DISTRIBUTED ALGORITHMS FOR CYCLE FINDING AND MATCHING

by

Philip Sau-Tak Pun

B.Math. (Honors), University of Waterloo, 1982

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the School

of

Computing Science

SIMON FRASER UNIVERSITY

April 1986

APPROVAL

Name: Philip Sau-Tak Pun

Degree: Master of Science

Title of Thesis: Distributed Algorithms for Cycle Finding and Matching

Examining Committee:

Chairperson: Dr. Robert F. Hadley

Dr. Arthur L. Liestman
Senior Supervisor

Dr. Binay Bhattacharya

Dr. Joseph Peters

Dr. Anthony H. Dixon
External Examiner
School of Computer Science
Simon Fraser University

10 April 1986
Date Approved

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

Distributed Algorithms for Cycle Finding and Matching

Author:

(signature)

Philip Sau-Tak Pun

(name)

16 April 1986

(date)

ABSTRACT

As microtechnology becomes more advanced, it is necessary to consider local networks with thousands of processors as opposed to the tens that were possible with minicomputers alone. As local networks become larger and larger, it is infeasible to store global topological information at each processor. In addition to the extra memory required to store this information, every time there is a change in the network topology, all the processors would need to be notified to update their topological information. This would require a considerable amount of time and would affect the mutual consistency of the topological information. It is therefore a practical and important problem to design algorithms that are fully distributed, which require only local topological knowledge at each processor during execution.

We study two graph problems: **cycle finding** and **maximum matching**. A synchronous cycles finding algorithms, is presented with message complexity $O(e(d-1)^{\lfloor k/2 \rfloor - 1})$, where d is the maximum degree and e is the number of edges in the graph for cycles of length k. An asynchronous algorithm with message complexity $O(n^2e)$, where n is the number of vertices in the graph, for maximum matching is also presented. The algorithm for maximum matching has been implemented and tested on a variety of graphs using a distributed network simulator.

ACKNOWLEDGEMENTS

My greatest thanks to my Senior Supervisor Dr. Arthur Liestman, for supporting and helping me throughout my work on this research, in particular, his enthusiasm and patience to proofread this thesis. Also, thanks to my committee members, Dr. Binay Bhattacharya, Dr. Joseph Peters and to the external examiner Dr. Anthony Dixon for helping with the revision of the thesis. Special thanks to Judy Muir for allowing us to use her DC package to test our algorithms.

To my parents.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

As microtechnology becomes more advanced, it is necessary to consider local networks with thousands of processors as opposed to the tens that were possible with minicomputers alone. As local networks become larger and larger, it is infeasible to store global topological information at each processor. In addition to the extra memory required to store this information, every time there is a change in the network topology, all the processors would need to update their topological information. This would require a considerable amount of time and would affect the mutual consistency of the topological information. It is therefore a practical and important problem to design algorithms that are fully distributed, which require only local topological knowledge at each processor during execution.

Many distributed algorithms have been proposed, but this field of research is still young. Some of the problems which have been studied are: finding an extremum, also known as electing a leader (Chang & Roberts [ChR79]; Dolev et al. [DKR82]; Peterson [Pet82]; Matsushita [Mat83]), determining medians (Rodeh [Rod82]; Santoro & Sidney [SaS82]; Matsushita [Mat83]), ranking (Korach et al. [KRS82, KRS84]), selection (Shrira et al. [SFR83]; Rotem et al. [RSS84]), finding minimum-weight spanning trees (Spira [Spi77]; Parker & Samadi [JrS81]; Gallager et al. [GHS83]), knot detection (Misra & Chandy [MiC82]), sorting (Loui [Lou84]; Rotem et al. [RSS83]), shortest paths (Chandy & Misra [MiC82]; Chen [Che82]; Lakhani [Lak84]), and maximum flows (Segall [Seg82]). These papers use a variety of models of distributed computation.

The model used here requires no centralized control, admits no shared memory and permits data transfer only on a communication network. Each processor has an associated distinct identity, which is known only to itself. No processor has a priori knowledge of the number of processors in the network. Each processor communicates by exchanging messages with its neighbors in the network, and can detect from which neighbor a message is received. A link of the communication network is an un-ordered pair of processors on which messages can be sent in both directions. Processor x can send a message directly to processor y if and only if link (x,y) is in the network. The computation can be started at one or several processors and the result of the computation will be known at one or more processors. It is assumed that each processor executes an identical algorithm. Each processor is assumed to have an unbounded input buffer in the form of a queue from which the message must be removed in FIFO order. If processor x sends a message to its neighbor y, the message is appended to the end of y's input queue after a finite, arbitrary delay. These assumptions hold for the algorithms presented later unless stated otherwise.

CHAPTER 2

FINDING CYCLES OF A GIVEN LENGTH

2.1. Definitions and Previous Results

Let $G = (V,E)$ be a graph that represents a general communication network, with $|V| = n$ and $|E| = e$. The nodes and edges of the graph correspond to the processors and communication links of the network, respectively. We will use these terms interchangeably. A cycle is defined to be a path $(v_{i1},v_{i2}, \cdots ,v_{ik})$ in which the terminal vertex v_{ik} coincides with the initial vertex v_{i1} and which does not include any vertex twice. The length of such a cycle is the number of edges encountered.

It is known that finding the longest cycle in a graph is an NP-complete problem [GaJ79]. A sequential algorithm to find the shortest cycle in a graph was devised by Itai and Rodeh [ItR77] in average time $O(n^2)$. Finding a cycle of specified length was found to be solvable in polynominal time by Itai & Rodeh [ItR77] and Richards & Liestman [RiL85]. Itai & Rodeh present three algorithms to find a cycle of length 3 if one exists. The three algorithms are $O(n^{3/2})$, $O(ne)$, and $O(n^{\log_2(7)})$ in the worst case. Richards & Liestman present algorithms requiring times of $O(n^2)$, $O(n^3)$, $O(n^k k^{5/2})$ and $O(n^{k+1}k^{5/2})$ for finding a cycle of length 4, 5, $2k$ (for $k \geq 3$) and $2k + 1$ (for $k \geq 3$) in G, respectively, if one exists.

In this chapter we present distributed algorithms to find a cycle of length k in our network G with message complexity $O(e(d-1)^{\lfloor k/2 \rfloor - 1})$, where d is the maximum vertex degree. The algorithms are based on a breadth-first-search and are similar to one of the algorithms of Itai & Rodeh [ItR77]. If the network G contains cycles of the specified length (k), the algorithm may identify more than one such cycle in the network G. A subsequent election

could be used to select one of the cycles found if exactly one such cycle is required. We do not consider this election process when analyzing our algorithms. In this chapter, the word cycle will be used to denote a cycle of length k.

2.2. Preliminary Algorithm for Cycles of Even Length

In this chapter we make the following additional assumptions: 1) each processor can inspect its own input buffer in any order, but messages must be removed in FIFO order, 2) the network is phase synchronous, that is, each processor can locally determine the starting time for each phase, 3) each message transmitted in phase i arrives at beginning of phase i+1, and 4) a processor does not know the names of its neighbors but can distinguish among its incident edges. Initially all processors are active and will initiate messages at the same time. In each phase sufficient time is allowed so that every processor receives all the messages from its neighbors before the start of the next phase.

The algorithm is divided into two stages: cycle detection and cycle announcement. If a cycle of the specified length exists in G, members of the cycle found will detect its existence at the end of the cycle detection stage. All the members on that cycle will then be notified that they are the members of the cycle during the cycle announcement stage. If there is more than one cycle in G, the algorithm is not guaranteed to identify all of them but at least one cycle will be detected. After a processor detects a cycle, it will stop the cycle detection stage locally. At the end of the cycle announcement stage, a processor which has not been notified that it is in a cycle knows that the algorithm was terminated, but does not know whether it is in any cycle or not.

The algorithm uses two types of messages: explorer and announcement. An explorer is of the form *<ORIGINATOR,ID,PATH,COUNT>*, where *ORIGINATOR* contains the id of the originator of the message, *ID* is a list which contains the identities of those processors

which the message has visited in sorted order. *PATH* is a list which contains all the identities in *ID* in the order in which they were visited, and *COUNT* contains the length of the path traversed so far. Initially, each processor sends messages to all of its neighbors with its own id as *ORIGINATOR*, *COUNT* set to one and both *ID* and *PATH* set to nil. These messages then traverse the network in a breadth-first manner. In the cycle detection stage, each processor receiving a message of the above format determines whether the message has traveled less than $k/2$ edges and has not been to the current processor before (either as originator or along the path). If so, the processor inserts its own identity into *ID* (a list maintained in ascending order), appends its own identity to the end of *PATH*, and increments the variable *COUNT*. The message is then relayed to all its other neighbors. If the message has previously visited this processor, the message is discarded. A cycle of length k is found when a processor receives two messages which traveled exactly a distance of $k/2$ with disjoint sets of *ID*'s and matching *ORIGINATOR*. If such a cycle exists, the processor sends an announcement message to the members of the cycle found. An announcement message is of the form <*CYCLE*,*COUNT*,*INDEX*>, where *CYCLE* is a list which contains the identities of the processors in the cycle in the cycle order, *COUNT* is the distance the message needs to travel, *INDEX* is an index which indicates the position of the id of the most recent sender of the message in *CYCLE*. We use $CYCLE_{INDEX}$ to represent the $INDEX^{th}$ element in the list *CYCLE*. The processor which detects the cycle can construct *CYCLE* from the two *PATH* lists, the *ORIGINATOR* and it own id. There are two ways to perform the cycle announcement: 1) send an announcement message along the cycle with *COUNT* set to $k-1$, and 2) send two announcement messages around the cycle in opposite directions with *COUNT* set to $k/2$ and $k/2-1$, respectively. In each case, a processor which receives an announcement message, saves the *CYCLE*, decrements the value of *COUNT*, and then relays the message to its other neighbor on the cycle if *COUNT* > 0. The algorithm in the next section uses the former method above to ease the explanation and analysis of the algorithm.

2.2.1. Algorithm

Step 1 and each iteration of steps 2 and 3 require one synchronous phase. The algorithm executes steps 1 and 2 during the cycle detection stage and step 3 during the cycle announcement stage. The correctness of the algorithm is shown in Theorem 2.1. The cycle finding algorithm for processor P is as follows:

(1) Send the message $<P,NIL,NIL,1>$ to all neighbors.

(2) For phase = 1 to $k/2$ DO
  WHILE more incoming messages from the previous phase DO
    take a message $<ORIGINATOR_p,ID_p,PATH_p,COUNT_p>$ from the input queue
    IF $P \neq ORIGINATOR_p$ and P is not contained in ID_p THEN DO
      IF $COUNT_p < k/2$ THEN DO
        insert P into ID_p
        append P to $PATH_p$
        send $<ORIGINATOR_p,ID_p,PATH_p,COUNT_p+1>$ to all other neighbors
        OD
      ELSE IF $COUNT_p = k/2$ THEN DO
        IF there is a message $<ORIGINATOR_q,ID_q,PATH_q,COUNT_q>$ in the input queue, s.t.
        1. $ORIGINATOR_q = ORIGINATOR_p$,
        2. ID_q does not contain P,
        3. $COUNT_q = k/2$, and
        4. ID_p and ID_q are disjoint THEN DO
            set up the $CYCLE$ list as follows:
              i) starting with P,
              ii) followed by the ids in $PATH_p$,
              iii) then $ORIGINATOR_p$, and
              iv) followed by the ids in $PATH_q$ in reverse order.
            empty the input buffer
            send $<CYCLE,k-1,1>$ to $CYCLE_2$
            OD
        OD
      OD
    OD
  OD.

(3) For phase = 1 to $k-1$ DO
  WHILE more incoming messages from the previous phase DO
    take a message $<CYCLE,COUNT,INDEX>$ from the input queue
    save the $CYCLE$
    IF $COUNT > 1$ and $P = CYCLE_{INDEX-1}$ THEN
      send $<CYCLE,COUNT-1,INDEX-1>$ to $CYCLE_{INDEX-2}$
    ELSE IF $COUNT > 1$ and $P = CYCLE_{INDEX+1}$ THEN
      send $<CYCLE,COUNT-1,INDEX+1>$ to $CYCLE_{INDEX+2}$
    OD
  OD.

Theorem 2.1: If there is a cycle of length k in the network, the algorithm will find it.

Proof: Assume that a cycle of length k exists. Each processor will initiate messages containing its own *ID*, and relay messages received to each of the other neighbors after incrementing *COUNT*. The messages initiated by the members of the cycle will travel around the cycle in both directions. Each pair of them will meet half way around the cycle and cause the termination of the cycle detection stage. In this case, an announcement message is formed an being relayed around the cycle found until *COUNT* reached 0. If no such cycle exists, each message will be relayed until *COUNT* exceeds $k/2$. At this point, the message will be ignored. The cycle detection stage terminates when all of the messages *COUNT*s exceed $k/2$. The cycle announcement stage terminates when all of the message *COUNT*s exceed $k-1$.

□

2.2.2. Message Complexity

In phase 0 of the cycle detection stage, all processors initiate their messages (step 1). During the other phases of the cycle detection stage, processors analyze each message in the input buffer (from the previous phase) and relay the message to each of the other neighbors (at most $d-1$) after incrementing the variable *COUNT*. During each phase of the cycle announcement stage, processors analyze each message in the input buffer and relay the message to one neighbor after changing the variable *COUNT* and *INDEX*. Since each processor detects at most one cycle, at most n announcement messages are sent in each phase of this stage. Each of these messages will be forwarded to at most $k-1$ vertices.

Let e be the number of links and d be the maximum degree of the network. The number of messages sent in the first stage is as follows:

phase 0 : $2e$ messages sent,
phase 1 : $\leqslant 2e(d-1)$ messages sent,
phase 2 : $\leqslant 2e(d-1)^2$ messages sent,

.
.

phase $k/2-1$: $\leqslant 2e(d-1)^{k/2-1}$ messages sent.

Notice that, phase $k/2$ of the cycle detection stage coincides with the phase 0 of the cycle announcement stage. The number of messages sent in each phase of the second stage is as follows:

phase 0 : $\leqslant n$ messages sent,
phase 1 : $\leqslant n$ messages sent,

.
.

phase $k-2$: $\leqslant n$ messages sent.

Therefore, the total number of messages sent is

$$2e[1 + (d-1) + (d-1)^2 + \cdots + (d-1)^{k/2-1}] + n(k-1)$$

$$\leqslant \begin{cases} ke + n(k-1) & d=2 \\ 4e(d-1)^{k/2-1} + n(k-1) & d \geqslant 3. \end{cases}$$

2.2.3. Time complexity for local processing

We use the term local processing time to denote the worst case time complexity of a single processor in the network. The local time complexity of the algorithm during the cycle detection stage is dominated by the testing parts in various phases. The time required to transmit the initial messages is at most d units of time. During phase i, for i = 1 to $k/2$ - 2, each message received will be checked for the distance it has traveled and $\log_2(i)$ comparisons are used to determine whether it has visited the current processor before. Each valid message is forwarded to at most $d-1$ other processors. The time required during these phases is at most

$$\sum_{i=1}^{i=k/2-2} [(2 + log(i) + (d-1)) \times (2e(d-1)^i)]$$

$$= \sum_{i=1}^{i=k/2-2} 4e(d-1)^i + \sum_{i=1}^{i=k/2-2} 4elog(i)(d-1)^i + \sum_{i=1}^{k/2-2} 2e(d-1)^{i+1}$$

$$\leq 8e(d-1)^{k/2-2} + 4e(d-1)^{k/2-1} + 4e\sum_{i=1}^{i=k/2-2} log(i)(d-1)^i.$$

So the time required in the last phase is at most $4ke^2(d-1)^{k-2} + 4e^2log(k)(d-1)^{k-2} + 8e^2(d-1)^{k-2}$. In phase $k/2 - 1$, each message received requires at most $2e(d-1)^{k/2-1}(k + log(k) + 2)$ comparisons to go through the messages in the input buffer. During the cycle announcement stage, the total local time complexity is at most $2nk$. Thus, the total local processing time is $O(ke^2(d-1)^{k-2})$.

2.3. An Improved Algorithm for Cycles of Even Length

In the above algorithm, every cycle of length k is discovered by k processors. Each of them sends two messages half way around the cycle. Our strategy is to try to eliminate this redundancy by allowing only the messages initiated by the processor with the smallest *ID* on the cycle to survive. This can be achieved, in part, if each processor relays to its neighbors only those messages that contain an *ORIGINATOR* smaller than its own.

2.3.1. Algorithm

(1) Send the message $<P,NIL,NIL,1>$ to all neighbors.

