Please type a plus sign (+) inside this box [+]

08-02-00

PTO/SB/05 (4/98) A

UTILITY PATENT APPLICATION TRANSMITTAL

(Only for nonprovisional applications under 37 CFR § 1.53(b))

Attorney Docket No.	030048002US
First Inventor or Application Identifier	Virgil E. Bourassa
Title	JOINING A BROADCAST CHANNEL
Express Mail Label No.	EL404935398US

APPLICATION ELEMENTS

See MPEP chapter 600 concerning utility patent application contents.

ADDRESS TO: Box Patent Application
Assistant Commissioner for Patents
Washington, D.C. 20231

1. [ ] Authorization for Extensions & Fee Transmittal *(Submit an original and a duplicate for fee processing)*
2. [X] Specification [Total Pages] **48**
   (preferred arrangement set forth below)
   - Descriptive Title of the Invention
   - Cross References to Related Applications
   - Statement Regarding Fed sponsored R & D
   - Reference to Microfiche Appendix
   - Background of the Invention
   - Brief Summary of the Invention
   - Brief Description of the Drawings *(if filed)*
   - Detailed Description
   - Claim(s)
   - Abstract of the Disclosure
3. [X] Drawing(s) *(35 USC 113)* [Total Sheets] **39**
4. Oath or Declaration [Total Pages] ____
   a. [ ] Newly executed (original or copy)
   b. [ ] Copy from a prior application (37 CFR 1.63(d)) *(for continuation/divisional with Box 16 completed)*
      i. [ ] DELETION OF INVENTOR(S) Signed statement attached deleting inventor(s) named in the prior application, see 37 CFR 1.63(d)(2) and 1.33(b)

**NOTE FOR ITEMS 1 & 13: IN ORDER TO BE ENTITLED TO PAY SMALL ENTITY FEES, A SMALL ENTITY STATEMENT IS REQUIRED (37 C.F.R. § 1.27), EXCEPT IF ONE FILED IN A PRIOR APPLICATION IS RELIED UPON (37 C.F.R. § 1.28).*

5. [ ] Microfiche Computer Program *(Appendix)*
6. Nucleotide and/or Amino Acid Sequence Submission *(if applicable, all necessary)*
   a. [ ] Computer-Readable Copy
   b. [ ] Paper Copy (identical to computer copy)
   c. [ ] Statement verifying identity of above copies

ACCOMPANYING APPLICATION PARTS

7. [ ] Assignment Papers (cover sheet & document(s))
8. [ ] 37 CFR 3.73(b) Statement *(when there is an assignee)* [ ] Power of Attorney
9. [ ] English Translation Document *(if applicable)*
10. [ ] Information Disclosure Statement (IDS)/PTO-1449 [ ] Copies of IDS Citations
11. [ ] Preliminary Amendment
12. [X] Return Receipt Postcard
12. [ ] Small Entity Statement(s) [ ] Statement filed in prior application, Status still proper and desired
14. [ ] Certified Copy of Priority Document(s) *(if foreign priority is claimed)*
15. [ ] Other: ____

16. If a CONTINUING APPLICATION, *check appropriate box and supply the requisite information below and in a preliminary amendment*

[ ] Continuation [ ] Divisional [ ] Continuation-In-Part (CIP) of prior Application No.: ____

Prior application information: Examiner ____ *Group / Art Unit* ____

For CONTINUATION or DIVISIONAL apps only: The entire disclosure of the prior application, from which an oath or declaration is supplied under Box 4b, is considered a part of the disclosure of the accompanying continuation or divisional application and is hereby incorporated by reference. The incorporation can only be relied upon when a portion has been inadvertently omitted from the submitted application parts.

[ ] Claims the benefit of Provisional Application No. ____

17. CORRESPONDENCE ADDRESS

Customer Number 25096 / Barcode

25096
PATENT TRADEMARK OFFICE

Respectfully submitted,

TYPED or PRINTED NAME Maurice ~~J. Pirio~~ REGISTRATION NO. 33,273

SIGNATURE Maurice J Pirio Date July 31 2000

JOINING A BROADCAST CHANNEL

CROSS-REFERENCE TO RELATED APPLICATIONS

This application is related to U.S. Patent Application No. ______________, entitled "BROADCASTING NETWORK," filed on July 31, 2000 (Attorney Docket No. 030048001 US); U.S. Patent Application No.______________, entitled "JOINING A BROADCAST CHANNEL," filed on July 31, 2000 (Attorney Docket No. 030048002 US); U.S. Patent Application No.______________, "LEAVING A BROADCAST CHANNEL," filed on July 31, 2000 (Attorney Docket No. 030048003 US); U.S. Patent Application No.______________, entitled "BROADCASTING ON A BROADCAST CHANNEL," filed on July 31, 2000 (Attorney Docket No. 030048004 US); U.S. Patent Application No.______________, entitled "CONTACTING A BROADCAST CHANNEL," filed on July 31, 2000 (Attorney Docket No. 030048005 US); U.S. Patent Application No.______________, entitled "DISTRIBUTED AUCTION SYSTEM," filed on July 31, 2000 (Attorney Docket No. 030048006 US); U.S. Patent Application No.______________, entitled "AN INFORMATION DELIVERY SERVICE," filed on July 31, 2000 (Attorney Docket No. 030048007 US); U.S. Patent Application No.______________, entitled "DISTRIBUTED CONFERENCING SYSTEM," filed on July 31, 2000 (Attorney Docket No. 030048008 US); and U.S. Patent Application No.______________, entitled "DISTRIBUTED GAME ENVIRONMENT," filed on July 31, 2000 (Attorney Docket No. 030048009 US), the disclosures of which are incorporated herein by reference.

TECHNICAL FIELD

The described technology relates generally to a computer network and more particularly, to a broadcast channel for a subset of a computers of an underlying network.

BACKGROUND

There are a wide variety of computer network communications techniques such as point-to-point network protocols, client/server middleware, multicasting network

protocols, and peer-to-peer middleware. Each of these communications techniques have their advantages and disadvantages, but none is particularly well suited to the simultaneous sharing of information among computers that are widely distributed. For example, collaborative processing applications, such as a network meeting programs, have a need to distribute information in a timely manner to all participants who may be geographically distributed.

The point-to-point network protocols, such as UNIX pipes, TCP/IP, and UDP, allow processes on different computers to communicate via point-to-point connections. The interconnection of all participants using point-to-point connections, while theoretically possible, does not scale well as a number of participants grows. For example, each participating process would need to manage its direct connections to all other participating processes. Programmers, however, find it very difficult to manage single connections, and management of multiple connections is much more complex. In addition, participating processes may be limited to the number of direct connections that they can support. This limits the number of possible participants in the sharing of information.

The client/server middleware systems provide a server that coordinates the communications between the various clients who are sharing the information. The server functions as a central authority for controlling access to shared resources. Examples of client/server middleware systems include remote procedure calls ("RPC"), database servers, and the common object request broker architecture ("CORBA"). Client/server middleware systems are not particularly well suited to sharing of information among many participants. In particular, when a client stores information to be shared at the server, each other client would need to poll the server to determine that new information is being shared. Such polling places a very high overhead on the communications network. Alternatively, each client may register a callback with the server, which the server then invokes when new information is available to be shared. Such a callback technique presents a performance bottleneck because a single server needs to call back to each client whenever new information is to be shared. In addition, the reliability of the entire sharing of information depends upon the reliability of the single server. Thus, a failure at a single computer (*i.e.*, the server) would prevent communications between any of the clients.

The multicasting network protocols allow the sending of broadcast messages to multiple recipients of a network. The current implementations of such multicasting network

protocols tend to place an unacceptable overhead on the underlying network. For example, UDP multicasting would swamp the Internet when trying to locate all possible participants. IP multicasting has other problems that include needing special-purpose infrastructure (*e.g.*, routers) to support the sharing of information efficiently.

The peer-to-peer middleware communications systems rely on a multicasting network protocol or a graph of point-to-point network protocols. Such peer-to-peer middleware is provided by the T.120 Internet standard, which is used in such products as Data Connection's D.C.-share and Microsoft's NetMeeting. These peer-to-peer middleware systems rely upon a user to assemble a point-to-point graph of the connections used for sharing the information. Thus, it is neither suitable nor desirable to use peer-to-peer middleware systems when more than a small number of participants is desired. In addition, the underlying architecture of the T.120 Internet standard is a tree structure, which relies on the root node of the tree for reliability of the entire network. That is, each message must pass through the root node in order to be received by all participants.

It would be desirable to have a reliable communications network that is suitable for the simultaneous sharing of information among a large number of the processes that are widely distributed.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 illustrates a graph that is 4-regular and 4-connected which represents a broadcast channel.

Figure 2 illustrates a graph representing 20 computers connected to a broadcast channel.

Figures 3A and 3B illustrate the process of connecting a new computer Z to the broadcast channel.

Figure 4A illustrates the broadcast channel of Figure 1 with an added computer.

Figure 4B illustrates the broadcast channel of Figure 4A with an added computer.

Figure 4C also illustrates the broadcast channel of Figure 4A with an added computer.

Figure 5A illustrates the disconnecting of a computer from the broadcast channel in a planned manner.

Figure 5B illustrates the disconnecting of a computer from the broadcast channel in an unplanned manner.

Figure 5C illustrates the neighbors with empty ports condition.

Figure 5D illustrates two computers that are not neighbors who now have empty ports.

Figure 5E illustrates the neighbors with empty ports condition in the small regime.

Figure 5F illustrates the situation of Figure 5E when in the large regime.

Figure 6 is a block diagram illustrating components of a computer that is connected to a broadcast channel.

Figure 7 is a block diagram illustrating the sub-components of the broadcaster component in one embodiment.

Figure 8 is a flow diagram illustrating the processing of the connect routine in one embodiment.

Figure 9 is a flow diagram illustrating the processing of the seek portal computer routine in one embodiment.

Figure 10 is a flow diagram illustrating the processing of the contact process routine in one embodiment.

Figure 11 is a flow diagram illustrating the processing of the connect request routine in one embodiment.

Figure 12 is a flow diagram of the processing of the check for external call routine in one embodiment.

Figure 13 is a flow diagram of the processing of the achieve connection routine in one embodiment.

Figure 14 is a flow diagram illustrating the processing of the external dispatcher routine in one embodiment.

Figure 15 is a flow diagram illustrating the processing of the handle seeking connection call routine in one embodiment.

Figure 16 is a flow diagram illustrating processing of the handle connection request call routine in one embodiment.

Figure 17 is a flow diagram illustrating the processing of the add neighbor routine in one embodiment.

Figure 18 is a flow diagram illustrating the processing of the forward connection edge search routine in one embodiment.

Figure 19 is a flow diagram illustrating the processing of the handle edge proposal call routine.

Figure 20 is a flow diagram illustrating the processing of the handle port connection call routine in one embodiment.

Figure 21 is a flow diagram illustrating the processing of the fill hole routine in one embodiment.

Figure 22 is a flow diagram illustrating the processing of the internal dispatcher routine in one embodiment.

Figure 23 is a flow diagram illustrating the processing of the handle broadcast message routine in one embodiment.

Figure 24 is a flow diagram illustrating the processing of the distribute broadcast message routine in one embodiment.

Figure 26 is a flow diagram illustrating the processing of the handle connection port search statement routine in one embodiment.

Figure 27 is a flow diagram illustrating the processing of the court neighbor routine in one embodiment.

Figure 28 is a flow diagram illustrating the processing of the handle connection edge search call routine in one embodiment.

Figure 29 is a flow diagram illustrating the processing of the handle connection edge search response routine in one embodiment.

Figure 30 is a flow diagram illustrating the processing of the broadcast routine in one embodiment.

Figure 31 is a flow diagram illustrating the processing of the acquire message routine in one embodiment.

Figure 32 is a flow diagram illustrating processing of the handle condition check message in one embodiment.

Figure 33 is a flow diagram illustrating processing of the handle condition repair statement routine in one embodiment.

Figure 34 is a flow diagram illustrating the processing of the handle condition double check routine.

DETAILED DESCRIPTION

A broadcast technique in which a broadcast channel overlays a point-to-point communications network is provided. The broadcasting of a message over the broadcast channel is effectively a multicast to those computers of the network that are currently connected to the broadcast channel. In one embodiment, the broadcast technique provides a logical broadcast channel to which host computers through their executing processes can be connected. Each computer that is connected to the broadcast channel can broadcast messages onto and receive messages off of the broadcast channel. Each computer that is connected to the broadcast channel receives all messages that are broadcast while it is connected. The logical broadcast channel is implemented using an underlying network system (*e.g.*, the Internet) that allows each computer connected to the underlying network system to send messages to each other connected computer using each computer's address. Thus, the broadcast technique effectively provides a broadcast channel using an underlying network system that sends messages on a point-to-point basis.

The broadcast technique overlays the underlying network system with a graph of point-to-point connections (*i.e.*, edges) between host computers (*i.e.*, nodes) through which the broadcast channel is implemented. In one embodiment, each computer is connected to four other computers, referred to as neighbors. (Actually, a process executing on a computer is connected to four other processes executing on this or four other computers.) To broadcast a message, the originating computer sends the message to each of its neighbors using its point-to-point connections. Each computer that receives the message then sends the message to its three other neighbors using the point-to-point connections. In this way, the message is propagated to each computer using the underlying network to effect the broadcasting of the message to each computer over a logical broadcast channel. A graph in which each node is connected to four other nodes is referred to as a 4-regular graph. The use of a 4-regular graph means that a computer would become disconnected from the broadcast channel only if all four of the connections to its neighbors fail. The graph used by the broadcast technique also has the property that it would take a failure of four computers to

divide the graph into disjoint sub-graphs, that is two separate broadcast channels. This property is referred to as being 4-connected. Thus, the graph is both 4-regular and 4-connected.

Figure 1 illustrates a graph that is 4-regular and 4-connected which represents the broadcast channel. Each of the nine nodes A-I represents a computer that is connected to the broadcast channel, and each of the edges represents an "edge" connection between two computers of the broadcast channel. The time it takes to broadcast a message to each computer on the broadcast channel depends on the speed of the connections between the computers and the number of connections between the originating computer and each other computer on the broadcast channel. The minimum number of connections that a message would need to traverse between each pair of computers is the "distance" between the computers (*i.e.*, the shortest path between the two nodes of the graph). For example, the distance between computers A and F is one because computer A is directly connected to computer F. The distance between computers A and B is two because there is no direct connection between computers A and B, but computer F is directly connected to computer B. Thus, a message originating at computer A would be sent directly to computer F, and then sent from computer F to computer B. The maximum of the distances between the computers is the "diameter" of broadcast channel. The diameter of the broadcast channel represented by Figure 1 is two. That is, a message sent by any computer would traverse no more than two connections to reach every other computer. Figure 2 illustrates a graph representing 20 computers connected to a broadcast channel. The diameter of this broadcast channel is 4. In particular, the shortest path between computers 1 and 3 contains four connections (1-12, 12-15, 15-18, and 18-3).

The broadcast technique includes (1) the connecting of computers to the broadcast channel (*i.e.*, composing the graph), (2) the broadcasting of messages over the broadcast channel (*i.e.*, broadcasting through the graph), and (3) the disconnecting of computers from the broadcast channel (*i.e.*, decomposing the graph) composing the graph.

<u>Composing the Graph</u>

To connect to the broadcast channel, the computer seeking the connection first locates a computer that is currently fully connected to the broadcast channel and then

establishes a connection with four of the computers that are already connected to the broadcast channel. (This assumes that there are at least four computers already connected to the broadcast channel. When there are fewer than five computers connected, the broadcast channel cannot be a 4-regular graph. In such a case, the broadcast channel is considered to be in a "small regime." The broadcast technique for the small regime is described below in detail. When five or more computers are connected, the broadcast channel is considered to be in the "large regime." This description assumes that the broadcast channel is in the large regime, unless specified otherwise.) Thus, the process of connecting to the broadcast channel includes locating the broadcast channel, identifying the neighbors for the connecting computer, and then connecting to each identified neighbor. Each computer is aware of one or more "portal computers" through which that computer may locate the broadcast channel. A seeking computer locates the broadcast channel by contacting the portal computers until it finds one that is currently fully connected to the broadcast channel. The found portal computer then directs the identifying of four computers (*i.e.*, to be the seeking computer's neighbors) to which the seeking computer is to connect. Each of these four computers then cooperates with the seeking computer to effect the connecting of the seeking computer to the broadcast channel. A computer that has started the process of locating a portal computer, but does not yet have a neighbor, is in the "seeking connection state." A computer that is connected to at least one neighbor, but not yet four neighbors, is in the "partially connected state." A computer that is currently, or has been, previously connected to four neighbors is in the "fully connected state."

Since the broadcast channel is a 4-regular graph, each of the identified computers is already connected to four computers. Thus, some connections between computers need to be broken so that the seeking computer can connect to four computers. In one embodiment, the broadcast technique identifies two pairs of computers that are currently connected to each other. Each of these pairs of computers breaks the connection between them, and then each of the four computers (two from each pair) connects to the seeking computer. Figures 3A and 3B illustrate the process of a new computer Z connecting to the broadcast channel. Figure 3A illustrates the broadcast channel before computer Z is connected. The pairs of computers B and E and computers C and D are the two pairs that are identified as the neighbors for the new computer Z. The connections between each of these pairs is broken, and a connection between computer Z and each of computers B, C, D, and E

is established as indicated by Figure 3B. The process of breaking the connection between two neighbors and reconnecting each of the former neighbors to another computer is referred to as "edge pinning" as the edge between two nodes may be considered to be stretched and pinned to a new node.

Each computer connected to the broadcast channel allocates five communications ports for communicating with other computers. Four of the ports are referred to as "internal" ports because they are the ports through which the messages of the broadcast channels are sent. The connections between internal ports of neighbors are referred to as "internal" connections. Thus, the internal connections of the broadcast channel form the 4-regular and 4-connected graph. The fifth port is referred to as an "external" port because it is used for sending non-broadcast messages between two computers. Neighbors can send non-broadcast messages either through their internal ports of their connection or through their external ports. A seeking computer uses external ports when locating a portal computer.

In one embodiment, the broadcast technique establishes the computer connections using the TCP/IP communications protocol, which is a point-to-point protocol, as the underlying network. The TCP/IP protocol provides for reliable and ordered delivery of messages between computers. The TCP/IP protocol provides each computer with a "port space" that is shared among all the processes that may execute on that computer. The ports are identified by numbers from 0 to 65,535. The first 2056 ports are reserved for specific applications (*e.g.*, port 80 for HTTP messages). The remainder of the ports are user ports that are available to any process. In one embodiment, a set of port numbers can be reserved for use by the computer connected to the broadcast channel. In an alternative embodiment, the port numbers used are dynamically identified by each computer. Each computer dynamically identifies an available port to be used as its call-in port. This call-in port is used to establish connections with the external port and the internal ports. Each computer that is connected to the broadcast channel can receive non-broadcast messages through its external port. A seeking computer tries "dialing" the port numbers of the portal computers until a portal computer "answers," a call on its call-in port. A portal computer answers when it is connected to or attempting to connect to the broadcast channel and its call-in port is dialed. (In this description, a telephone metaphor is used to describe the connections.) When a computer receives a call on its call-in port, it transfers the call to another port. Thus, the

seeking computer actually communicates through that transfer-to port, which is the external port. The call is transferred so that other computers can place calls to that computer via the call-in port. The seeking computer then communicates via that external port to request the portal computer to assist in connecting the seeking computer to the broadcast channel. The seeking computer could identify the call-in port number of a portal computer by successively dialing each port in port number order. As discussed below in detail, the broadcast technique uses a hashing algorithm to select the port number order, which may result in improved performance.

A seeking computer could connect to the broadcast channel by connecting to computers either directly connected to the found portal computer or directly connected to one of its neighbors. A possible problem with such a scheme for identifying the neighbors for the seeking computer is that the diameter of the broadcast channel may increase when each seeking computer uses the same found portal computer and establishes a connection to the broadcast channel directly through that found portal computer. Conceptually, the graph becomes elongated in the direction of where the new nodes are added. Figures 4A-4C illustrate that possible problem. Figure 4A illustrates the broadcast channel of Figure 1 with an added computer. Computer J was connected to the broadcast channel by edge pinning edges C-D and E-H to computer J. The diameter of this broadcast channel is still two. Figure 4B illustrates the broadcast channel of Figure 4A with an added computer. Computer K was connected to the broadcast channel by edge pinning edges E-J and B-C to computer K. The diameter of this broadcast channel is three, because the shortest path from computer G to computer K is through edges G-A, A-E, and E-K. Figure 4C also illustrates the broadcast channel of Figure 4A with an added computer. Computer K was connected to the broadcast channel by edge pinning edges D-G and E-J to computer K. The diameter of this broadcast channel is, however, still two. Thus, the selection of neighbors impacts the diameter of the broadcast channel. To help minimize the diameter, the broadcast technique uses a random selection technique to identify the four neighbors of a computer in the seeking connection state. The random selection technique tends to distribute the connections to new seeking computers throughout the computers of the broadcast channel which may result in smaller overall diameters.

Broadcasting Through the Graph

As described above, each computer that is connected to the broadcast channel can broadcast messages onto the broadcast channel and does receive all messages that are broadcast on the broadcast channel. The computer that originates a message to be broadcast sends that message to each of its four neighbors using the internal connections. When a computer receives a broadcast message from a neighbor, it sends the message to its three other neighbors. Each computer on the broadcast channel, except the originating computer, will thus receive a copy of each broadcast message from each of its four neighbors. Each computer, however, only sends the first copy of the message that it receives to its neighbors and disregards subsequently received copies. Thus, the total number of copies of a message that is sent between the computers is 3N+1, where N is the number of computers connected to the broadcast channel. Each computer sends three copies of the message, except for the originating computer, which sends four copies of the message.

The redundancy of the message sending helps to ensure the overall reliability of the broadcast channel. Since each computer has four connections to the broadcast channel, if one computer fails during the broadcast of a message, its neighbors have three other connections through which they will receive copies of the broadcast message. Also, if the internal connection between two computers is slow, each computer has three other connections through which it may receive a copy of each message sooner.

Each computer that originates a message numbers its own messages sequentially. Because of the dynamic nature of the broadcast channel and because there are many possible connection paths between computers, the messages may be received out of order. For example, the distance between an originating computer and a certain receiving computer may be four. After sending the first message, the originating computer and receiving computer may become neighbors and thus the distance between them changes to one. The first message may have to travel a distance of four to reach the receiving computer. The second message only has to travel a distance of one. Thus, it is possible for the second message to reach the receiving computer before the first message.

When the broadcast channel is in a steady state (*i.e.*, no computers connecting or disconnecting from the broadcast channel), out-of-order messages are not a problem because each computer will eventually receive both messages and can queue messages until all earlier ordered messages are received. If, however, the broadcast channel is not in a

steady state, then problems can occur. In particular, a computer may connect to the broadcast channel after the second message has already been received and forwarded on by its new neighbors. When a new neighbor eventually receives the first message, it sends the message to the newly connected computer. Thus, the newly connected computer will receive the first message, but will not receive the second message. If the newly connected computer needs to process the messages in order, it would wait indefinitely for the second message.

One solution to this problem is to have each computer queue all the messages that it receives until it can send them in their proper order to its neighbors. This solution, however, may tend to slow down the propagation of messages through the computers of the broadcast channel. Another solution that may have less impact on the propagation speed is to queue messages only at computers who are neighbors of the newly connected computers. Each already connected neighbor would forward messages as it receives them to its other neighbors who are not newly connected, but not to the newly connected neighbor. The already connected neighbor would only forward messages from each originating computer to the newly connected computer when it can ensure that no gaps in the messages from that originating computer will occur. In one embodiment, the already connected neighbor may track the highest sequence number of the messages already received and forwarded on from each originating computer. The already connected computer will send only higher numbered messages from the originating computers to the newly connected computer. Once all lower numbered messages have been received from all originating computers, then the already connected computer can treat the newly connected computer as its other neighbors and simply forward each message as it is received. In another embodiment, each computer may queue messages and only forwards to the newly connected computer those messages as the gaps are filled in. For example, a computer might receive messages 4 and 5 and then receive message 3. In such a case, the already connected computer would forward queue messages 4 and 5. When message 3 is finally received, the already connected computer will send messages 3, 4, and 5 to the newly connected computer. If messages 4 and 5 were sent to the newly connected computer before message 3, then the newly connected computer would process messages 4 and 5 and disregard message 3. Because the already connected computer queues messages 4 and 5, the newly connected computer will be able to process message 3. It is possible that a newly connected computer will receive a set of messages from an originating computer through one neighbor and then receive another set of message from the

same originating computer through another neighbor. If the second set of messages contains a message that is ordered earlier than the messages of the first set received, then the newly connected computer may ignore that earlier ordered message if the computer already processed those later ordered messages.

Decomposing the Graph

A connected computer disconnects from the broadcast channel either in a planned or unplanned manner. When a computer disconnects in a planned manner, it sends a disconnect message to each of its four neighbors. The disconnect message includes a list that identifies the four neighbors of the disconnecting computer. When a neighbor receives the disconnect message, it tries to connect to one of the computers on the list. In one embodiment, the first computer in the list will try to connect to the second computer in the list, and the third computer in the list will try to connect to the fourth computer in the list. If a computer cannot connect (*e.g.*, the first and second computers are already connected), then the computers may try connecting in various other combinations. If connections cannot be established, each computer broadcasts a message that it needs to establish a connection with another computer. When a computer with an available internal port receives the message, it can then establish a connection with the computer that broadcast the message. Figures 5A-5D illustrate the disconnecting of a computer from the broadcast channel. Figure 5A illustrates the disconnecting of a computer from the broadcast channel in a planned manner. When computer H decides to disconnect, it sends its list of neighbors to each of its neighbors (computers A, E, F and I) and then disconnects from each of its neighbors. When computers A and I receive the message they establish a connection between them as indicated by the dashed line, and similarly for computers E and F.

When a computer disconnects in an unplanned manner, such as resulting from a power failure, the neighbors connected to the disconnected computer recognize the disconnection when each attempts to send its next message to the now disconnected computer. Each former neighbor of the disconnected computer recognizes that it is short one connection (*i.e.*, it has a hole or empty port). When a connected computer detects that one of its neighbors is now disconnected, it broadcasts a port connection request on the broadcast channel, which indicates that it has one internal port that needs a connection. The port connection request identifies the call-in port of the requesting computer. When a connected

computer that is also short a connection receives the connection request, it communicates with the requesting computer through its external port to establish a connection between the two computers. Figure 5B illustrates the disconnecting of a computer from the broadcast channel in an unplanned manner. In this illustration, computer H has disconnected in an unplanned manner. When each of its neighbors, computers A, E, F, and I, recognizes the disconnection, each neighbor broadcasts a port connection request indicating that it needs to fill an empty port. As shown by the dashed lines, computers F and I and computers A and E respond to each other's requests and establish a connection.

It is possible that a planned or unplanned disconnection may result in two neighbors each having an empty internal port. In such a case, since they are neighbors, they are already connected and cannot fill their empty ports by connecting to each other. Such a condition is referred to as the "neighbors with empty ports" condition. Each neighbor broadcasts a port connection request when it detects that it has an empty port as described above. When a neighbor receives the port connection request from the other neighbor, it will recognize the condition that its neighbor also has an empty port. Such a condition may also occur when the broadcast channel is in the small regime. The condition can only be corrected when in the large regime. When in the small regime, each computer will have less than four neighbors. To detect this condition in the large regime, which would be a problem if not repaired, the first neighbor to receive the port connection request recognizes the condition and sends a condition check message to the other neighbor. The condition check message includes a list of the neighbors of the sending computer. When the receiving computer receives the list, it compares the list to its own list of neighbors. If the lists are different, then this condition has occurred in the large regime and repair is needed. To repair this condition, the receiving computer will send a condition repair request to one of the neighbors of the sending computer which is not already a neighbor of the receiving computer. When the computer receives the condition repair request, it disconnects from one of its neighbors (other than the neighbor that is involved with the condition) and connects to the computer that sent the condition repair request. Thus, one of the original neighbors involved in the condition will have had a port filled. However, two computers are still in need of a connection, the other original neighbor and the computer that is now disconnected from the computer that received the condition repair request. Those two computers send out port connection requests. If those two computers are not neighbors, then they will connect to

each other when they receive the requests. If, however, the two computers are neighbors, then they repeat the condition repair process until two non-neighbors are in need of connections.

It is possible that the two original neighbors with the condition may have the same set of neighbors. When the neighbor that receives the condition check message determines that the sets of neighbors are the same, it sends a condition double check message to one of its neighbors other than the neighbor who also has the condition. When the computer receives the condition double check message, it determines whether it has the same set of neighbors as the sending computer. If so, the broadcast channel is in the small regime and the condition is not a problem. If the set of neighbors are different, then the computer that received the condition double check message sends a condition check message to the original neighbors with the condition. The computer that receives that condition check message directs one of it neighbors to connect to one of the original neighbors with the condition by sending a condition repair message. Thus, one of the original neighbors with the condition will have its port filled.

