.

Proof. The initial matching and the spanning tree can be obtained with communication complexity lower than O(mn). For example, if Algorithm DSM is used, it has communication complexity O(m). For constructing a spanning tree, the communication complexity is O(m) [CHAN 79]. For the augmentation process of DMM, Lemma 4.2 states that O(n) phases are needed. Hence, all we have to prove is that O(m) messages are transmitted in each phase.

Consider the four steps in the global-view description of DMM (Section 4.3). Step I (Initializes each node) Exactly n STARTPHASE and n READY are involved. Step II has two stages: (1) (Generating VAT's) Exactly n GENALT are involved. (2) (Searching for augmenting paths) Along each edge(x,y) at most 4 messages are exchanged starting at x. Since each edge (x,y) is involved in at most two VAT's (according to Rule 4 in Section 4.1), at most two EXPANDs are sent from x to y. In response to each EXPAND, either an AUGFOUND or an ECHO-S, is sent from y to x. Therefore, at most 4 messages are exchanged on (x,y) from x to y and at most 8 messages are transmitted along each edge. Step III (Augment the matching along the discovered augmenting paths). Since each edge (x,y) can be involved in at most one augmenting path p in this step. x forwards at most one CHANGE-REQ along p. In response to this CHANGE-REQ, either a CHANGE or a NO-CHANGE is sent from y to x. Therefore, starting from x, at most

2 messages are exchanged on (x,y). That means at most 4 messages are transmitted along each edge. Step IV (Inform the root of ST to end the phase) Exactly n ECHO-G are involved. ♦

Theorem 4.2. The time complexity of Algorithm DMM is $O(n^2)$, where n is the number of nodes.

Proof. The initial matching and the spanning tree can be obtained with time complexity lower than $O(n^2)$. For example, if Algorithm DSM is used, it has time complexity O(n). For constructing a spanning tree, the time complexity is O(n) [CHAN 79]. For the augmentation process of DMM, Lemma 4.2 states that O(n) phases are needed. Hence, all we have to prove is that the time complexity is O(n) in each phase.

In the worst case, the network is a linear structure. Consider the four steps in the global-view description of DMM (Section 4.1). Step I (Initialize each node) 2n units of time are needed to complete the initialization. Step II has two stages: (1) (Generating VAT's) n units of time are needed to broadcast GENALT. (2) (Searching for augmenting paths) 2n units of time are needed to expand a VAT in the linear network. Step III (Augment the matching along the discovered augmenting paths) 2n units of time are needed to augment the augmenting path on a linear network. Step IV (Inform the root of ST to end the phase) n units of time are needed to complete the convergecasting. ♦

Chapter 5

CONCLUSION

5.1 Comparison with Other Distributed Algorithms

We have presented two new asynchronous distributed algorithms (DSM and DMM) for finding spanning and maximum-cardinality matchings on an undirected graph. There exists no algorithm in the literature for finding spanning matching. The communication complexities of DSM and DMM are O(m) and O(mn), respectively. The time complexities of DSM and DMM are O(n) and $O(n^2)$, respectively. In Table 5.1, we compare DMM with the other two distributed maximum matching algorithms of Schieber (1986) and Wu (1987) existing in the literature.

	Schieber's	Wu's	DMM
Network types	sync.	async.	async.
Communication complexity	$O(n^2m)$	$O(n^{5/2})$	O(mn)
Time complexity	O(n log n)	$O(n^{5/2})$	$O(n^2)$
Global knowledge required	number of nodes	none	none

Table 5.1. Comparison of DMM with other distributed algorithms
(m = the number of edges, n = the number of nodes of the graph).

Since, in many practical cases, $n^{3/2}$ is greater than m, DMM has the lowest communication complexity.