(2) For phase = 1 to $k/2$ DO
  WHILE more incoming messages from the previous phase DO
    take a message $<ORIGINATOR_p,ID_p,PATH_p,COUNT_p>$ from the input queue
    IF $P \neq ORIGINATOR_p$ and P is not contained in ID_p THEN DO
      IF $COUNT_p < k/2$ THEN DO
        IF P is $>$ the $ORIGINATOR_p$ THEN DO
          insert P into ID_p
          append P to $PATH_p$
          send $<ORIGINATOR_p,ID_p,PATH_p,COUNT_p+1>$ to all other neighbors
          OD
        DO
      ELSE IF $COUNT_p = k/2$ THEN DO
        IF there is a message $<ORIGINATOR_q,ID_q,PATH_q,COUNT_q>$ in the input queue, s.t.
          1. $ORIGINATOR_q = ORIGINATOR_p$,
          2. ID_q does not contain P,
          3. $COUNT_q = k/2$, and

4. ID_p and ID_q are disjoint THEN DO
set up the *CYCLE* list as follows:
i) starting with P,
ii) followed by the ids in $PATH_p$,
iii) then $ORIGINATOR_p$, and
iv) followed by the ids in $PATH_q$ in reverse order.
empty the input buffer
send <*CYCLE*,$k-1$,1> to $CYCLE_2$
OD
OD
OD
OD
OD.

(3) For phase = 1 to $k-1$ DO
WHILE more incoming messages from the previous phase DO
take a message <*CYCLE*,*COUNT*,*INDEX*> from the input queue
save the *CYCLE*
IF *COUNT* > 1 and $P = CYCLE_{INDEX-1}$ THEN
send <*CYCLE*,*COUNT*-1,*INDEX*-1> to $CYCLE_{INDEX-2}$
ELSE IF *COUNT* > 1 and $P = CYCLE_{INDEX+1}$ THEN
send <*CYCLE*,*COUNT*-1,*INDEX*$+1$> to $CYCLE_{INDEX+2}$
OD
OD.

Again step 1 and each iteration of steps 2 and 3 of the modified algorithm require one synchronous phase.

2.3.2. Correctness of the modified algorithm

Before analyzing the message complexity of the modified algorithm we present the following lemmas which show that the modified algorithm does find such a cycle if one exists.

Lemma 2.2: If there is a cycle of length k, at least one processor will discover its existence.

Proof: Assume there exists a cycle $C=(c_1,c_2,\cdots,c_k)$, and that processor c_1 has the smallest *ID* in C. The messages initiated by c_1 will be relayed by the other processors of C since c_1 is the smallest identity among them, and will reach $c_{k/2}$.

□

Lemma 2.3: If a cycle of length k exists, at most two pairs of messages travel completely around the cycle.

Proof: By Lemma 2.2, there is at least one such pair of messages. Let us assume that c_1 has the smallest and c_i the second smallest id in C. Any message initiated by any other processor will be stopped by either c_1 or c_i if it reaches them. The messages initiated by c_i will only be stopped by c_1. If $i=k/2$, c_i's messages will reach c_1 and c_1's messages will reach c_i.

□

2.3.3. Message Complexity

In phase 0 of the cycle detection stage, all processors initiate messages. During the other phases of the cycle detection stage, processors analyze and may relay the messages received. Notice that two messages sent between two incident processors during the first phase will only have one survivor as we assumed all the ids are distinct. The number of messages sent in each phase of the first stage is as follows:

phase 0 :	$2e$ messages sent,
phase 1 :	$\leqslant e(d-1)$ messages sent,
phase 2 :	$\leqslant e(d-1)^2$ messages sent,
.	
.	
phase $k/2-1$:	$\leqslant e(d-1)^{k/2-1}$ messages sent.

The number of messages sent during the cycle announcement stage is at most $n(k-1)$. Therefore the total number of messages sent is

$$e + e[1 + (d-1) + (d-1)^2 + \ldots + (d-1)^{k/2-1}] + n(k-1)$$

$$\leqslant \begin{cases} e + 1/2\ ke + n(k-1) & d=2 \\ e + 2e(d-1)^{k/2-1} + n(k-1) & d \geqslant 3 \end{cases}$$

During the notification stage at most $2k$ announcement messages will travel around each cycle found instead of k^2 in the preliminary algorithm. This is a significant improvement in the number of message sent.

2.3.4. Local processing time

As for the previous algorithm, the local processing time in each phase depends on the number of messages sent during the previous phase. The reduction of the number of messages sent substantially reduces the overall processing time. However, some additional local processing results from the necessity of determining whether P is smaller than the *ORIGINATOR* of the message received. Thus the improved algorithm reduces both the number of messages sent and local processing time.

2.4. An Algorithm for Cycles of Odd Length

There are two obvious ways to extend the algorithm of Section 2.2 to work for odd length cycles: 1) finding two disjoint paths of length $\lfloor k/2 \rfloor$ and $\lfloor k/2 \rfloor$, respectively or 2) build two length $\lfloor k/2 \rfloor$ disjoint paths from endpoints of an edge. We choose the latter method which requires fewer messages in the average case. After we present a simple version of the algorithm, we will show how to reduce the number of messages sent.

To find a cycle of odd length, every edge becomes a "core" of a search. The *ID* of a core is the ordered set of ids of the two incident nodes. Messages containing the core id will be initiated by both nodes. The algorithm of Section 2.2 can be modified easily by adding a preliminary phase in which every processor informs his neighbors of his own id. Each processor then forms a list of cores to which it belongs and then begins the cycle detection stage replacing "originator id" with "core id". The resulting algorithm requires $2de$ initial messages rather than the $2e$ message used in Section 2.2.

To reduce the number of messages sent, we observe that all of the initial messages for any node sent along any one of its edges propagate along the same path until they reach the specified lengths. We may combine these messages into a single message with many core ids, and proceed as in Section 2.2. If such a message arrives at a node that has been visited before (either as originator or along the path), discard the message. Otherwise, remove the id of the current node from the core ids if it is there. Two messages determine a cycle of length k if they have a common core id, all of the intermediate ids encountered are distinct, and the *COUNT* value are both $\lfloor k/2 \rfloor$. Each processor in the network initiates only one message to each neighbor and then relays messages received. This results in an algorithm with $2e$ initial messages.

Further improvement can be made by applying the technique of Section 2.3. This does not guarantee a substantial reduction in the number of messages sent as it did in Section 2.3 but the total number of bits sent in all the messages can be reduced. We define the current "core id" of a processor to be its own *ID* together with the *ID* of the neighbor that sent it the current message. A core id (a,b) is said to be smaller than core id (c,d) if and only if $a<c$ or if $a=c$ and $b<d$. Recall that $a<b$ and $c<d$. In this way, a processor only relays those messages that have a core id smaller than its own current core id. If a message contains more than one core id, we remove the *ID*s from *CORE* whose corresponding core id is greater than the current core id. The message is then updated and relayed to all the other neighbors if the *CORE* field of the message is not empty. Otherwise, the message is discarded.

The algorithm uses messages with five fields: *ORIGINATOR*, *CORE*, *ID*, *PATH* and *COUNT*. The *ORIGINATOR* contains the id of the originator of the message. *CORE* is a list which contains the *ID*s of the originator's neighbors: each one of them together with the originator corresponds to a core id. *ID* is a sorted list which contains the identities of those

processors which the message has visited. *PATH* is a list which contains the identities in *ID* in the order which they were visited. *COUNT* contains the length of the path traversed and is initialized to one. The algorithm is similar to the algorithm of the previous section, with the conditions for relaying a message modified. When a message arrives at node *P*: 1) If *P* is contained in the *ID* field of the message, delete the message. 2) If *P* is the originator of the message, delete the message. 3) If *P* is contained in *CORE* and is not the only id in *CORE*, remove it from the *CORE*, insert the id of *P* into *ID*, and append the id *P* to *PATH*. 4) If *P* is the only id in *CORE*, delete the message. If the message has not been deleted, increment *COUNT*. If $COUNT < \lfloor k/2 \rfloor$, send the message to all neighbors other than the neighbor from which it was received. If two messages have traveled exactly $\lfloor k/2 \rfloor$ edges, the *ORIGINATOR* of each message is in the *CORE* of the other message and the two *ID* fields are disjoint, then a cycle of length k has been found. The cycle found contains the two *ORIGINATOR*s and all the ids in the *PATH* fields. The members of the cycle can now be informed as in the previous algorithms.

2.4.1. Algorithm

The algorithm for processor *P* is as follows:

(1) Send the message $<P>$ to all the neighbors.

(2) Sort the ids received from neighbors and store them in *CORE*. Send the message $<P,CORE,NIL,NIL,1>$ to all neighbors.

(3) For phase = 1 to $k/2$ DO
While more incoming message from the previous phase DO
take a message $<ORIGINATOR_p, CORE_p, ID_p, PATH_p, COUNT_p>$ from the input queue
If $P \neq ORIGINATOR_p$ and *P* is not contained in ID_p Then DO
If $COUNT_p < \lfloor k/2 \rfloor$ Then DO
If the size of $CORE_p - P \geq 1$ Then
remove *P* from $CORE_p$
insert *P* into ID_p
append P to $PATH_p$
For id $\in CORE_p$ DO
If the core id corresponding to id > the current core id Then

remove id from $CORE_p$
OD
If $CORE_p \neq NIL$ Then
send $<ORIGINATOR_p, CORE_p, ID_p, PATH_p, COUNT_p+1>$ to all other neighbors
OD
OD
Else If $COUNT_p = \lfloor k/2 \rfloor$ Then
If there has a message $<ORIGINATOR_q, CORE_q, ID_q, PATH_q, COUNT_q>$ is in the input queue, s.t.
1. $ORIGINATOR_p$ is in $CORE_q$,
2. $ORIGINATOR_q$ is in $CORE_p$,
3. ID_q does not contain P,
4. both $COUNT_p$ and $COUNT_q = \lfloor k/2 \rfloor$, and
5. ID_p & ID_q are disjoint THEN DO
set up the $CYCLE$ list as follows:
i) starting with P,
ii) followed by the ids in $PATH_p$,
iii) then $ORIGINATOR_p$, and
iv) followed by the ids in $PATH_q$ in reverse order.
empty the input buffer
send $<CYCLE, k-1, 1>$ to $CYCLE_2$
OD
OD
OD
OD.

(4) For phase = 1 to $k-1$ DO
WHILE more incoming messages from the previous phase DO
take a message $<CYCLE, COUNT, INDEX>$ from the input queue
save the $CYCLE$
IF $COUNT > 1$ and $P = CYCLE_{INDEX-1}$ THEN
send $<CYCLE, COUNT-1, INDEX-1>$ to $CYCLE_{INDEX-2}$
ELSE IF $COUNT > 1$ and $P = CYCLE_{INDEX+1}$ THEN
send $<CYCLE, COUNT-1, INDEX+1>$ to $CYCLE_{INDEX+2}$
OD
OD.

Note that step 1 and each iteration of steps 2 and 3 require one synchronous phase as in previous sections.

2.4.2. Message Complexity

In the preliminary phase, each processor sends its id to its neighbors. After collecting all of the neighbors' ids, each processor initiates messages in the next phase. During the

later phases of the first stage, processors analyze, modify and relay the messages received. The number of messages sent in each phase of the first stage is as follows:

phase 0 :	$2e$ messages,
phase 1 :	$2e$ messages,
phase 2 :	$\leqslant 2e(d-1)$ messages,
phase 3 :	$\leqslant 2e(d-1)^2$ messages,
.	.
.	.
phase $\lfloor k/2 \rfloor$:	$\leqslant 2\,e\,(d-1)^{\lfloor k/2 \rfloor - 1}$ messages.

Since at most $n(k-1)$ announcement messages sent during the second stage. Therefore the total number of messages sent is at most

$$2e + 2e\,(1 + (d-1) + (d-1)^2 + \ldots + (d-1)^{\lfloor k/2 \rfloor - 1}) + n(k-1)$$

$$\leqslant \begin{cases} e + ke + n(k-1) & d=2 \\ 2e + 4e(d-1)^{\lfloor k/2 \rfloor - 1} + n(k-1) & d \geqslant 3. \end{cases}$$

2.5. Conclusion

The synchronous algorithms derived in this chapter successfully detect a cycle if one exists, and terminate even if no such cycle exists. The message complexity for the algorithms to find cycles of length k for both even and odd k is $O(ed^{\lfloor k/2 \rfloor - 1})$ with local processing time $O(ke^2d^{k-2})$. A straightforward sequential implementation of our algorithm would use time $O(nd^{k/2+1})$. Although it is desirable to have an asynchronous algorithm for this problem, we have been unable to devise such an algorithm. The main difficulty appears to be terminating the algorithm when no such cycle exists.

CHAPTER 3

MAXIMUM MATCHING FOR GENERAL GRAPHS

3.1. Introduction

Let $G = (V,E)$ be a simple connected graph with $|V| = n$ and $|E| = e$. The graph represents a communication network, where the nodes and edges of the graph correspond to the processors and communication links of the network respectively. A subset M of E is called a matching in G if no two edges of M are adjacent in G. If an edge is contained in M, then it is said to be **matched**, otherwise it is said to be **unmatched**. A maximum matching M is a matching whose cardinality is maximum. A node is **free** or **unmatched** if all the edges incident on it are unmatched, and **matched** otherwise. An **augmenting path** is a simple path between two free nodes whose edges are alternately in M and not in M.

The problem of finding sequential algorithms to find a maximum matching has long been studied. In 1957 Berge [Ber57] proved that a matching is maximum if and only if the graph has no augmenting paths. Since then many researchers have proposed algorithms for the problem [EvK75, Gab72, HoK73, MiV80] . Currently, the best algorithm has a running time of $O(n^{1/2} e)$ for general graphs.

In this chapter, we present an asynchronous distributed algorithm to find a maximum matching in the network G using $O(n^2 e)$ messages. The nodes do not need to know the topology of the graph, the maximum degree or the size of the graph. Although the algorithm is not fully decentralized, the central control processor will be altered from time to time throughout the execution of the MATCHING algorithm.

The algorithm presented in this chapter is a distributed version of Witzgall & Zahn's algorithm [WiZ65], which is a modification of Edmonds' algorithm [Edm65]. Both of these algorithms search for outer vertices, although their definitions of **outer vertex** are slightly different. Edmonds defined outer vertices to be those vertices which are free (with respect to a maximum matching). Witzgall & Zahn observed that the notion of outer vertex is closely related to accessibility by simple alternating paths. They defined a vertex v to be an outer vertex rooted at some vertex r, if r is an unmatched vertex and there is a simple alternating path of even length from r to v. Otherwise, v is an **inner vertex.** The latter definition is used throughout this chapter. Witzgall & Zahn restated Berge's theorem as: A matching is maximum if and only if no free vertex is adjacent to an outer vertex which is rooted at a free vertex different from the vertex r. This statement is true for all matchings in a graph [WiZ65].

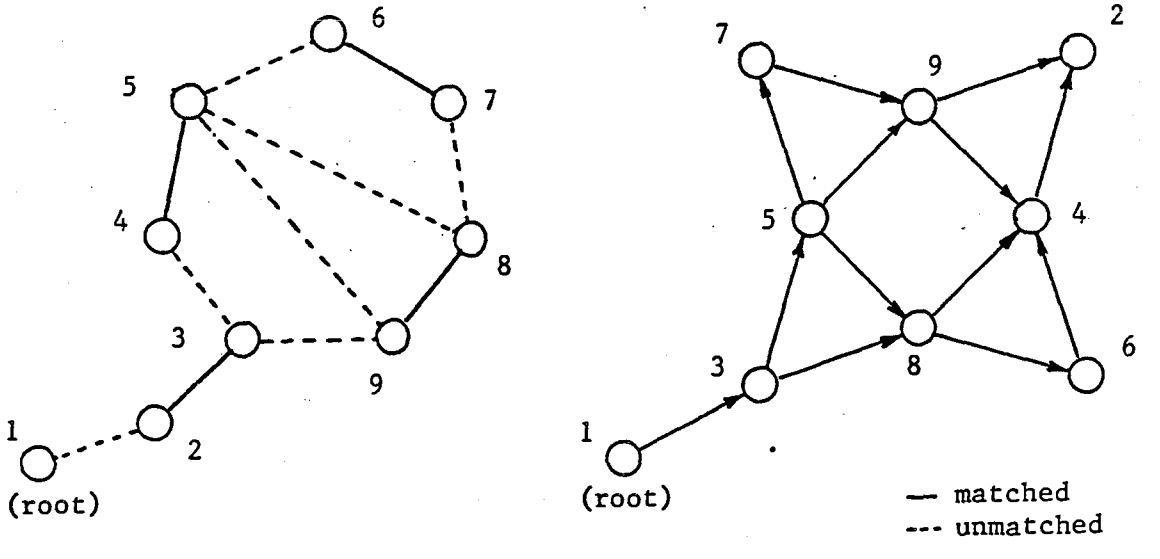

Figure 3.1: A matched graph G and its directed graph DG.