Figure 5C illustrates the neighbors with empty ports condition. In this illustration, computer H disconnected in an unplanned manner, but computers F and I responded to the port connection request of the other and are now connected together. The other former neighbors of computer H, computers A and E, are already neighbors, which gives rise to the neighbors with empty ports condition. In this example, computer E received the port connection request from computer A, recognized the possible condition, and sent (since they are neighbors via the internal connection) a condition check message with a list of its neighbors to computer A. When computer A received the list, it recognized that computer E has a different set of neighbor (*i.e.*, the broadcast channel is in the large regime). Computer A selected computer D, which is a neighbor of computer E and sent it a condition repair request. When computer D received the condition repair request, it disconnected from one of its neighbors (other than computer E), which is computer G in this example. Computer D then connected to computer A. Figure 5D illustrates two computers that are not neighbors who now have empty ports. Computers E and G now have empty ports and are not currently neighbors. Therefore, computers E and G can connect to each other.

Figures 5E and 5F further illustrate the neighbors with empty ports condition. Figure 5E illustrates the neighbors with empty ports condition in the small regime. In this

example, if computer E disconnected in an unplanned manner, then each computer broadcasts a port connection request when it detects the disconnect. When computer A receives the port connection request form computer B, it detects the neighbors with empty ports condition and sends a condition check message to computer B. Computer B recognizes that it has the same set of neighbors (computer C and D) as computer A and then sends a condition double check message to computer C. Computer C recognizes that the broadcast channel is in the small regime because is also has the same set of neighbors as computers A and B, computer C may then broadcast a message indicating that the broadcast channel is in the small regime.

Figure 5F illustrates the situation of Figure 5E when in the large regime. As discussed above, computer C receives the condition double check message from computer B. In this case, computer C recognizes that the broadcast channel is in the large regime because it has a set of neighbors that is different from computer B. The edges extending up from computer C and D indicate connections to other computers. Computer C then sends a condition check message to computer B. When computer B receives the condition check message, it sends a condition repair message to one of the neighbors of computer C. The computer that receives the condition repair message disconnects from one of its neighbors, other than computer C, and tries to connect to computer B and the neighbor from which it disconnected tries to connect to computer A.

Port Selection

As described above, the TCP/IP protocol designates ports above number 2056 as user ports. The broadcast technique uses five user port numbers on each computer: one external port and four internal ports. Generally, user ports cannot be statically allocated to an application program because other applications programs executing on the same computer may use conflicting port numbers. As a result, in one embodiment, the computers connected to the broadcast channel dynamically allocate their port numbers. Each computer could simply try to locate the lowest number unused port on that computer and use that port as the call-in port. A seeking computer, however, does not know in advance the call-in port number of the portal computers when the port numbers are dynamically allocated. Thus, a seeking computer needs to dial ports of a portal computer starting with the lowest port number when locating the call-in port of a portal computer. If the portal computer is

connected to (or attempting to connect to) the broadcast channel, then the seeking computer would eventually find the call-in port. If the portal computer is not connected, then the seeking computer would eventually dial every user port. In addition, if each application program on a computer tried to allocate low-ordered port numbers, then a portal computer may end up with a high-numbered port for its call-in port because many of the low-ordered port numbers would be used by other application programs. Since the dialing of a port is a relatively slow process, it would take the seeking computer a long time to locate the call-in port of a portal computer. To minimize this time, the broadcast technique uses a port ordering algorithm to identify the port number order that a portal computer should use when finding an available port for its call-in port. In one embodiment, the broadcast technique uses a hashing algorithm to identify the port order. The algorithm preferably distributes the ordering of the port numbers randomly through out the user port number space and only selects each port number once. In addition, every time the algorithm is executed on any computer for a given channel type and channel instance, it generates the same port ordering. As described below, it is possible for a computer to be connected to multiple broadcast channels that are uniquely identified by channel type and channel instance. The algorithm may be "seeded" with channel type and channel instance in order to generate a unique ordering of port numbers for each broadcast channel. Thus, a seeking computer will dial the ports of a portal computer in the same order as the portal computer used when allocating its call-in port.

If many computers are at the same time seeking connection to a broadcast channel through a single portal computer, then the ports of the portal computer may be busy when called by seeking computers. The seeking computers would typically need to keep on redialing a busy port. The process of locating a call-in port may be significantly slowed by such redialing. In one embodiment, each seeking computer may each reorder the first few port numbers generated by the hashing algorithm. For example, each seeking computer could randomly reorder the first eight port numbers generated by the hashing algorithm. The random ordering could also be weighted where the first port number generated by the hashing algorithm would have a 50% chance of being first in the reordering, the second port number would have a 25% chance of being first in the reordering, and so on. Because the seeking computers would use different orderings, the likelihood of finding a busy port is reduced. For example, if the first eight port numbers are randomly selected, then it is

possible that eight seeking computers could be simultaneously dialing ports in different sequences which would reduce the chances of dialing a busy port.

Locating a Portal Computer

Each computer that can connect to the broadcast channel has a list of one or more portal computers through which it can connect to the broadcast channel. In one embodiment, each computer has the same set of portal computers. A seeking computer locates a portal computer that is connected to the broadcast channel by successively dialing the ports of each portal computer in the order specified by an algorithm. A seeking computer could select the first portal computer and then dial all its ports until a call-in port of a computer that is fully connected to the broadcast channel is found. If no call-in port is found, then the seeking computer would select the next portal computer and repeat the process until a portal computer with such a call-in port is found. A problem with such a seeking technique is that all user ports of each portal computer are dialed until a portal computer fully connected to the broadcast channel is found. In an alternate embodiment, the seeking computer selects a port number according to the algorithm and then dials each portal computer at that port number. If no acceptable call-in port to the broadcast channel is found, then the seeking computer selects the next port number and repeats the process. Since the call-in ports are likely allocated at lower-ordered port numbers, the seeking computer first dials the port numbers that are most likely to be call-in ports of the broadcast channel. The seeking computers may have a maximum search depth, that is the number of ports that it will dial when seeking a portal computer that is fully connected. If the seeking computer exhausts its search depth, then either the broadcast channel has not yet been established or, if the seeking computer is also a portal computer, it can then establish the broadcast channel with itself as the first fully connected computer.

When a seeking computer locates a portal computer that is itself not fully connected, the two computers do not connect when they first locate each other because the broadcast channel may already be established and accessible through a higher-ordered port number on another portal computer. If the two seeking computers were to connect to each other, then two disjoint broadcast channels would be formed. Each seeking computer can share its experience in trying to locate a portal computer with the other seeking computer. In particular, if one seeking computer has searched all the portal computers to a depth of eight,

then the one seeking computer can share that it has searched to a depth of eight with another seeking computer. If that other seeking computer has searched to a depth of, for example, only four, it can skip searching through depths five through eight and that other seeking computer can advance its searching to a depth of nine.

In one embodiment, each computer may have a different set of portal computers and a different maximum search depth. In such a situation, it may be possible that two disjoint broadcast channels are formed because a seeking computer cannot locate a fully connected port computer at a higher depth. Similarly, if the set of portal computers are disjoint, then two separate broadcast channels would be formed.

Identifying Neighbors for a Seeking Computer

As described above, the neighbors of a newly connecting computer are preferably selected randomly from the set of currently connected computers. One advantage of the broadcast channel, however, is that no computer has global knowledge of the broadcast channel. Rather, each computer has local knowledge of itself and its neighbors. This limited local knowledge has the advantage that all the connected computers are peers (as far as the broadcasting is concerned) and the failure of any one computer (actually any three computers when in the 4-regular and 4-connect form) will not cause the broadcast channel to fail. This local knowledge makes it difficult for a portal computer to randomly select four neighbors for a seeking computer.

To select the four computers, a portal computer sends an edge connection request message through one of its internal connections that is randomly selected. The receiving computer again sends the edge connection request message through one of its internal connections that is randomly selected. This sending of the message corresponds to a random walk through the graph that represents the broadcast channel. Eventually, a receiving computer will decide that the message has traveled far enough to represent a randomly selected computer. That receiving computer will offer the internal connection upon which it received the edge connection request message to the seeking computer for edge pinning. Of course, if either of the computers at the end of the offered internal connection are already neighbors of the seeking computer, then the seeking computer cannot connect through that internal connection. The computer that decided that the message has

traveled far enough will detect this condition of already being a neighbor and send the message to a randomly selected neighbor.

In one embodiment, the distance that the edge connection request message travels is established by the portal computer to be approximately twice the estimated diameter of the broadcast channel. The message includes an indication of the distance that it is to travel. Each receiving computer decrements that distance to travel before sending the message on. The computer that receives a message with a distance to travel that is zero is considered to be the randomly selected computer. If that randomly selected computer cannot connect to the seeking computer (*e.g.*, because it is already connected to it), then that randomly selected computer forwards the edge connection request to one of its neighbors with a new distance to travel. In one embodiment, the forwarding computer toggles the new distance to travel between zero and one to help prevent two computers from sending the message back and forth between each other.

Because of the local nature of the information maintained by each computer connected to the broadcast channel, the computers need not generally be aware of the diameter of the broadcast channel. In one embodiment, each message sent through the broadcast channel has a distance traveled field. Each computer that forwards a message increments the distance traveled field. Each computer also maintains an estimated diameter of the broadcast channel. When a computer receives a message that has traveled a distance that indicates that the estimated diameter is too small, it updates its estimated diameter and broadcasts an estimated diameter message. When a computer receives an estimated diameter message that indicates a diameter that is larger than its own estimated diameter, it updates its own estimated diameter. This estimated diameter is used to establish the distance that an edge connection request message should travel.

<u>External Data Representation</u>

The computers connected to the broadcast channel may internally store their data in different formats. For example, one computer may use 32-bit integers, and another computer may use 64-bit integers. As another example, one computer may use ASCII to represent text and another computer may use Unicode. To allow communications between heterogeneous computers, the messages sent over the broadcast channel may use the XDR ("eXternal Data Representation") format.

The underlying peer-to-peer communications protocol may send multiple messages in a single message stream. The traditional technique for retrieving messages from a stream has been to repeatedly invoke an operating system routine to retrieve the next message in the stream. The retrieval of each message may require two calls to the operating system: one to retrieve the size of the next message and the other to retrieve the number of bytes indicated by the retrieved size. Such calls to the operating system can, however, be very slow in comparison to the invocations of local routines. To overcome the inefficiencies of such repeated calls, the broadcast technique in one embodiment, uses XDR to identify the message boundaries in a stream of messages. The broadcast technique may request the operating system to provide the next, for example, 1,024 bytes from the stream. The broadcast technique can then repeatedly invoke the XDR routines to retrieve the messages and use the success or failure of each invocation to determine whether another block of 1,024 bytes needs to be retrieved from the operating system. The invocation of XDR routines do not involve system calls and are thus more efficient than repeated system calls.

<u>M-Regular</u>

In the embodiment described above, each fully connected computer has four internal connections. The broadcast technique can be used with other numbers of internal connections. For example, each computer could have 6, 8, or any even number of internal connections. As the number of internal connections increase, the diameter of the broadcast channel tends to decrease, and thus propagation time for a message tends to decrease. The time that it takes to connect a seeking computer to the broadcast channel may, however, increase as the number of internal connections increases. When the number of internal connectors is even, then the broadcast channel can be maintained as m-regular and m-connected (in the steady state). If the number of internal connections is odd, then when the broadcast channel has an odd number of computers connected, one of the computers will have less than that odd number of internal connections. In such a situation, the broadcast network is neither m-regular nor m-connected. When the next computer connects to the broadcast channel, it can again become m-regular and m-connected. Thus, with an odd number of internal connections, the broadcast channel toggles between being and not being m-regular and m-connected.

Components

Figure 6 is a block diagram illustrating components of a computer that is connected to a broadcast channel. The above description generally assumed that there was only one broadcast channel and that each computer had only one connection to that broadcast channel. More generally, a network of computers may have multiple broadcast channels, each computer may be connected to more than one broadcast channel, and each computer can have multiple connections to the same broadcast channel. The broadcast channel is well suited for computer processes (*e.g.*, application programs) that execute collaboratively, such as network meeting programs. Each computer process can connect to one or more broadcast channels. The broadcast channels can be identified by channel type (*e.g.*, application program name) and channel instance that represents separate broadcast channels for that channel type. When a process attempts to connect to a broadcast channel, it seeks a process currently connected to that broadcast channel that is executing on a portal computer. The seeking process identifies the broadcast channel by channel type and channel instance.

Computer 600 includes multiple application programs 601 executing as separate processes. Each application program interfaces with a broadcaster component 602 for each broadcast channel to which it is connected. The broadcaster component may be implement as an object that is instantiated within the process space of the application program. Alternatively, the broadcaster component may execute as a separate process or thread from the application program. In one embodiment, the broadcaster component provides functions (*e.g.*, methods of class) that can be invoked by the application programs. The primary functions provided may include a connect function that an application program invokes passing an indication of the broadcast channel to which the application program wants to connect. The application program may provide a callback routine that the broadcaster component invokes to notify the application program that the connection has been completed, that is the process enters the fully connected state. The broadcaster component may also provide an acquire message function that the application program can invoke to retrieve the next message that is broadcast on the broadcast channel. Alternatively, the application program may provide a callback routine (which may be a virtual function provided by the application program) that the broadcaster component invokes to notify the application program that a broadcast message has been received. Each broadcaster component allocates a call-in port using the hashing algorithm. When calls are answered at the call-in port, they are transferred to other ports that serve as the external and internal ports.

The computers connecting to the broadcast channel may include a central processing unit, memory, input devices (*e.g.*, keyboard and pointing device), output devices (*e.g.*, display devices), and storage devices (*e.g.*, disk drives). The memory and storage devices are computer-readable medium that may contain computer instructions that implement the broadcaster component. In addition, the data structures and message structures may be stored or transmitted via a signal transmitted on a computer-readable media, such as a communications link.

Figure 7 is a block diagram illustrating the sub-components of the broadcaster component in one embodiment. The broadcaster component includes a connect component 701, an external dispatcher 702, an internal dispatcher 703 for each internal connection, an acquire message component 704 and a broadcast component 712. The application program may provide a connect callback component 710 and a receive response component 711 that are invoked by the broadcaster component. The application program invokes the connect component to establish a connection to a designated broadcast channel. The connect component identifies the external port and installs the external dispatcher for handling messages that are received on the external port. The connect component invokes the seek portal computer component 705 to identify a portal computer that is connected to the broadcast channel and invokes the connect request component 706 to ask the portal computer (if fully connected) to select neighbor processes for the newly connecting process. The external dispatcher receives external messages, identifies the type of message, and invokes the appropriate handling routine 707. The internal dispatcher receives the internal messages, identifies the type of message, and invokes the appropriate handling routine 708. The received broadcast messages are stored in the broadcast message queue 709. The acquire message component is invoked to retrieve messages from the broadcast queue. The broadcast component is invoked by the application program to broadcast messages in the broadcast channel.

The following tables list messages sent by the broadcaster components.

EXTERNAL MESSAGES

Message Type	Description
seeking_connection_call	Indicates that a seeking process would like to know whether the receiving process is fully connected to the broadcast channel
connection_request_call	Indicates that the sending process would like the receiving process to initiate a connection of the sending process to the broadcast channel
edge_proposal_call	Indicates that the sending process is proposing an edge through which the receiving process can connect to the broadcast channel (*i.e.*, edge pinning)
port_connection_call	Indicates that the sending process is proposing a port through which the receiving process can connect to the broadcast channel
connected_stmt	Indicates that the sending process is connected to the broadcast channel
condition_repair_stmt	Indicates that the receiving process should disconnect from one of its neighbors and connect to one of the processes involved in the neighbors with empty port condition

INTERNAL MESSAGES

Message Type	Description
broadcast_stmt	Indicates a message that is being broadcast through the broadcast channel for the application programs
connection_port_search_stmt	Indicates that the designated process is looking for a port through which it can connect to the broadcast channel
connection_edge_search_call	Indicates that the requesting process is looking for an edge through which it can connect to the broadcast channel
connection_edge_search_resp	Indicates whether the edge between this process and the sending neighbor has been accepted by the requesting party
diameter_estimate_stmt	Indicates an estimated diameter of the broadcast channel
diameter_reset_stmt	Indicates to reset the estimated diameter to indicated diameter
disconnect_stmt	Indicates that the sending neighbor is disconnecting from the broadcast channel
condition_check_stmt	Indicates that neighbors with empty port condition have

	been detected
condition_double_check_stmt	Indicates that the neighbors with empty ports have the same set of neighbors
shutdown_stmt	Indicates that the broadcast channel is being shutdown

Flow Diagrams

Figures 8-34 are flow diagrams illustrating the processing of the broadcaster component in one embodiment. Figure 8 is a flow diagram illustrating the processing of the connect routine in one embodiment. This routine is passed a channel type (*e.g.*, application name) and channel instance (*e.g.*, session identifier), that identifies the broadcast channel to which this process wants to connect. The routine is also passed auxiliary information that includes the list of portal computers and a connection callback routine. When the connection is established, the connection callback routine is invoked to notify the application program. When this process invokes this routine, it is in the seeking connection state. When a portal computer is located that is connected and this routine connects to at least one neighbor, this process enters the partially connected state, and when the process eventually connects to four neighbors, it enters the fully connected state. When in the small regime, a fully connected process may have less than four neighbors. In block 801, the routine opens the call-in port through which the process is to communicate with other processes when establishing external and internal connections. The port is selected as the first available port using the hashing algorithm described above. In block 802, the routine sets the connect time to the current time. The connect time is used to identify the instance of the process that is connected through this external port. One process may connect to a broadcast channel of a certain channel type and channel instance using one call-in port and then disconnects, and another process may then connect to that same broadcast channel using the same call-in port. Before the other process becomes fully connected, another process may try to communicate with it thinking it is the fully connected old process. In such a case, the connect time can be used to identify this situation. In block 803, the routine invokes the seek portal computer routine passing the channel type and channel instance. The seek portal computer routine attempts to locate a portal computer through which this process can connect to the broadcast channel for the passed type and instance. In decision block 804, if the seek portal computer routine is successful in locating a fully connected process on that portal computer, then the routine continues at block 805, else the routine returns an unsuccessful indication. In decision block 805, if no portal computer other than the portal computer on which the process is executing was located, then this is the first process to fully connect to broadcast channel and the routine continues at block 806, else the routine continues at block 808. In block 806, the routine invokes the achieve connection routine to change the state of this process to fully connected. In block 807, the routine installs the external dispatcher for processing messages received through this process' external port for the passed channel type and channel instance. When a message is received through that external port, the external dispatcher is invoked. The routine then returns. In block 808, the routine installs an external dispatcher. In block 809, the routine invokes the connect request routine to initiate the process of identifying neighbors for the seeking computer. The routine then returns.

Figure 9 is a flow diagram illustrating the processing of the seek portal computer routine in one embodiment. This routine is passed the channel type and channel instance of the broadcast channel to which this process wishes to connect. This routine, for each search depth (*e.g.*, port number), checks the portal computers at that search depth. If a portal computer is located at that search depth with a process that is fully connected to the broadcast channel, then the routine returns an indication of success. In blocks 902-911, the routine loops selecting each search depth until a process is located. In block 902, the routine selects the next search depth using a port number ordering algorithm. In decision block 903, if all the search depths have already been selected during this execution of the loop, that is for the currently selected depth, then the routine returns a failure indication, else the routine continues at block 904. In blocks 904-911, the routine loops selecting each portal computer and determining whether a process of that portal computer is connected to (or attempting to connect to) the broadcast channel with the passed channel type and channel instance. In block 904, the routine selects the next portal computer. In decision block 905, if all the portal computers have already been selected, then the routine loops to block 902 to select the next search depth, else the routine continues at block 906. In block 906, the routine dials the selected portal computer through the port represented by the search depth. In decision block 907, if the dialing was successful, then the routine continues at block 908, else the routine loops to block 904 to select the next portal computer. The dialing will be successful if the dialed port is the call-in port of the broadcast channel of the passed channel type and channel instance of a process executing on that portal computer. In block 908, the routine invokes a contact process routine, which contacts the answering process of the portal computer through the dialed port and determines whether that process is fully connected to the broadcast channel. In block 909, the routine hangs up on the selected portal computer. In decision block 910, if the answering process is fully connected to the broadcast channel, then the routine returns a success indicator, else the routine continues at block 911. In block 911, the routine invokes the check for external call routine to determine whether an external call has been made to this process as a portal computer and processes that call. The routine then loops to block 904 to select the next portal computer.

Figure 10 is a flow diagram illustrating the processing of the contact process routine in one embodiment. This routine determines whether the process of the selected portal computer that answered the call-in to the selected port is fully connected to the broadcast channel. In block 1001, the routine sends an external message (*i.e.*, seeking_connection_call) to the answering process indicating that a seeking process wants to know whether the answering process is fully connected to the broadcast channel. In block 1002, the routine receives the external response message from the answering process. In decision block 1003, if the external response message is successfully received (*i.e.*, seeking_connection_resp), then the routine continues at block 1004, else the routine returns. Wherever the broadcast component requests to receive an external message, it sets a time out period. If the external message is not received within that time out period, the broadcaster component checks its own call-in port to see if another process is calling it. In particular, the dialed process may be calling the dialing process, which may result in a deadlock situation. The broadcaster component may repeat the receive request several times. If the expected message is not received, then the broadcaster component handles the error as appropriate. In decision block 1004, if the answering process indicates in its response message that it is fully connected to the broadcast channel, then the routine continues at block 1005, else the routine continues at block 1006. In block 1005, the routine adds the selected portal computer to a list of connected portal computers and then returns. In block 1006, the routine adds the answering process to a list of fellow seeking processes and then returns.

Figure 11 is a flow diagram illustrating the processing of the connect request routine in one embodiment. This routine requests a process of a portal computer that was identified as being fully connected to the broadcast channel to initiate the connection of this process to the broadcast channel. In decision block 1101, if at least one process of a portal computer was located that is fully connected to the broadcast channel, then the routine continues at block 1103, else the routine continues at block 1102. A process of the portal computer may no longer be in the list if it recently disconnected from the broadcast channel. In one embodiment, a seeking computer may always search its entire search depth and find multiple portal computers through which it can connect to the broadcast channel. In block 1102, the routine restarts the process of connecting to the broadcast channel and returns. In block 1103, the routine dials the process of one of the found portal computers through the call-in port. In decision block 1104, if the dialing is successful, then the routine continues at block 1105, else the routine continues at block 1113. The dialing may be unsuccessful if, for example, the dialed process recently disconnected from the broadcast channel. In block 1105, the routine sends an external message to the dialed process requesting a connection to the broadcast channel (*i.e.*, connection_request_call). In block 1106, the routine receives the response message (*i.e.*, connection_request_resp). In decision block 1107, if the response message is successfully received, then the routine continues at block 1108, else the routine continues at block 1113. In block 1108, the routine sets the expected number of holes (*i.e.*, empty internal connections) for this process based on the received response. When in the large regime, the expected number of holes is zero. When in the small regime, the expected number of holes varies from one to three. In block 1109, the routine sets the estimated diameter of the broadcast channel based on the received response. In decision block 1111, if the dialed process is ready to connect to this process as indicated by the response message, then the routine continues at block 1112, else the routine continues at block 1113. In block 1112, the routine invokes the add neighbor routine to add the answering process as a neighbor to this process. This adding of the answering process typically occurs when the broadcast channel is in the small regime. When in the large regime, the random walk search for a neighbor is performed. In block 1113, the routine hangs up the external connection with the answering process computer and then returns.

Figure 12 is a flow diagram of the processing of the check for external call routine in one embodiment. This routine is invoked to identify whether a fellow seeking process is attempting to establish a connection to the broadcast channel through this process. In block 1201, the routine attempts to answer a call on the call-in port. In decision block 1202, if the answer is successful, then the routine continues at block 1203, else the routine returns. In block 1203, the routine receives the external message from the external port. In decision block 1204, if the type of the message indicates that a seeking process is calling (*i.e.*, seeking_connection_call), then the routine continues at block 1205, else the routine returns. In block 1205, the routine sends an external message (*i.e.*, seeking_connection_resp) to the other seeking process indicating that this process is also is seeking a connection. In decision block 1206, if the sending of the external message is successful, then the routine continues at block 1207, else the routine returns. In block 1207, the routine adds the other seeking process to a list of fellow seeking processes and then returns. This list may be used if this process can find no process that is fully connected to the broadcast channel. In which case, this process may check to see if any fellow seeking process were successful in connecting to the broadcast channel. For example, a fellow seeking process may become the first process fully connected to the broadcast channel.

Figure 13 is a flow diagram of the processing of the achieve connection routine in one embodiment. This routine sets the state of this process to fully connected to the broadcast channel and invokes a callback routine to notify the application program that the process is now fully connected to the requested broadcast channel. In block 1301, the routine sets the connection state of this process to fully connected. In block 1302, the routine notifies fellow seeking processes that it is fully connected by sending a connected external message to them (*i.e.*, connected_stmt). In block 1303, the routine invokes the connect callback routine to notify the application program and then returns.

Figure 14 is a flow diagram illustrating the processing of the external dispatcher routine in one embodiment. This routine is invoked when the external port receives a message. This routine retrieves the message, identifies the external message type, and invokes the appropriate routine to handle that message. This routine loops processing each message until all the received messages have been handled. In block 1401, the routine answers (*e.g.*, picks up) the external port and retrieves an external message. In decision block 1402, if a message was retrieved, then the routine continues at block 1403, else the routine hangs up on the external port in block 1415 and returns. In decision block 1403, if the message type is for a process seeking a connection (*i.e.*, seeking_connection_call), then the routine invokes the handle seeking connection call routine in block 1404, else the routine continues at block 1405. In decision block 1405, if the message type is for a connection request call (*i.e.*, connection_request_call), then the routine invokes the handle connection request call routine in block 1406, else the routine continues at block 1407. In decision block 1407, if the message type is edge proposal call (*i.e.*, edge_proposal_call), then the routine invokes the handle edge proposal call routine in block 1408, else the routine continues at block 1409. In decision block 1409, if the message type is port connect call (*i.e.*, port_connect_call), then the routine invokes the handle port connection call routine in block 1410, else the routine continues at block 1411. In decision block 1411, if the message type is a connected statement (*i.e.*, connected_stmt), the routine invokes the handle connected statement in block 1112, else the routine continues at block 1212. In decision block 1412, if the message type is a condition repair statement (*i.e.*, condition_repair_stmt), then the routine invokes the handle condition repair routine in block 1413, else the routine loops to block 1414 to process the next message. After each handling routine is invoked, the routine loops to block 1414. In block 1414, the routine hangs up on the external port and continues at block 1401 to receive the next message.

Figure 15 is a flow diagram illustrating the processing of the handle seeking connection call routine in one embodiment. This routine is invoked when a seeking process is calling to identify a portal computer through which it can connect to the broadcast channel. In decision block 1501, if this process is currently fully connected to the broadcast channel identified in the message, then the routine continues at block 1502, else the routine continues at block 1503. In block 1502, the routine sets a message to indicate that this process is fully connected to the broadcast channel and continues at block 1505. In block 1503, the routine sets a message to indicate that this process is not fully connected. In block 1504, the routine adds the identification of the seeking process to a list of fellow seeking processes. If this process is not fully connected, then it is attempting to connect to the broadcast channel. In block 1505, the routine sends the external message response (*i.e.*, seeking_connection_resp) to the seeking process and then returns.

Figure 16 is a flow diagram illustrating processing of the handle connection request call routine in one embodiment. This routine is invoked when the calling process wants this process to initiate the connection of the process to the broadcast channel. This routine either allows the calling process to establish an internal connection with this process (*e.g.*, if in the small regime) or starts the process of identifying a process to which the calling process can connect. In decision block 1601, if this process is currently fully connected to the broadcast channel, then the routine continues at block 1603, else the routine hangs up on the external port in block 1602 and returns. In block 1603, the routine sets the number of holes that the calling process should expect in the response message. In block 1604, the routine sets the estimated diameter in the response message. In block 1605, the routine indicates whether this process is ready to connect to the calling process. This process is ready to connect when the number of its holes is greater than zero and the calling process is not a neighbor of this process. In block 1606, the routine sends to the calling process an external message that is responsive to the connection request call (*i.e.*, connection_request_resp). In block 1607, the routine notes the number of holes that the calling process needs to fill as indicated in the request message. In decision block 1608, if this process is ready to connect to the calling process, then the routine continues at block 1609, else the routine continues at block 1611. In block 1609, the routine invokes the add neighbor routine to add the calling process as a neighbor. In block 1610, the routine decrements the number of holes that the calling process needs to fill and continues at block 1611. In block 1611, the routine hangs up on the external port. In decision block 1612, if this process has no holes or the estimated diameter is greater than one (*i.e.*, in the large regime), then the routine continues at block 1613, else the routine continues at block 1616. In blocks 1613-1615, the routine loops forwarding a request for an edge through which to connect to the calling process to the broadcast channel. One request is forwarded for each pair of holes of the calling process that needs to be filled. In decision block 1613, if the number of holes of the calling process to be filled is greater than or equal to two, then the routine continues at block 1614, else the routine continues at block 1616. In block 1614, the routine invokes the forward connection edge search routine. The invoked routine is passed to an indication of the calling process and the random walk distance. In one embodiment, the distance is twice in the estimated diameter of the broadcast channel. In block 1614, the routine decrements the holes left to fill by two and loops to block 1613. In decision block 1616, if there is still a hole to fill, then the routine continues at block 1617, else the routine returns. In block 1617, the routine invokes the fill hole routine passing the identification of the calling process. The fill hole routine broadcasts a connection port search statement (*i.e.*, connection_port_search_stmt) for a hole of a connected process through which the calling process can connect to the broadcast channel. The routine then returns.