5.2 Further Research

In the following, we point out an area in which our algorithm may be improved. In each phase, DMM augments at least one augmenting path because of the asynchronous nature of the augmenting routines. If DMM can augment all the discovered augmenting paths in each

phase, the number of phases will be reduced. Two possible ways are described in the following:

1) After a 'free' node receives an AUGFOUND with an augmenting path from its child, it does not try to augment that path. Instead, it immediately convergecasts back to the root of ST with that augmenting path. After the root receives all the convergecasting messages, DMM then augments the augmenting paths sequentially.
2) By adding some waiting facilities in augmenting procedures for both CHANGE and NO-CHANGE messages, some 'no-changed' augmenting paths may be augmented.

These two modifications reduce the number of phases but may not improve the complexities of our algorithm. By pursuing the research in this direction, we hope that a better distributed algorithm may be discovered.

Other problems for further consideration include distributed maximum weighted matching algorithms for both bipartite and general graphs, and distributed *b*-matching algorithm in a general graph. A *b*-matching of a given graph G is an assignment of integer weights to the edges of G so that the sum of the weights on the edges incident with a vertex v is at most b_v (b denotes the vectors of b_v's). When $b_v = 1$ for all vertices v in G, then *b*-matching is the maximum-cardinality matching. The *b*-matching problem [ANST 87] asks for a *b*-matching of maximum cost where the edges of G have been assigned costs and the cost of a *b*-matching is the sum of weights times the cost. No such distributed algorithms exist in the literature.

REFERENCES

[ANST 87] Anstee, Richard P., "A polynomial algorithm for *b*-matchings: An alternative approach", Information Processing Letters, Vol. 24 (1987), pp.153-157.

[AWER 84] Awerbuch, B., "An efficient network synchronization protocol", Proc. 16th ACM Symp. on the Theory of Computing, Washington, D.C., (1984), pp. 522-525.

[AWER 85a] Awerbuch, B., "Complexity of network synchronization", J. ACM, Vol. 32, No. 4 (1985), pp. 804-823.

[AWER 85b] Awerbuch, B., "Reducing complexities of the distributed max-flow and breadth-first-search algorithms by means of network synchronization", Networks, Vol. 15 (1985), pp. 425-437.

[AWER 85c] Awerbuch, B., "A new distributed depth-first-serach algorithm", Information Processing Letters., Vol. 20 (3) (1985), pp. 147-150.

[BERG 57] Berge, C., "Two theorems in graph theory", Proc. National Academic Science USA, Vol. 43 (1957), pp. 842-844.

[CHAN 79] Chang, E. J. H., "Decentralized algorithms in distributed systems", Ph.D dissertation, Technical Report CSRG-103, Univ. of Toronto, Toronto, Ontario, Canada (1979).

[CHAN 82] Chang, E. J. H., "Echo algorithms : depth parallel operations on general graphs", IEEE Trans. on Software Engineering, Vol. SE-8, No. 4 (1982), pp. 391-401.

[CHEU 83] Cheung, T. Y., "Graph traversal techniques and the maximum flow problem in distributed computation", IEEE Trans. on Software Engineering, Vol. SE-9 (4) (1983), pp. 504-512.

[CHEU 89] Cheung, T. Y., "An algorithm with decentralized control for sorting files in a network", Journal of Parallel and Distributed Computing, Vol. 7 (1989), pp. 464-481.

[DERI 80] Derigs, U., "Methods for solving the cardinality matching problem (a state of the art study)", Industrieseminar Universität zu Köln (1980).

[DERI 81] Derigs, U., "A shortest augmenting path method for solving minimal perfect matching problems", Networks, Vol. 11 (1981), pp. 379-390.

[EDMO 65] Edmonds, J., "Paths, trees and flowers", Canadian J. Mathematics, Vol. 17 (1965), pp. 449-467.

[EVEN 75] Even, S., and O. Kariv, "An O($n^{2.5}$) algorithm for maximum matching in general graphs", 16^{th} Annual Symp. on Fundations of Computer Science (1975), pp. 100-112.

[FLYN 66] Flynn, M. J., "Very high speed computing systems", Proc. IEEE, Vol. 14 (1966), pp. 1901-1909.

[GABO 76] Gabow, H., "An efficient implementation of Edmonds' algorithm for maximum matchings on graphs", J. ACM, Vol. 23, No. 2 (1976), pp. 221-234.