Witzgall & Zahn modified Edmonds' algorithm with the introduction of **bi-edges** and the notion of **predecessors**. Observe that an alternating path between an unmatched vertex and an outer vertex can be broken into segments of length two. These segments consist of an unmatched edge followed by a matched edge. Such a segment is called a bi-edge. A bi-edge from u to v, $(u,m(v),v)$, indicates a sense of direction from u to v, where $m(v)$ is used to denote the match mate of v. For any bi-edge $(u,m(v),v)$, u is called the predecessor of v. We use $p(v)$ to denote the predecessor of v. Given a matched graph G, we can construct a directed graph DG with the same vertex set by replacing each bi-edge from u to v in G with a directed edge from u to v in DG (as in figure 3.1). Thus, each path in DG corresponds to an alternating path of even length in G and vice versa. However, a simple path in DG may yield a non-simple alternating path in G. That is, the alternating path in G may reuse some edges. We therefore call a directed path in DG legal if it corresponds to a simple alternating path in G. Witzgall & Zahn proved the following theorem:

> **Theorem 3.1:** Let DG be the graph of bi-edges of a matched graph G, let r be an arbitrary vertex of DG and denote by Ω the set of all vertices of DG that are legally accessible from r. Then DG contains as a subgraph a tree T which is rooted at r, such that 1) T has Ω as its vertex set, and 2) every simple path in T that joins a vertex v from the root r is legal.

Since every matched graph G is in one-to-one correspondence with such a directed graph DG, we do not actually need to construct the directed graph DG in order to construct the tree T. By labeling the vertices of G as inner and outer vertices with respect to a free vertex r and carefully maintaining the predecessor relation as in figure 3.2, we can obtain the same effect as converting G into its DG (w.r.t vertex r) in order to construct the tree T. Witzgall & Zahn termed such a labeling of graph G a **labeled subgraph**, denoted as (A,Ω,I,p), where A is a subgraph of G. The vertices of A are labeled either outer or inner, sets Ω and I,

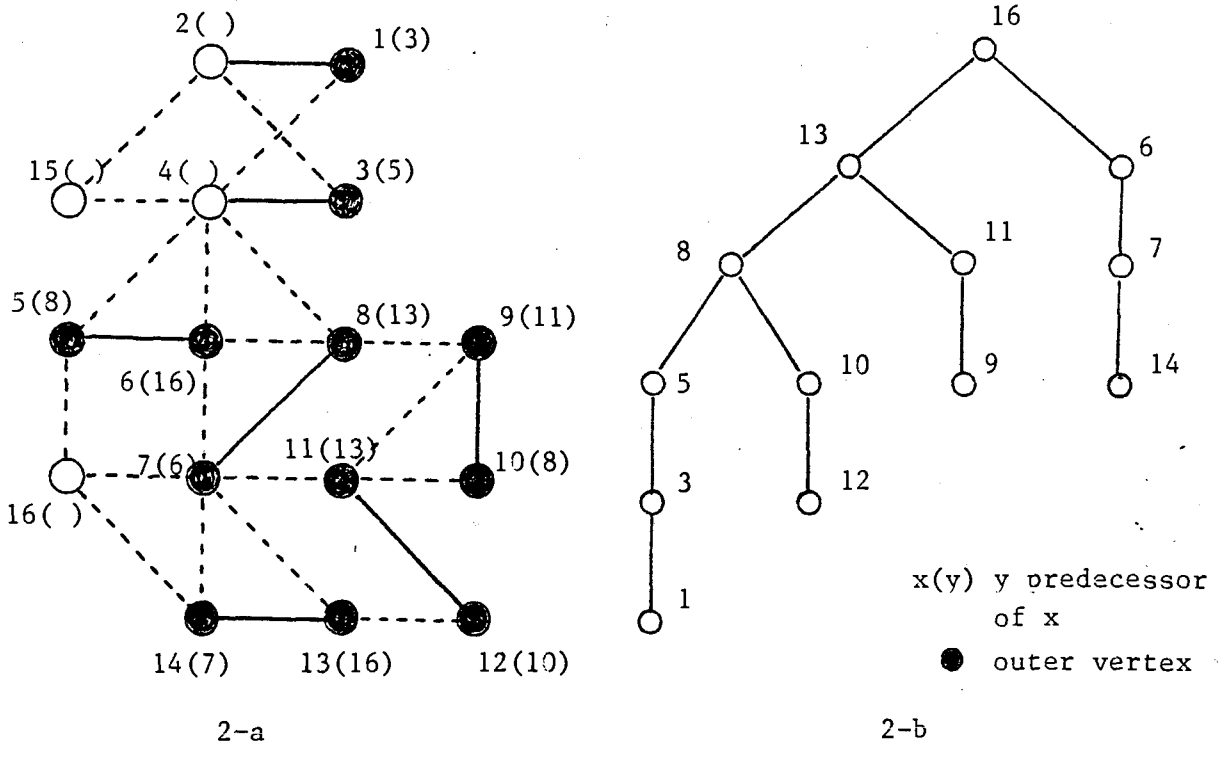

Figure 3.2: a) A labeled graph A and b) a particular tree T of the graph.

respectively. The two sets are disjoint. The last component p is a single-valued function p: Ω-$\{r\} \rightarrow \Omega$ called the predecessor function. Witzgall & Zahn proved that all the outer vertices in A are legally accessible from the root vertex. Thus, the path for any outer vertex v in A towards the root of A in the predecessor relation can be represented by a simple alternating path as $(v,m(v),p(v),mp(v),p^2(v),mp^2(v),\cdots,p^n(v)=r)$. We write $mp(v)$ for $m(p(v))$ and $p^2(v)$ for $p(p(v))$ and so on. We use the term ancestor of v to mean any vertex $p^i(v)$ for $i \geq 1$. A path formed by the predecessor relation is called a back-tracing path (abbreviated btp below).

Witzgall & Zahn presented their algorithm as two sub-algorithms: LL and MM. The sub-algorithm LL constructs the labeled subgraph (A,Ω,I,p) with respect to a free vertex. The sub-algorithm MM uses the result of LL (a labeled subgraph with a free vertex, other than the root, that is adjacent to an outer vertex) to construct a maximum matching. LL is divided into two steps: the forward step and the blossom step. These two steps alternate to label the matched graph until either a potential augmenting path is found or no augmenting path can be found. For the latter case, LL is repeated by using another free vertex that has not be examined before. The two sub-algorithms repeat the labeling and updating of the matching until a matching with the maximum cardinality is found or until all the free vertices have been given the chance to search for an augmenting path.

The forward step of LL enlarges the labeled subgraph if u is an outer vertex with unexplored neighbor x (neither an outer nor an inner vertex) and the edge (x,u) is matched. These three vertices form the new bi-edge (v,x,u). The labeled subgraph (A,Ω,I,p) is enlarged by setting $A := A \cup (v,x) \cup (x,u)$, $\Omega := \Omega \cup v$, $I := I \cup x$, $p(z) := p(z)$ for $z \in \Omega$, and $p(z) := u$ for $z = v$.

The blossom step enlarges the subgraph if there is an edge (s,t) between two outer vertices s and t which is not in the current labeled subgraph. These two vertices have at least one common ancestor in the predecessor relation. A blossom in a matched graph is defined to be an odd circuit with one more unmatched edge than the matched edges. Witzgall & Zahn's blossom step includes a special structure (figure 3.3) as a blossom, and handles it in the same way as other regular blossoms as defined above. The base vertex for such vertices s and t is defined to be their nearest common ancestor with respect to the back-tracing paths of s and t. The back-tracing paths from s and t to their common ancestor x can be represented by a sequence of vertices in terms of s and t. Without loss of generality, the btp from s to x can be represented by $(s=p^0(s),m(s),p(s),\cdots,mp^{k-1}(s),x=p^k(s))$. The ver-

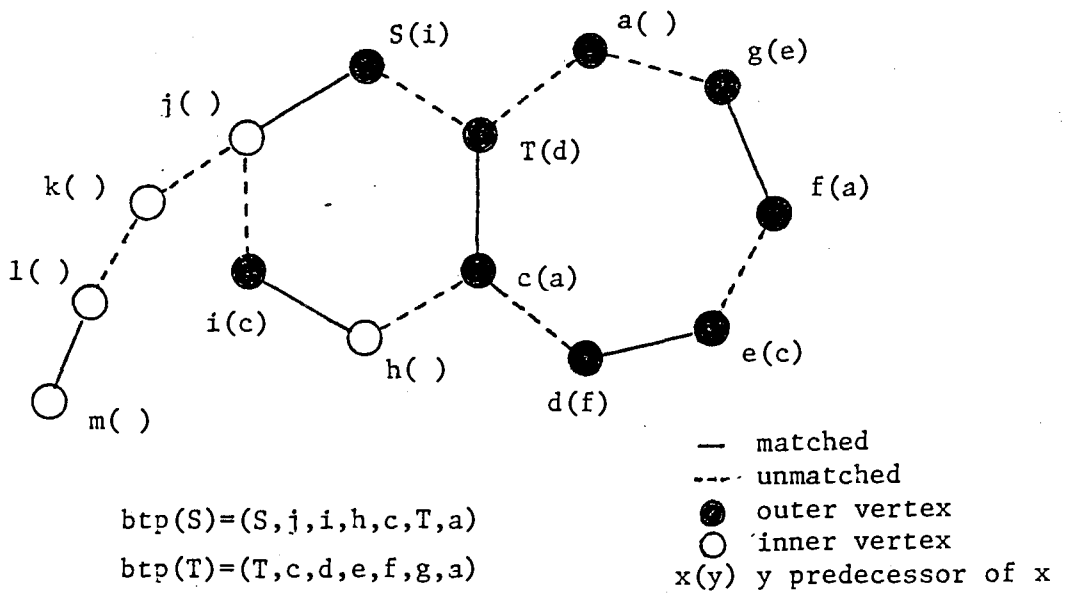

Figure 3.3: A special case for the blossom step.

tices of the form $p^i(s)$ for $i \geq 0$ are said to be **even** or **outer** with respect to the btp of s. The others are said to be **odd** or **inner** with respect to the btp of s. Consider the vertices which are inner with respect to the btp of t and which lie between x and t on the btp of t. We define the blossom splitting vertex of s, denoted $b(s)$, to be the inner vertex with respect to the btp of t closest to x on the btp of t which does not have a predecessor (if such a vertex exists). The vertices between t and $b(s)$ (inclusive) which are odd with respect to the btp of t will be added to the new subgraph A. The blossom step makes s the ancestor of all of the inner vertices between t and $b(s)$ (including $b(s)$) on the btp of t. This is done by changing the predecessor of each inner vertex on this path to be the next inner vertex in the direction of t. The vertex s becomes the predecessor of $m(t)$ (unless there was no vertex $b(s)$). The inner vertices on the btp of s are treated similarly with t becoming their ances-

tor. All those vertices involved are removed from the set I and added to the set Ω.

The distributed algorithm is divided into three phases. The first phase of the algorithm elects an initial control processor. During the second phase, this processor conducts a depth-first search (abbreviated DFS below) of the network and an initial matching is formed from the DFS tree. The last phase of the algorithm is further divided into two sub-phases, which will be repeated until a maximum matching is found, i.e. until no augmenting path exists. The first sub-phase elects a control processor among all free processors that have not been elected before. During the second sub-phase the elected processor supervises a search for an augmenting path, and updates the existing matching if any augmenting path is found. We will show that the free processors with degree one can be excluded from these elections.

Throughout this chapter, when we refer to the size of a message, we will temporarily ignore the bits that are required to represent the message type. The additional bits required to distinguish the message types will be considered in the conclusion (see Lemma 3.10).

3.2. The First Phase Of The Matching Algorithm

The first phase is based on the Multi-source Biggest Finder algorithm of Chang [Cha79]. We first negate the *ID*s of leaf nodes (degree one processors), so that they won't be elected. By doing this, the root of any subtree in a DFS tree (assuming more than 2 nodes) is a non-leaf node. In this way we maximize the number of leaf nodes in the initial matching because all leaf nodes have the highest priority to be matched with their parents in a DFS tree. This will be explained in more detail in the next section. The purpose of phase one is to choose a control processor to carry out the DFS of the network. Allowing only non-leaf nodes to be elected (to maximize the number of leaf nodes in the initial matching) will not affect the end result of the algorithm as shown in Lemma 3.2 below.

Any processor may initiate the MATCHING algorithm. A sleeping or inactive processor will wake up when a message arrives and will then join the others in the election. The number of messages in this phase is no more than $4ne$. Each message sent during this phase carries at most one field of information, the *ID* of the initiator, that can be represented by a string of $\log_2(n)$ bits. Therefore the total number of bits sent during this phase is no more than $4ne\log_2(n)$.

We can reduce the number of messages by allowing only the processors that awakened before a message arrived to be candidate in this election. To ensure that the controller of the DFS is a non-leaf node, if the elected node is a leaf node we make its neighbor the controller. In the worst case, all processors initiate the election and the number of messages does not change.

Lemma 3.2: Each vertex of degree one is in some maximum matching.

Proof: By way of contradiction, assume that this is not the case. To avoid trivialities, we assume that there are at least three vertices. Consider a maximum matching M and an unmatched vertex u of degree one. Its neighbor v must be matched to some other vertex w (otherwise M is not a maximum matching). Edge uv can replace edge vw in the matching M to form a matching M' with same cardinality.

□

3.3. The Second Phase of The Matching Algorithm

The second phase is to create a DFS tree and construct an initial matching among the processors based on the structure of that tree (see figure 3.4 for an example). The elected processor from the first phase begins the DFS by sending out an explorer to one of its neighbors. The explorer is relayed to the other processors in the network in a depth-first

manner. When a node receives its first explorer and it declares the sender to be its parent node. An explorer reaching a terminus (a node of degree one or a node previously visited in this phase) causes a reply (an echo to the parent or a reject message to the sender). Both the sender and the receiver of an explorer message mark the edge as scanned and that edge will not be used for transmitting explorer messages again. The parent node or sender then tries to explore other neighbors and returns an echo to its parent when no unexplored neighbors remain. The process is repeated until the root (the controller) has explored all of its neighbors. Only the control processor knows when the second phase terminates and signals the start of the next phase by initiating an election command. Each processor keeps track of some local information, such as its parent (the neighbor that first sent it an explorer) and its match-mate (the processor's mate in the matching).

Seven message types are used to construct the initial matching: explorer, echo-leaf, echo-candidate, echo-eliminate, echo-reject, confirm-match and echo-confirm. We may classify these message types into three groups: explorer messages, echo messages and confirm messages. Explorer messages are used in the forward sweep of the DFS. An explorer is relayed throughout the network until terminated by an appropriate response message. The echo messages are replies to an explorer. The first three types of echo messages convey information to the parent about the initial matching on the subtree rooted at the sender of these echo messages. A processor will not send these echo messages to its parent until all of its neighbors have been explored and have replied. The echo-leaf message indicates that it is a preferred match-mate for the parent. The echo-candidate message indicates that it is a possible match-mate for the parent. The echo-eliminate message indicates that it is already matched and can be ignored. A parent processor makes a local matching decision based on the responses sent from all of its children. A parent processor will try to match with an

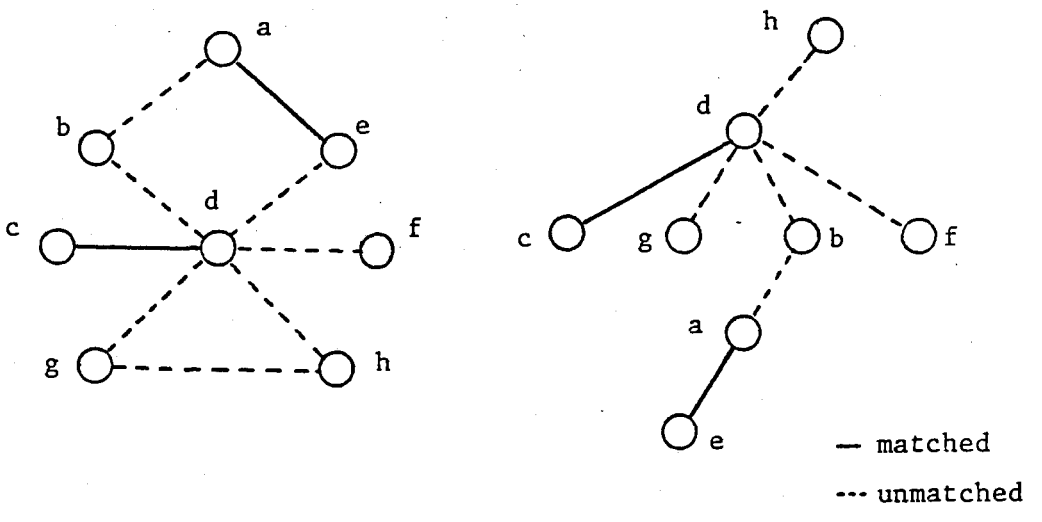

Figure 3.4: A DFS tree and an initial matching.

unmatched child before it permits matching with its own parent (by sending the request message echo-candidate to the parent). If there is more than one child, it will first try to match with a leaf, if any. Otherwise it matches with any unmatched child. Requests of the same type are chosen arbitrarily. When a processor decides to match with one of its children, it sends a confirm-match to the child and waits for its reply. After receiving the reply, it returns the message echo-eliminate to its parent. The echo-reject message is sent by a previously visited processor in response to an explorer. The confirm-match message is sent from a parent node to its child to inform the child that it is being chosen as match-mate. The echo-confirm message is a reply to the confirm-match message, which is used to ensure that both nodes are aware that they are matched before the next election phase starts.