Figure 17 is a flow diagram illustrating the processing of the add neighbor routine in one embodiment. This routine adds the process calling on the external port as a neighbor to this process. In block 1701, the routine identifies the calling process on the external port. In block 1702, the routine sets a flag to indicate that the neighbor has not yet received the broadcast messages from this process. This flag is used to ensure that there are no gaps in the messages initially sent to the new neighbor. The external port becomes the internal port for this connection. In decision block 1703, if this process is in the seeking connection state, then this process is connecting to its first neighbor and the routine continues at block 1704, else the routine continues at block 1705. In block 1704, the routine sets the connection state of this process to partially connected. In block 1705, the routine adds the calling process to the list of neighbors of this process. In block 1706, the routine installs an internal dispatcher for the new neighbor. The internal dispatcher is invoked when a message is received from that new neighbor through the internal port of that new neighbor. In decision block 1707, if this process buffered up messages while not fully connected, then the routine continues at block 1708, else the routine continues at block 1709. In one embodiment, a process that is partially connected may buffer the messages that it receives through an internal connection so that it can send these messages as it connects to new neighbors. In block 1708, the routine sends the buffered messages to the new neighbor through the internal port. In decision block 1709, if the number of holes of this process equals the expected number of holes, then this process is fully connected and the routine continues at block 1710, else the routine continues at block 1711. In block 1710, the routine invokes the achieve connected routine to indicate that this process is fully connected. In decision block 1711, if the number of holes for this process is zero, then the routine continues at block 1712, else the routine returns. In block 1712, the routine deletes any pending edges and then returns. A pending edge is an edge that has been proposed to this process for edge pinning, which in this case is no longer needed.

Figure 18 is a flow diagram illustrating the processing of the forward connection edge search routine in one embodiment. This routine is responsible for passing along a request to connect a requesting process to a randomly selected neighbor of this process through the internal port of the selected neighbor, that is part of the random walk. In decision block 1801, if the forwarding distance remaining is greater than zero, then the routine continues at block 1804, else the routine continues at block 1802. In decision block 1802, if the number of neighbors of this process is greater than one, then the routine continues at block 1804, else this broadcast channel is in the small regime and the routine continues at block 1803. In decision block 1803, if the requesting process is a neighbor of this process, then the routine returns, else the routine continues at block 1804. In blocks 1804-1807, the routine loops attempting to send a connection edge search call internal message (*i.e.*, connection_edge_search_call) to a randomly selected neighbor. In block 1804, the routine randomly selects a neighbor of this process. In decision block 1805, if all the neighbors of this process have already been selected, then the routine cannot forward the message and the routine returns, else the routine continues at block 1806. In block 1806, the routine sends a connection edge search call internal message to the selected neighbor. In decision block 1807, if the sending of the message is successful, then the routine continues at block 1808, else the routine loops to block 1804 to select the next neighbor. When the sending of an internal message is unsuccessful, then the neighbor may have disconnected from the broadcast channel in an unplanned manner. Whenever such a situation is detected by the broadcaster component, it attempts to find another neighbor by invoking the fill holes routine to fill a single hole or the forward connecting edge search routine to fill two holes. In block 1808, the routine notes that the recently sent connection edge search call has not yet been acknowledged and indicates that the edge to this neighbor is reserved if the remaining forwarding distance is less than or equal to one. It is reserved because the selected neighbor may offer this edge to the requesting process for edge pinning. The routine then returns.

Figure 19 is a flow diagram illustrating the processing of the handle edge proposal call routine. This routine is invoked when a message is received from a proposing process that proposes to connect an edge between the proposing process and one of its neighbors to this process for edge pinning. In decision block 1901, if the number of holes of this process minus the number of pending edges is greater than or equal to one, then this process still has holes to be filled and the routine continues at block 1902, else the routine continues at block 1911. In decision block 1902, if the proposing process or its neighbor is a neighbor of this process, then the routine continues at block 1911, else the routine continues at block 1903. In block 1903, the routine indicates that the edge is pending between this process and the proposing process. In decision block 1904, if a proposed neighbor is already pending as a proposed neighbor, then the routine continues at block 1911, else the routine continues at block 1907. In block 1907, the routine sends an edge proposal response as an external message to the proposing process (*i.e.*, edge_proposal_resp) indicating that the proposed edge is accepted. In decision block 1908, if the sending of the message was successful, then the routine continues at block 1909, else the routine returns. In block 1909, the routine adds the edge as a pending edge. In block 1910, the routine invokes the add neighbor routine to add the proposing process on the external port as a neighbor. The routine then returns. In block 1911, the routine sends an external message (*i.e.*, edge_proposal_resp) indicating that this proposed edge is not accepted. In decision block 1912, if the number of holes is odd, then the routine continues at block 1913, else the routine returns. In block 1913, the routine invokes the fill hole routine and then returns.

Figure 20 is a flow diagram illustrating the processing of the handle port connection call routine in one embodiment. This routine is invoked when an external message is received then indicates that the sending process wants to connect to one hole of this process. In decision block 2001, if the number of holes of this process is greater than zero, then the routine continues at block 2002, else the routine continues at block 2003. In decision block 2002, if the sending process is not a neighbor, then the routine continues at block 2004, else the routine continues to block 2003. In block 2003, the routine sends a port connection response external message (*i.e.*, port_connection_resp) to the sending process that indicates that it is not okay to connect to this process. The routine then returns. In block 2004, the routine sends a port connection response external message to the sending process that indicates that is okay to connect this process. In decision block 2005, if the sending of the message was successful, then the routine continues at block 2006, else the routine continues at block 2007. In block 2006, the routine invokes the add neighbor routine to add the sending process as a neighbor of this process and then returns. In block 2007, the routine hangs up the external connection. In block 2008, the routine invokes the connect request routine to request that a process connect to one of the holes of this process. The routine then returns.

Figure 21 is a flow diagram illustrating the processing of the fill hole routine in one embodiment. This routine is passed an indication of the requesting process. If this process is requesting to fill a hole, then this routine sends an internal message to other processes. If another process is requesting to fill a hole, then this routine invokes the routine to handle a connection port search request. In block 2101, the routine initializes a connection port search statement internal message (*i.e.*, connection_port_search_stmt). In decision block 2102, if this process is the requesting process, then the routine continues at block 2103, else the routine continues at block 2104. In block 2103, the routine distributes the message to the neighbors of this process through the internal ports and then returns. In block 2104, the routine invokes the handle connection port search routine and then returns.

Figure 22 is a flow diagram illustrating the processing of the internal dispatcher routine in one embodiment. This routine is passed an indication of the neighbor who sent the internal message. In block 2201, the routine receives the internal message. This routine identifies the message type and invokes the appropriate routine to handle the message. In block 2202, the routine assesses whether to change the estimated diameter of the broadcast channel based on the information in the received message. In decision block 2203, if this process is the originating process of the message or the message has already been received (*i.e.*, a duplicate), then the routine ignores the message and continues at block 2208, else the routine continues at block 2203A. In decision block 2203A, if the process is partially connected, then the routine continues at block 2203B, else the routine continues at block 2204. In block 2203B, the routine adds the message to the pending connection buffer and continues at block 2204. In decision blocks 2204-2207, the routine decodes the message type and invokes the appropriate routine to handle the message. For example, in decision block 2204, if the type of the message is broadcast statement (*i.e.*, broadcast_stmt), then the routine invokes the handle broadcast message routine in block 2205. After invoking the appropriate handling routine, the routine continues at block 2208. In decision block 2208, if the partially connected buffer is full, then the routine continues at block 2209, else the routine continues at block 2210. The broadcaster component collects all its internal messages in a buffer while partially connected so that it can forward the messages as it connects to new neighbors. If, however, that buffer becomes full, then the process assumes that it is now fully connected and that the expected number of connections was too high, because the broadcast channel is now in the small regime. In block 2209, the routine invokes the achieve connection routine and then continues in block 2210. In decision block 2210, if the application program message queue is empty, then the routine returns, else the routine continues at block 2212. In block 2212, the routine invokes the receive response routine passing the acquired message and then returns. The received response routine is a callback routine of the application program.

Figure 23 is a flow diagram illustrating the processing of the handle broadcast message routine in one embodiment. This routine is passed an indication of the originating process, an indication of the neighbor who sent the broadcast message, and the broadcast message itself. In block 2301, the routine performs the out of order processing for this message. The broadcaster component queues messages from each originating process until it can send them in sequence number order to the application program. In block 2302, the routine invokes the distribute broadcast message routine to forward the message to the neighbors of this process. In decision block 2303, if a newly connected neighbor is waiting to receive messages, then the routine continues at block 2304, else the routine returns. In block 2304, the routine sends the messages in the correct order if possible for each originating process and then returns.

Figure 24 is a flow diagram illustrating the processing of the distribute broadcast message routine in one embodiment. This routine sends the broadcast message to each of the neighbors of this process, except for the neighbor who sent the message to this process. In block 2401, the routine selects the next neighbor other than the neighbor who sent the message. In decision block 2402, if all such neighbors have already been selected, then the routine returns. In block 2403, the routine sends the message to the selected neighbor and then loops to block 2401 to select the next neighbor.

Figure 26 is a flow diagram illustrating the processing of the handle connection port search statement routine in one embodiment. This routine is passed an indication of the neighbor that sent the message and the message itself. In block 2601, the routine invokes the distribute internal message which sends the message to each of its neighbors other than the sending neighbor. In decision block 2602, if the number of holes of this process is greater than zero, then the routine continues at block 2603, else the routine returns. In decision block 2603, if the requesting process is a neighbor, then the routine continues at block 2605, else the routine continues at block 2604. In block 2604, the routine invokes the court neighbor routine and then returns. The court neighbor routine connects this process to the requesting process if possible. In block 2605, if this process has one hole, then the neighbors with empty ports condition exists and the routine continues at block 2606, else the routine returns. In block 2606, the routine generates a condition check message (*i.e.*, condition_check) that includes a list of this process' neighbors. In block 2607, the routine sends the message to the requesting neighbor.

Figure 27 is a flow diagram illustrating the processing of the court neighbor routine in one embodiment. This routine is passed an indication of the prospective neighbor for this process. If this process can connect to the prospective neighbor, then it sends a port connection call external message to the prospective neighbor and adds the prospective neighbor as a neighbor. In decision block 2701, if the prospective neighbor is already a neighbor, then the routine returns, else the routine continues at block 2702. In block 2702, the routine dials the prospective neighbor. In decision block 2703, if the number of holes of this process is greater than zero, then the routine continues at block 2704, else the routine continues at block 2706. In block 2704, the routine sends a port connection call external message (*i.e.*, port_connection_call) to the prospective neighbor and receives its response (*i.e.*, port_connection_resp). Assuming the response is successfully received, in block 2705, the routine adds the prospective neighbor as a neighbor of this process by invoking the add neighbor routine. In block 2706, the routine hangs up with the prospect and then returns.

Figure 28 is a flow diagram illustrating the processing of the handle connection edge search call routine in one embodiment. This routine is passed a indication of the neighbor who sent the message and the message itself. This routine either forwards the message to a neighbor or proposes the edge between this process and the sending neighbor to the requesting process for edge pinning. In decision block 2801, if this process is not the requesting process or the number of holes of the requesting process is still greater than or equal to two, then the routine continues at block 2802, else the routine continues at block 2813. In decision block 2802, if the forwarding distance is greater than zero, then the random walk is not complete and the routine continues at block 2803, else the routine continues at block 2804. In block 2803, the routine invokes the forward connection edge search routine passing the identification of the requesting process and the decremented forwarding distance. The routine then continues at block 2815. In decision block 2804, if the requesting process is a neighbor or the edge between this process and the sending neighbor is reserved because it has already been offered to a process, then the routine continues at block 2805, else the routine continues at block 2806. In block 2805, the routine invokes the forward connection edge search routine passing an indication of the requesting party and a toggle indicator that alternatively indicates to continue the random walk for one or two more computers. The routine then continues at block 2815. In block 2806, the routine dials the requesting process via the call-in port. In block 2807, the routine sends an edge proposal call external message (*i.e.*, edge_proposal_call) and receives the response (*i.e.*, edge_proposal_resp). Assuming that the response is successfully received, the routine continues at block 2808. In decision block 2808, if the response indicates that the edge is acceptable to the requesting process, then the routine continues at block 2809, else the routine continues at block 2812. In block 2809, the routine reserves the edge between this process and the sending neighbor. In block 2810, the routine adds the requesting process as a neighbor by invoking the add neighbor routine. In block 2811, the routine removes the sending neighbor as a neighbor. In block 2812, the routine hangs up the external port and continues at block 2815. In decision block 2813, if this process is the requesting process and the number of holes of this process equals one, then the routine continues at block 2814, else the routine continues at block 2815. In block 2814, the routine invokes the fill hole routine. In block 2815, the routine sends an connection edge search response message (*i.e.*, connection_edge_search_response) to the sending neighbor indicating acknowledgement and then returns. The graphs are sensitive to parity. That is, all possible paths starting from a node and ending at that node will have an even length unless the graph has a cycle whose length is odd. The broadcaster component uses a toggle indicator to vary the random walk distance between even and odd distances.

Figure 29 is a flow diagram illustrating the processing of the handle connection edge search response routine in one embodiment. This routine is passed as indication of the requesting process, the sending neighbor, and the message. In block 2901, the routine notes that the connection edge search response (*i.e.*, connection_edge_search_resp) has been received and if the forwarding distance is less than or equal to one unreserves the edge between this process and the sending neighbor. In decision block 2902, if the requesting process indicates that the edge is acceptable as indicated in the message, then the routine continues at block 2903, else the routine returns. In block 2903, the routine reserves the edge between this process and the sending neighbor. In block 2904, the routine removes the sending neighbor as a neighbor. In block 2905, the routine invokes the court neighbor routine to connect to the requesting process. In decision block 2906, if the invoked routine was unsuccessful, then the routine continues at block 2907, else the routine returns. In decision block 2907, if the number of holes of this process is greater than zero, then the routine continues at block 2908, else the routine returns. In block 2908, the routine invokes the fill hole routine and then returns.

Figure 30 is a flow diagram illustrating the processing of the broadcast routine in one embodiment. This routine is invoked by the application program to broadcast a message on the broadcast channel. This routine is passed the message to be broadcast. In decision block 3001, if this process has at least one neighbor, then the routine continues at block 3002, else the routine returns since it is the only process connected to be broadcast channel. In block 3002, the routine generates an internal message of the broadcast statement type (*i.e.*, broadcast _stmt). In block 3003, the routine sets the sequence number of the message. In block 3004, the routine invokes the distribute internal message routine to broadcast the message on the broadcast channel. The routine returns.

Figure 31 is a flow diagram illustrating the processing of the acquire message routine in one embodiment. The acquire message routine may be invoked by the application program or by a callback routine provided by the application program. This routine returns a message. In block 3101, the routine pops the message from the message queue of the broadcast channel. In decision block 3102, if a message was retrieved, then the routine returns an indication of success, else the routine returns indication of failure.

Figures 32-34 are flow diagrams illustrating the processing of messages associated with the neighbors with empty ports condition. Figure 32 is a flow diagram illustrating processing of the handle condition check message in one embodiment. This message is sent by a neighbor process that has one hole and has received a request to connect to a hole of this process. In decision block 3201, if the number of holes of this process is equal to one, then the routine continues at block 3202, else the neighbors with empty ports condition does not exist any more and the routine returns. In decision block 3202, if the sending neighbor and this process have the same set of neighbors, the routine continues at block 3203, else the routine continues at block 3205. In block 3203, the routine initializes a condition double check message (*i.e.*, condition_double_check) with the list of neighbors of this process. In block 3204, the routine sends the message internally to a neighbor other than sending neighbor. The routine then returns. In block 3205, the routine selects a neighbor of the sending process that is not also a neighbor of this process. In block 3206, the routine sends a condition repair message (*i.e.*, condition_repair_stmt) externally to the selected process. In block 3207, the routine invokes the add neighbor routine to add the selected neighbor as a neighbor of this process and then returns.

Figure 33 is a flow diagram illustrating processing of the handle condition repair statement routine in one embodiment. This routine removes an existing neighbor and connects to the process that sent the message. In decision block 3301, if this process has no holes, then the routine continues at block 3302, else the routine continues at block 3304. In block 3302, the routine selects a neighbor that is not involved in the neighbors with empty ports condition. In block 3303, the routine removes the selected neighbor as a neighbor of this process. Thus, this process that is executing the routine now has at least one hole. In block 3304, the routine invokes the add neighbor routine to add the process that sent the message as a neighbor of this process. The routine then returns.

Figure 34 is a flow diagram illustrating the processing of the handle condition double check routine. This routine determines whether the neighbors with empty ports condition really is a problem or whether the broadcast channel is in the small regime. In decision block 3401, if this process has one hole, then the routine continues at block 3402, else the routine continues at block 3403. If this process does not have one hole, then the set of neighbors of this process is not the same as the set of neighbors of the sending process. In decision block 3402, if this process and the sending process have the same set of neighbors, then the broadcast channel is not in the small regime and the routine continues at block 3403, else the routine continues at block 3406. In decision block 3403, if this process has no holes, then the routine returns, else the routine continues at block 3404. In block 3404, the routine sets the estimated diameter for this process to one. In block 3405, the routine broadcasts a diameter reset internal message (*i.e.*, diameter_reset) indicating that the estimated diameter is one and then returns. In block 3406, the routine creates a list of neighbors of this process. In block 3407, the routine sends the condition check message (*i.e.*, condition_check_stmt) with the list of neighbors to the neighbor who sent the condition double check message and then returns.

From the above description, it will be appreciated that although specific embodiments of the technology have been described, various modifications may be made without deviating from the spirit and scope of the invention. For example, the communications on the broadcast channel may be encrypted. Also, the channel instance or session identifier may be a very large number (*e.g.*, 128 bits) to help prevent an unauthorized user to maliciously tap into a broadcast channel. The portal computer may also enforce security and not allow an unauthorized user to connect to the broadcast channel. Accordingly, the invention is not limited except by the claims.

CLAIMS

1. A computer-based method for adding a participant to a network of participants, each participant being connected to three or more other participants, the method comprising:

identifying pair of participants of the network that are connected;

disconnecting the participants of the identified pair from each other; and

connecting each participant of the identified pair of participants to the added participant.

2. The method of claim 1 wherein each participant is connected to 4 participants.

3. The method of claim 1 wherein the identifying of a pair includes randomly selecting a pair of participants that are connected.

4. The method of claim 3 wherein the randomly selecting of a pair includes sending a message through the network on a randomly selected path.

5. The method of claim 4 wherein when a participant receives the message, the participant sends the message to a randomly selected participant to which it is connected.

6. The method of claim 4 wherein the randomly selected path is approximately proportional to the diameter of the network.

7. The method of claim 1 wherein the participant to be added requests a portal computer to initiate the identifying of the pair of participants.

8. The method of claim 7 wherein the initiating of the identifying of the pair of participants includes the portal computer sending a message to a connected participant requesting an edge connection.

9. The method of claim 8 wherein the portal computer indicates that the message is to travel a certain distance and wherein the participant that receives the message after the message has traveled that certain distance is one of the participants of the identified pair of participants.

10. The method of claim 9 wherein the certain distance is approximately twice the diameter of the network.

11. The method of claim 1 wherein the participants are connected via the Internet.

12. The method of claim 1 wherein the participants are connected via TCP/IP connections.

13. The method of claim 1 wherein the participants are computer processes.

14. A computer-based method for adding nodes to a graph that is m-regular and m-connected to maintain the graph as m-regular, where m is four or greater, the method comprising:

identifying p pairs of nodes of the graph that are connected, where p is one half of m;

disconnecting the nodes of each identified pair from each other; and

connecting each node of the identified pairs of nodes to the added node.

15. The method of claim 14 wherein identifying of the p pairs of nodes includes randomly selecting a pair of connected nodes.

16. The method of claim 14 wherein the nodes are computers and the connections are point-to-point communications connections.

17. The method of claim 14 wherein m is even.

18. A method of initiating adding of a participant to a network, the method comprising:

receiving a connection message from the participant to be added; and

sending a connection edge search message to a neighbor participant of the participant that received the message wherein the connection edge search message is forwarded to neighbor participants until a participant that receives the connection edge search message decides to connect to the participant to be added.

19. The method of claim 18 wherein the sent connection edge search message includes an indication of the number of participants to which the connection edge search message should be forwarded.

20. The method of claim 19 wherein the number of participants is based on the diameter of the network.

21. The method of claim 19 wherein the number of participants is approximately twice the diameter.

22. The method of claim 18 wherein when a participant decides to connect to the participant to be added, the neighbor participant that sent the connection edge search message to the participant that decided to connect also decides to connect to the participant to be added.

23. The method of claim 18 wherein participants that receive the connection edge search message forward the connection edge search message to a randomly selected neighbor.

24. A method in a computer system for connecting to a new participant of a network, the method comprising:

receiving at a participant a connection edge search message;

identifying a neighbor participant of the participant that received the connection edge search message;

notifying the neighbor participant to connect to the new participant;

disconnecting the participant from the identified neighbor participant; and

connecting the participant to the new participant.

25. The method of claim 24 including determining whether the participant is the last participant in a path of participants through which the connection edge search message was sent.

26. The method of claim 25 wherein when the participant is not the last participant in the path, sending the connection edge search message to a neighbor of the participant.

27. The method of claim 26 including randomly selecting the neighbor participant to which the connection edge search message is to be sent.

28. The method of claim 24 wherein the received connection edge search message includes an indication of the number of participants through which the connection edge search message is to be sent.

29. The method of claim 24 including when the participant is already a neighbor of the new participant, sending the connection edge search message to a neighbor participant of the participant.

30. The method of claim 24 wherein the participants are computer processes.

31. The method of claim 24 wherein the connections are point-to-point connections.

32. A computer-readable medium containing instructions for controlling a computer system to connect a participant to a network of participants, each participant being connected to three or more other participants, the network representing a broadcast channel wherein each participant forwards broadcast messages that it receives to its neighbor participants, by a method comprising:

identifying a pair of participants of the network that are connected;

disconnecting the participants of the identified pair from each other; and

connecting each participant of the identified pair of participants to the added participant.

33. The computer-readable medium of claim 32 wherein each participant is connected to 4 participants.

34. The computer-readable medium of claim 32 wherein the identifying of a pair includes randomly selecting a pair of participants that are connected.

35. The computer-readable medium of claim 34 wherein the randomly selecting of a pair includes sending a message through the network on a randomly selected path.

36. The computer-readable medium of claim 35 wherein when a participant receives the message, the participant sends the message to a randomly selected participant to which it is connected.

37. The computer-readable medium of claim 35 wherein the randomly selected path is approximately twice a diameter of the network.

38. The computer-readable medium of claim 32 wherein the participant to be added requests a portal computer to initiate the identifying of the pair of participants.

39. The computer-readable medium of claim 38 wherein the initiating of the identifying of the pair of participants includes the portal computer sending a message to a connected participant requesting an edge connection.

40. The computer-readable medium of claim 38 wherein the portal computer indicates that the message is to travel a certain distance and wherein the participant that receives the message after the message has traveled that certain distance is one of the identified pair of participants.

41. A method in a computer system for connecting to a participant of a network, the method comprising:

receiving at a participant a connection port search message sent by a requesting participant; and

when the participant has a port that is available through which it can connect to the requesting participant,

sending a port connection message to the requesting participant proposing that the requesting participant connect to the available port of the participant; and

when the participant receives a port proposal response message that indicates the requesting participant accepts to connect to the available port, connecting the participant to the requesting participant.

42. The method of claim 41 including:

when the participant does not have a port that is available through which it can connect to the requesting participant, sending the connection port search message to a neighbor participant.

43. The method of claim 41 wherein a port is available when the requesting participant is not already connected to the participant and the participant has an empty port.

44. A method in a computer system of detecting neighbors with empty ports condition in a network, the method comprising:

receiving at a first participant a connection port search message indicating that a second participant has an empty port; and

when the first participant is already connected to the second participant and the first participant has an empty port, sending a condition check message from the first participant to the second participant wherein the condition check message identifies neighbors of the first participant.

45. The method of claim 44 including:

when the second participant receives the condition check message,

when the second participant does not have the same neighbors as the first participant, sending a condition repair message to third participant that is a neighbor of the first participant but is not a neighbor of the second participant.

46. The method of claim 45 including:

when the third participant receives the condition repair message,

disconnecting from a neighbor of the third participant other than the first participant; and

connecting to the second participant.

47. The method of claim 44 including:

when the second participant receives the condition check message,

when the second participant has the same neighbors as the first participant, sending a condition double check message to a third participant that is a neighbor of the second participant.

48. The method of claim 47 including:

when the third participant receives the condition double check message, when the third participant does not have the same neighbors as the first participant, sending a condition check message to a fourth participant that is not the first participant or the second participant.

49. The method of claim 48 including:

when the fourth participant receives the condition check message, sending a condition repair message to a fifth participant directing the fifth participant to connect to the first participant or the second participant.