[GALI 86] Galil, Z., "Efficient algorithms for finding maximum matching in graphs", ACM Computing Surveys, Vol. 18, No. 1 (1986), pp.23-38.

[GALL 83] Gallagher, R., P. Humblet, and P. Spira, "A distributed algorithm for minimum-weight spanning trees", ACM Trans. on Programming Languages and Systems, Vol. 5, No. 1 (1983), pp. 66-77.

[GROT 81] Grötschel, M., and L. Lovász and A. Schrijver, "The ellipsoid method and its consequences in combinatorial optimization", Combinatorica, Vol. 1 (1981), pp. 169-197.

[GUAN 62] Guan, M., "Graphic programming using odd and even points", Chinese Mathematics, Vol. 1 (1962), pp. 273-277.

[HOPC 73] Hopcroft, J. E., and R. M. Karp, "An $n^{5/2}$ algorithm for matchings in bipartite graphs", SIAM J. Computing, Vol. 2, No. 4 (1973), pp.225-231.

[KARI 76] Kariv, O., "An O($n^{2.5}$) algorithm for maximal matching in general graphs", Ph.D dissertation, Dept. of Applied Mathematics, Weizmann Institute, Rehovot, Israel (1976).

[KRON 86] Kronsjö, L., Computational Complexity of Sequential and Parallel Algorithms, John Wiley & Sons (1986).

[LAVA 85] Lavallée, I., and C. Lavault, "Scheme for efficiency-performance measures of distributed and parallel algorithms", Distributed Algorithms on Graphs, Carleton University - Distributed Computing Group (1985), pp.69-101.

[LOVA 86] Lovász, L., and M. D. Plummer, Matching Theory, North-Holland Mathematics Studies 121, Elsevier Science Publishing Company Inc. (1986).

[MICA 80] Micali, S., and V. V. Vazirani, "An O($n^{1/2}m$) algorithm for finding maximum matching in general graphs", 21st Annual Symp. on Foundations of Computer Science, Syracuse, New York, (1980), pp. 17-27.

[PAPE 80] Pape, U., and D. Conradt, "Maximales matching in graphen", Ausgewählte Operations Research Software in FORTRAN, Oldenburg, Munich (1980), pp. 103-114.

[PETE 87] Peterson, P. A., and M. C. Loui, "The general maximum matching algorithm of Micali and Vazirani", Technical Report T-163, Coordinated Sci. Lab., Univ. of Illinois at Urbana-Champaign (1985).

[SCHI 86] Schieber, B., and S. Moran, "Slowing sequential algorithms for obtaining fast distributed and parallel algorithms : maximum matchings", ACM Symp. on Principles of Distributed Computing, Calgary, Alberta, Canada, Vol. 5 (1986), pp. 282-292.

[SCHN 77] Schneider, H., "The concepts of irreducibility and full indecomposability of a matrix in the works of Frobenius, König & Markov", Linear Algebra Appl., Vol. 18 (1977), pp. 139-162.

[SHAM 82] Shamir, E., and E. Upfal, "N-processors graphs distributively achieve perfect matchings in O($\log^2 N$) beats", ACM Symp. on Principles of Distributed Computing, Ottawa, Ontario, Canada, Vol. 1 (1982), pp. 238-241.

[WITZ 65] Witzgall, C., and C. T. Zahn, Jr., "Modification of Edmonds' maximum matching algorithm", J. Res. National Bureau Standards, Vol. 69B (1965), pp. 91-98.

[WU 87] Wu, M. M., "An efficient distributed algorithm for maximum matching in general graphs", Master Thesis, Univ. of of Illinois at Urbana-Champaign (1987).

[WU 90] Wu, M. M., and Loui, M. C., "An efficient distributed algorithm for maximum matching in general graphs", Algorithmica, Vol. 5 (1990), pp.383-406.

[ZHU 87] Zhu, Y. and T.Y. Cheung; "A new distributed breadth-first-search algorithm", Information Processing Letter, Vol. 25 (1987), pp. 329-333.