3.3.1. Algorithm for phase 2

(1) The controller starts the DFS traversal by sending an explorer to one of its neighbors. The controller then waits for an echo from that neighbor. After an echo is received, the process is repeated with the remaining unexplored neighbors. This phase terminates when the last echo has arrived.

(2) When an explorer arrives at a node of degree one, the node marks the edge scanned and 'echo-leaf' is returned.

(3) When an explorer arrives at a previously visited node, the node marks the edge scanned and 'echo-reject' is returned.

(4) When an explorer comes to a node p of degree greater than one for the first time, the node from which it arrived is marked as parent and scanned. The explorer is relayed to another unscanned neighbor and the processor waits for an echo from the neighbor. When the echo is received, the process is repeated with the remaining neighbors. After all the neighbors have been explored, an echo is sent to the parent of the node. The content of the echo depends on the replies received by p:

1. if all replies are 'echo-reject', return 'echo-candidate',
2. if all replies are 'echo-eliminate', return 'echo-candidate',
3. if at least one echo of the form 'echo-leaf' is received, choose one of these replies arbitrarily, mark the corresponding neighbor as match-mate, send 'confirm-match' to that neighbor and return 'echo-eliminate' after receiving an 'echo-confirm' back from that neighbor.
4. otherwise, choose a candidate node, mark it as match-mate, send 'confirm-match' to that neighbor and return 'echo-eliminate' after receiving an 'echo-confirm' back from that neighbor.

(5) When a confirm-match message arrives at a node, mark its parent as match-mate and return 'echo-confirm'.

3.3.2. Analysis of phase 2

Theorem 3.3 shows that the phase two algorithm successfully creates a matching, and guarantees that the matching created contains the maximum number of degree one vertices. Lemma 3.4 states that no more degree one vertices can be added to the matching without sacrificing other degree one vertices already in the matching. This observation is important to the algorithm in the next phase of the algorithm, which will be explained more later.

Theorem 3.3: Phase two creates a matching.

Proof: The control processor initiates a single explorer which traverses the tree in a depth-first fashion. No other processor initiates an explorer. Each processor keeps track of its parent in the DFS tree. The only edges which may be placed in the matching are the edges of the DFS tree. The edges for the matching are chosen during a leaf to root sweep of the tree with each parent choosing to match with at most one of its unmatched children. A parent never matches with a previously matched child. Obviously, no two edges chosen are adjacent.

□

Lemma 3.4: A leaf node will never be matched during the search for augmenting paths if it is not matched during the construction of the initial matching above.

Proof: Berge [Ber57] proved that a matching is maximum if and only if the graph has no augmenting paths. To extend the initial matching to a maximum matching, we search for augmenting paths. By our construction, if any leaf node u is not matched in the initial matching, its neighbor must be matched to another leaf node. Thus, no augmenting path can add u to the matching.

□

The message complexity of the phase 2 algorithm is shown in the Lemma 3.5. Since the messages used in this phase carry no information other than the message type, we can postpone the analysis of the bit complexity until the total number of messages types is determined in the next section.

Lemma 3.5: At most $2e + n$ messages are sent in phase 2.

Proof: Three groups of messages are used. An explorer is sent along each edge once in either direction. Thus e explorers are sent. Each explorer causes exactly one echo-leaf, echo-candidate, echo-eliminate or echo-reject message, so another e echos messages are required. Each matched pair causes exactly one confirm-match and one echo-confirm message. Thus at most $n/2$ confirm-match and $n/2$ echo-confirm messages are sent. Therefore, the total number of messages sent during phase two is at most $2e + n$.

□

3.4. The Third Phase of The Matching Algorithm

Phase 3 is divided into two parts which are repeated until a maximum matching is found. In Phase 2, the maximum possible number of degree one vertices were placed in the

matching. They can be ignored in this phase which tends to reduce the number of iterations required. A controller is elected among the unmatched non-leaf processors which have not been elected previously in phase 3. The controller initiates a search for an augmenting path from itself to some other vertex x. If such a path is found, an augmentation will be performed. Otherwise, the controller initiates a new election. When there are no further candidates for controller, the algorithm terminates.

3.4.1. Election Phase

During this phase, each processor keeps track of some local information: max-ID, vot-ID, parent, maxID-node, message-out, match-mate and candidate-count. The variable max-ID contains the maximum *ID* encountered so far (including its own). The variable vot-ID contains the *ID* used for the election. Initially, all previously elected, matched or leaf processors have vot-ID equal to zero and any other processor has vot-ID equal to its own *ID*. When a processor is elected, its vot-ID is set to zero. The parent of a processor is the neighbor that first sent it an explorer in the current election. The variable maxID-node contains the neighbor from which the processor received the largest *ID* encountered other than its own. Initially, each processor uses its own *ID* as the value of maxID-node. The variable maxID-node will indicate the direction from the controller to the newly elected processor. The variable messages-out contains the number of replies expected and is reset to zero for each election. The match-mate is the processor's mate in the matching. Its initial value is determined in the previous phase. The variable candidate-count contains the number of descendents with a non-zero vot-ID. Its initial value is zero or one, depending on the value of the processor's own vot-ID.

This election algorithm uses three types of messages: explorer, echo and control. The control processor begins the election process by broadcasting an explorer message to all of

its neighbors in parallel. When a processor first receives an explorer message, it marks that neighbor as parent. The explorer is then relayed to the other processors in the network. Each processor upon receiving the same number of explorer and echo messages as the number of neighbors it has returns an echo to its parent. An echo message contains the max-ID that it has encountered and the number of vertices that it knew with non-zero vot-ID. Besides, it marks the neighbor that sent it the largest max-ID value as maxID-node. When the control processor receives the max-ID, it knows the election is finished. The newly elected controller will be notified to start the next phase if the max-ID is non-zero and the candidate-count is greater than one. Otherwise, all processors will be notified to terminate the MATCHING algorithm.

3.4.1.1. Algorithm for the Election Phase

(1) The control processor p starts the election by sending explorers to all of its neighbors. After all the replies are received, if the max-ID is non-zero and candidate-count is greater than one, p passes control to the newly elected controller (with *ID* equal to the max-ID) who will start the next phase. If no processor is elected, the MATCHING algorithm terminates.

(2) When an explorer arrives at an unvisited node, the neighbor from which it arrived is marked as parent. Explorers are then relayed in parallel to the other neighbors of the node.

(3) When an explorer arrives at a leaf, the neighbor from which it arrived is marked as parent. The explorer terminates and 'echo <vot-ID,candidate-count>' is sent to the parent.

(4) Subsequent explorers arriving at a node are terminated and treated as a reply from that neighbor.

(5) When 'echo <echo-ID,candidate-count>' arrives at a node, echo-ID is compared with max-ID. The larger value is saved in maxID and both the maxID-node and candidate-count is updated if necessary.

(6) After receiving replies from all neighbors, a node sends 'echo <max-ID,candidate-count>' to its parent.

3.4.1.2. Message Complexity for the Election Phase

Lemma 3.6: At most $2(e + n - 1)$ messages are sent during the first sub-phase.

Proof: Only the control processor initiates an explorer. When a node first receives an explorer it sends explorers to all of its other neighbors. Since these explorers are sent in parallel, an explorer may be sent along an edge from each end-point at the same moment. Thus at most $2e$ explorers are sent. An echo message is only sent from a child to its parent. Thus at most $n-1$ echos are sent. Passing control from the initiator to the newly elected processor requires at most another $n-1$ messages. Therefore, the total number of messages sent during the election is

$$\leqslant 2e + n - 1 + n - 1$$

$$= 2(e + n - 1).$$

□

Lemma 3.7: The number of bits sent in the first sub-phase is no more than $(n-1)\log_2(n)$.

Proof: The explorer and control passing messages contain no information other than the message type representation which will be discussed later in this chapter. Each echo message carries an *ID* value and the number of candidate-count, which requires $2\log_2(n)$ bits. Therefore the total number of bits sent duing this phase is at most $2(n-1)\log_2(n)$.

□

3.4.2. Augmentation Phase

The phase three algorithm is based on the algorithm of Witzgall & Zahn [WiZ65], which does not require a complicated data structure and can be more easily simulated in a distributed network. This distributed algorithm to find an augmenting path is invoked by the newly elected control processor p. It begins by performing a depth-first-search of G and constructing the labeled subgraph as Witzgall & Zahn did in their algorithm. We call this process bi-edge depth-first-search (abbreviated BDFS below). The algorithm will terminate if there is a free processor other than the control processor p that is adjacent to an outer vertex rooted at the controller p (that is, an augmenting path is found), or if all edges incident to the control processor have been scanned and have replied negatively. In both cases, an election is initiated for the next iteration of phase 3.

The following local variables are kept by each processor: predecessor, match-mate, node-status, blossom-edge, blossom-number, unconfirm-predecessor, scan-replied, edge-scan and btp-edge. The predecessor of a vertex v (denoted $p(v)$) is a vertex u such that there is a bi-edge from u to v and both u and v are outer vertices. The match-mate of a vertex v (denoted $m(v)$) is v's current mate in the matching. The variable node-status indicates whether the node is an outer vertex or an inner vertex with respect to the labeled subgraph A rooted at the controller. The variable blossom-edge contains the label of an edge on which the processor received the notification that it is in a blossom. The variable blossom-

number is the number of the blossom found. This value may differ from processor to processor. Each processor will update the value of its blossom-number when it receives a higher value. The variable unconfirm-predecessor is the processor's potential predecessor which will become its predecessor if a confirmation message arrives from some neighbor. The variable scan-replied is used to indicate whether a processor has replied to its parent or not. The list edge-scan contains the status of the incident edges, one element for each edge, indicating whether each edge was scanned or not. The variable btp-edge is the edge on the current back-tracing path.

The algorithm is divided into three tasks: BI-EDGE, BLOSSOM and AUGMENTATION. At any one time, only one task is active. Both BI-EDGE and BLOSSOM (corresponding to Witzgall & Zahn's forward and blossom step, respectively) are used to enlarge the labeled subgraph A, whose predecessor relation corresponds to the BDFS tree mentioned in Section 3.1. BI-EDGE expands the BDFS tree by searching for bi-edges formed by the vertices that have not been searched before. BLOSSOM expands the BDFS tree by searching for bi-edges formed by the vertices that have been searched before. During the BI-EDGE expansion, BLOSSOM is called when a blossom is detected. Control is returned to BI-EDGE once the BLOSSOM task is finished. AUGMENTATION updates the existing matching once an augmenting path is found (a free vertex is found adjacent to an outer vertex during BI-EDGE).

During the BI-EDGE expansion only one processor is active. A BDFS is similar to a DFS, with branching occurring only at an outer vertex. An outer vertex t starts the search for a bi-edge by sending an explorer along an unmatched edge to one of its neighbors s. Whenever a vertex sends an explorer along an unscanned edge, it marks that edge scanned. When an unmatched vertex s receives an explorer (from its neighbor t) an augmenting path is found. In this case, AUGMENTATION is called to trace the back-tracing path of t back

to the control processor and update the matching along this path. When an unmatched vertex s receives its first explorer (from vertex t) along an unmatched edge, a new bi-edge is formed among the vertices t, s and $m(s)$. The node status of s is set to inner, and s marks the edge to t as scanned. The explorer is forwarded to $m(s)$. Vertices s and $m(s)$ mark the edge between them as scanned and the node status of $m(s)$ is set to outer. The labeling process is continued by $m(s)$ which sends the explorer along an unmatched edge to one of its neighbors. When a subsequent explorer arrives at outer vertex s (from t), both s and t mark the edge scanned. Since both s and t are outer, BLOSSOM is called to enlarge the subgraph A. When a subsequent explorer arrives at inner vertex s (from t), $m(s)$ already has a predecessor and both s and $m(s)$ are in A. In this case, vertex s returns an echo-scan message to t. When a vertex receives an echo-scan message, it starts the search for a bi-edge if it is an outer vertex and has an unexplored neighbor. Otherwise, it returns an echo-scan to its parent (or calls for an election if it is the control processor).

BLOSSOM is called from BI-EDGE when an edge (s,t) not in A (i.e. an edge have not marked scanned) is found joining two outer vertices in A. To perform BLOSSOM distributively, we need to identify the nearest common ancestor of s and t. This is done by sending a message (blossom-scan) carrying the current blossom number along the back-tracing paths of s and t. This message also informs each vertex of its match-mate's potential predecessor. The first common ancestor (x) will notice these two messages and determine that it is the base vertex. It then sends messages (blossom-retreat) back along the two back-tracing paths to the vertices s and t. As these messages travel to s and t, vertices on the paths update their predecessors and each inner vertex on the path changes its node status to outer.

So far, the distributed algorithm is a straightforward implementation of the sequential algorithm. However, a complication arises at this point in the distributed algorithm

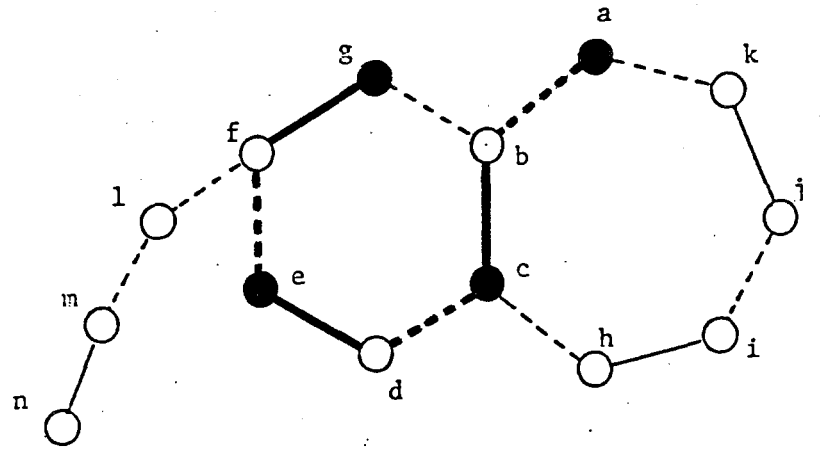

figure 3-5(a)

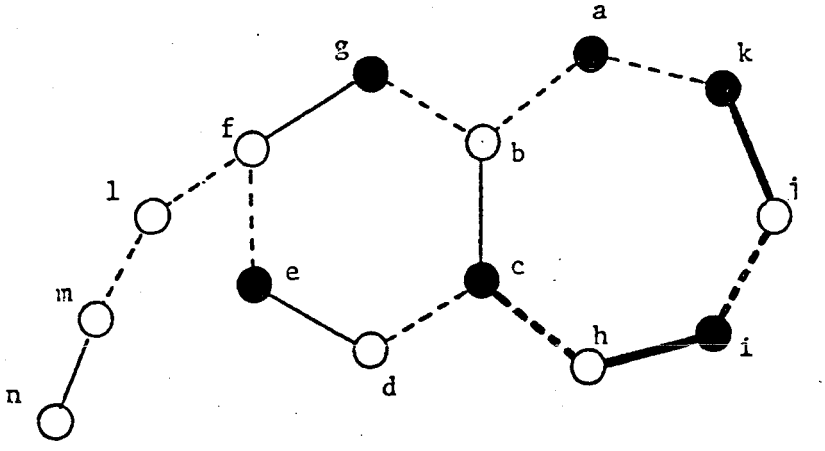

figure 3-5(b)

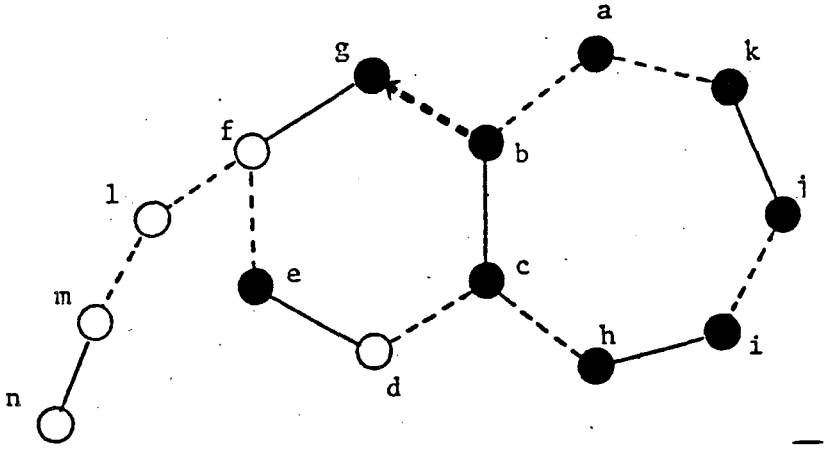

figure 3-5(c)

— matched
--- unmatched
● outer vertex

Figure 3.5: A particular BDFS of a matched graph.

which does not occur in the sequential algorithm. A vertex which has just been changed from inner to outer and has already replied to its parent during the previous BI-EDGE search must be re-examined and allowed to explore its neighbors. The BDFS only allows outer vertices to explore their neighbors. During the BDFS, a vertex v replies to its parent when all the vertices in the subtree rooted at v are scanned. Unfortunately, some inner vertices in such a subtree may change to outer during a BLOSSOM expansion as shown in figure 3.5. Vertex a in the figure is the control processor. In figure 3.5-a, the BDFS terminates at vertex g, since the unscanned edge incident on vertex g leads to an inner vertex (b). The BDFS backs up to vertex c and the labeling continues until a blossom is detected by vertices a and k (figure 3.5-b). The BLOSSOM process changes all these vertices (a,b,c,h,i,j,k) to outer. BI-EDGE is resumed and the BDFS construction continues from k. Eventually, the BDFS backs up to vertex b. Vertex b has not marked its edge to g as scanned. A blossom is detected when b sends an explorer to g (as shown in figure 3.5-c). At this point, all the vertices in the graph, except vertices l, m and n are outer and scanned. Notice that, vertices f and d were previously inner and did not explore their neighbors (in figure 3.5-a). If we now allow vertex b (which has scanned all of its neighbors) to reply to its parent a, the augmenting path search fails to locate the augmenting path (a,k,j,i,h,c,b,g,f,e).