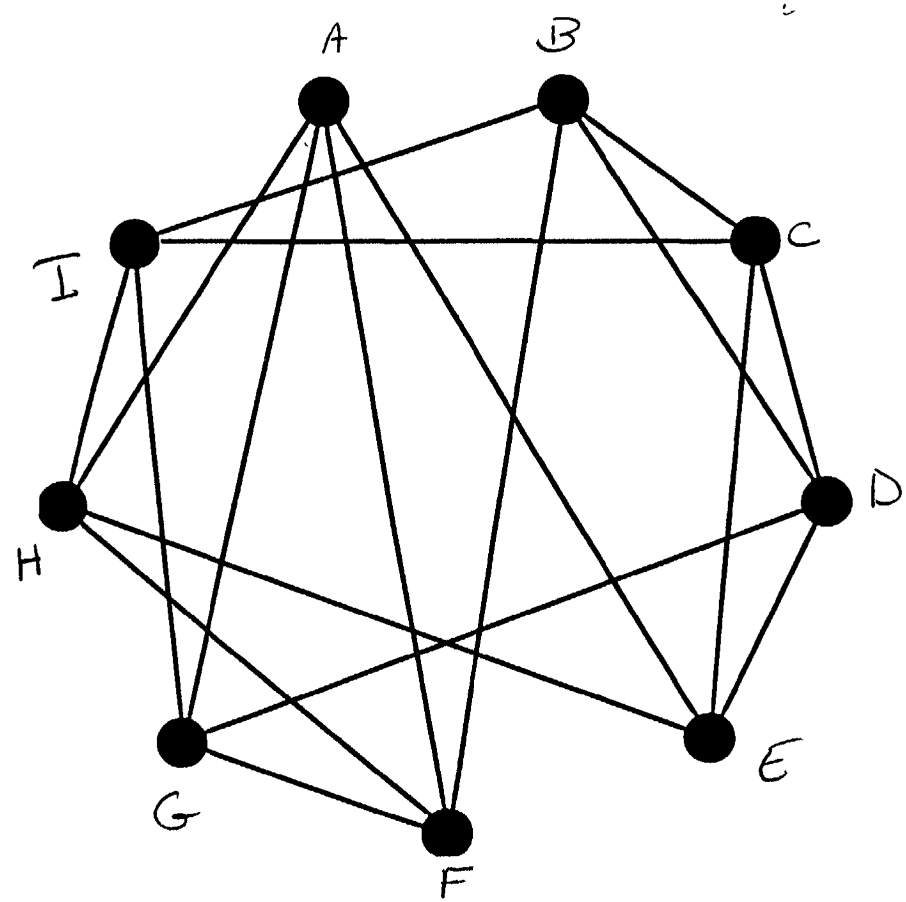


Fig 1

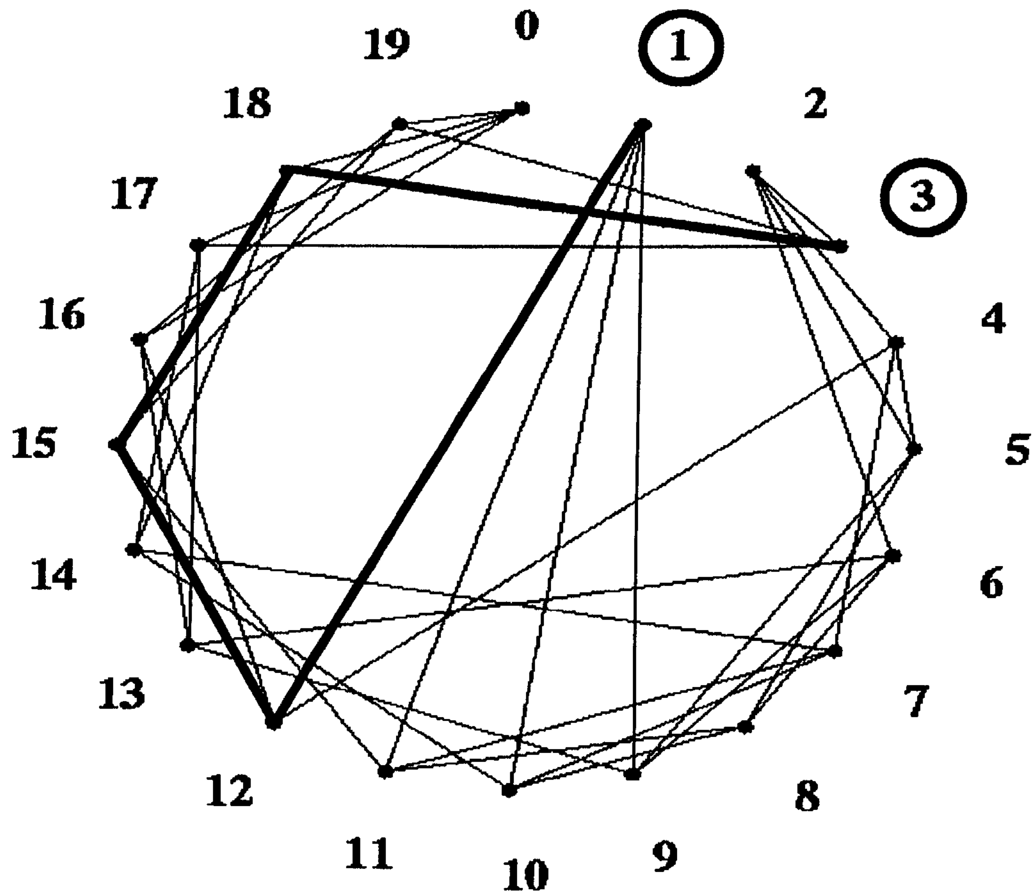


Fig 2

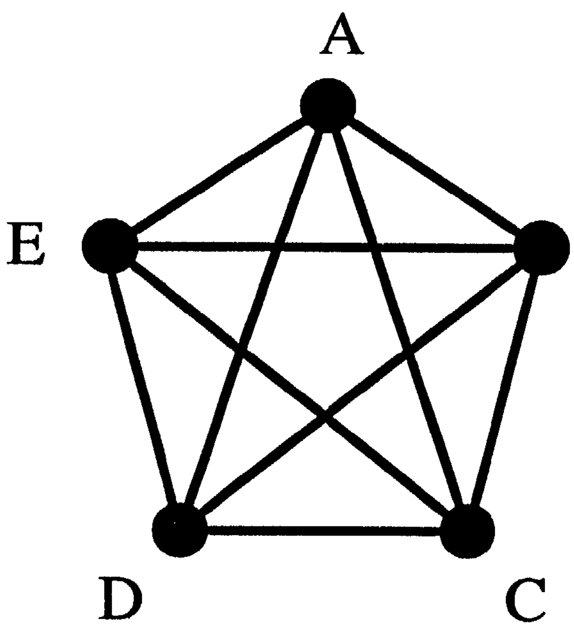


Fig 3A

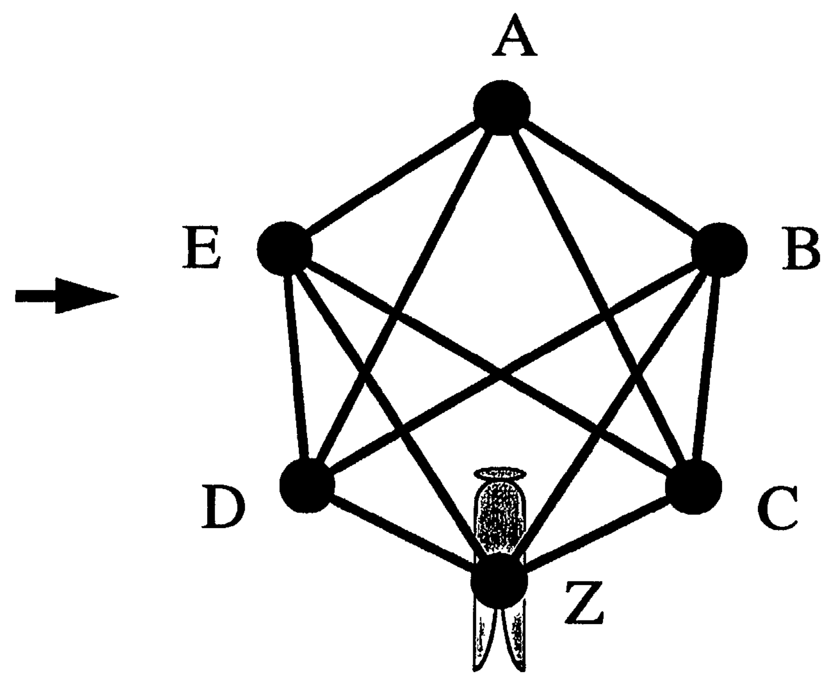


Fig 3B

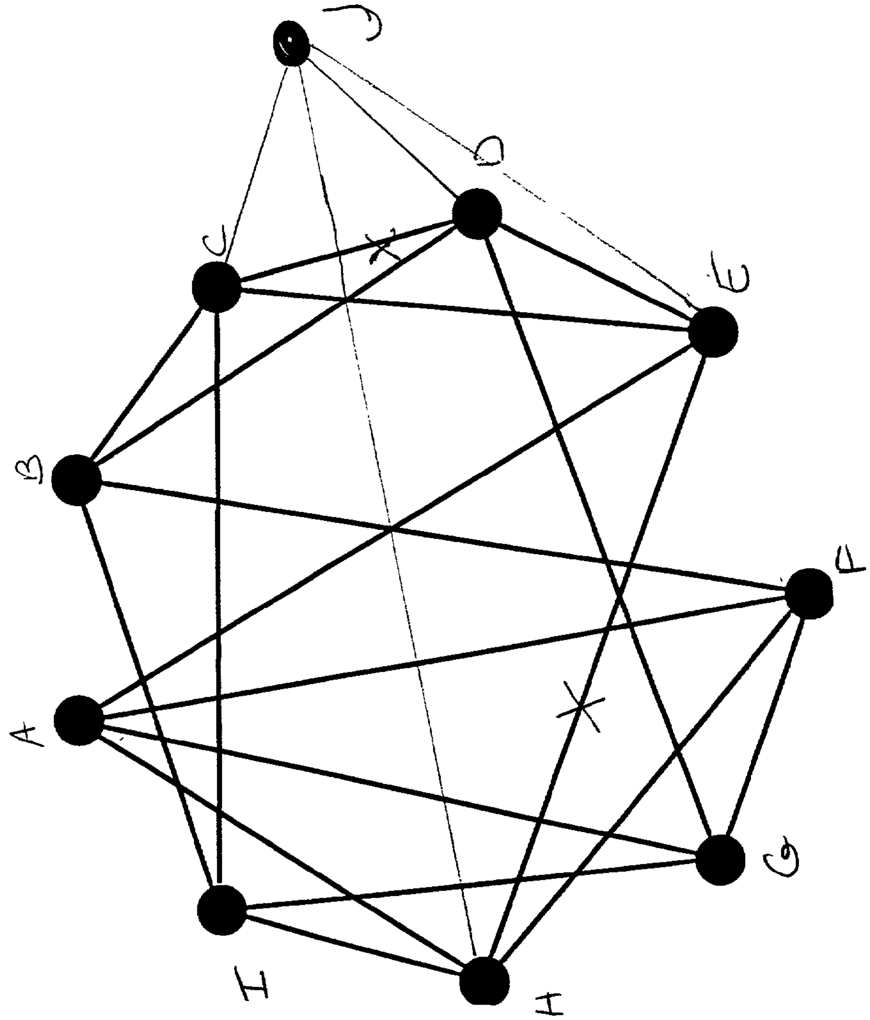


Fig 4A

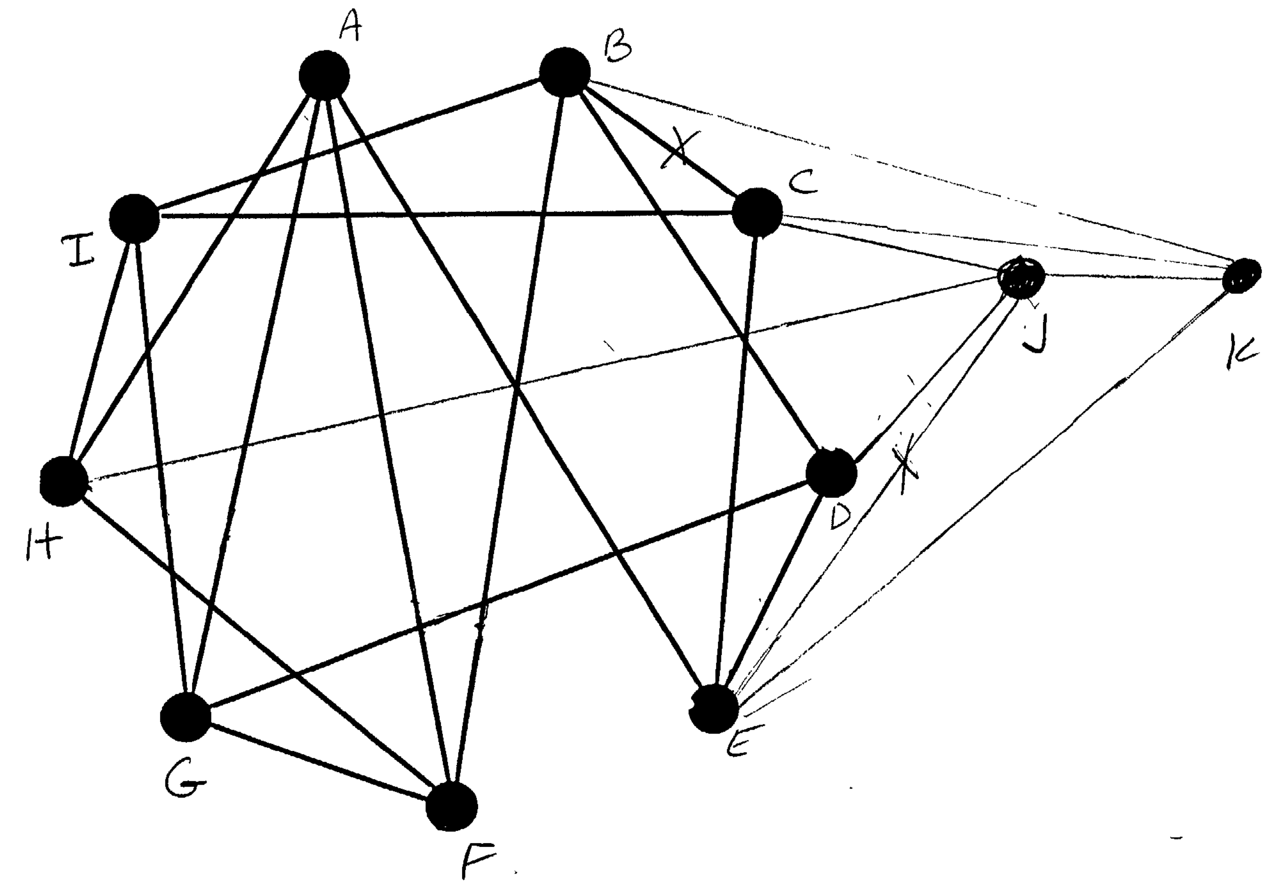


Fig 4B

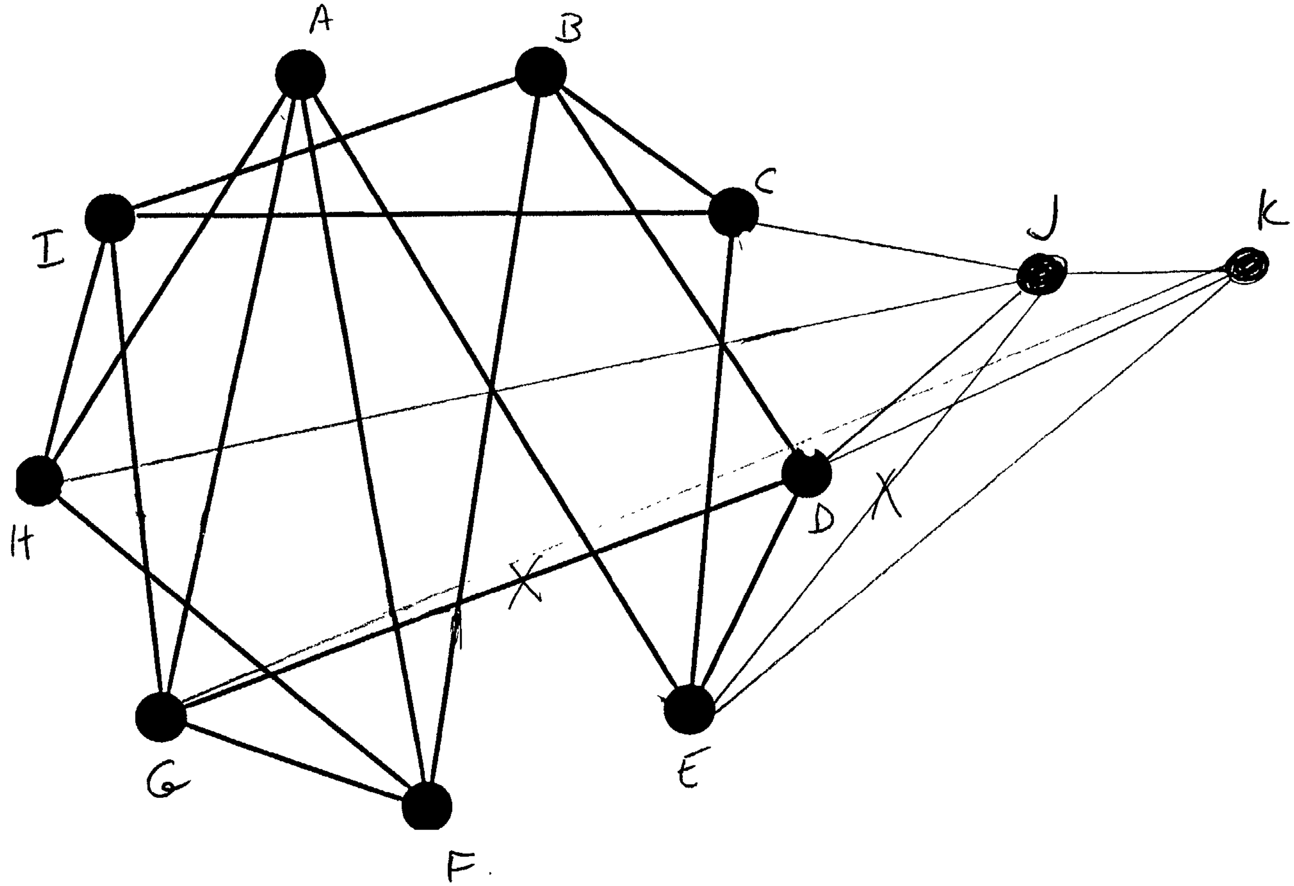


Fig 4C

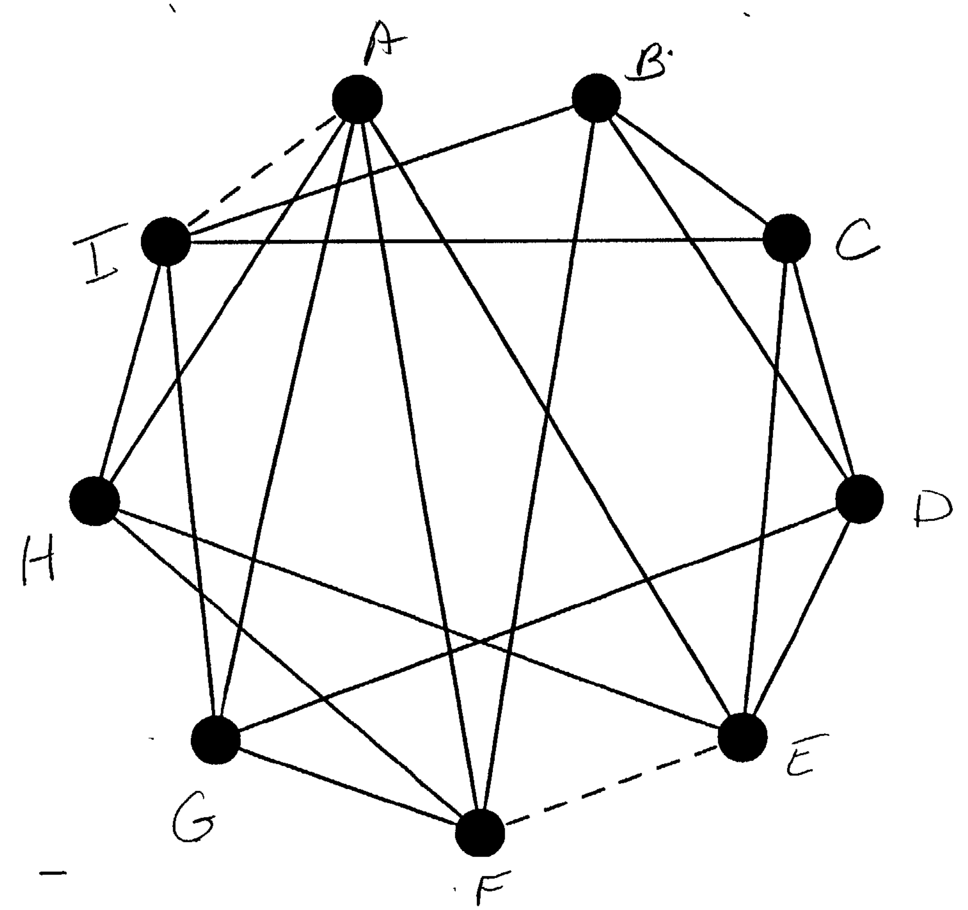


Fig 5A

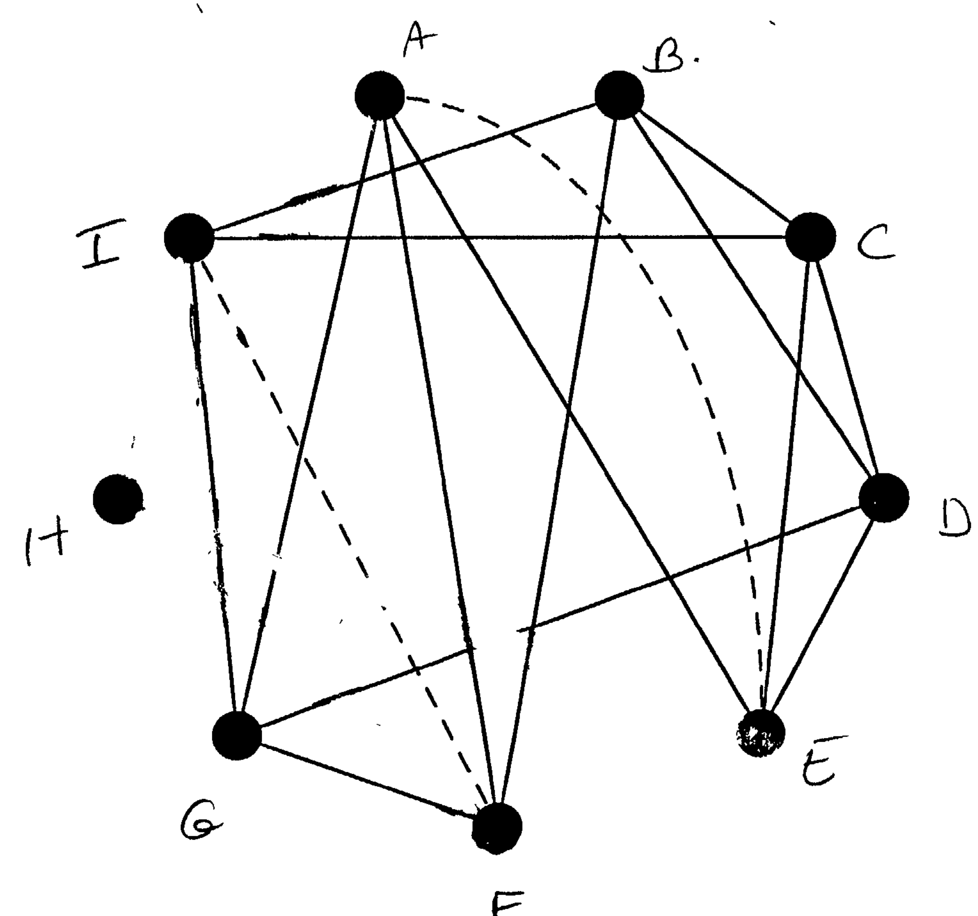


Fig 5B

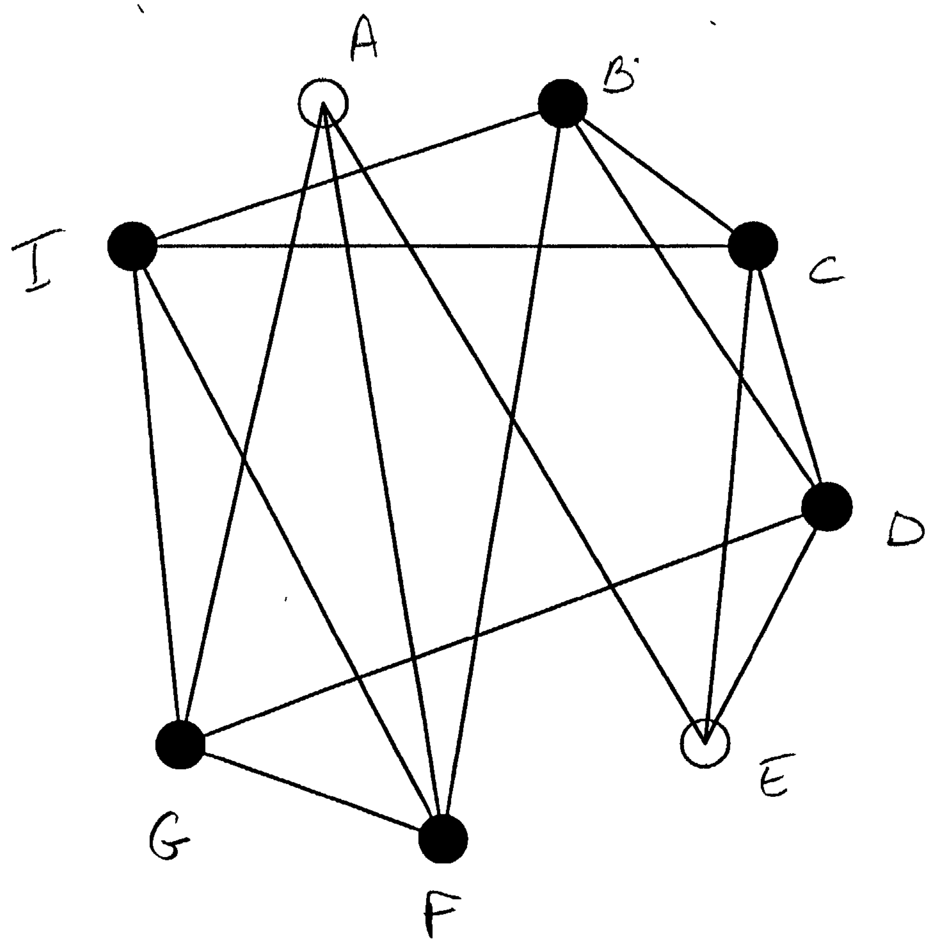


Fig 5C

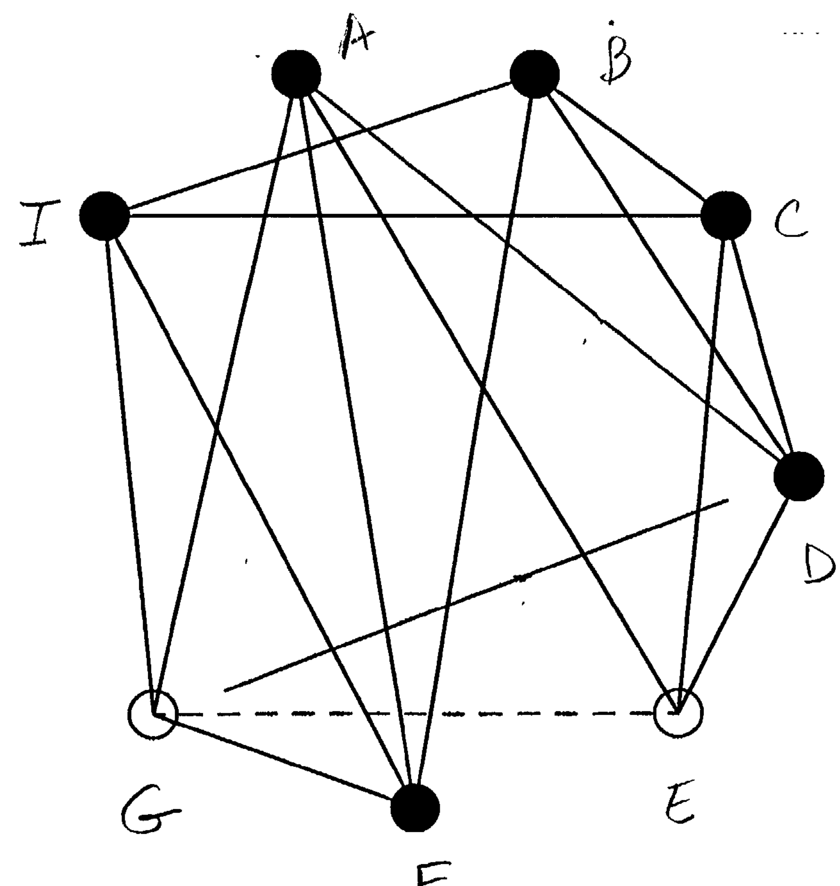


Fig 5D

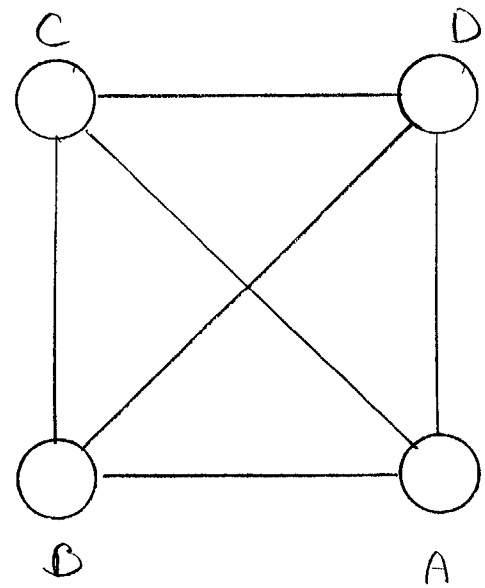


Fig 5E

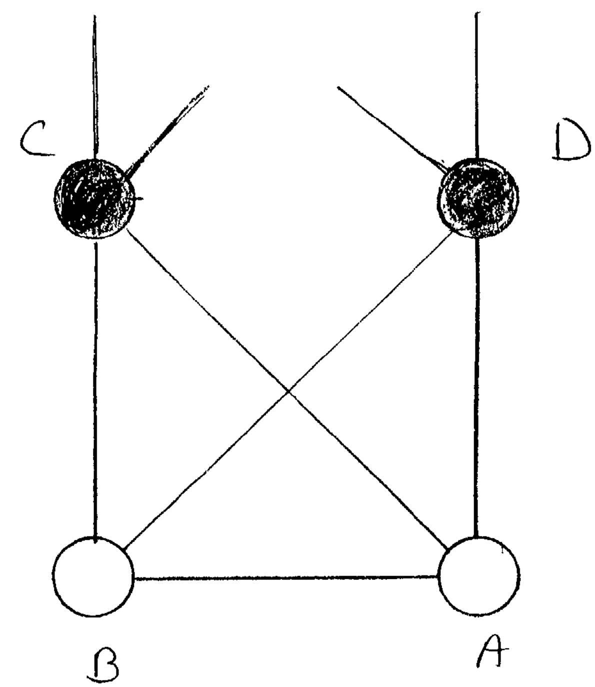


Fig 5F

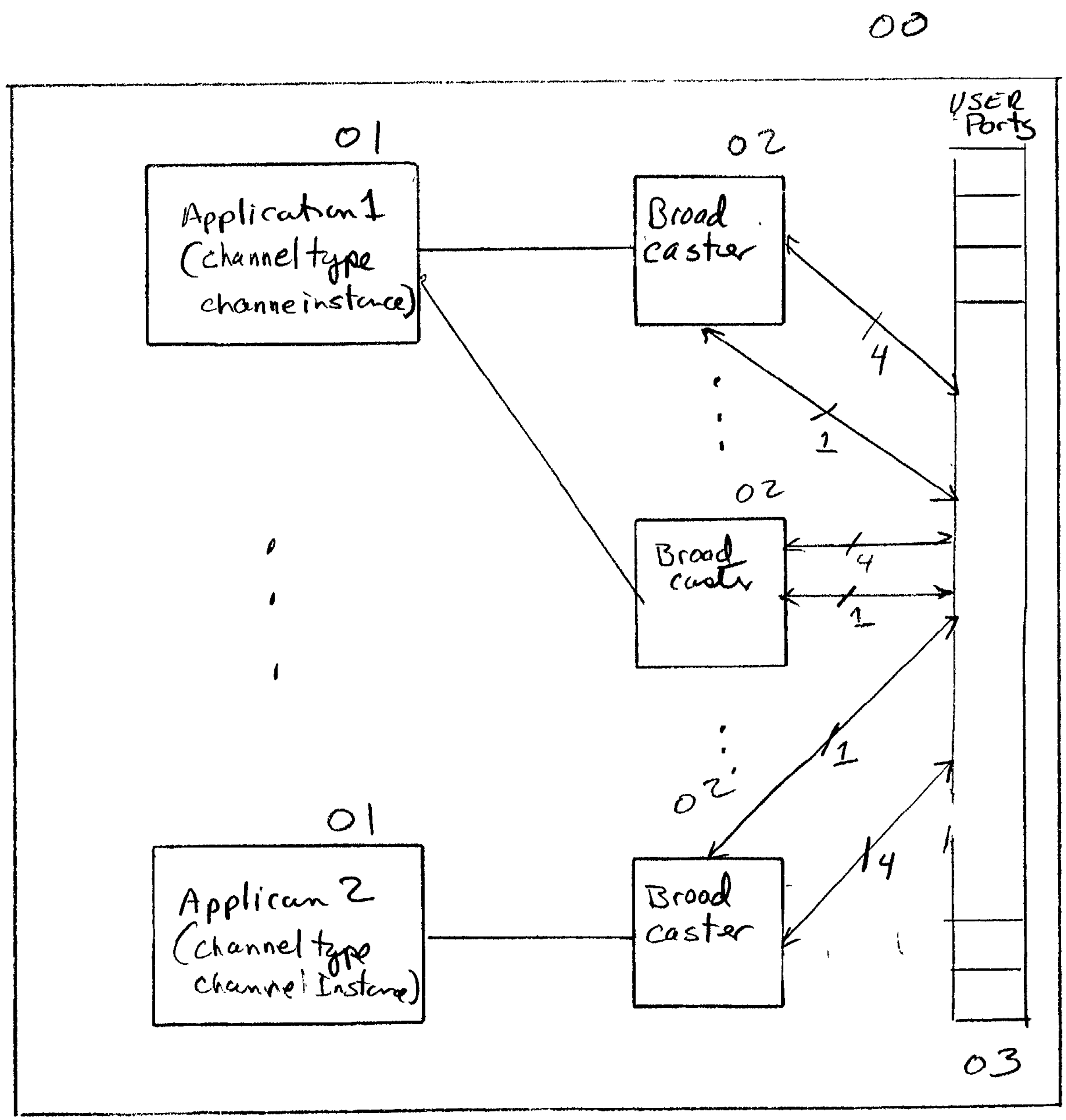

00
USER Ports
01
Application 1 (channel type channe instance)
02
Broad caster
4
1
02
Broad caster
4
1
1
02
4
01
Applican 2 (channel type channel Instance)
Broad caster
03