To overcome this problem, we allow these vertices (f and d) to continue the BDFS before a reply is sent to vertex b's parent. If vertices such as f and d exist, they will be discovered during BLOSSOM. It is shown below (Property 3.2) that if such vertices exist, either they all lie along the back-tracing path of s or they all lie along the back-tracing path of t. Without loss of generality, assume the back-tracing path of s contains such vertices, namely $mp^j(s), mp^k(s), \cdots, mp^l(s)$, with $1 \leqslant j < k < l$, such that $mp^j(s)$ is the closest to the base vertex x on the btp of s. If these vertices exist, we make t the parent of s, s the parent

of $m(s)$, $m(s)$ the parent of $p(s)$, $\cdots$, $mp^{j-1}(s)$ the parent of $p^j(s)$, $p^j(s)$ the parent of $mp^j(s)$ and so on. An example of this process is shown in Figure 3.6. We then allow $mp^j(s)$ to resume BI-EDGE. After $mp^j(s)$ finishes exploring its neighbors, it returns an echo-scan message to its parent. Subsequent vertices along this path are allowed to explore their neighbors if they haven't done so previously. If no such vertex exists, t resumes the BI-EDGE search.

Ten types of messages are used in the augmenting phase: explorer, echo-found, echo-blossom, echo-scanned, echo-resume, echo-extend, blossom-scan, blossom-retreat, extend and request-ok. The explorer message contains both the *ID* of the initiator and the blossom number (denoted blossom-number). The value of blossom-number indicates the current

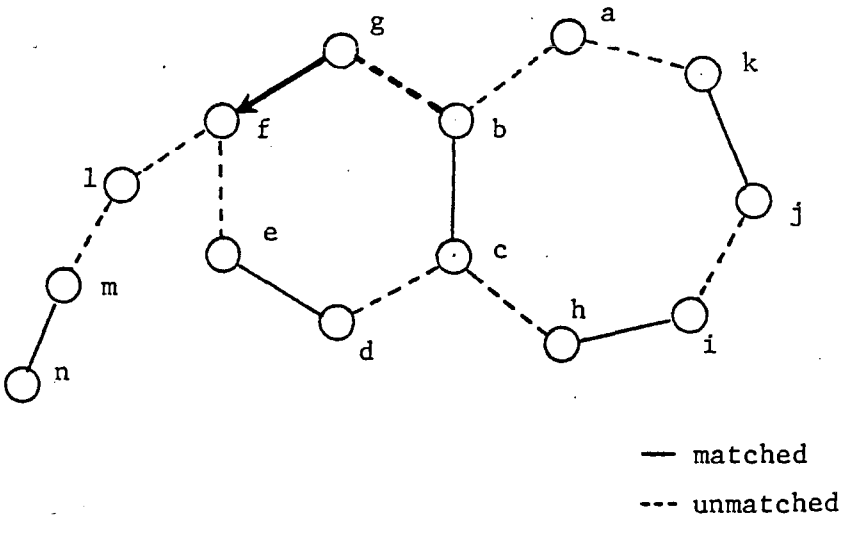

Figure 3.6: An extension of BI-EDGE search.

blossom number and its value will be increment when a new blossom is found. This value allows x to determine that it is the base vertex of the blossom. If any vertex receives two blossom-scan messages with different blossom-numbers, it discards the message with the smaller blossom-number. The echo-found message contains the information for it to trace back to the root of T. The echo-blossom message contains the *ID* of the initiator. The echo-scanned message contains the blossom-number. The blossom-scan message contains 3 fields: btp-edge, unconfirm-predecessor and the current blossom-number. The btp-edge field is used to back-tracing to the root of T. The unconfirm-predecessor field carries the potential predecessor that may be the receiving processor's future predecessor. The blossom-retreat message contains 3 fields: blossom-number, node-status and extend-ID. The node-status field has two values: inner and outer, which are used to indicate the occurrence of the first inner vertex. Subsequent processors receiving this message with node-status equal to inner can update their predecessors. Extend-ID is the *ID* of the new outer vertex v closest to x (on the path of the blossom-retreat message travel back to vertices s and t) that has replied to its parent before the current BLOSSOM took place. Under that circumstance the current BDFS tree should restart the BI-EDGE task from this vertex. The extend message carries an extend-ID and routing information to the processor extend-ID, is used to signal the vertex (with id equal to extend-ID) to resume the execution. The echo-extend is the message for vertex s to signal t that there exists a non-nil extend-ID on its btp. The request-ok message is the reply for t to signal s to let extend-ID start BI-EDGE. The echo-resume is to signal t to resume it suspended job.

3.4.2.1. Algorithm for the Augmentation Phase

(1) The controller p starts the alternating path labeling by sending out an 'explorer <p,blossom-number>' bearing its own *ID* on one of its incident edges. It marks the

edge scanned and waits for an echo along that edge. After each echo from an explored neighbor is received, the traversal is repeated on the remaining unscanned incident edges. This sub-phase is terminated after all edges incident on the control processor are scanned and have replied, or when the controller receives an echo-found message from an explored neighbor.

(2) When an 'explorer <*ID*,blossom-number>' message arrives at *p* along an unmatched edge for the first time, *p* saves the blossom-number, becomes an inner vertex and marks the edge as both scanned and parent. Then the 'explorer <*ID*,blossom-number>' is relayed on a matched edge. If no such edge exists, an 'echo-found <*p*>' message is returned to its parent.

(3) When a subsequent 'explorer <*ID*,blossom-number>' message arrives at an outer vertex *p* along an unmatched edge, it saves the blossom-number, and marks the edge scanned. Vertex *p* is the vertex *s* of the blossom. Then *p* sends an 'echo-blossom <*p*>' message as a reply along the edge that it received the explorer message. Furthermore, *p* initiates a 'blossom-scan <*p(p)*,*ID*,blossom-number+1>' message along the matched edge if there exists one (that is, if *p* is not the controller).

(4) When a subsequent 'explorer <*ID*,blossom-number>' message arrives at an inner vertex *p* along an unmatched edge, it saves the blossom-number and returns an 'echo-scan <blossom-number>' message.

(5) When an 'explorer <*ID*,blossom-number>' message arrives at a node *p* along a matched edge, *p* becomes an outer vertex, assigns *ID* as its predecessor, and marks the edge as both scanned and parent. The message 'explorer <*p*,blossom-number>' is relayed on an unscanned edge incident to *p* and *p* waits for an echo along that edge. This process is repeated until all the neighbors of *p* are scanned. At this point, an 'echo-scan <blossom-number>' message is returned to the parent.

(6) When an 'echo-found <*ID*>' message arrives at a node *p* along an unmatched edge, then the edge is mark as matched (match-mate) and the message 'echo-found <*p*(*p*)>' is relayed along the previous matched edge. When an 'echo-found <*ID*>' message arrives along a matched edge, it is relayed along the *ID* edge and the edge is marked matched (match-mate).

(7) When an 'echo-scanned <blossom-number>' message arrives at an outer vertex *p*, it updates its blossom-number and sends an 'explorer <*p*,blossom-number>' message along one of its unscanned edges. This process is repeated on the remaining incident edges after each reply is returned along an explored edge until all the incident edges are scanned. After all incident edges are scanned, an 'echo-scanned <blossom-number>' message is returned to the parent.

(8) When an 'echo-scanned <blossom-number>' message arrives at an inner vertex *p*, it updates its blossom-number and relays the 'echo-scanned <blossom-number>' message to the parent.

(9) When an 'echo-blossom <*b*>' message arrives at a node *p*, the execution of the explore traversal is suspended. The node marks itself as vertex *t* and starts the blossom traversal by initiating a 'blossom-scan <*p*(*p*),*b*,blossom-number+1> along its matched edge.

(10) When a 'blossom-scan <btp-edge,unconfirm-predecessor,blossom-number>' message arrives at a node *p* along a matched edge, *p* determines whether it came from the current blossom. If so, it saves both the blossom-number and unconfirm-predecessor and relays a 'blossom-scan <NIL,*p*,blossom-number>' message along the btp-edge. Otherwise, discard the message.

(11) When a 'blossom-scan <btp-edge,unconfirm-predecessor,blossom-number>' message arrives at a node *p* along an unmatched edge, determine whether it came from the

current blossom. If it is not from the current blossom, discard the message. If it came from the current blossom and is arriving at p for the first time p updates its blossom-number, saves the unconfirm-predecessor and relays the 'blossom-scan <$p(p)$,unconfirm-predecessor,blossom-number>' along its matched edge. If it came from the current blossom and is the second arrival of such a message at p, a 'blossom-retreat <blossom-number,outer,NIL>' message is sent along the blossom-edge(s).

(12) A 'blossom-retreat <blossom-number,status,extend-ID>' message arrives at a node p along a matched edge. If p is not one of the two vertices that detected the blossom, it relays the message along the blossom-edge. If p is vertex s, it signals vertex t to resume BI-EDGE. If p is vertex t (the one that suspended BI-EDGE), it resumes BI-EDGE if the extend-ID during BLOSSOM is equal to NIL, otherwise BI-EDGE is resumed at the vertex with id equal to extend-ID. For the latter case, p sends 'extend <extend-ID,$p(p)$>' along the matched edge.

(13) When a 'blossom-retreat <blossom-number,status,extend-ID>' message arrives at an outer vertex p along an unmatched edge, p updates its predecessor relation if the variable status in the message it received indicates an inner vertex has been encountered during the retreat process. The message is relayed along the blossom-edge.

(14) When a 'blossom-retreat <blossom-number,status,extend-ID>' message arrives at an inner vertex p along an unmatched edge, p changes its node-status to outer and updates its predecessor relation. If p has already replied to its parent and the extend-ID it received is NIL, it sends a 'blossom-retreat <blossom-number,inner,p>' along the blossom-edge. Otherwise, it sends a 'blossom-retreat <blossom-number,inner,extend>' along the blossom edge.

(15) When an 'extend <extend-ID,pred-edge>' message arrives at a node p along a matched edge, p marks the edge both scanned and parent. The message is then relayed along the pred-edge.

(16) When an 'extend <extend-ID,pred>' message arrives at a node p along an unmatched edge, p marks the edge both scanned and parent. If p equal to the extend-ID, then p starts BI-EDGE. Otherwise, p sends 'extend <extend-ID,$p(p)$>' along its matched edge.

We observe the following properties:

Property 3.1: A vertex with a node-status "outer" will never change its node-status. A vertex with node status "inner" may change its node-status to "outer".

Property 3.2: When a blossom is detected by two adjacent outer vertices s and t, at most one of the back-tracing paths of vertices of s and t to their nearest common ancestor x may contain inner vertices which have replied to their parents previously.

Proof: If either s or t is the nearest common ancestor x, then there is only one back-tracing path. Otherwise, since the algorithm performs a bi-edge depth-first-search of the graph G, one vertex, say t, must have been an inner vertex at the time s did its BDFS (as in figure 3.5-c). In this case all the vertices on the back-tracing path of t which already replied to their parents are outer and only the btp of s may contain a vertex that is inner and has replied to its parent.

□

Theorem 3.8: If there is an augmenting path with the control processor at one end, the augmentation algorithm will find such a path.

Proof: An augmenting path exists if there is a free vertex other than the control processor adjacent to an outer vertex with respect to the control processor. BI-EDGE simulates Witzgall & Zahn's forward step, enlarging the labeled subgraph A by searching for a vertex that is not already in A. BLOSSOM updates the labeled subgraph A by searching for an edge not in A that joins two outer vertices of A, as does Witzgall & Zahn's blossom step. The distributed algorithm repeats the search for those vertices which are changed to outer vertices during BLOSSOM and which have previously replied to their parents. This guarantees that all vertices adjacent to an outer vertex are labeled and ensures that an edge joining two outer vertices in A will be found if such an edge exists.

□

3.4.2.2. Analysis of the Augmenting Phase

Lemma 3.9: The number of messages sent during the second sub-phase is at most $2en + 4e + 2 - n^2 - 2n$.

Proof: An explorer may traverse an edge in both directions except for those edges upon which a processor receives its first explorer. Thus, at most $2e-n+1$ explorers will be sent. Each explorer causes an echo (-found, -blossom or -scan) in return. Thus, at most $2e-n+1$ echoes are sent. A blossom may be detected when an edge which is not in A is found joining two outer vertices. There can be at most $e-n+1$ of these, as a blossom can only be found between two outer vertices, connected by an edge that joins two branches of the tree T mentioned in Section 3.1. Each blossom detection will cause at most $n-1$ blossom-scan messages and an equal or smaller number of blossom-retreat messages. Thus at most $2(e-n+1)(n-1)$ blossom-scan and blossom-retreat messages are sent. Every time a blossom is detected, either one echo-resume or one echo-extend message and one request-ok message is used for vertices s and t to conclude the BLOSSOM step. Thus no more than $2(e-n+1)$ such messages are required. At most $n/2$ vertices are inner. Therefore, no more than $(n-2)n/2$ extend messages and the same number of echo-scanned messages are used for this extension process. Therefore, the total number of messages used in this sub-phase is:

$$\leqslant 2e - n + 1 + 2e - n + 1 + 2(e-n+1)(n-1) + 2(e-n+1) + n(n-2)$$

$$= 2en + 4e + 2 - n^2 - 2n.$$

□

Lemma 3.10: The number of bits sent during the second sub-phase is at most $7en\log_2(n) + 3e\log_2(n) + 9n\log_2(n) + en + 2n$.

Proof: An explorer carries two fields of information, the *ID* and the blossom-number which can be any value in the range from 0 to n^2-1. Thus, we need $3\log_2(n)$ bits for each explorer message. Thus $3\log_2(n)(2e-n+1)$ bits are required for all the explorer messages. The echo-found and echo-blossom messages carry a value less than n, so at most $\log_2(n)$ bits are required. An echo-scanned message carries the blossom-number which requires $2\log_2(n)$ bits. Since each echo message causes one of the above echo messages to be returned, at most $2\log_2(n)(2e-n+1)$ bits are required. The blossom-scan message carries three fields: the blossom-number which requires $2\log_2(n)$ bits and the other two which require $\log_2(n)$ bits each. Therefore, each blossom-scan message uses $4\log_2(n)$ bits. The blossom-retreat message carries three fields, using 1 bit to represent the node-status, $\log_2(n)$ bits to represent extend-ID and $2\log_2(n)$ bits for the blossom-number. Thus each blossom-retreat uses $3\log_2(n) + 1$ bits and all blossom messages require at most

$$7en\log_2(n) + 14n\log_2(n) + en + 2n$$

$$- (7n^2\log_2(n) + 7e\log_2(n) + 7\log_2(n) + n^2 + e + 1) \text{ bits.}$$

The echo-resume, echo-extend and request-ok messages require no bits other than the message type. The extend message and the additional echos passing the control back to t require another $2n^2\log_2(n)$ bits. Therefore the total number of bits is:

$$\leqslant 7en\log_2(n) + 3e\log_2(n) + 9n\log_2(n) + en + 2n$$

$$- (5n^2\log_2(n) + n^2 + e + 2\log_2(n) + 1).$$

□

3.5. Conclusion

After the second phase, at most $n-2$ non-leaf processors remain unmatched. As mentioned in [PaS82, WiZ65], once a free processor P has failed to find an augmenting path,

there will never be an augmenting path starting with P. Each iteration of phase 3 will eliminate at least one processor either by matching it with a neighbor if an augmenting path is found, or by setting its voting identity (vot-ID) to zero otherwise. Phase 3 will be repeated at most $n-2$ times.