Fig 6

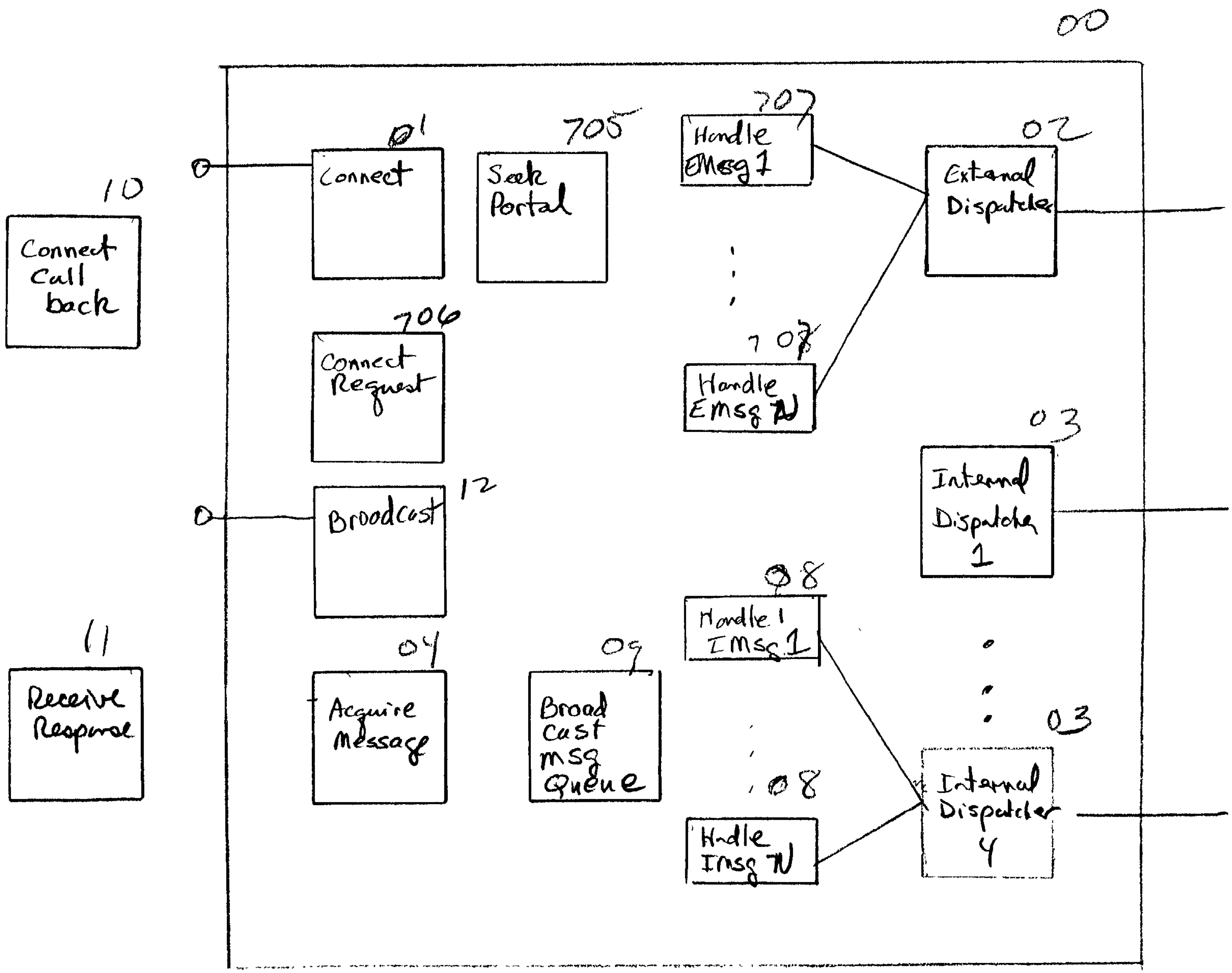

00
10
Connect Call back
11
Receive Response
01
Connect
705
Seek Portal
707
Handle EMsg 1
708
Handle EMsg N
02
Extanal Dispatcher
706
Connect Request
12
Broadcast
03
Internal Dispatcher 1
04
Acquire Message
09
Broad cast msg Queue
08
Handle I IMsg 1
08
Handle IMsg N
03
Internal Dispatcher 4

Figure 7

p2

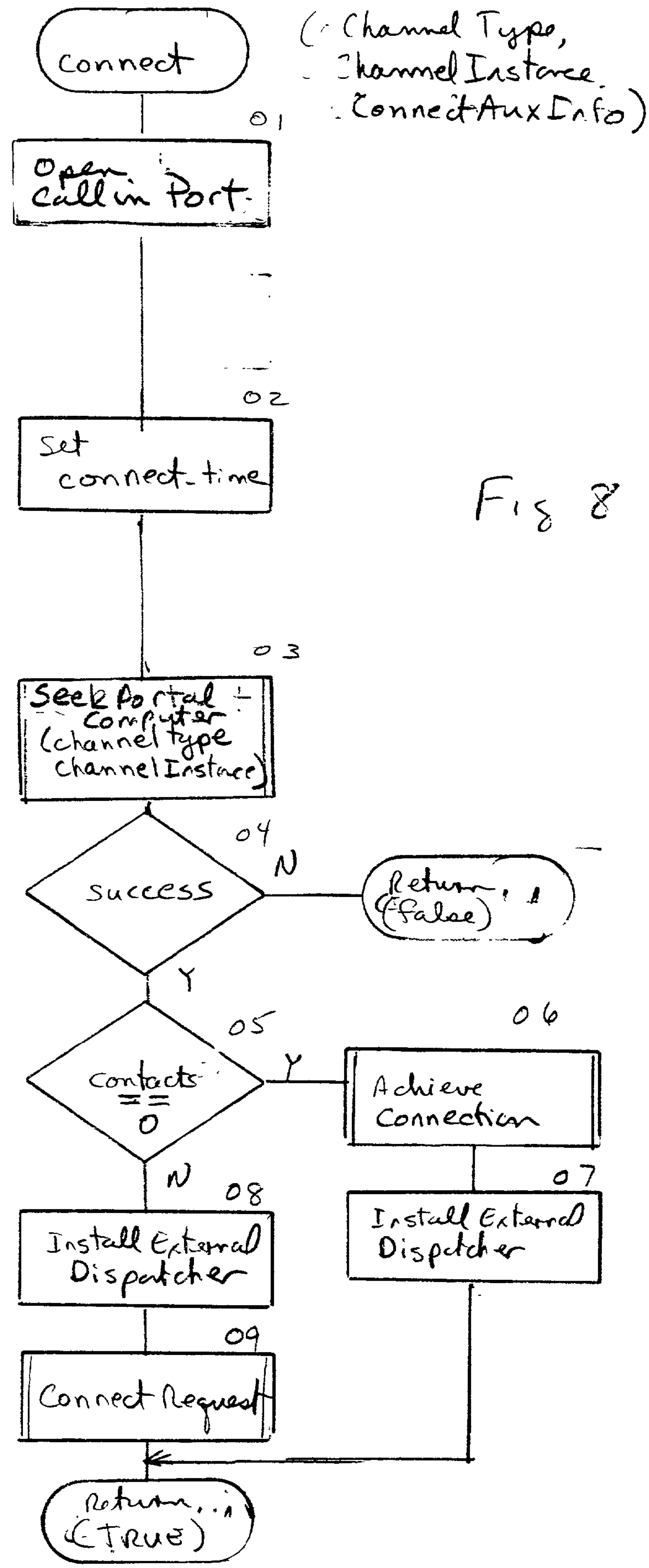

Connect
(Channel Type, Channel Instance, Connect Aux Info)
01
Open Call in Port
02
Set connect_time
03
Seek Portal Computer (Channel type Channel Instance)
04
Success
N
Return (False)
Y
05
Contacts == 0
Y
06
Achieve Connection
07
Install External Dispatcher
N
08
Install External Dispatcher
09
Connect Request
Return (TRUE)

Fig 8

P4
Quest

Channel type
channel Instance

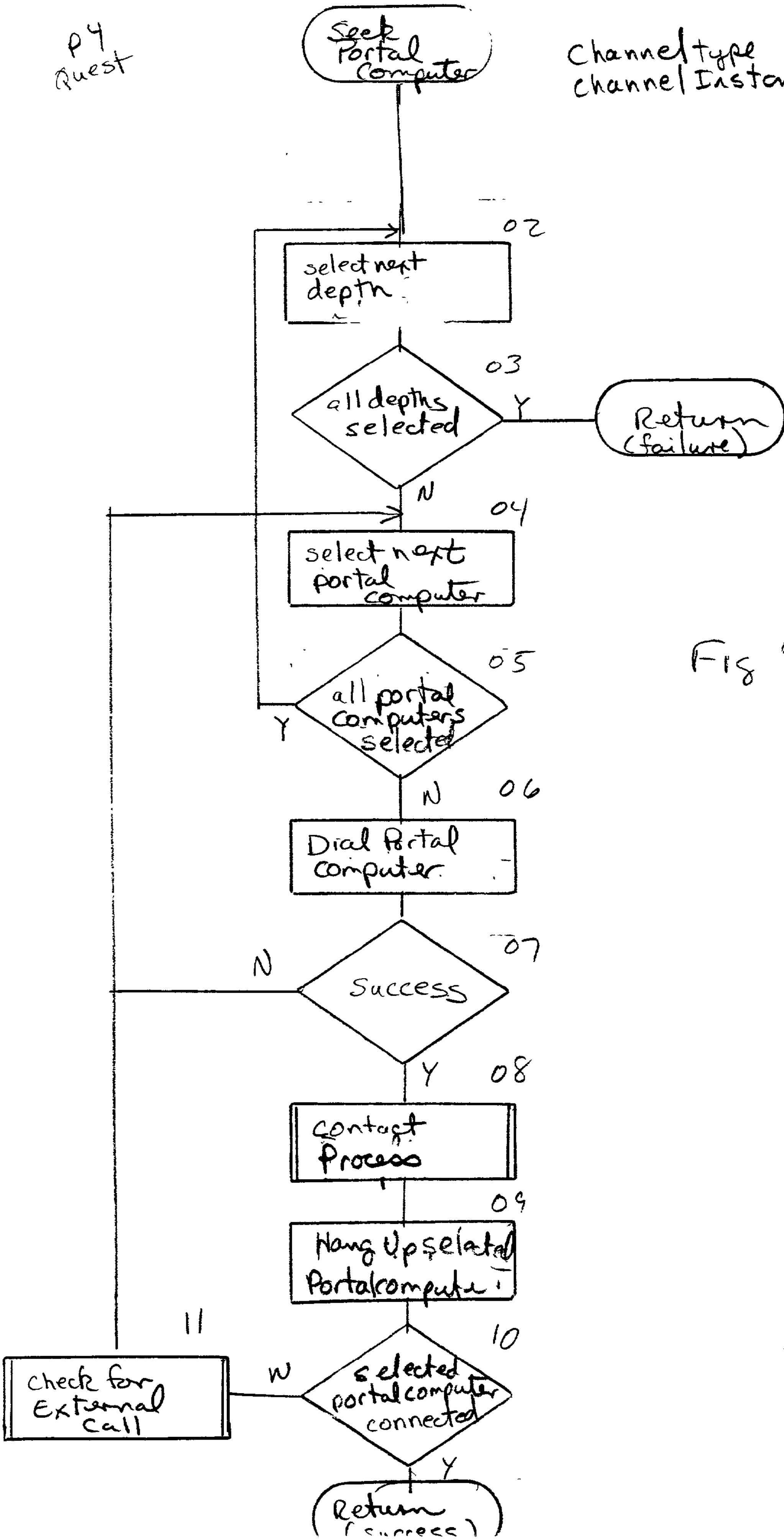


Fig 9

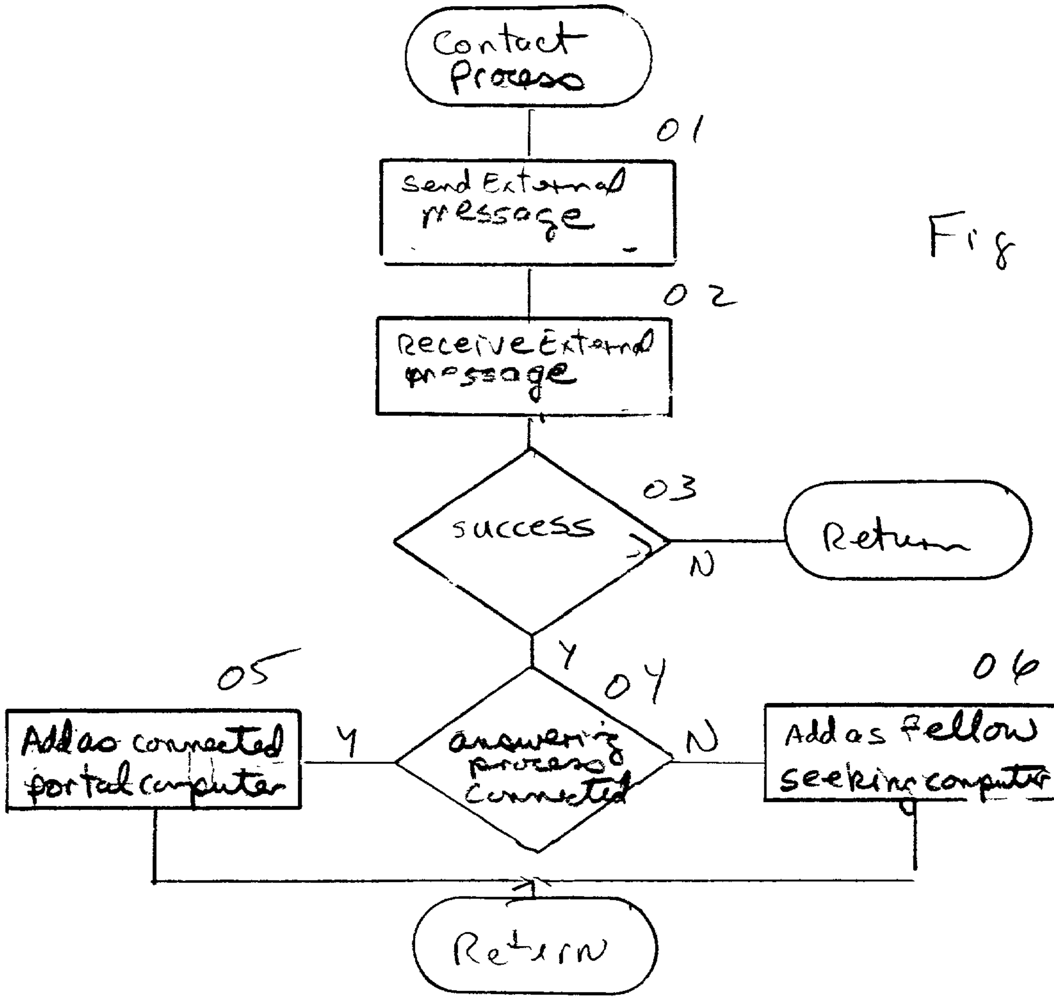
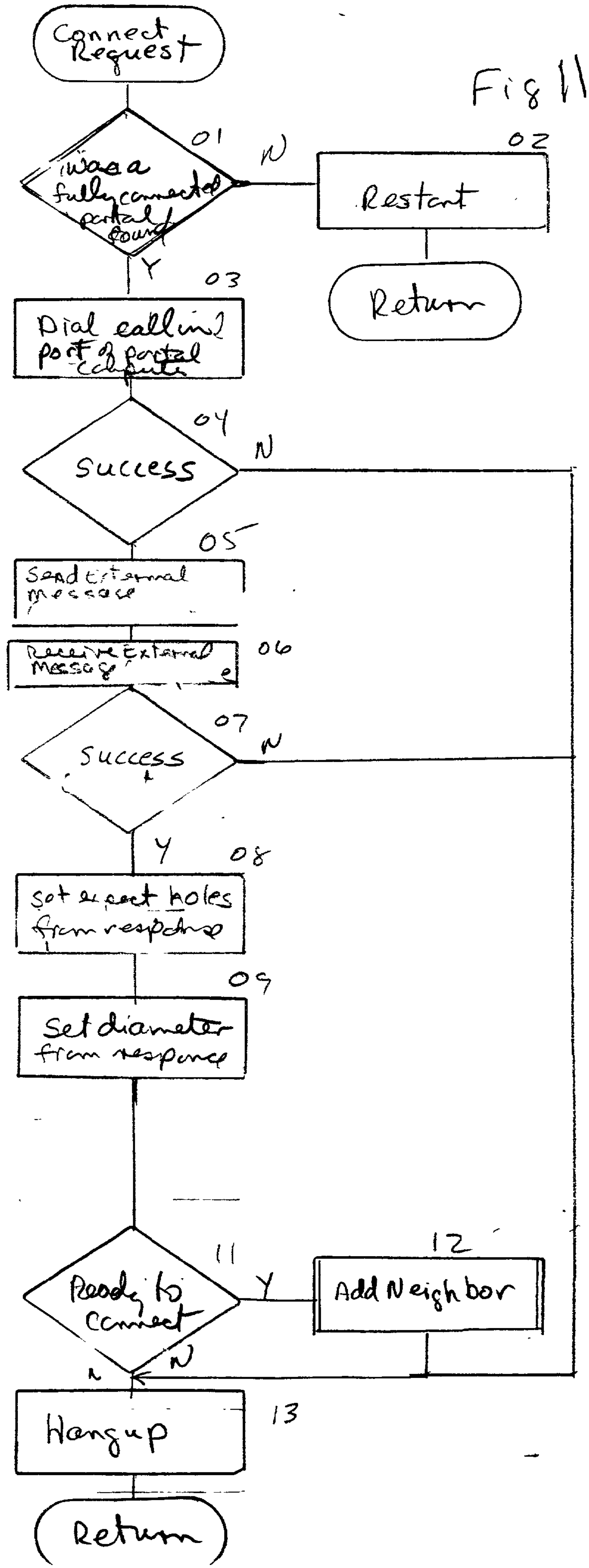

Contact Process
01
Send External message
02
Receive External message
03
success
N
Return
Y
04
answering process connected
05
Y
Add as connected portal computer
N
06
Add as fellow seeking computer
Return

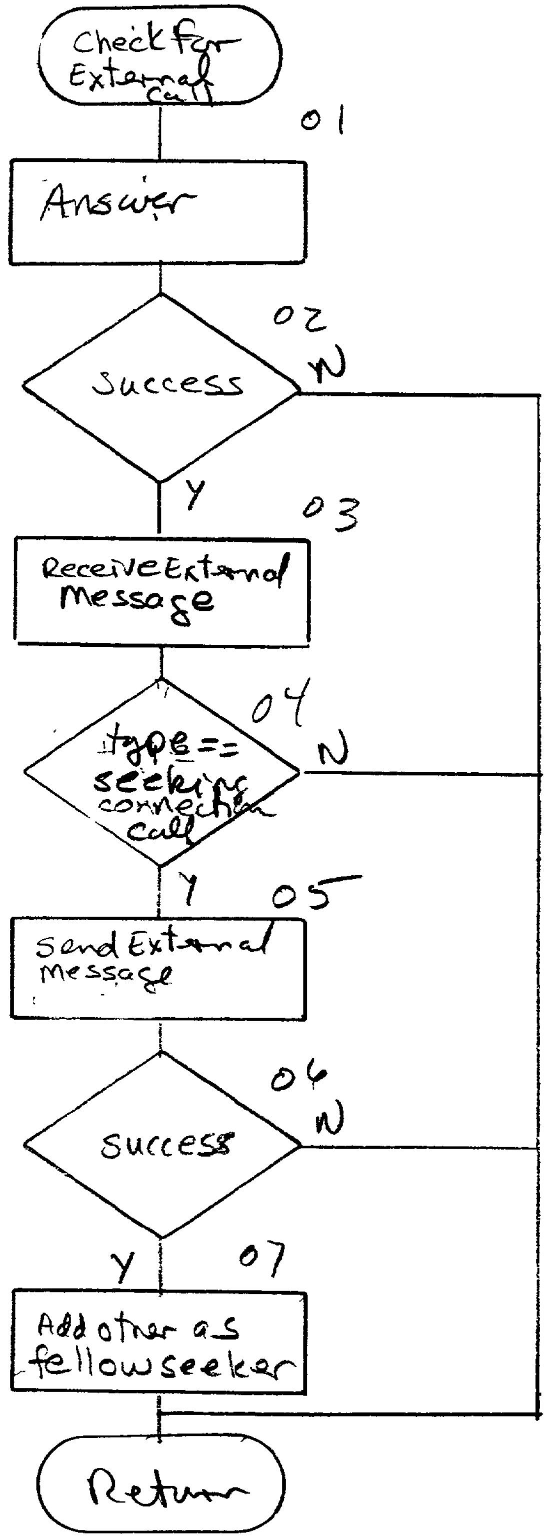
Fig 10

p23

Fig 11



Connect Request
01
was a fully connected portal found
N
02
Restart
Return
Y
03
Dial calling port of portal computer
04
success
N
05
Send External message
Receive External Message
06
07
success
N
Y
08
set expect holes from response
09
Set diameter from responce
11
Ready to connect
Y
12
Add Neighbor
N
13
Hangup
Return



Check for External call
01
Answer
02
Success
N
Y
03
Receive External message
04
type == seeking connection call
N
Y
05
Send External message
06
Success
N
Y
07
Add other as fellow seeker
Return

Fig 12

Achieve Nirvana
22

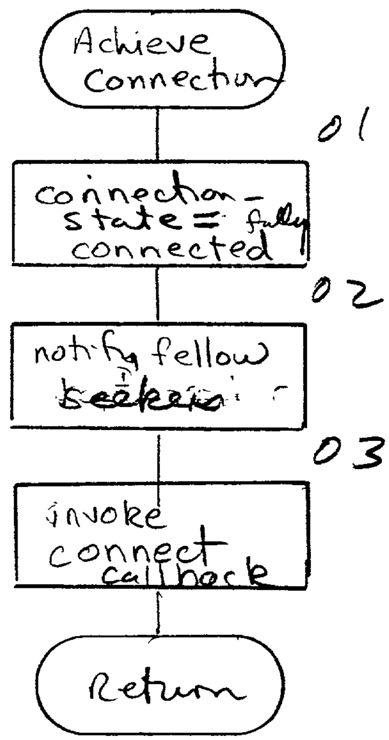
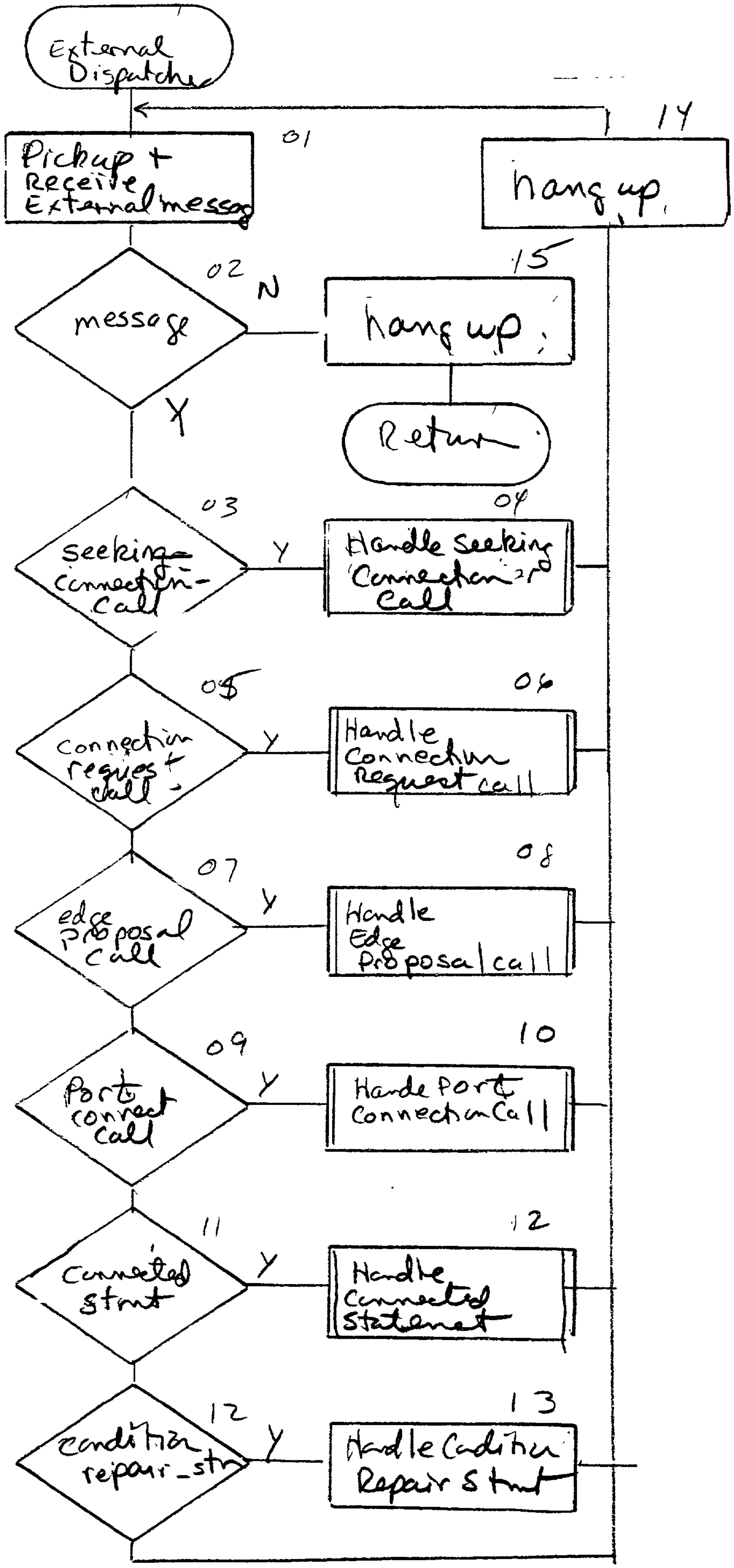


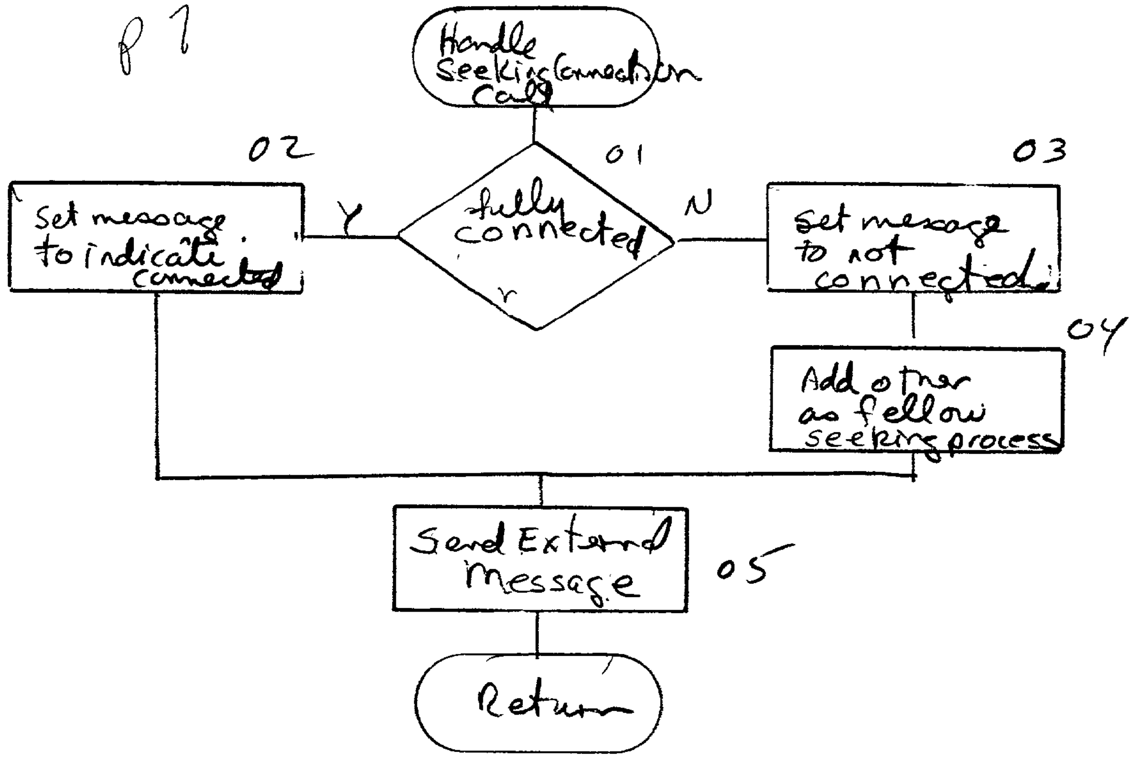
Fig 13

p7

Fig 14


External Dispatcher
01
Pickup + Receive External message
14
hang up
02
message
N
15
hang up
Y
Return
03
seeking connection call
Y
04
Handle seeking Connection Call
05
connection request call
Y
06
Handle Connection Request call
07
edge proposal call
Y
08
Handle Edge Proposal call
09
Port connect call
Y
10
Handle Port Connection Call
11
Connected stmt
Y
12
Handle Connected Statement
12
condition repair_str
Y
13
Handle Condition Repair Stmt

p 1

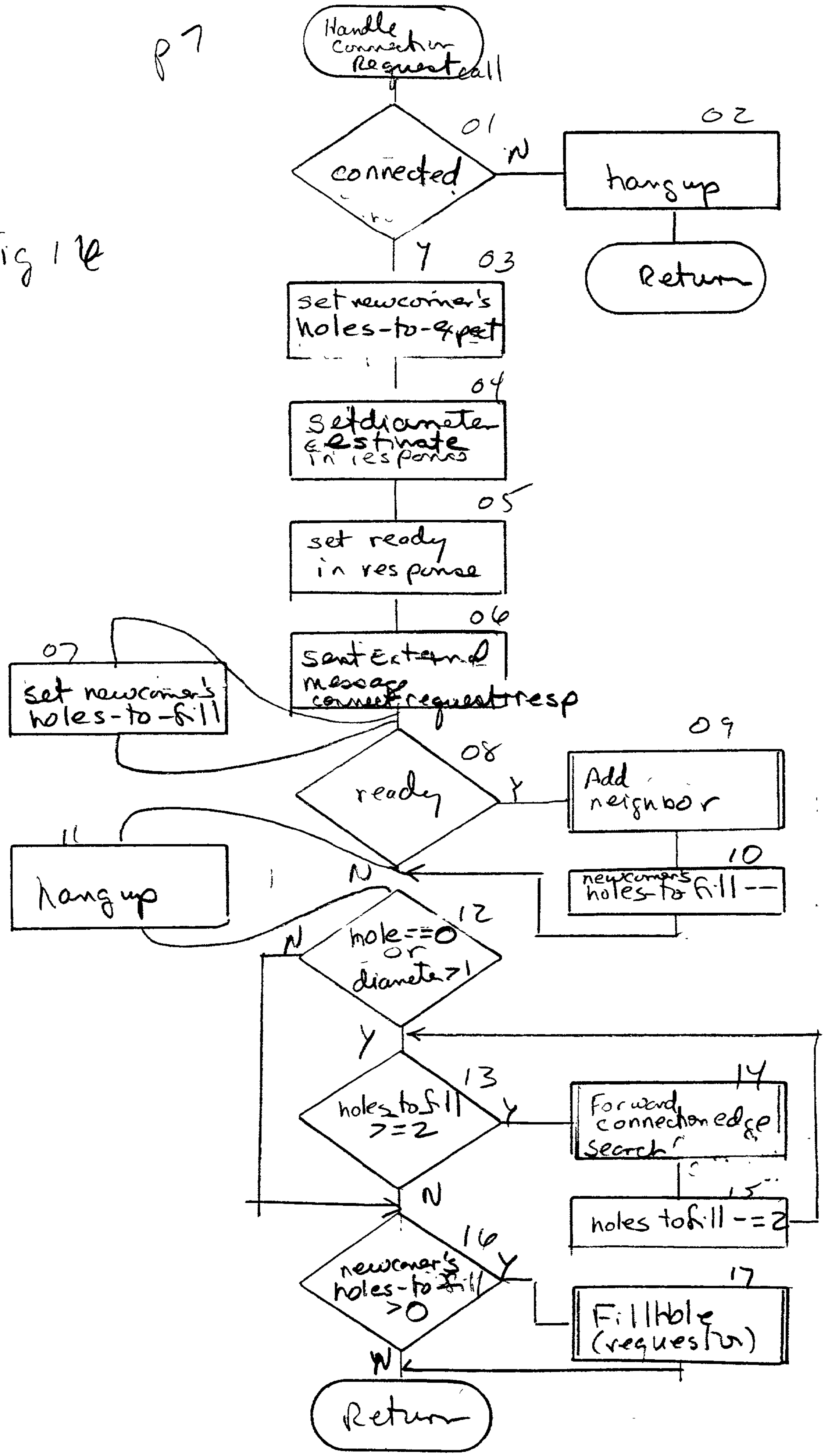


Handle Seeking Connection Call
01
fully connected
Y
N
02
Set message to indicate connected
03
Set message to not connected
04
Add other as fellow seeking process
Send External Message
05
Return

Fig 15

p 7

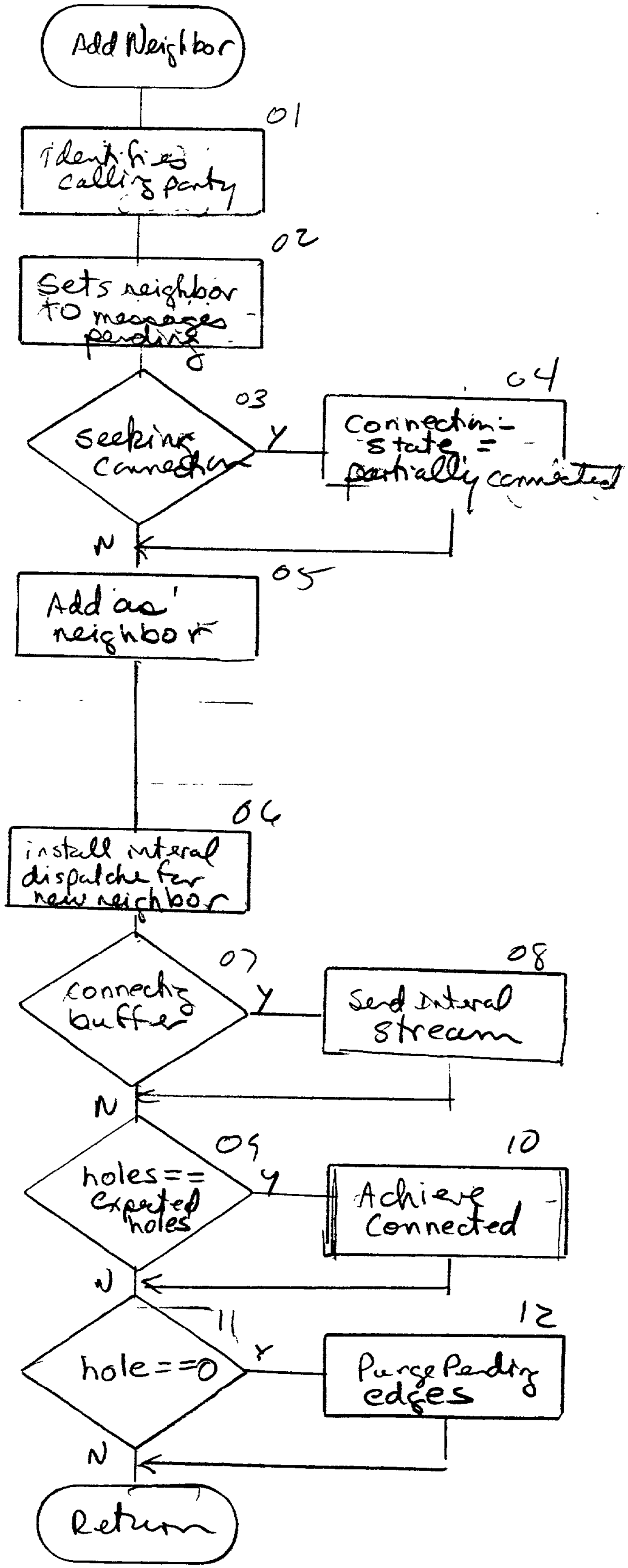
Fig 1 ℓ



Handle Connection Request call
01
connected
N
02
hangup
Return
Y
03
set newcomer's holes-to-expect
04
set diameter estimate in response
05
set ready in response
06
send Extend message connect request resp
07
set newcomer's holes-to-fill
08
ready
Y
09
Add neighbor
10
newcomer's holes-to-fill --
N
11
hang up
12
hole==0 or diameter>1
N
Y
13
holes to fill >=2
Y
14
Forward connection edge search
15
holes to fill -=2
N
16
newcomer's holes-to-fill >0
Y
17
Fill hole (requestor)
N
Return

p9

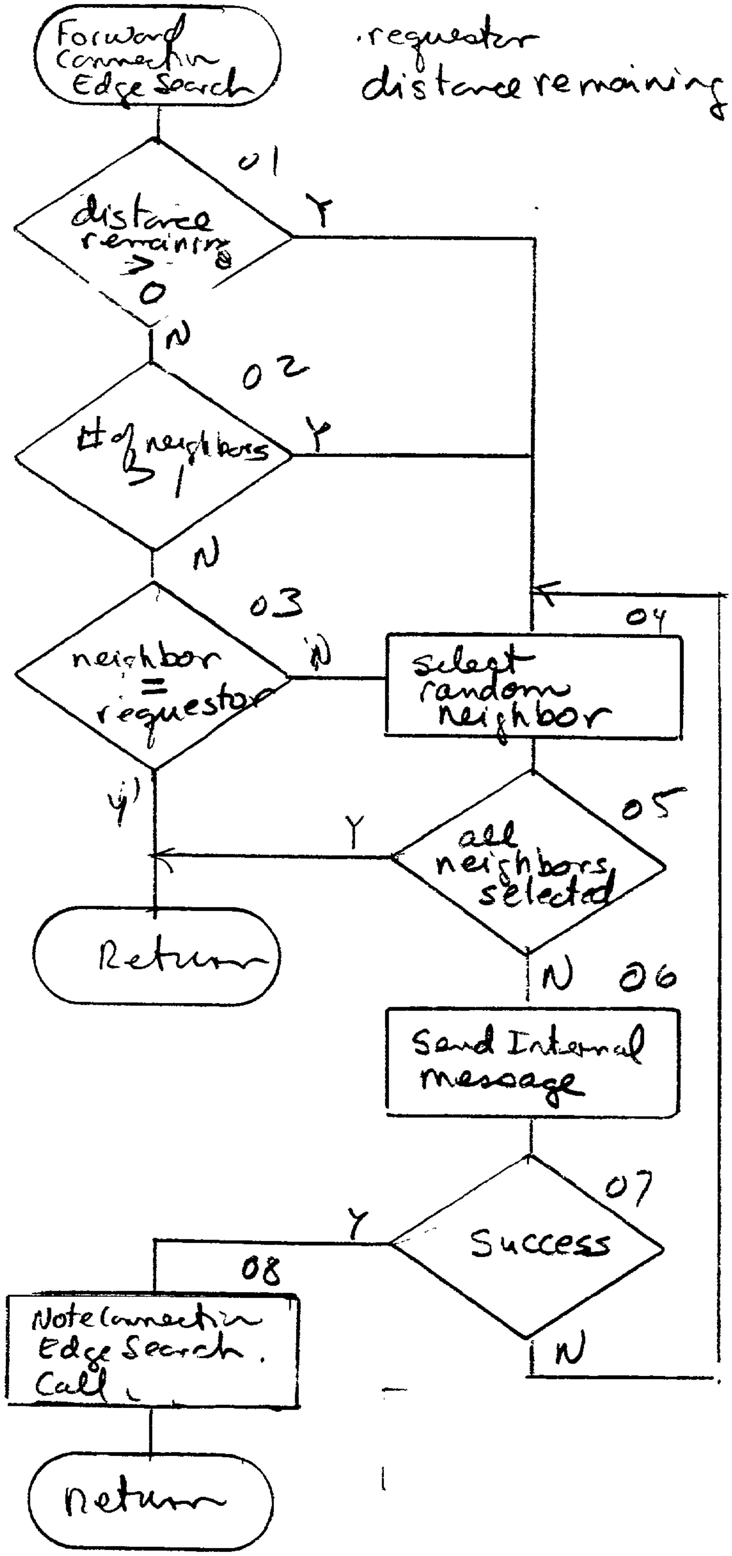
Fig 17



0072

15

Fig 18



Forward Connection Edge Search
requester
distance remaining
01
distance remaining > 0
Y
N
02
# of neighbors > 1
Y
N
03
neighbor = requestor
N
Y
04
select random neighbor
05
all neighbors selected
Y
N
Return
06
Send Internal message
07
Success
Y
N
08
Note Connection Edge Search Call
Return

p 8

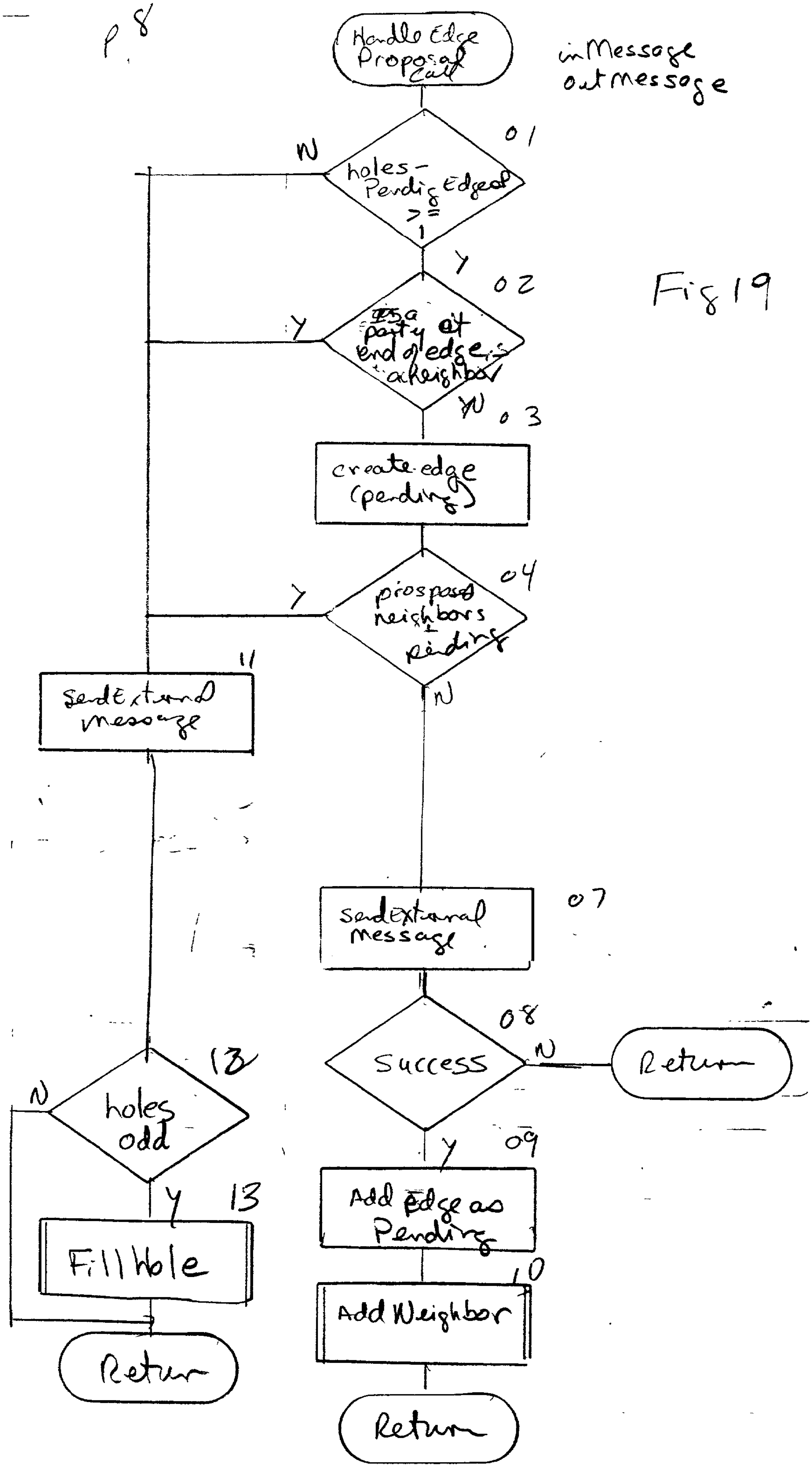
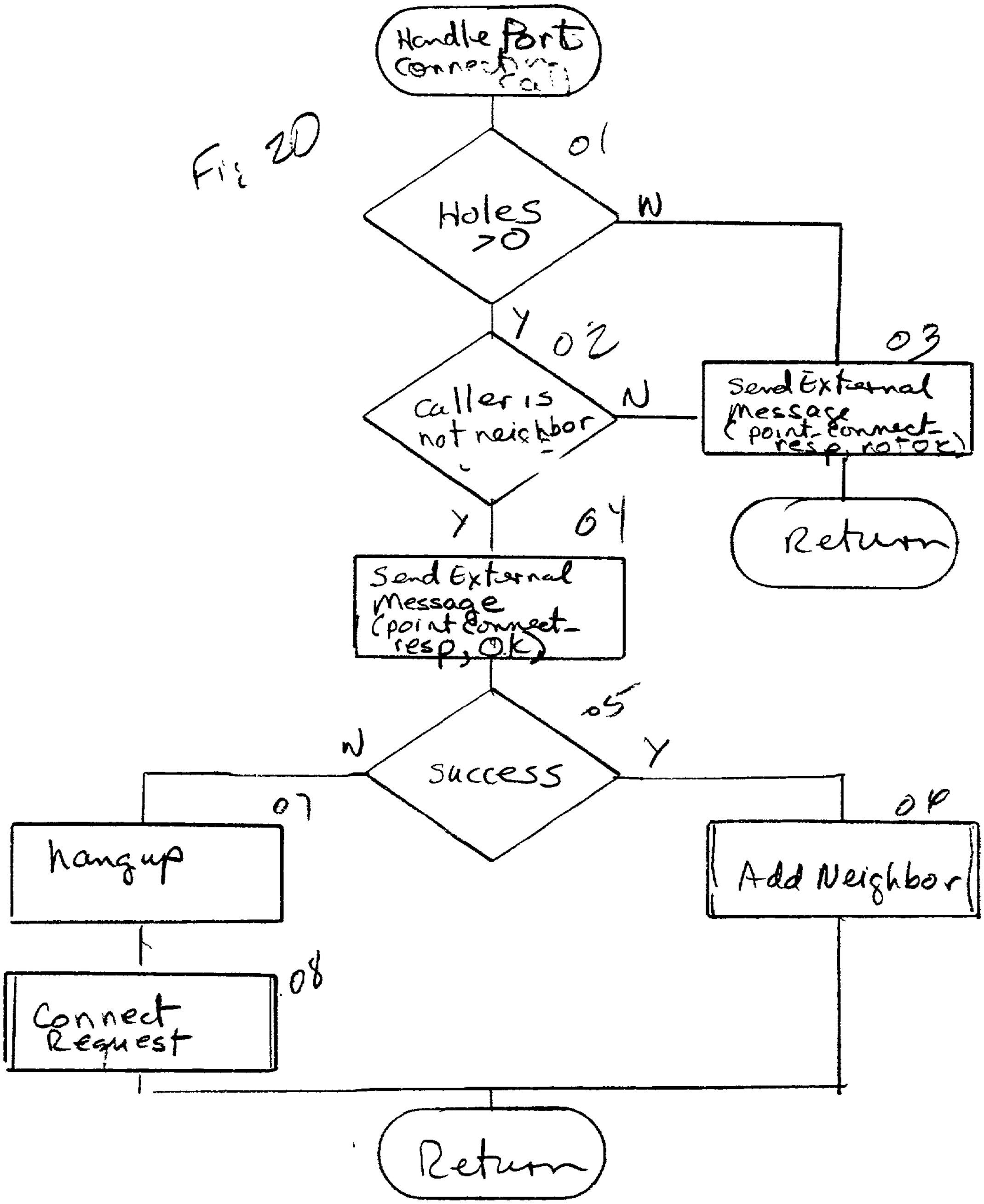

Handle Edge Proposal Call
inMessage
outMessage
01
holes-Pending Edges >= 1
N
Y
02
is a party at end of edge is a neighbor
Y
N
03
create edge (pending)
04
prospose neighbors pending
Y
N
11
sendExternal message
07
sendExternal message
08
Success
N
Return
Y
09
Add Edge as Pending
10
Add Weighbor
Return
12
holes odd
N
Y
13
FillHole
Return

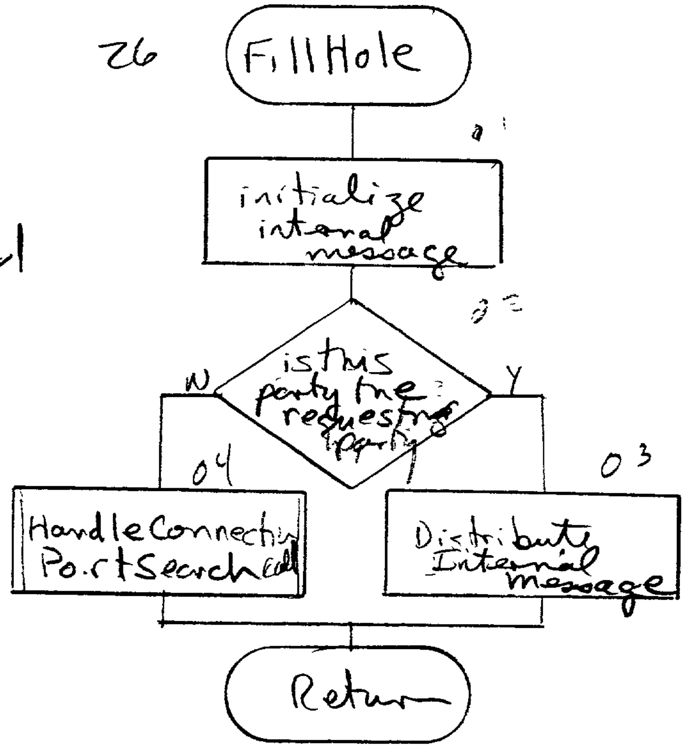
Fig 19

Fig 2D

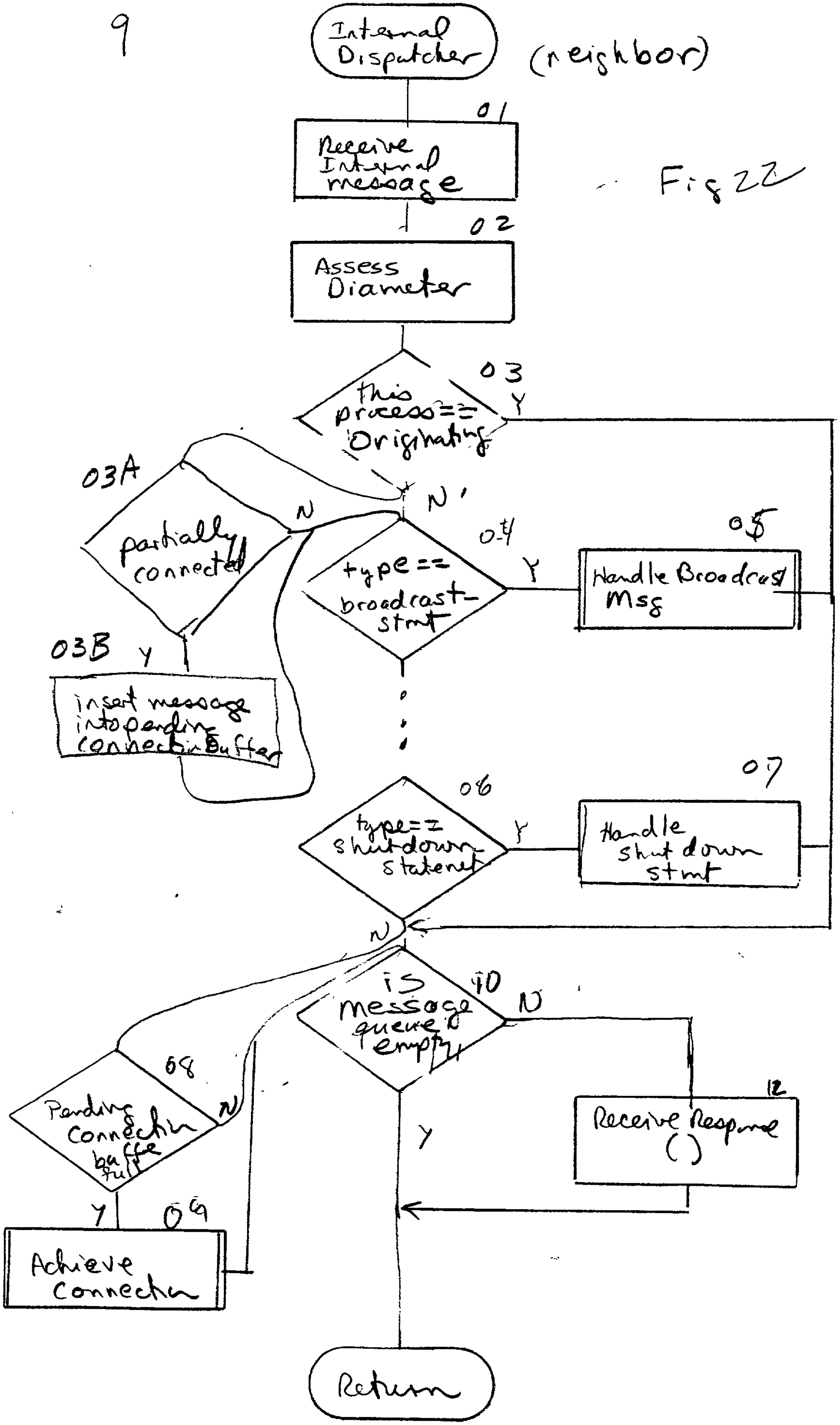


Handle Port Connection Call
01
Holes >0
N
Y
02
Caller is not neighbor
N
03
Send External Message (point_connect_resp, not OK)
Return
Y
04
Send External Message (point_connect_resp, OK)
05
Success
N
Y
07
hangup
06
Add Neighbor
08
Connect Request
Return

Fig 21


26
FillHole
initialize internal message
is this party the requesting party
N
Y
04
03
HandleConnection PortSearch
Distribute Internal message
Return