Lemma 3.11: The MATCHING algorithm correctly finds a maximum matching using $O(n^2 e)$ messages.

Proof: The algorithm terminates when the election in phase 3 concludes with zero as the elected *ID*. This implies that either no free processor exists or that every free processor has already been elected and failed to find an augmenting path. Therefore the current matching is maximum. The number of messages sent in phase 1 and phase 2 are $4ne$ and $2e + n$, respectively. The number of messages sent in each iteration of phase 3 is at most $2en + 6e - n^2$. Phase 3 will be repeated no more than $n-2$ times. Therefore, the total number of messages sent for the MATCHING algorithm is

$$\leqslant 4ne + 2e + n + (n-2)(2en + 6e - n^2)$$
$$= 2en^2 + 6en + 2n^2 + 2 - n^3 - 6e.$$

□

Lemma 3.12: The number of bits sent during the MATCHING algorithm is $O(n^2 e \log_2(n))$.

Proof: In Phase 1 each message requires $\log_2(n)$ bits. Thus, at most $4en\log_2(n)$ bits are used in the first phase. The messages used in the second phase carry no other information besides the message type representation, which will be take care together with the other messages in the other phase. The total number of bits used in each iteration of phase 3 is the sum of the values given in Lemmas 3.7 and 3.10. Phase 3 will repeat at most $n-2$ times. Therefore, the total number of bits sent for the MATCHING algorithm (excluding the bits for message representation) is

$$\leqslant 4en\log_2(n)+(n-2)(7en\log_2(n) + 3e\log_2(n) + 11n\log_2(n) + en + 2n)$$
$$- (n-2)(5n^2\log_2(n) + n^2 + e + 4\log_2(n) + 1)$$

Since there are 22 different types of messages, 5 bits are required in each message to represent the message type. Therefore, the total number of bits sent during the MATCHING algorithm including the message type bits is

$$\leqslant 7en^2\log_2(n) + 21n^2\log_2(n) + 7en + 6\log_2(n) + en^2.$$

□

The time required by the MATCHING algorithm can be calculated if we assume that each message sent takes one time unit to arrive and that each processor can simultaneously send and receive a message. Most of the algorithm, except for the election in the first phase and some steps (blossom message), is done sequentially, that is only one processor is active at a time. Therefore the time required to find the maximum matching is approximately the same as the total number of messages sent.

CHAPTER 4

SIMULATION OF THE MATCHING ALGORITHM

In the previous chapters, we presented distributed algorithms **finding cycles of a given length** and for **maximum matching**. The worst case message complexity of each of the problems has been analyzed. In this chapter we report the results of an implementation of the maximum matching algorithm on various sizes of graphs and study the algorithm's performance as the density of edges of the graph varies. For each size and edge density, we consider the average number of messages sent, the average number of iterations required in phase 3 and the average time to complete the computation.

We begin by describing the simulator used for our implementation and the method used to generate the networks. The simulation results are shown in Section 4.4.

4.1. Simulator

The algorithm was implemented and tested on a distributed network simulator. The simulation package, "Distributed C" (abbreviated DC below), is an extension of the C programming language by Muir [Mui85a, Mui85b]. Within DC, which provides a message driven mechanism for testing distributed algorithms, the user defines his own network topology and the code executed by each node. A DC program consists of the DC software library and the user supplied algorithm. The DC software library consists of the run kernel, library routines, simulation and debugger utilities. The simulation package DC has been adapted for use at Simon Fraser University on SUN-2 workstations running Unix 4.2.

The DC software library routines are self contained and sufficient to handle many distributed algorithms. However, a few facilities were added, which we found useful but were not supported by DC. These additional facilities include edge scanning functions (reset-scan, scan-ports and more-unscan), random-send and random-relay. (These give the user flexibility to design an algorithm for a node to explore its neighbors arbitrarily (for example a DFS).) The functions random-relay and random-send examine all the incident edges that have not been scanned. They assign a probability to each of those edges and the edge with the highest probability is chosen to send the next message along. In this way, during different runs, we can expect to generate different DFS and BDFS trees (given different starting points for the random number generator). The scanning functions (reset-scan, scan-ports and more-scan) allow the user to mark all the edges unscanned, mark an edge as scanned, and test whether or not all the edges incident on a processor are marked scanned, respectively.

The user supplied algorithm consists of 3 basic parts: 1) DC language extensions, 2) coding of the processes, and 3) configuration of the network. The DC language extensions contain the macro definitions used by the user's algorithm to call the routines in the DC software library. The coding of the user's distributed algorithm begins with the optional declaration of states and messages, labeling of ports and process initialization. This is followed by the main body of the algorithm, which is divided into states and is message driven. The network configuration contains the scheduler information, processor definition and specification of the connections between processors in the network.

DC is a sequential program which simulates the execution of an algorithm in a distributed environment. The message manager of DC uses a logical system clock to assign a random arrival time (within specified limits) for each message sent. Each processor's behavior is independent of the clock time. The package's logical system clock increments whenever

all the messages scheduled to be received at the current time have been received by the appropriate processors. Since we assume that messages will arrive within some finite period of time and that the system is reliable, it is acceptable to assign an arrival time to each message sent. Each processor receives messages from its neighbors, performs local computations and sends messages to its neighbors in zero time. The time required to complete the computation is the number of logical system clock ticks required to complete the computation.

There are two schedulers in the simulator: the processor scheduler and the message scheduler. Both schedulers support fixed and random scheduling modes, which act differently in each scheduler. If the processor scheduler is in fixed mode, all processors wake up at the time the computation starts. If the processor scheduler is in random mode then the processors wake up arbitrarily within a specified (finite) period of time. If the message scheduler is in fixed mode, every message from one processor to another (its neighbor) takes exactly one unit of time to arrive. In random scheduling mode, a message sent from one processor to another takes an arbitrary amount of time to arrive (bounded by the user's specified maximum delay). Random scheduling mode is used for both the processor scheduler and the message scheduler throughout our computation with 21 clock ticks chosen for the maximum processor scheduler delay and 7 clock ticks chosen for the maximum message scheduler delay. Since the maximum delay for the processor scheduler is 3 times as much as the delay for the message scheduler, we expect that about 1/3 of the processors wake up by themselves and initiate election messages bearing their own ids. When the simulator begins execution, wakeup messages are sent to each processor by the system which arrive within the maximum delay specified. If a processor receives this message before receiving any message from its neighbors, it is considered to have awakened by itself. When a message is sent, it is given a propagation delay which is calculated by adding a random delay to the current value of the logical system clock. Thus, if messages are sent

on different edges, a message sent earlier (with respect to the logical system clock) may reach its destination after a message sent in a later period. However, messages sent along the same edge arrive in FIFO order. A message on an edge can arrive no sooner than 1 time unit after a previous message on the same edge. Furthermore, no message is lost by the system.

When a message is sent, it is assigned a propagation delay and is stored in a priority queue maintained by the simulator. The receiver of the first message in the priority queue is wakened by the simulator to perform the computation and send out responses if necessary. The priority queue is updated and the process is repeated for the next message in the priority queue. An execution is terminated when a processor reaches the stopping condition and signals termination. For empirical analysis we retain the total number of messages sent during this process, ignoring the wakeup messages sent by the simulator.

4.2. Network Topology

The networks used in our computation are random graphs generated by the **constant density model** (see Karp [Kar76]). A random graph generated by the constant density model is defined as follows: for a graph of n vertices, each pair of i, j $\in$ {1,..n}, include {i,j} as an edge with probability p, independent of what other edges are included. The networks used were generated in this manner with various values n and p, discarding those graphs which were not connected. For each network size n, we generated random connected graphs of varying densities. For networks of 23, 24 and 25 processors, we included edges with probabilities 0.1, 0.15, 0.2, 0.25, 0.3, 0.5, 0.75 and 1. For networks of 50 processors, we repeated these probabilities as well as 0.05. For networks of 100 processors, we generated edges with probabilities 0.03, 0.05 and 0.1. For networks of sizes 23, 24 and 25, the probability that a graph generated in this method is connected is near zero when the edge

probability used is smaller than 0.1. Similarly for size 50, the probability that the graph is connected is quite small when the edge probability is smaller than 0.05 (see [Kar76]). The results in the tables are based on the total number of runs for each graph size and probability. For each graph size and probability, the algorithm was executed 50 times, 10 times each on 5 random connected graphs, except for the case of 50 processors with probability 1. In this latter case, the algorithm was executed for a total of 10 runs. The average number of messages sent and the average time to complete an execution are based on these results.

4.3. Maximum Matching Algorithm

The algorithm presented in the previous chapter, has been coded in DC. Each processor is in one of five states: INITIATOR, PhaseOne, PhaseTwo, Phase3One and Phase3Two. Each processor is in the INITIATOR state until it is waken up. The PhaseOne and PhaseTwo states correspond to the multi-election and depth-first search phases, respectively. The last two states handle the augmentation of the matching problem. Phase3One is the mono-election and state Phase3Two combined BI-EDGE, BLOSSOM and AUGMENTING steps. Each state is divided into cases corresponding to the type of message that a processor may receive in that state. A processor upon receiving a message, performs local computations based on its local variables and the information contained in the message and makes an appropriate response. The following tables summarize the results of our simulations:

4.4. Simulation Results and Conclusion

The average number of messages sent during the execution of the maximum matching algorithm is shown in figure 4.1. The message complexity increases although the number of iterations in phase three (figure 4.7) tends to decrease as the density of the graph increases. This increase in the message complexity is in inverse proportion to the number of iterations

of the third phase. This can be explained by considering the number of messages sent in each phase.

The maximum matching algorithm is divided into three phases: multi-processor election, initial matching and augmentation. The average number of messages sent in each phase is shown in figures 4.2, 4.3 and 4.4, respectively. The first two phases are compulsory for graphs of any size and topology. The number of messages sent in these two phases is directly proportional to both the graph size and density.

The large number of messages sent in the first phase is partly due to the multiple initiators of the election (table 4.1) and partly due to the increase in edge density. An election initiated by a processor requires at most $4e$ messages. Ideally, restricting the number of processors which initiate the election could substantially reduce the number of messages. For those graphs with high edge density, the number of messages needed to construct the initial matching dominates the total number of messages sent. Theoretically, it is difficult to restrict the number of processors which initiate the election in an asynchronous distributed network. For our simulation we assumed a fairly high number of initiators (see Section 4.1). Fortunately, in a "real world" situation, the number of processors which initiate the election is likely to be small, so this is not a critical drawback of our algorithm.

Network	Edge Probability × 100									
Size	3	5	10	15	20	25	30	50	75	100
23			8.92	7.08	6.08	5.98	5.66	4.10	3.96	2.98
24			8.44	7.82	6.72	5.80	5.06	4.40	3.64	3.50
25			9.30	7.68	6.42	6:12	5.36	4.64	3.98	4.06
50		17.58	13.40	10.68	9.58	8.52	8.08	6.14	6.24	4.80
100	33.82	25.22	18.94							

Table 4.1: Average number of processors initiating the phase one election.

Network Size	Edge Probability × 100									
	3	5	10	15	20	25	30	50	75	100
23			29	43	52	64	70	129	188	253
24			34	43	57	72	83	143	202	276
25			35	48	61	73	94	144	230	300
50		68	118	185	251	308	382	609	913	1225
100	161	252	482							

Table 4.2: Average number of edges in graphs.

The initial matching uses the DFS approach to create a large initial matching with $2e + n$ messages. This is relatively small compared to the number of messages used to perform an election in phase three.

The initial matchings created by the DFS are close to maximum cardinality. This results in a small number of iterations of the third phase (figure 4.7). The number of messages sent during the election of phase three increases steadily as the edge density increases. However the total number of messages sent in phase three (figure 4.4) does not grow appreciably once the edge density has reached 30%. This is due to the decrease in iterations of phase three.

The average time for a processor to compute the maximum matching is shown in figure 4.8.

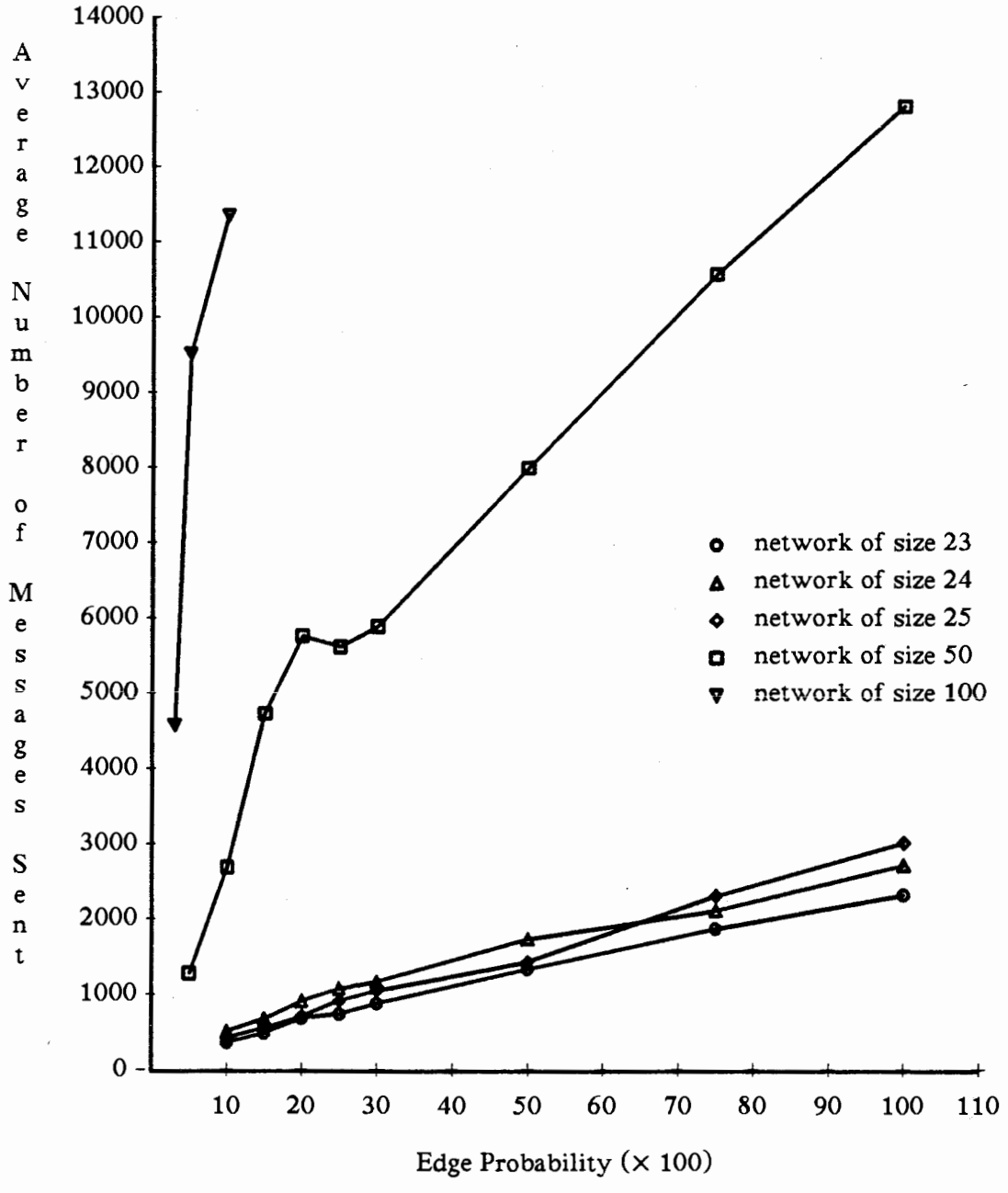

Figure 4.1: Average number of messages sent.