9



0096295 072300

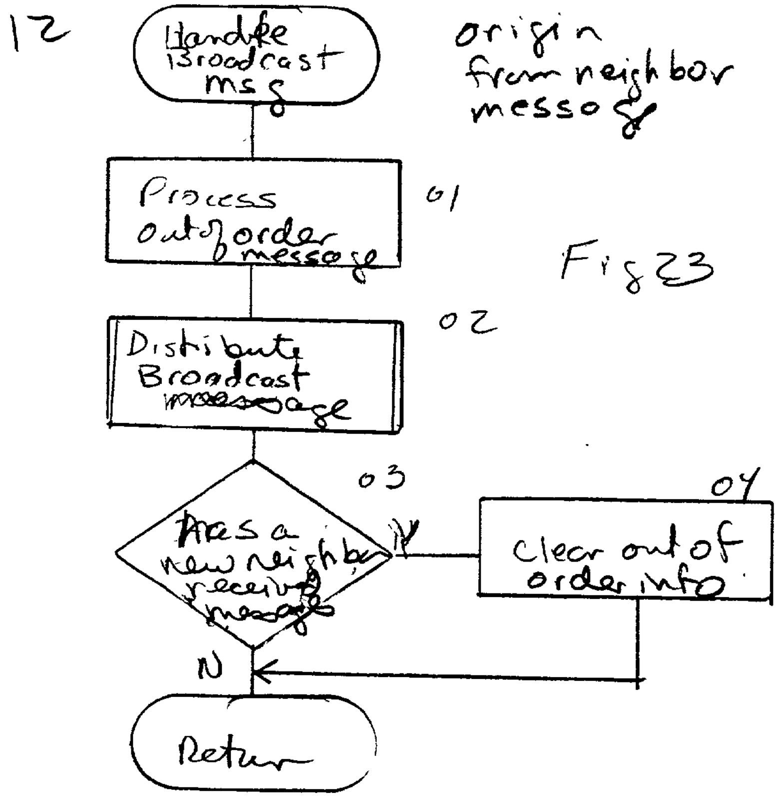

12
Handle Broadcast msg
origin from neighbor message
Process out of order message
01
02
Distribute Broadcast message
03
04
Has a new neighbor received message
Y
N
Clear out of order info
Return

Fig 23

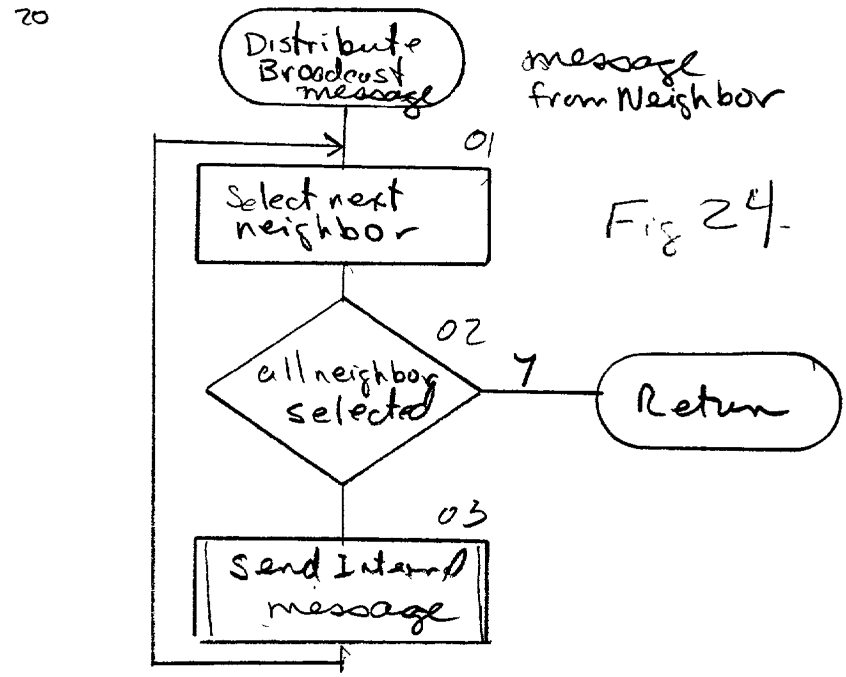
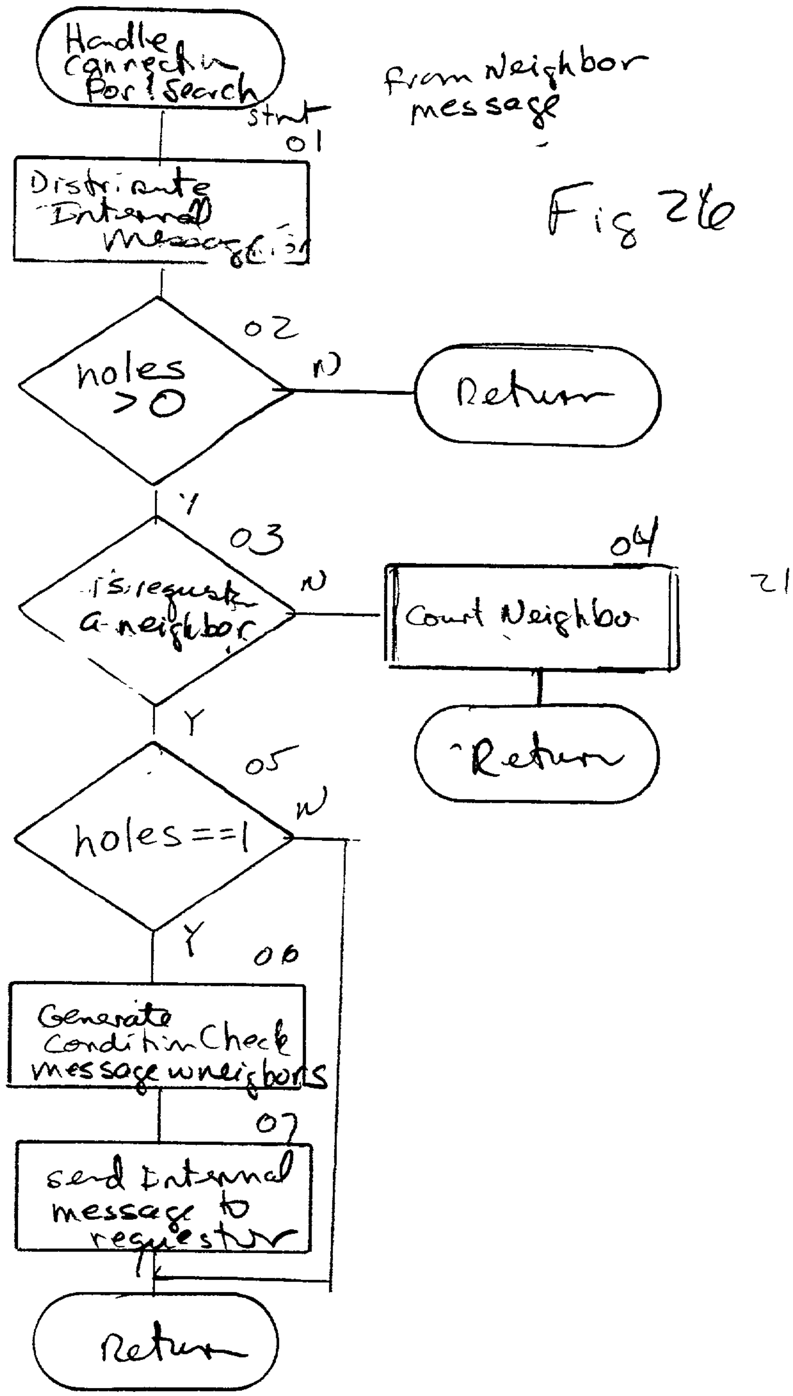

20
Distribute Broadcast message
message from Neighbor
01
Select next neighbor
02
all neighbor selected
Y
Return
03
Send Internal message

Fig 24.

13



from Neighbor message

Fig 26

21

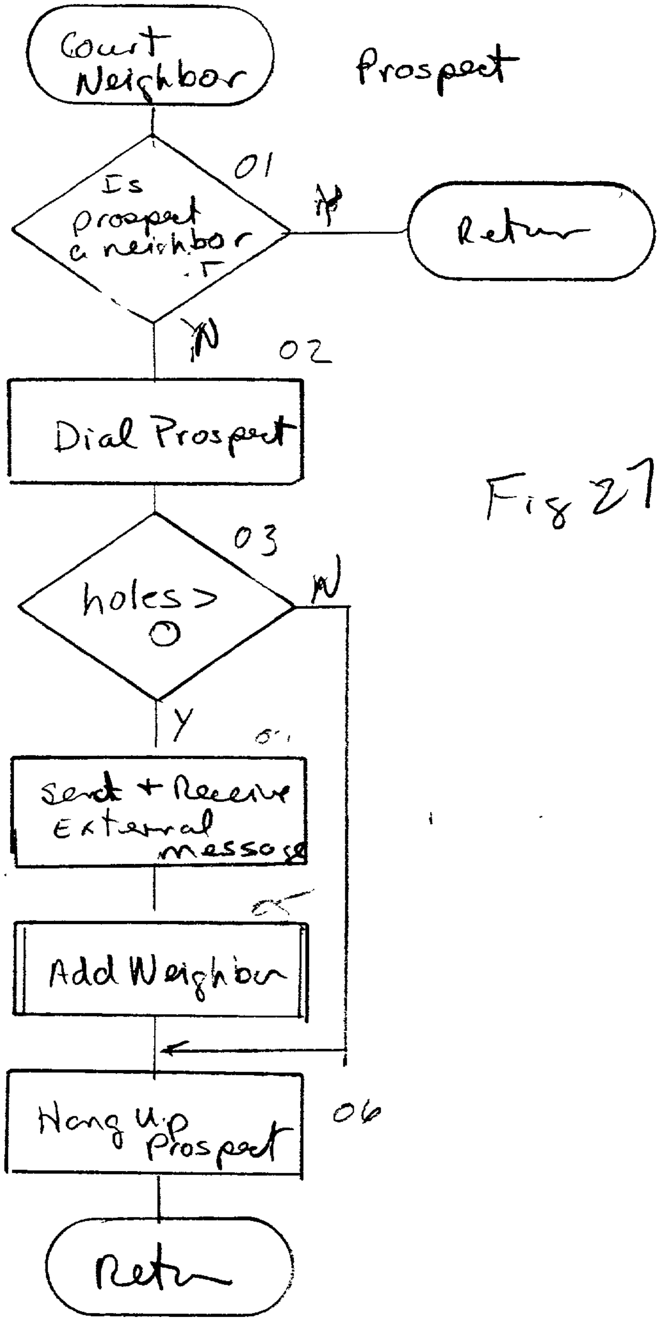

Court Neighbor
Prospect
01
Is prospect a neighbor?
N
Return
N
02
Dial Prospect
03
holes > 0
N
Y
Send + Receive External message
Add Neighbor
06
Hang up Prospect
Return

Fig 27

p14

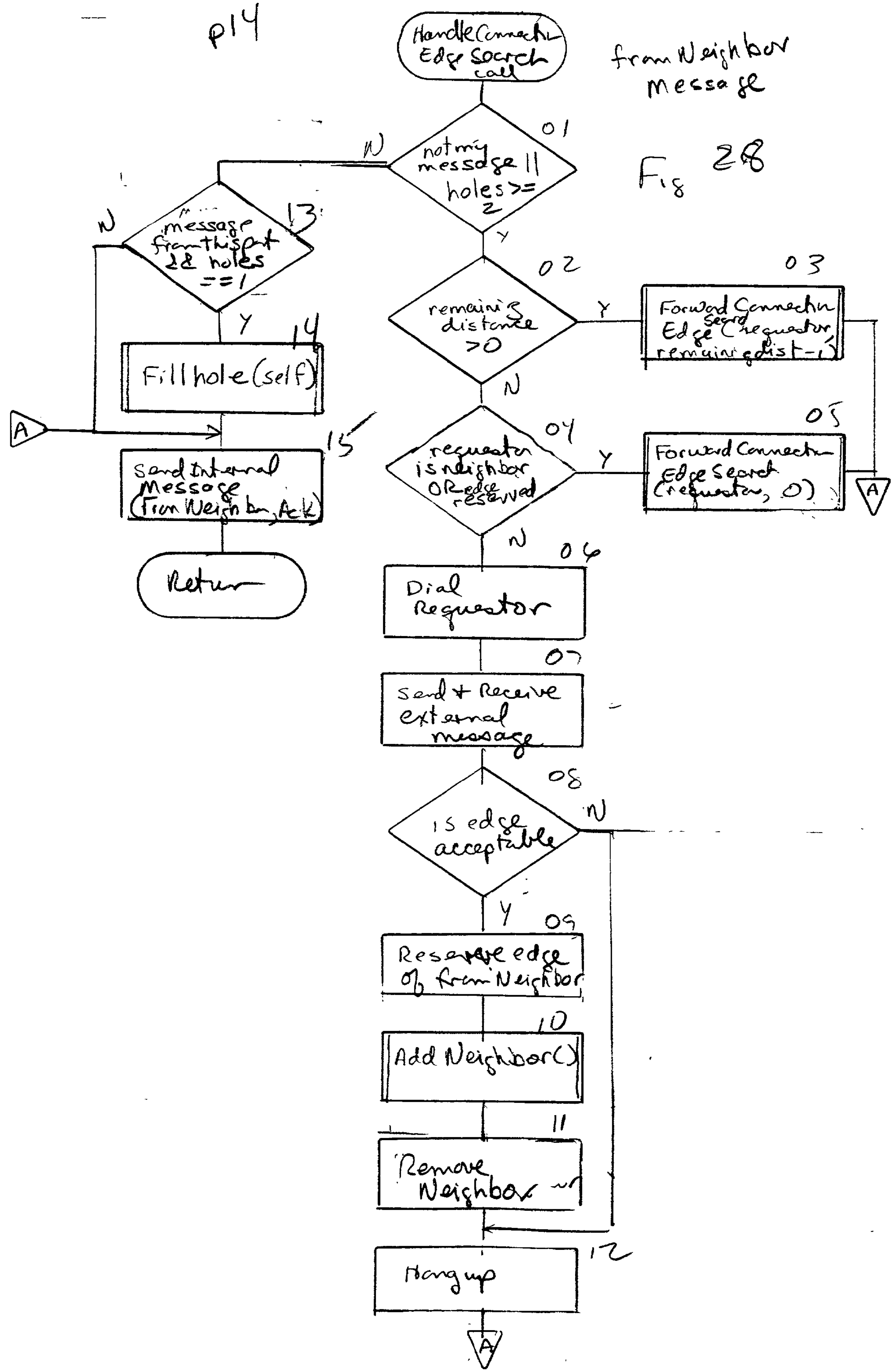


Fig 28

0025925.070301

15

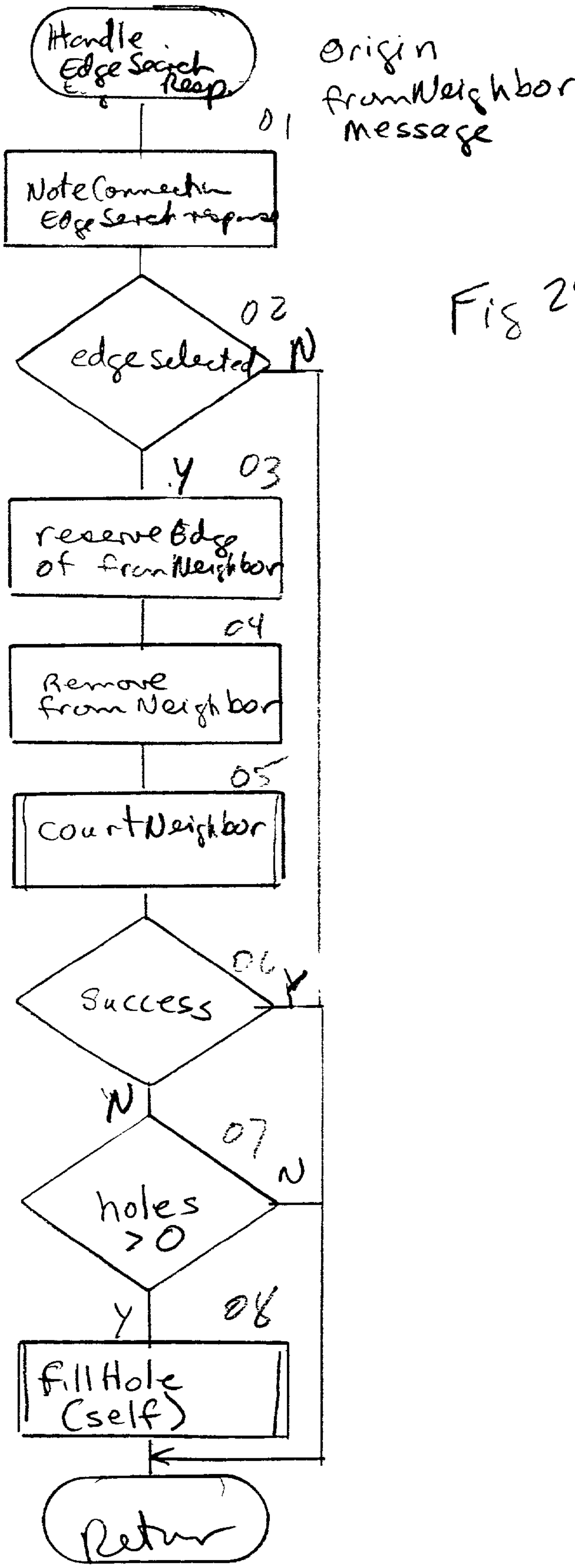

Handle EdgeSearch Resp
origin fromNeighbor message
01
NoteConnection EdgeSearch response
02
edge selected
N
Y
03
reserveEdge of fromNeighbor
04
Remove fromNeighbor
05
countNeighbor
06
Success
Y
N
07
holes >0
N
Y
08
fillHole (self)
Return

Fig 29

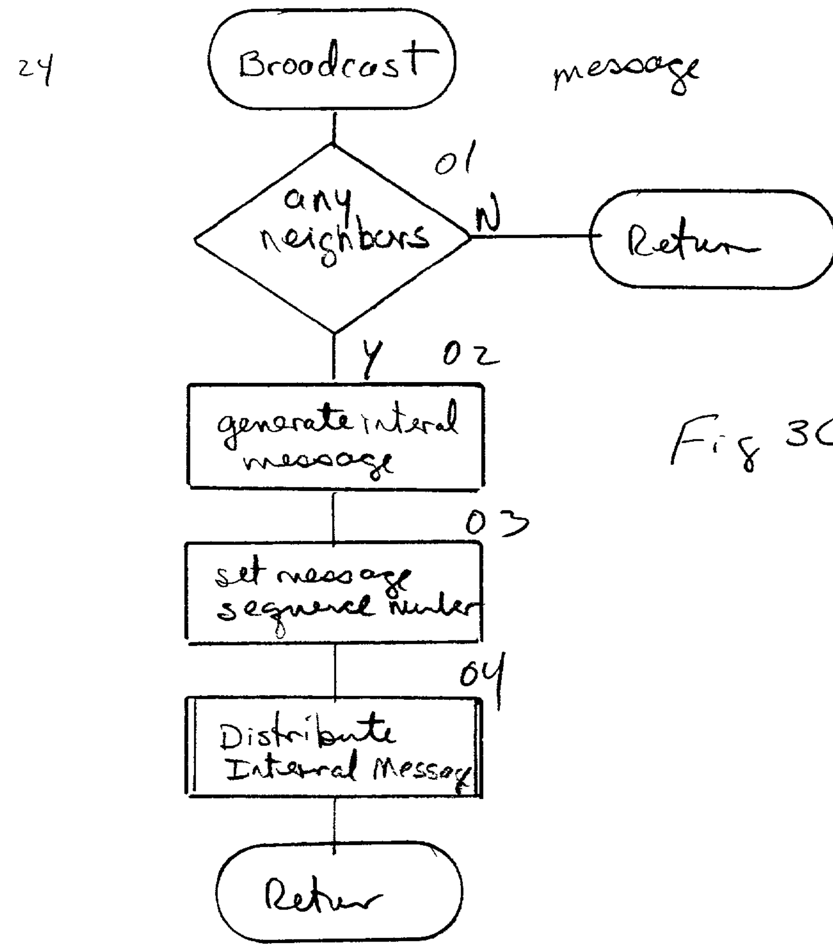
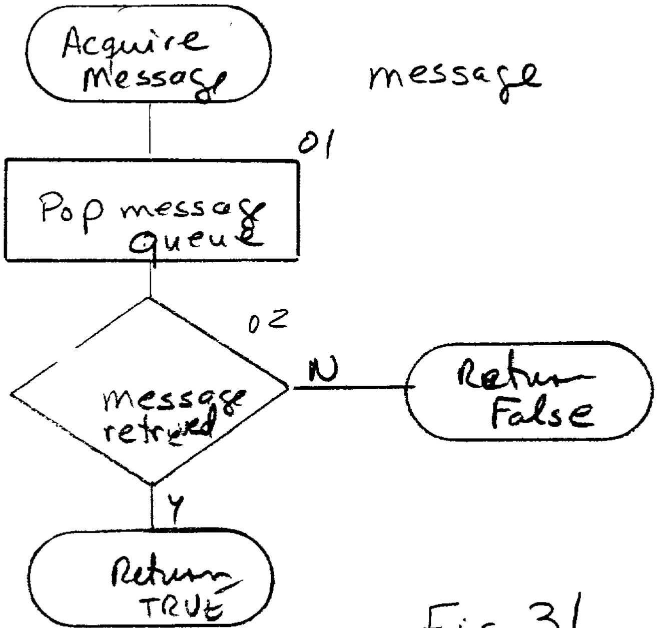

24
Broadcast message
01
any neighbors
N
Return
Y
02
generate internal message
03
set message sequence number
04
Distribute Internal Message
Return

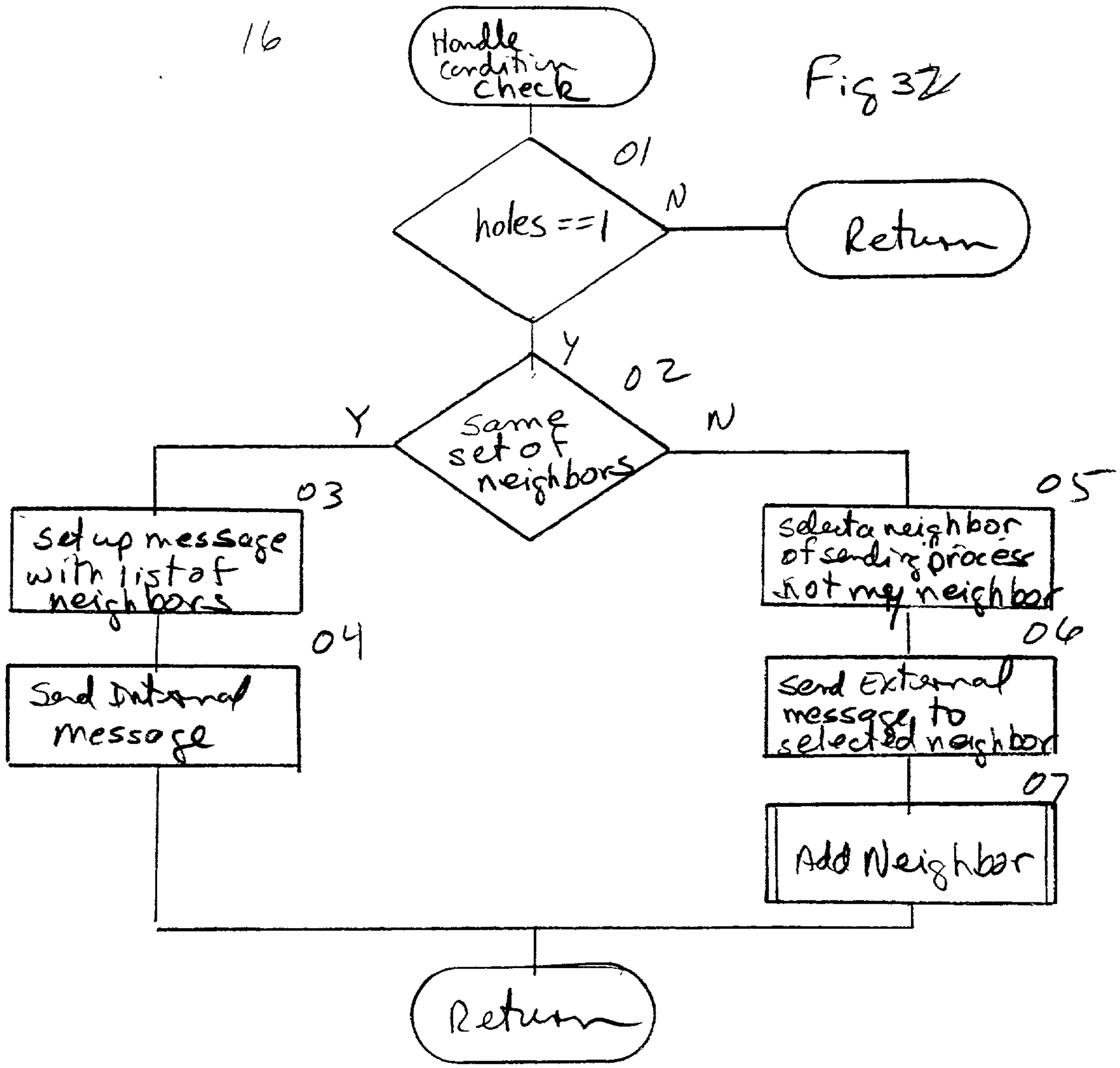
Fig 30

3



Acquire Message
message
01
Pop message queue
02
message retrieved
N
Return False
Y
Return TRUE

Fig 31



0096529D.D7D3DD

23

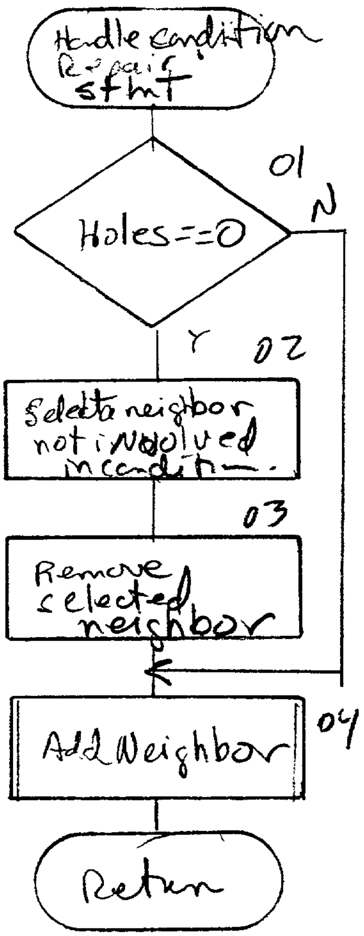
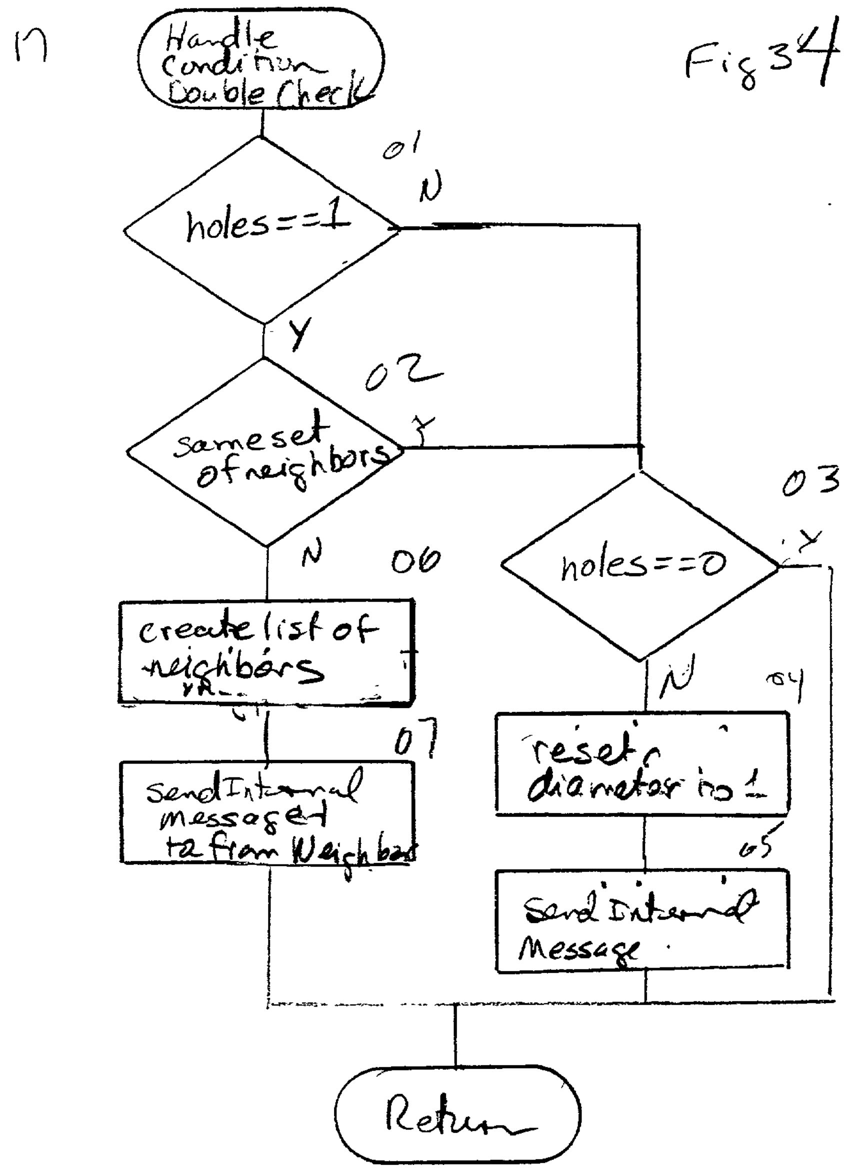


Fig 33

0039529570.072301

Fig 34



17
Handle Condition Double Check
01
N
holes==1
Y
02
sameset of neighbors
Y
N
03
holes==0
Y
06
create list of neighbors
N
04
reset diameter to 1
07
sendInternal message to from Neighbor
05
Send Internal Message
Return

SEARCHED

Class	Sub.	Date	Exmr.
709	221, 220, 223, 204, 238,	1/5/04	BE
370	225, 260, 400	1/6/04	BE
455	428	1/5/04	BE

SEARCH NOTES
(INCLUDING SEARCH STRATEGY)

	Date	Exmr.
Consulted Zarni Maung	10/17/03	NOW
EAST Text Search (see att)	1/2/04	BE
IEEE search (see att)	1/2/04	BE
Google Search (see att)	1/5/04	BE
Inventor Search (EAST)	1/5/04	BE
Consulted w/ Dang Dinh - search	1/5/04	BE

INTERFERENCE SEARCHED

Class	Sub.	Date	Exmr.