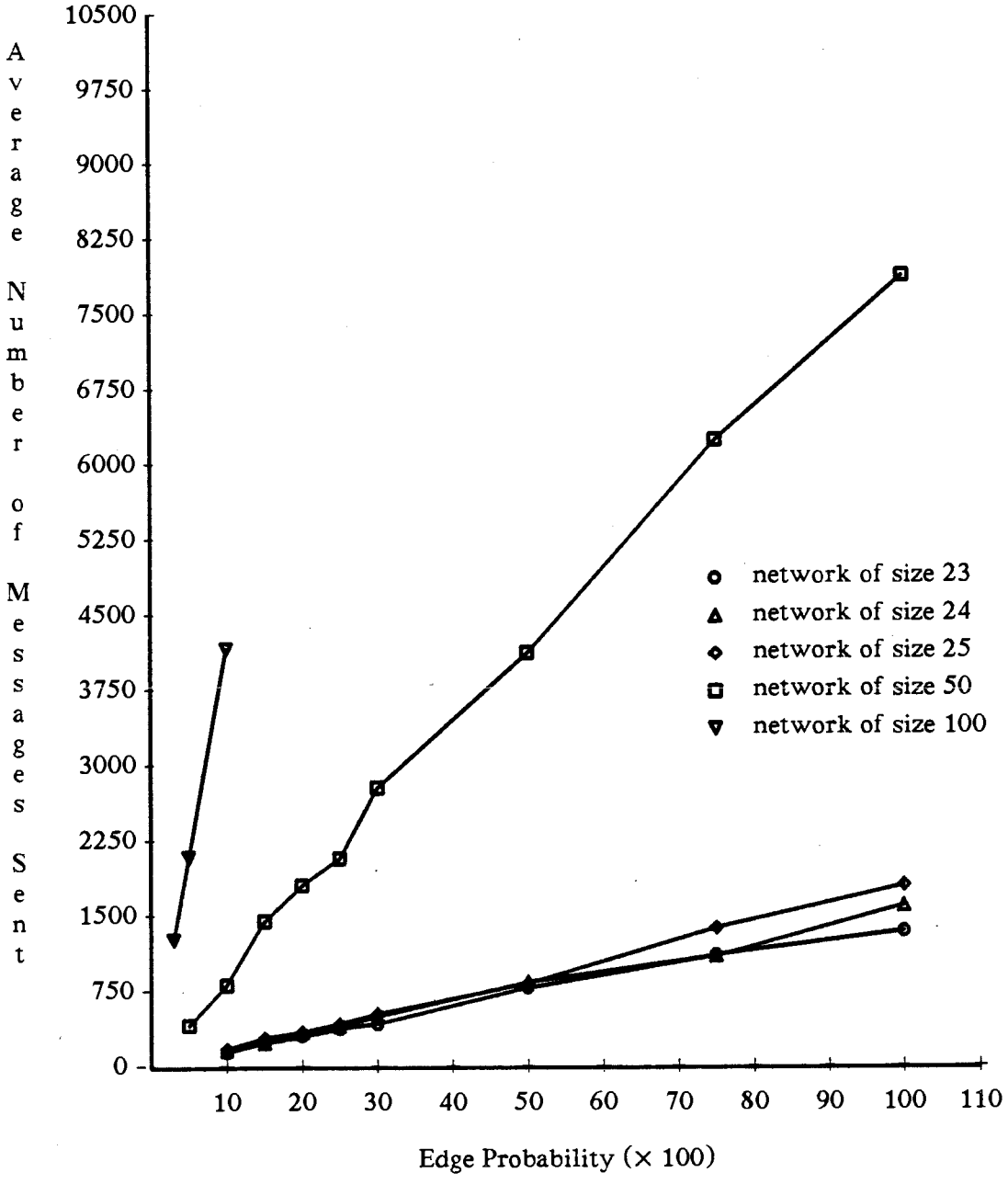

Figure 4.2: Average number of messages sent during phase one.

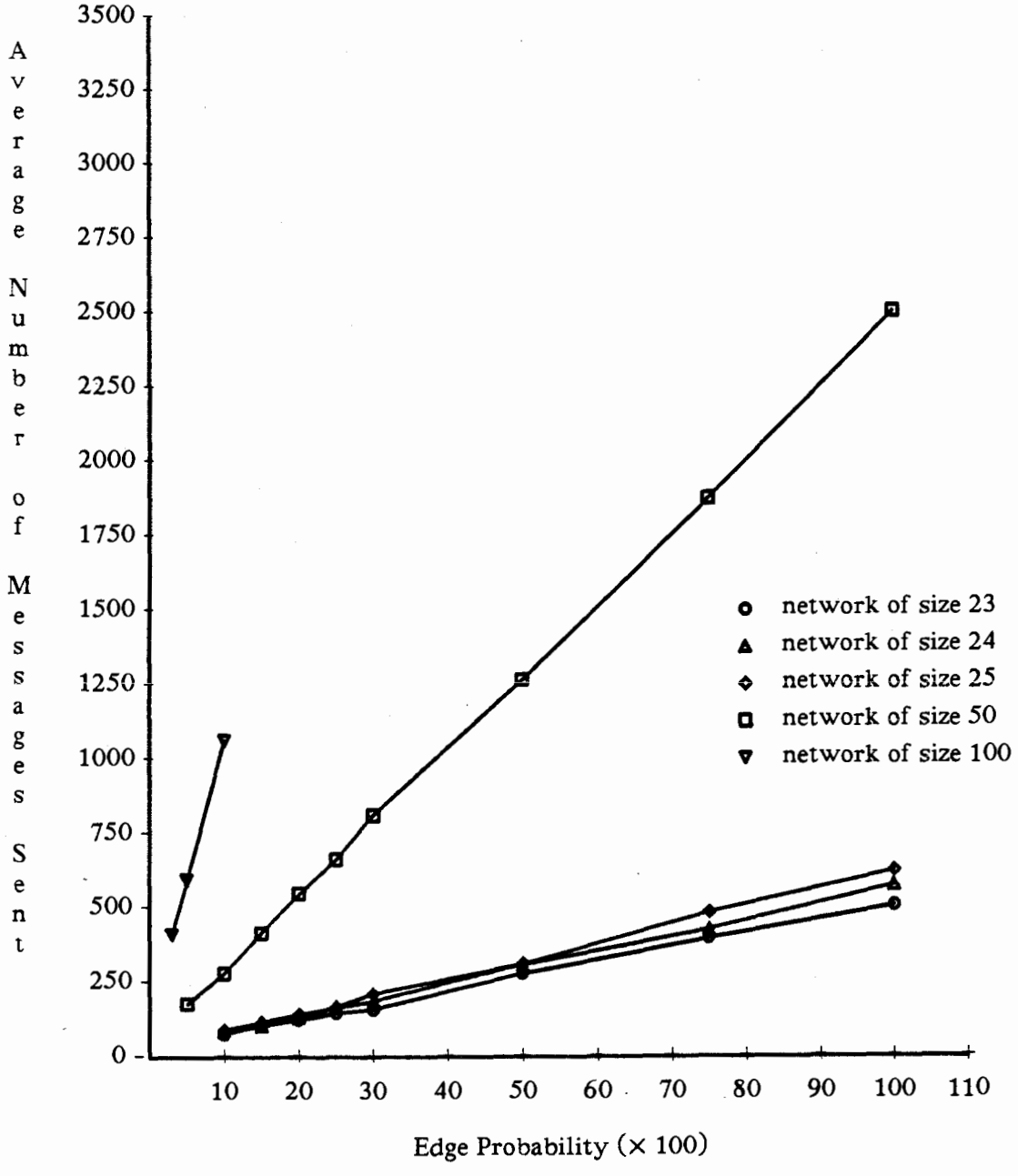

Figure 4.3: Average number of messages sent during phase two.

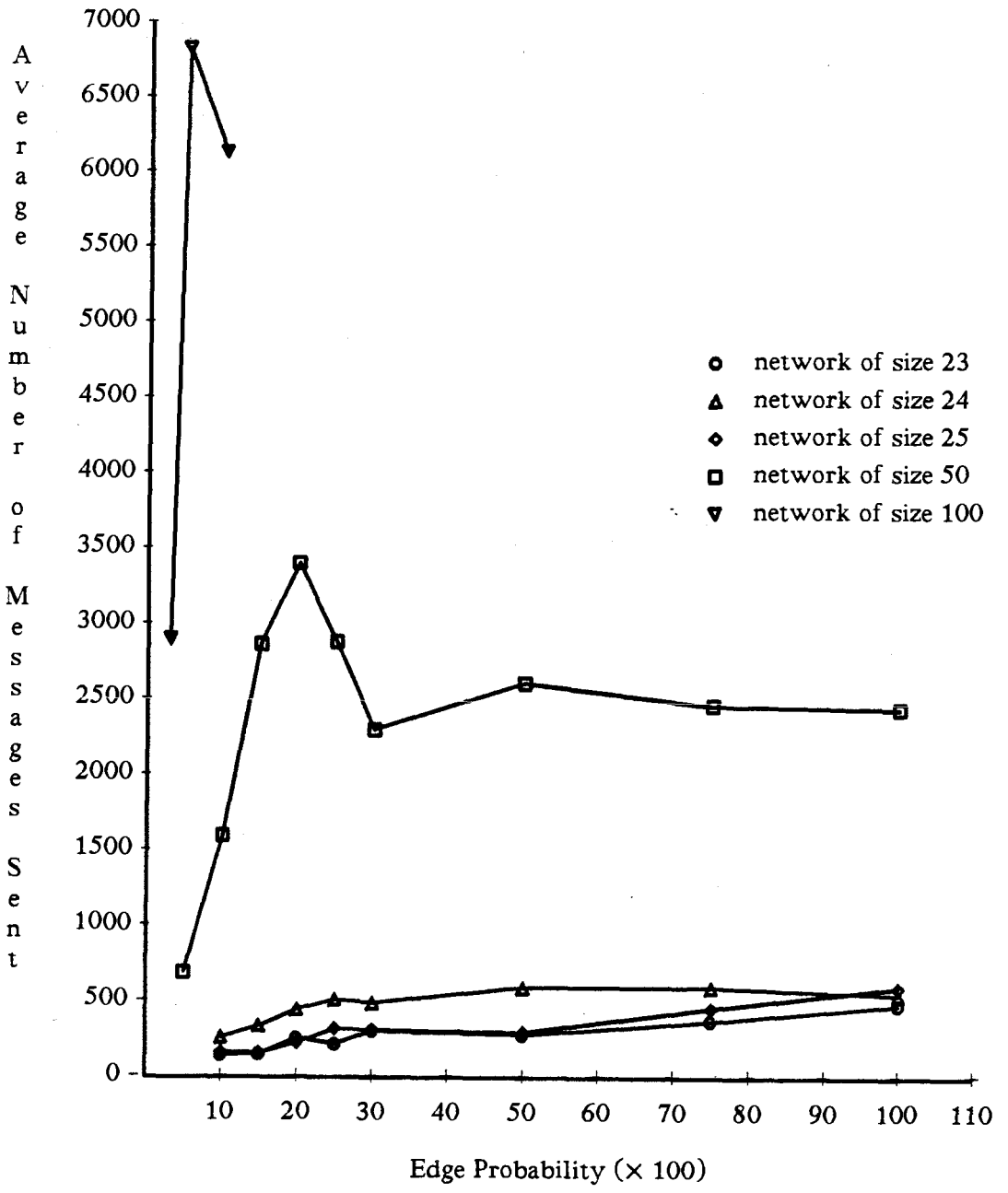

Figure 4.4: Average number of messages sent during phase three.

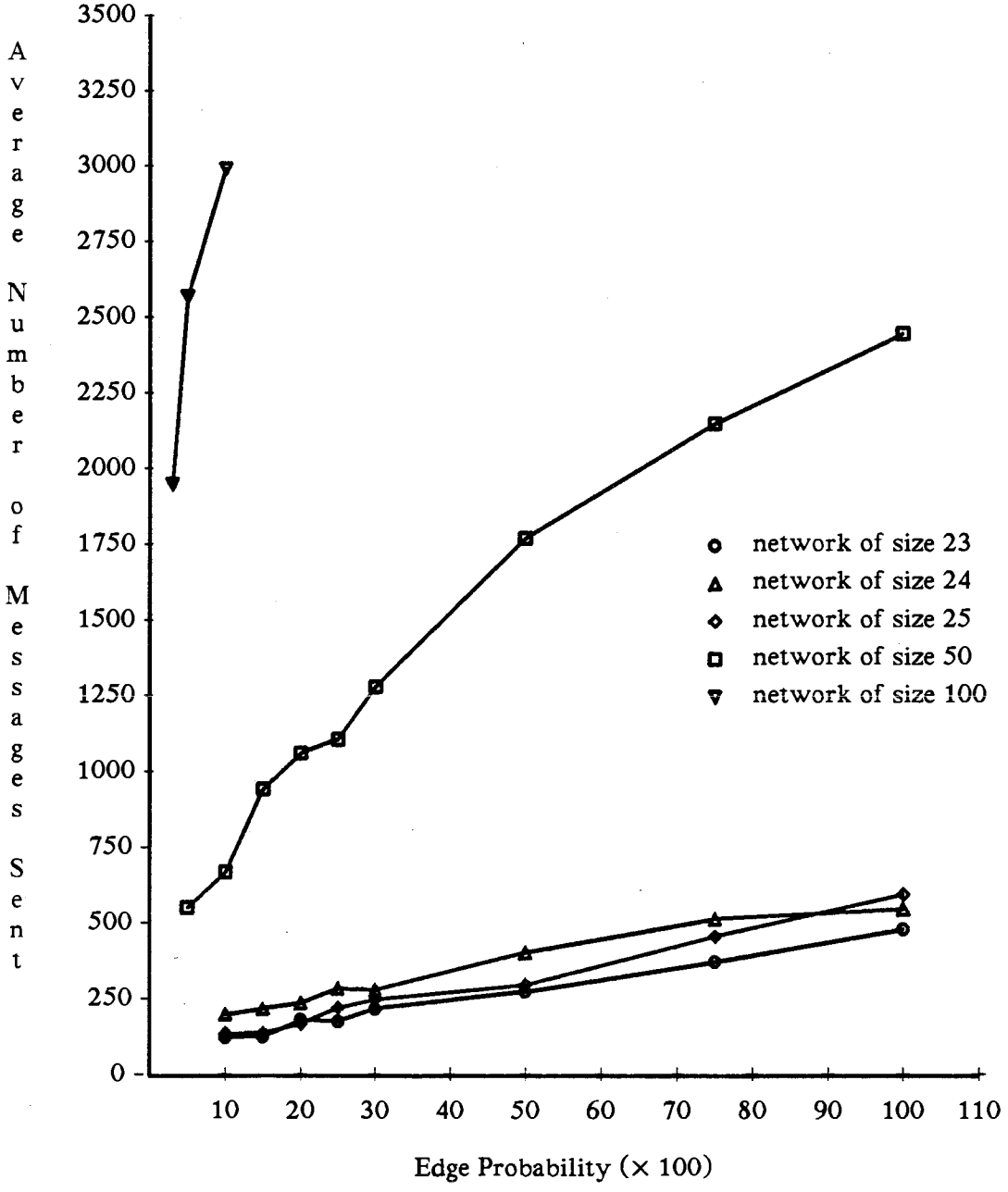

Figure 4.5: Average number of messages sent during the election of phase three.

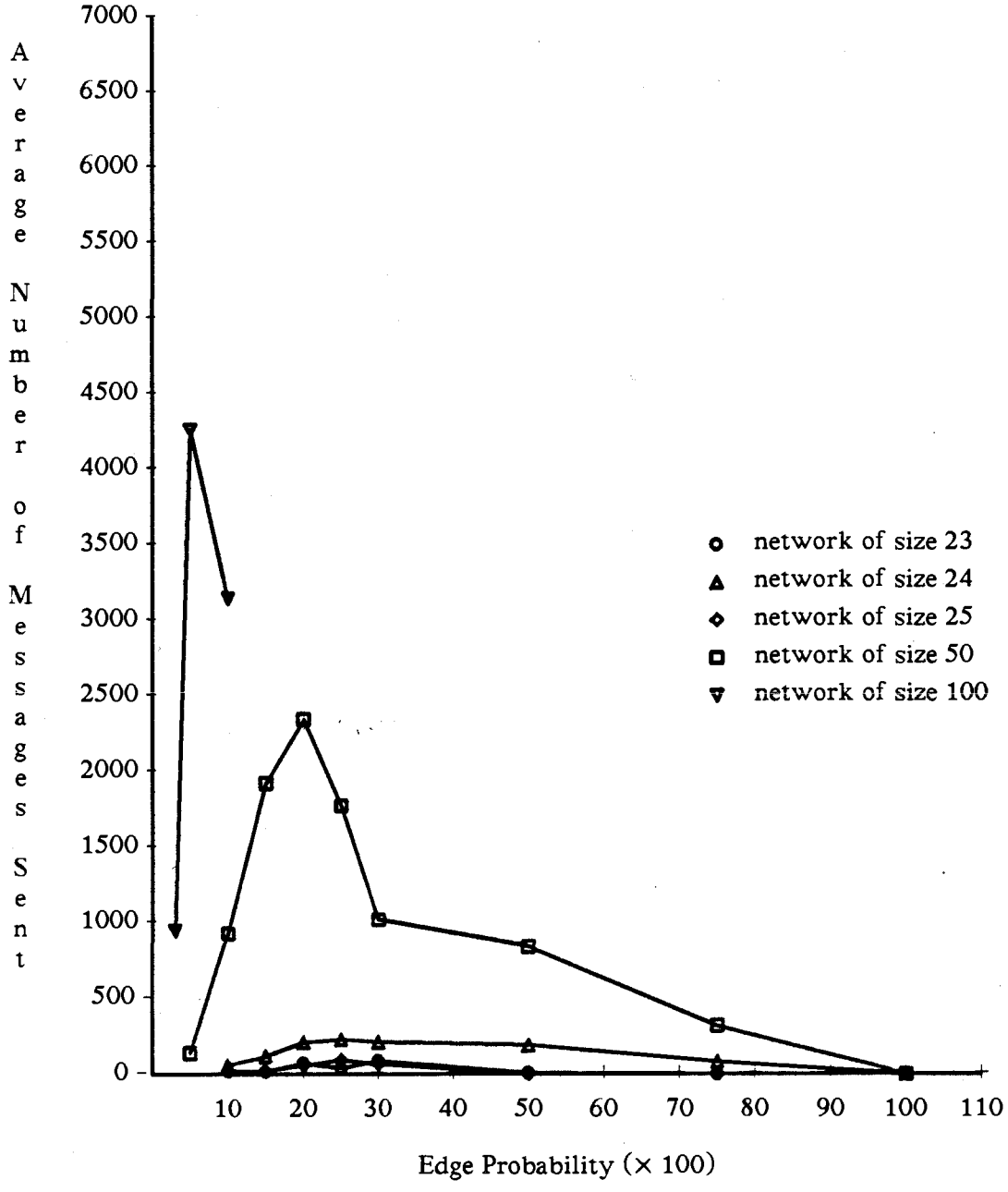

Figure 4.6: Average number of messages sent during the augmentation of phase three.

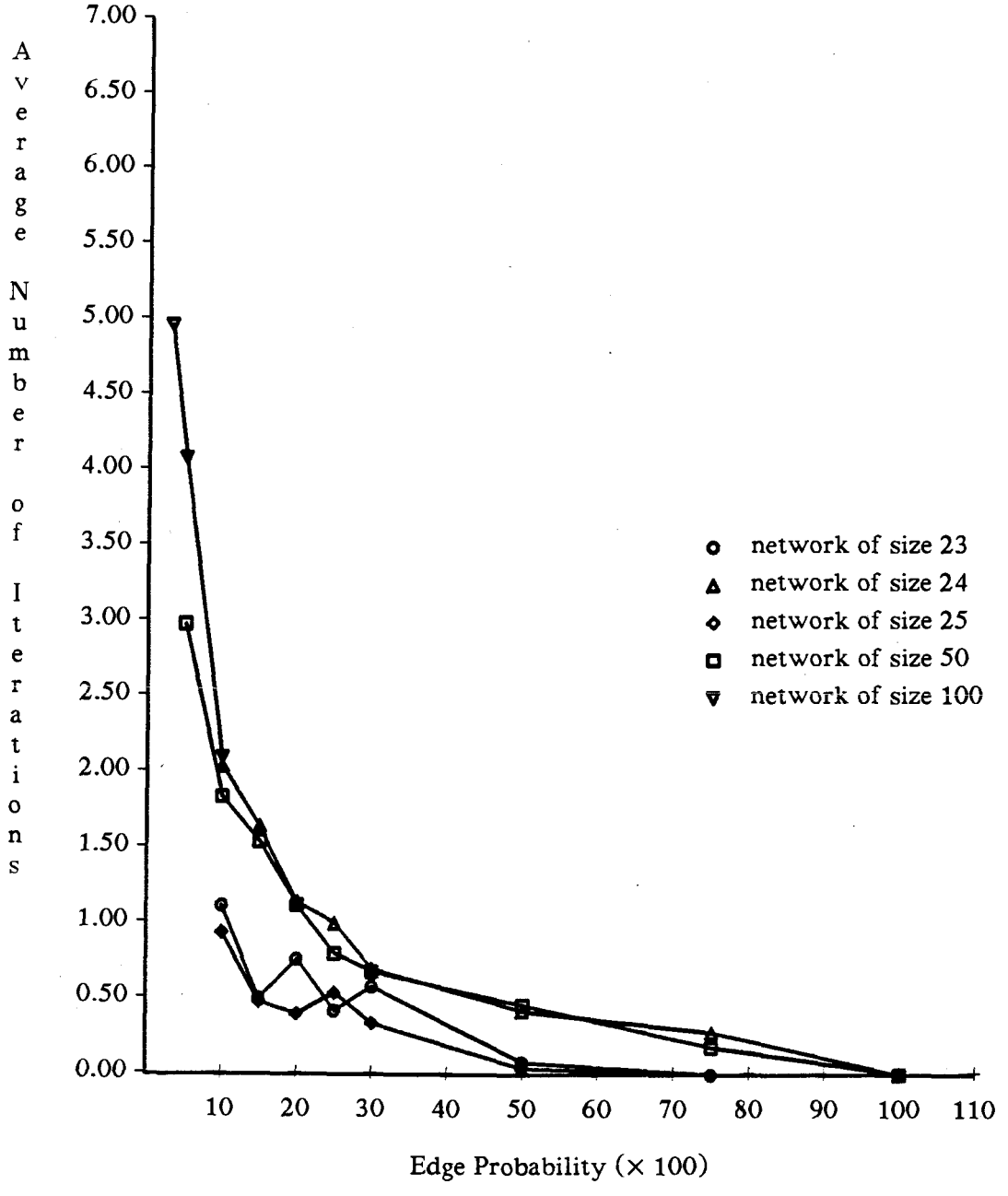

Figure 4.7: Average number of iterations in phase three.

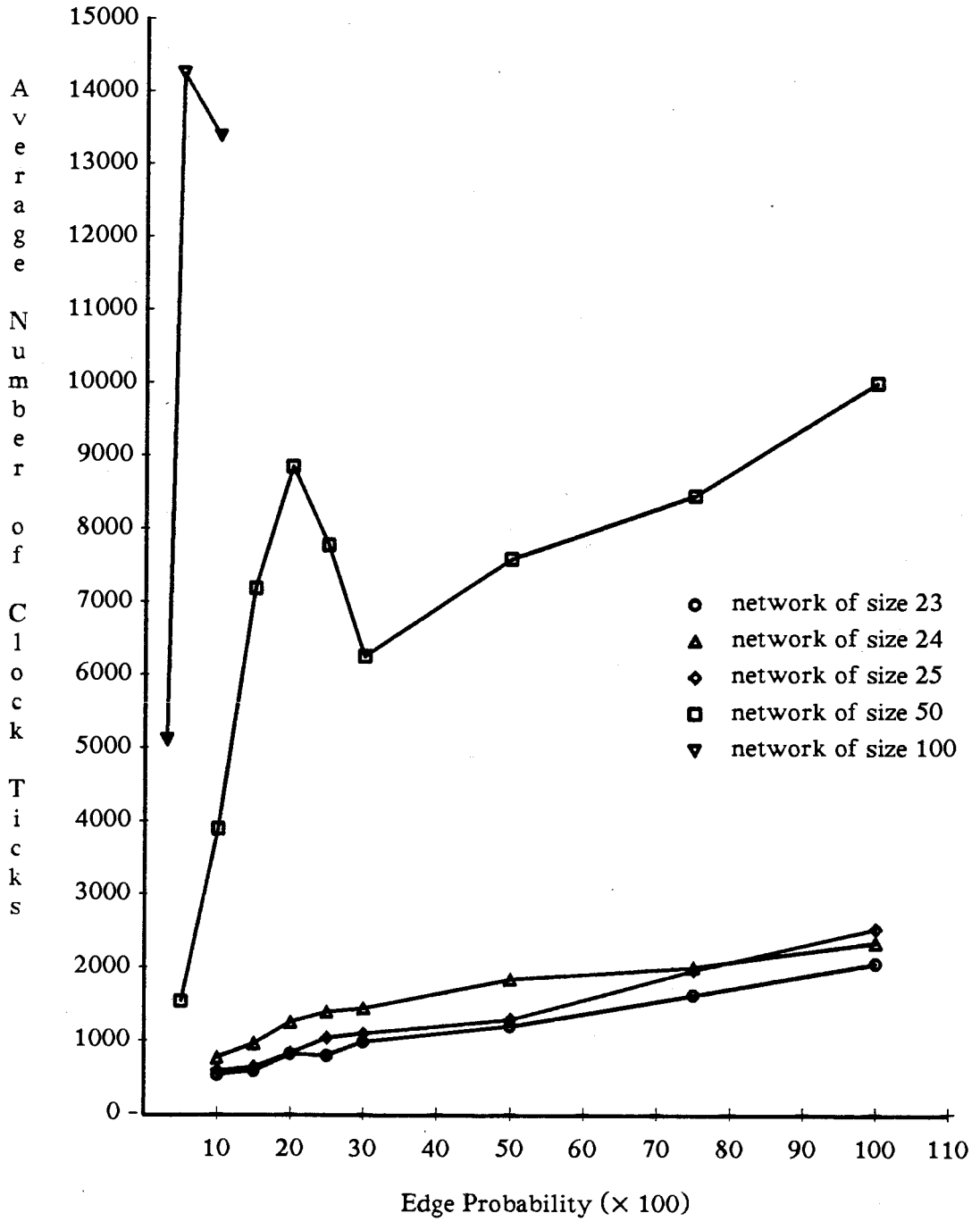

Figure 4.8: Average time to compute the maximum matching.

CHAPTER 5

CONCLUSION

The two graph problems studied in this thesis are finding cycles of a given length and maximum matching in a distributed fashion for general graphs. An algorithm for the former problem with message complexity of $O(e(d-1)^{\lfloor k/2 \rfloor - 1})$ for cycles of length k has been presented. The network is assumed to be synchronous for this problem. In this thesis we were unsuccessful in devising an algorithm for an asynchronous network. The main difficulty appears to be the termination of the algorithm in the event that no such cycle exists. It would be worthwhile to investigate this problem further for the asynchronous case. It would also be interesting to investigate the problem of finding a cycle of specified weight in a weighted graph. Our algorithms do not immediately generalize to solve this problem without requiring large amounts of storage at each processor.

An algorithm for the maximum matching problem has been presented which has $O(n^2 e)$ message complexity. For this problem, the network is assumed to be asynchronous and each processor requires no network topological information other than the number of neighbors that it has. This algorithm has been implemented and tested on a variety of graphs using the DC simulator. From the empirical results, we see that the complexity of the algorithm in a sparse graph is due to the number of iterations of the augmentation phase. As the density of the graph increase, the messages required for the election and the messages used to deal with blossoms dominate the complexity of the algorithm. This algorithm could be improved by a better election algorithm, a better method to deal with blossoms, or possibly by a different approach to the augmentation. Since the algorithm

presented is essentially a token passing algorithm it does not take full advantage of the parallelism which is possible in a distributed environment. It would be useful to design a more fully distributed algorithm for this problem. It would also be interesting to consider the problem of finding a maximum weight matching in a weighted graph.

References

[Ber57] C. Berge, Two Theorems in Graph Theory, Proceedings of the National Academy of Science, 1957.

[ChR79] E. Chang and R. Roberts, An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processes, *Comm. ACM 22*, (1979), 281-283.

[Cha79] E. Chang, An Introduction to Echo Algorithms, 1st Int. Conf. on Dist. Comp. Syst., 1979.

[Che82] C. C. Chen, A Distributed Algorithm for Shortest Paths, *IEEE Trans. Comp. 31*, (1982), 898-899.

[DKR82] D. Dolev, M. Klawe and M. Rodeh, An O(n log n) Unidirectional Distributed Algorithm for Extrema Finding in a Circle, *J. of Alg. 3*, (1982), 245-260.

[Edm65] J. Edmonds, Paths, Trees and Flowers, *Canadian J. of Math. 17*, (1965), 449-467.

[EvK75] S. Even and O. Kariv, An $O(n^{2.5})$ Algorithm for Maximum in General Graphs, Proceedings of the 16th Annual Symp. on Annual Symp. on Foundations of Computer Science, 1975.

[Gab72] H. Gabow, An Efficient Implementation of Edmonds' Maximum Matching Algorithm, Technical Report Stan-CS, June 1972.

[GHS83] R. G. Gallager, P. A. Humblet and P. M. Spira, A Distributed Algorithm for Minimum-Weight Spanning Trees, *ACM Trans. Prog. Lang. and Systems 5*, (1983), 66-77.

[GaJ79] M. R. Garey and D. S. Johnson, A Guide to the Theory of NP-Completeness, in *Computers and Intractability*, W. H. Freeman, New York, 1979.

[HoK73] J. E. Hopcroft and R. M. Karp, An $n^{2.5}$ Algorithm for Maximum Matching in Bipartite Graphs, *SIAM J. on Comp. 2*, (Dec. 1973), 225-231.

[ItR77] A. Itai and M. Rodeh, Finding a Minimum Circuit in a Graph, *Proc. 9th Ann. ACM Symp. on Theory of Computing*, , 1977, 1-10.

[JrS81] D. S. P. Jr. and B. Samadi, Adaptive Distributed Minimal Spanning Tree Algorithms, Proc. IEEE Symp. Rel. in Dist. Software and Database Syst., 1981.

[Kar76] R. M. Karp, The Probabilistic Analysis of Some Combinatorial Search Algorithms, Proceedings of the Symposium on Algorithm and Complexity: New Directions and Recent Results, 1976.

[KRS82] E. Korach, D. Rotem and N. Santoro, Distributed Algorithms for Ranking the Nodes of a Network, Proc. 13th SE Conf. on Comb., Graph Theory, and Comp., 1982.

[KRS84] E. Korach, D. Rotem and N. Santoro, Distributed Election in a circle without a global sense of orientation, *Intern. J. Computer Math 16*, (1984), 115-124.

[Lak84] G. D. Lakhani, An Improved Distribution Algorithm for Shortest Paths Problem, *IEEE Trans. Comp. 33*, (1984), 855-857.

[Lou84] M. C. Loui, The Complexity of Sorting on Distributed Systems, *Inf. and Control 60*, (1984), 70-85.

[Mat83] T. A. Matsushita, Distributed Algorithms for Selection, Technical Report ACT-37, Coordinated Science Lab, Univ. of Illinois at Urbana-Champaign, 1983.

[MiV80] S. Micali and V. V. Vazirani, A $O(|E|^{1/2}|E|)$ Algorithm for Finding Maximum Matching in General Graphs, *Proc. Twenty-first Annual Symposium on Annual Symp. on Foundations of Computer Science*, , 1980, 17-27.

[MiC82] J. Misra and K. M. Chandy, Distributed Computation on Graphs: Shortest Path Algorithms, *Comm. ACM 25*, (1982), 833-837.

[Mui85a] J. Muir, Distributed C Version 3: User Manual, Technical Report: SCS-Tech. Rep.-3, School of Computer Science, Carleton University, 1985.

[Mui85b] J. Muir, Distributed C Version 3: System Manual, Technical Report: SCS-Tech. Rep.-2, School of Computer Science, Carleton University, 1985.

[PaS82] C. H. Papadimitriou and K. Steiglitz, in *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Inc., New Jersey, 1982, 235.

[Pet82] G. L. Peterson, An O(n log n) Unidirectional Algorithm for the Circular Extrema Problem, *ACM Trans. Prog. Lang. and Systems 4*, (1982), 758-762.

[RiL85] D. Richards and A. L. Liestman, Finding Cycles of a Given Length, *Ann. of Disc. Math. 27*, (1985), .

[Rod82] M. Rodeh, Finding the Median Distributively, *J. Comp. and Syst. Sci. 24*, (1982), 162-166.

[RSS83] D. Rotem, N. Santoro and J. B. Sidney, A Shout-Echo Algorithm for Finding the Median of a Distributed Set, Proc. 14th SE Conf. on Comb., Graph Theory, and Comp., 1983.

[RSS84] D. Rotem, N. Santoro and J. B. Sidney, Shout Echo Selection in Distributed Files, Technical Report SCS-Tech. Rep.-21, School of Computer Science, Carleton University, 1984.

[SaS82] N. Santoro and J. B. Sidney, Order Statistics on Distributed Sets, Proc. 20th Allerton Conf., 1982.

[Seg82] A. Segall, Decentralized Maximum-flow Protocols, *Networks 12*, (1982), 213-220.

[SFR83] L. Shrira, N. Francez and M. Rodeh, Distributed k-Selection: From a Sequential to a Distributed Algorithm, Proc. 2nd Ann. ACM Symp. Princ. Dist. Comp., 1983.

[Spi77] P. M. Spira, Communication Complexity of Distributed Spanning-Tree Algorithms, Proc. 2nd Berkeley Conf on Distr. Data Management and Comp. Networks, 1977.

[WiZ65] C. Witzgall and C. T. Zahn, Modification of Edmonds' Maximum Matching Algorithm, *J. of Research of the National Bureau of Standards 69B*, (1965), 91-98.