(RIGHT OUTSIDE)

ISSUE SLIP STAPLE AREA (for additional cross references)

POSITION	INITIALS	ID NO.	DATE
FEE DETERMINATION			
O.I.P.E. CLASSIFIER			8-16-00
FORMALITY REVIEW		71435	9/22/00
RESPONSE FORMALITY REVIEW			
			11/24/00

INDEX OF CLAIMS

✔	Rejected	N	Non-elected
=	Allowed	I	Interference
— (Through numeral)	Canceled	A	Appeal
÷	Restricted	O	Objected

Claim Final	Claim Original	Date 10/17/03	Date 5/5/04
	1	÷	✓
	2	÷	
	3	÷	
	4	÷	
	5	÷	
	6	÷	
	7	÷	
	8	÷	
	9	÷	
	10	÷	
	11	÷	
	12	÷	
	13	÷	
	14	÷	
	15	÷	
	16	÷	
	17	÷	
	18	÷	
	19	÷	
	20	÷	
	21	÷	
	22	÷	
	23	÷	
	24	÷	
	25	÷	
	26	÷	
	27	÷	
	28	÷	
	29	÷	
	30	÷	
	31	÷	
	32	÷	✓
	33	÷	
	34	÷	
	35	÷	
	36	÷	
	37	÷	
	38	÷	
	39	÷	
	40	÷	
	41	÷	
	42	÷	
	43	÷	
	44	÷	
	45	÷	
	46	÷	
	47	÷	
	48	÷	
	49	÷	
	50		

Claim Final	Claim Original	Date
	51	
	52	
	53	
	54	
	55	
	56	
	57	
	58	
	59	
	60	
	61	
	62	
	63	
	64	
	65	
	66	
	67	
	68	
	69	
	70	
	71	
	72	
	73	
	74	
	75	
	76	
	77	
	78	
	79	
	80	
	81	
	82	
	83	
	84	
	85	
	86	
	87	
	88	
	89	
	90	
	91	
	92	
	93	
	94	
	95	
	96	
	97	
	98	
	99	
	100	

Claim Final	Claim Original	Date
	101	
	102	
	103	
	104	
	105	
	106	
	107	
	108	
	109	
	110	
	111	
	112	
	113	
	114	
	115	
	116	
	117	
	118	
	119	
	120	
	121	
	122	
	123	
	124	
	125	
	126	
	127	
	128	
	129	
	130	
	131	
	132	
	133	
	134	
	135	
	136	
	137	
	138	
	139	
	140	
	141	
	142	
	143	
	144	
	145	
	146	
	147	
	148	
	149	
	150	

If more than 150 claims or 10 actions
staple additional sheet her

(LEFT INSIDE)

Please type a plus sign (+) inside this box [+] 08-02-00 PTO/SB/05 (4/98) A

UTILITY PATENT APPLICA[illegible]N TRANSMITTAL

(Only for nonprovisional applications under 37 CFR § 1.53(b))

Attorney Docket No.	030048[illegible]S
First Inventor or Application Identifie[illegible]	Virgil E. Bourassa
Title	JOINING A BROADCAST CHANNEL
Express Mail Label No.	EL404935398US

APPLICATION ELEMENTS

See MPEP chapter 600 concerning utility patent application contents.

ADDRESS TO: Box Patent Application, Assistant Commissioner for Patents, Washington, D.C. 20231

1. [ ] Authorization for Extensions & Fee Transmittal *(Submit an original and a duplicate for fee processing)*
2. [X] Specification *(preferred arrangement set forth below)* [Total Pages] **48**
   - Descriptive Title of the Invention
   - Cross References to Related Applications
   - Statement Regarding Fed sponsored R & D
   - Reference to Microfiche Appendix
   - Background of the Invention
   - Brief Summary of the Invention
   - Brief Description of the Drawings *(if filed)*
   - Detailed Description
   - Claim(s)
   - Abstract of the Disclosure
3. [X] Drawing(s) *(35 USC 113)* [Total Sheets] **39**
4. Oath or Declaration [Total Pages] ____
   a. [ ] Newly executed (original or copy)
   b. [ ] Copy from a prior application (37 CFR 1.63(d)) *(for continuation/divisional with Box 16 completed)*
      i. [ ] DELETION OF INVENTOR(S) Signed statement attached deleting inventor(s) named in the prior application, see 37 CFR 1.63(d)(2) and 1.33(b)

**NOTE FOR ITEMS 1 & 13: IN ORDER TO BE ENTITLED TO PAY SMALL ENTITY FEES, A SMALL ENTITY STATEMENT IS REQUIRED (37 C.F.R. § 1.27), EXCEPT IF ONE FILED IN A PRIOR APPLICATION IS RELIED UPON (37 C.F.R. § 1.28).*

5. [ ] Microfiche Computer Program *(Appendix)*
6. Nucleotide and/or Amino Acid Sequence Submission *(if applicable, all necessary)*
   a. [ ] Computer-Readable Copy
   b. [ ] Paper Copy (identical to computer copy)
   c. [ ] Statement verifying identity of above copies

ACCOMPANYING APPLICATION PARTS

7. [ ] Assignment Papers (cover sheet & document(s))
8. [ ] 37 CFR 3.73(b) Statement *(when there is an assignee)* [ ] Power of Attorney
9. [ ] English Translation Document *(if applicable)*
10. [ ] Information Disclosure Statement (IDS)/PTO-1449 [ ] Copies of IDS Citations
11. [ ] Preliminary Amendment
12. [X] Return Receipt Postcard
12. [ ] Small Entity Statement(s) [ ] Statement filed in prior application, Status still proper and desired
14. [ ] Certified Copy of Priority Document(s) *(if foreign priority is claimed)*
15. [ ] Other: ____________________

16. If a CONTINUING APPLICATION, *check appropriate box and supply the requisite information below and in a preliminary amendment*

[ ] Continuation [ ] Divisional [ ] Continuation-In-Part (CIP) of prior Application No.: ____________

Prior application information: Examiner ____________ *Group / Art Unit* ____________

For CONTINUATION or DIVISIONAL apps only: The entire disclosure of the prior application, from which an oath or declaration is supplied under Box 4b, is considered a part of the disclosure of the accompanying continuation or divisional application and is hereby incorporated by reference. The incorporation can only be relied upon when a portion has been inadvertently omitted from the submitted application parts.

[ ] Claims the benefit of Provisional Application No. ____________

17. CORRESPONDENCE ADDRESS

Customer Number 25096 / Barcode

25096

PATENT TRADEMARK OFFICE

Respectfully submitted,

TYPED or PRINTED NAME Maurice J. Pirio REGISTRATION NO. 33,273

SIGNATURE *Maurice J Pirio* Date *July 31, 2000*

ISSUE CLASSIFICATION | Class

09/629570

PATENT NUMBER

U.S. UTILITY Patent Application

O.I.P.E.	PATENT DATE
KW SCANNED Am Q.A. A.G.	

APPLICATION NO.	CONT/PRIOR	CLASS	SUBCLASS	ART UNIT	EXAMINER
09/629570		~~455~~ 709	~~200~~ 221	~~2681~~ ~~2152~~ 2153	Edelman ~~M. Winters~~

APPLICANTS: Fred Holt, Virgil Bourassa

TITLE: Joining a broadcast channel

PTO-2040
12/99

ISSUING CLASSIFICATION

ORIGINAL		CROSS REFERENCE(S)						
CLASS	SUBCLASS	CLASS	SUBCLASS (ONE SUBCLASS PER BLOCK)					
INTERNATIONAL CLASSIFICATION								
				☐ Continued on Issue Slip Inside File Jacket				

☐ TERMINAL DISCLAIMER	DRAWINGS			CLAIMS ALLOWED	
	Sheets Drwg.	Figs. Drwg.	Print Fig.	Total Claims	Print Claim for O.G.
☐ The term of this patent subsequent to ________ (date) has been disclaimed.	(Assistant Examiner) (Date)			NOTICE OF ALLOWANCE MAILED	
☐ The term of this patent shall not extend beyond the expiration date of U.S Patent. No. ________	(Primary Examiner) (Date)			ISSUE FEE: Amount Due	Date Paid
☐ The terminal ____months of this patent have been disclaimed.	(Legal Instruments Examiner) (Date)			ISSUE BATCH NUMBER	

WARNING:
The information disclosed herein may be restricted. Unauthorized disclosure may be prohibited by the United States Code Title 35, Sections 122, 181 and 368. Possession outside the U.S. Patent & Trademark Office is restricted to authorized employees and contractors only.

Form PTO-436A
(Rev. 6/99)

FILED WITH: ☐ DISK (CRF) ☐ FICHE ☐ CD-ROM
(Attached in pocket on right inside flap)

(FACE)

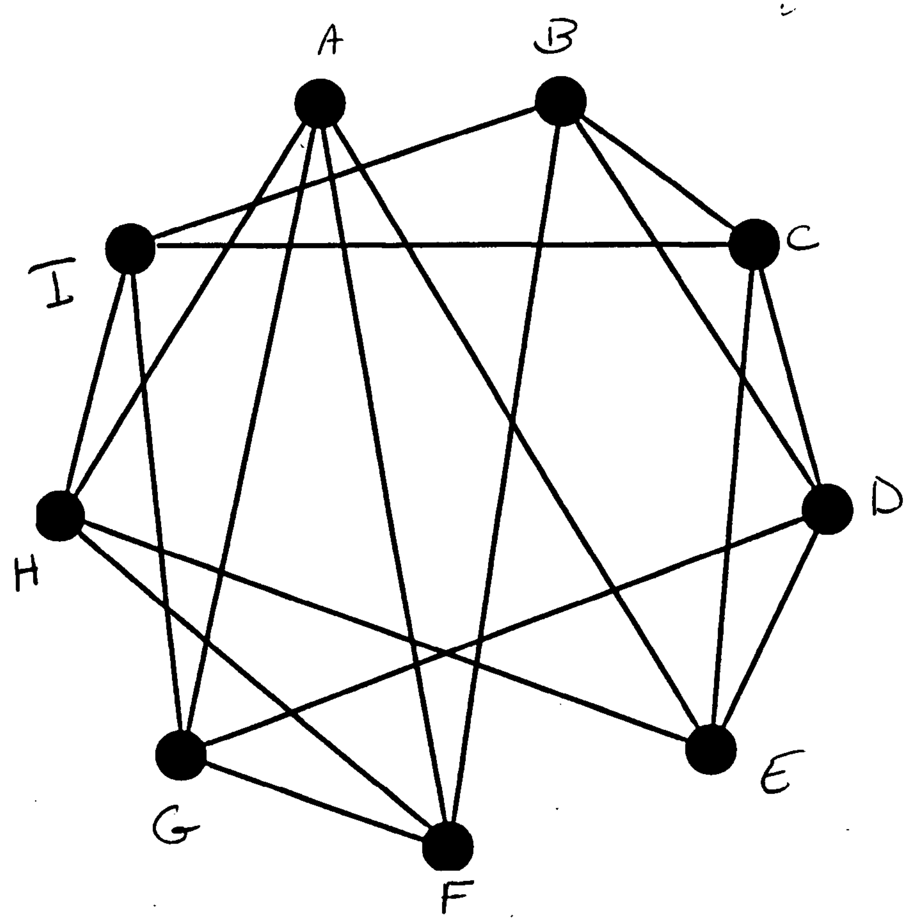


Fig 1.

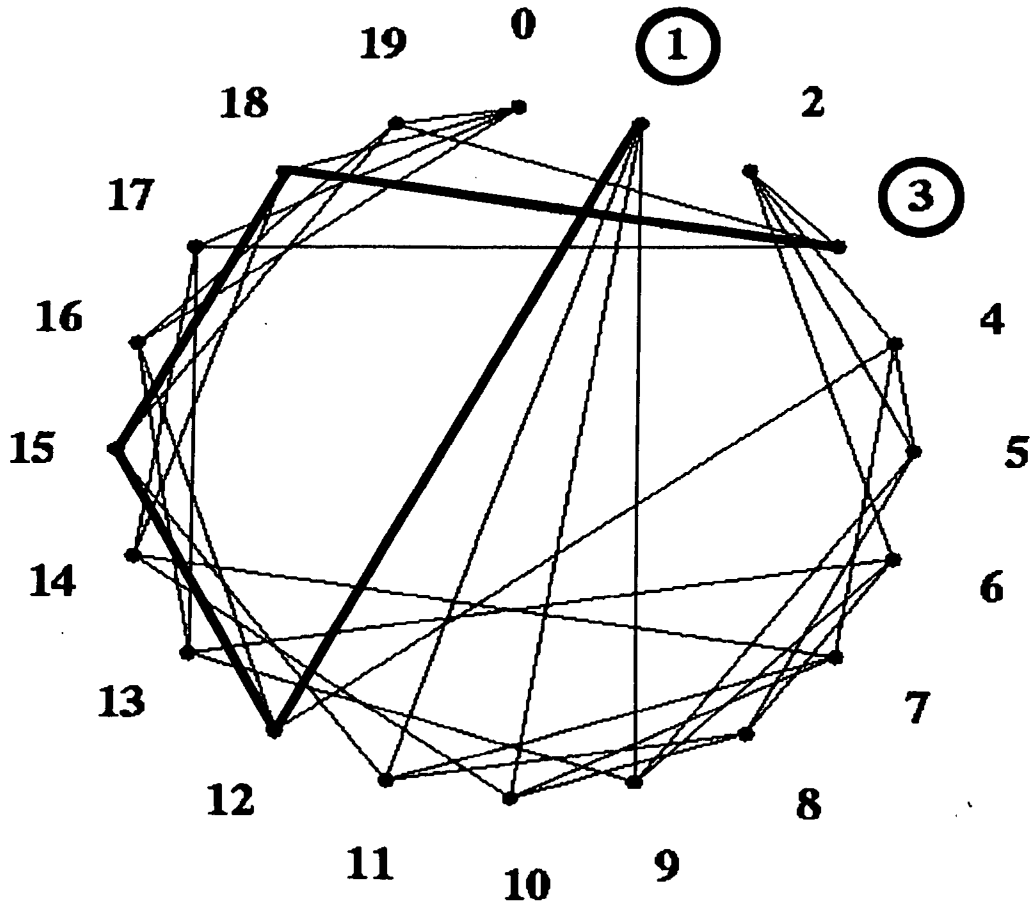


Fig 2

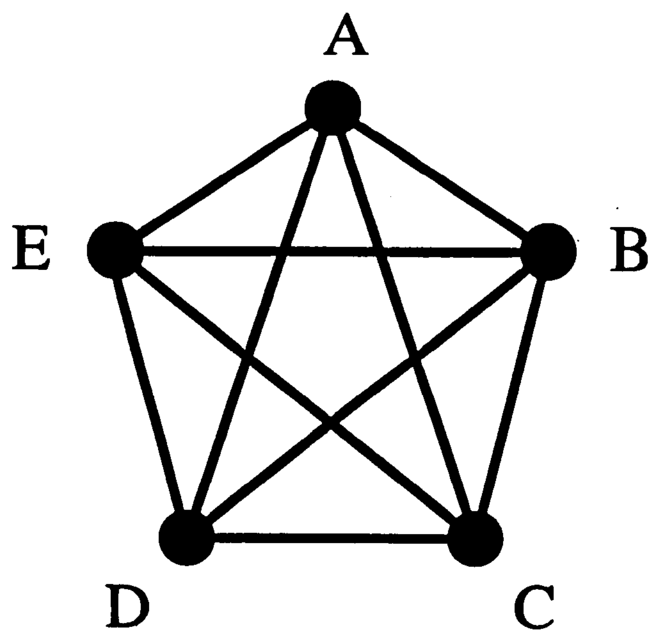


Fig 3A

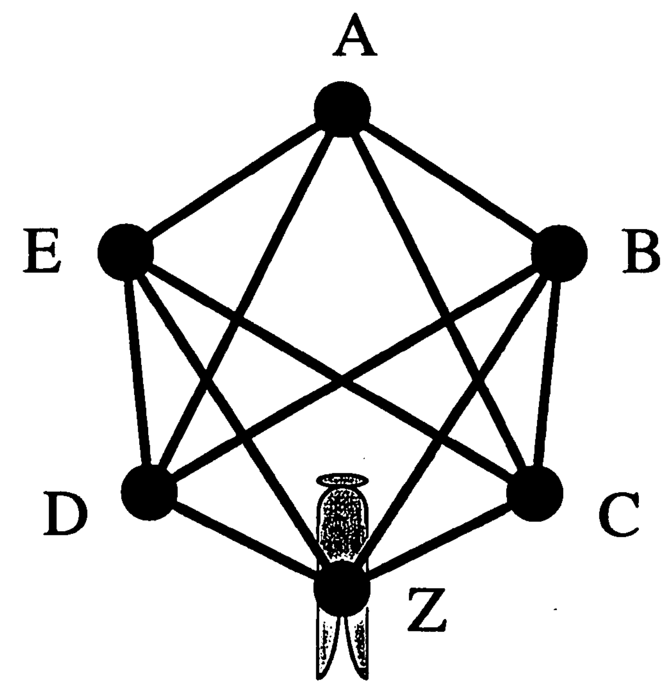


Fig 3B

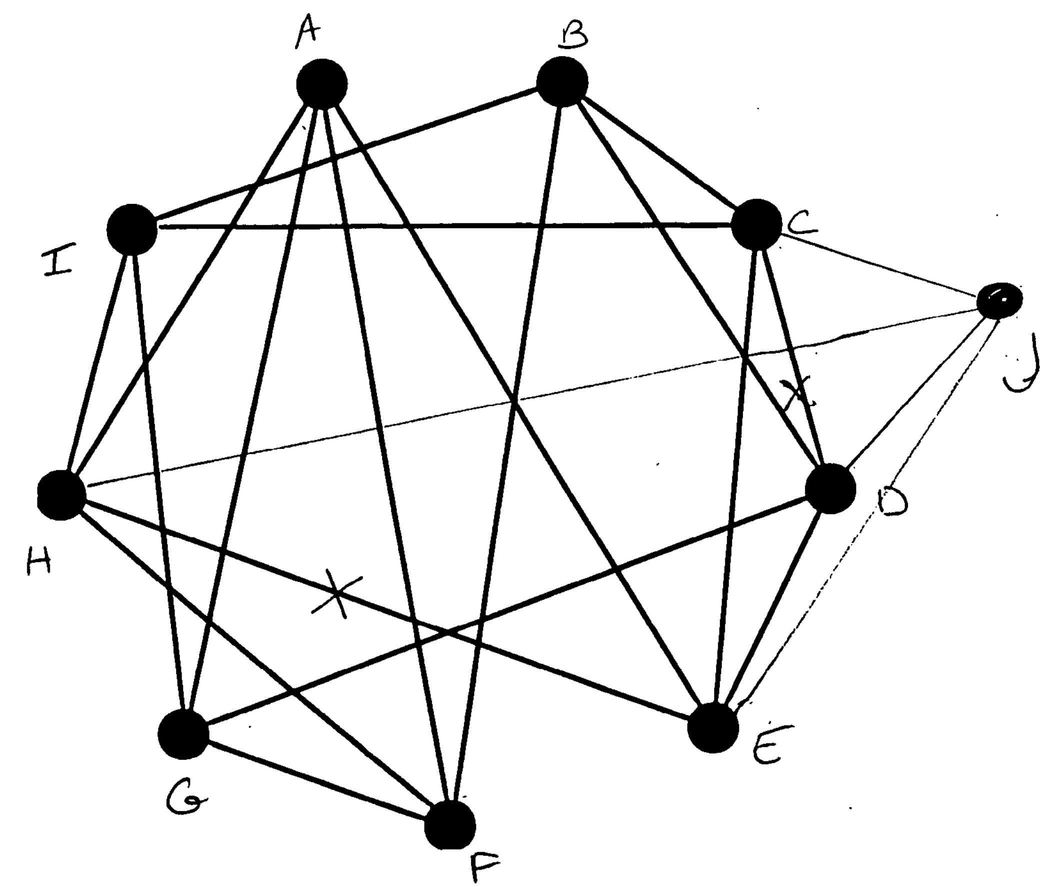


Fig 4A

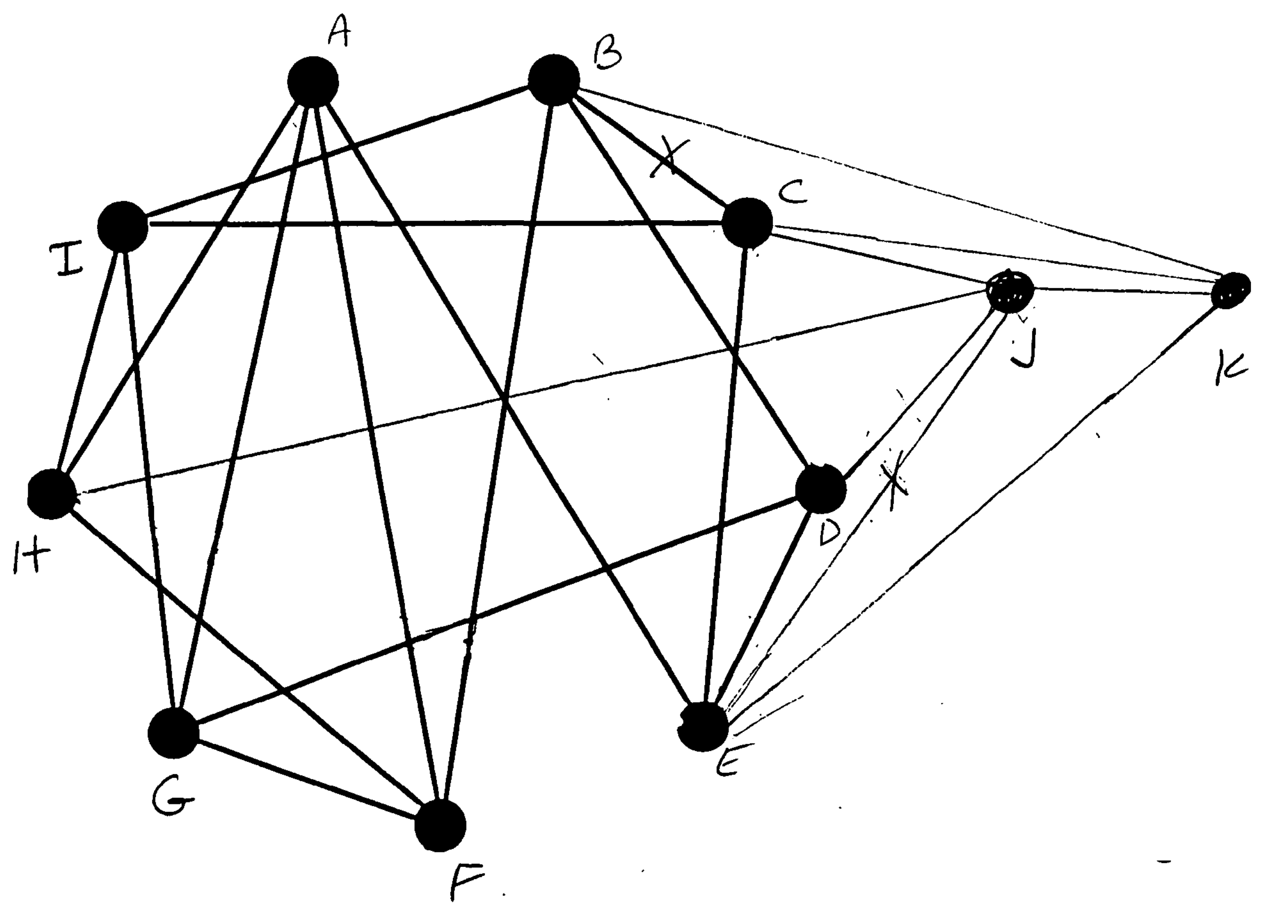


Fig 4B

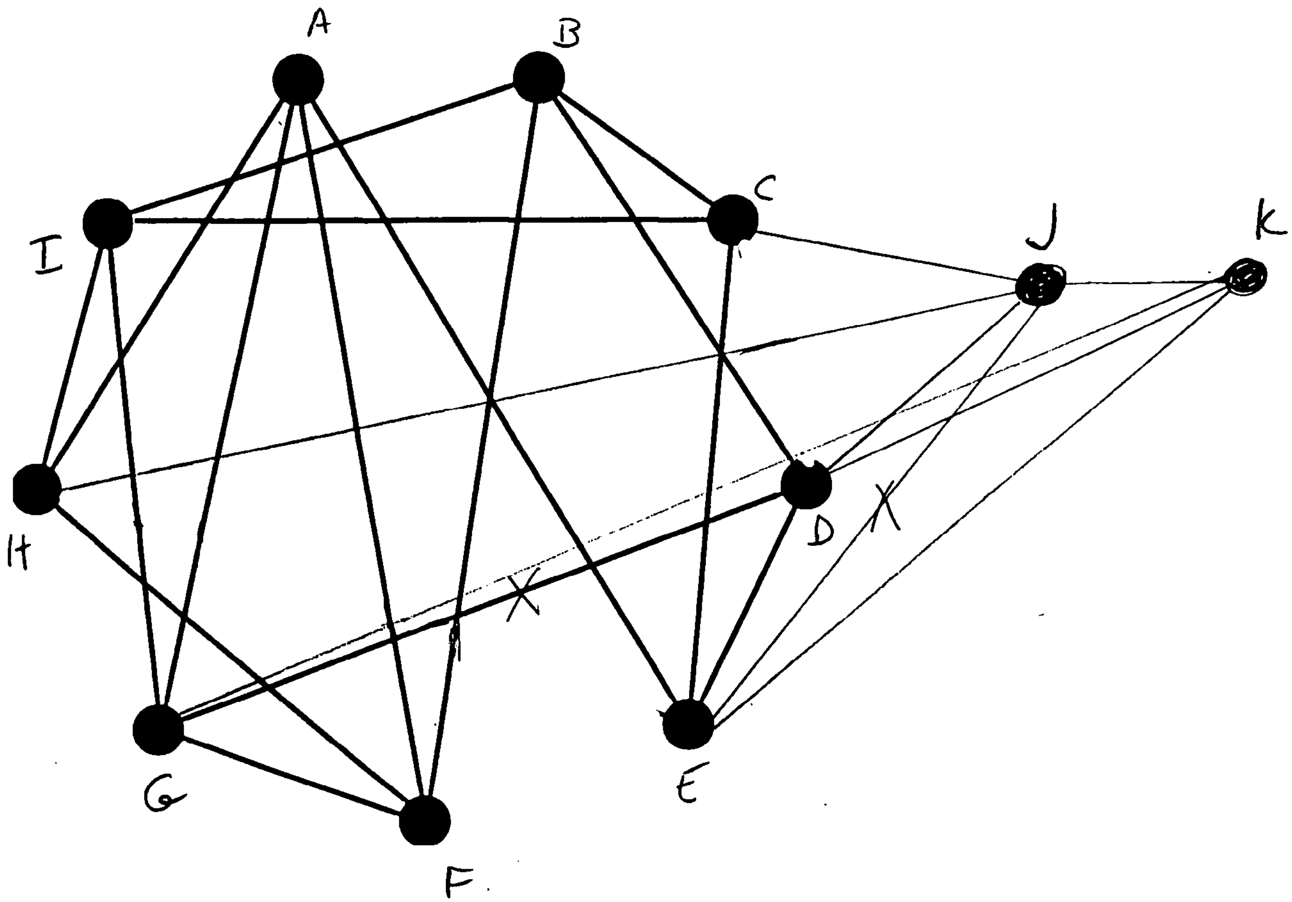


Fig 4C

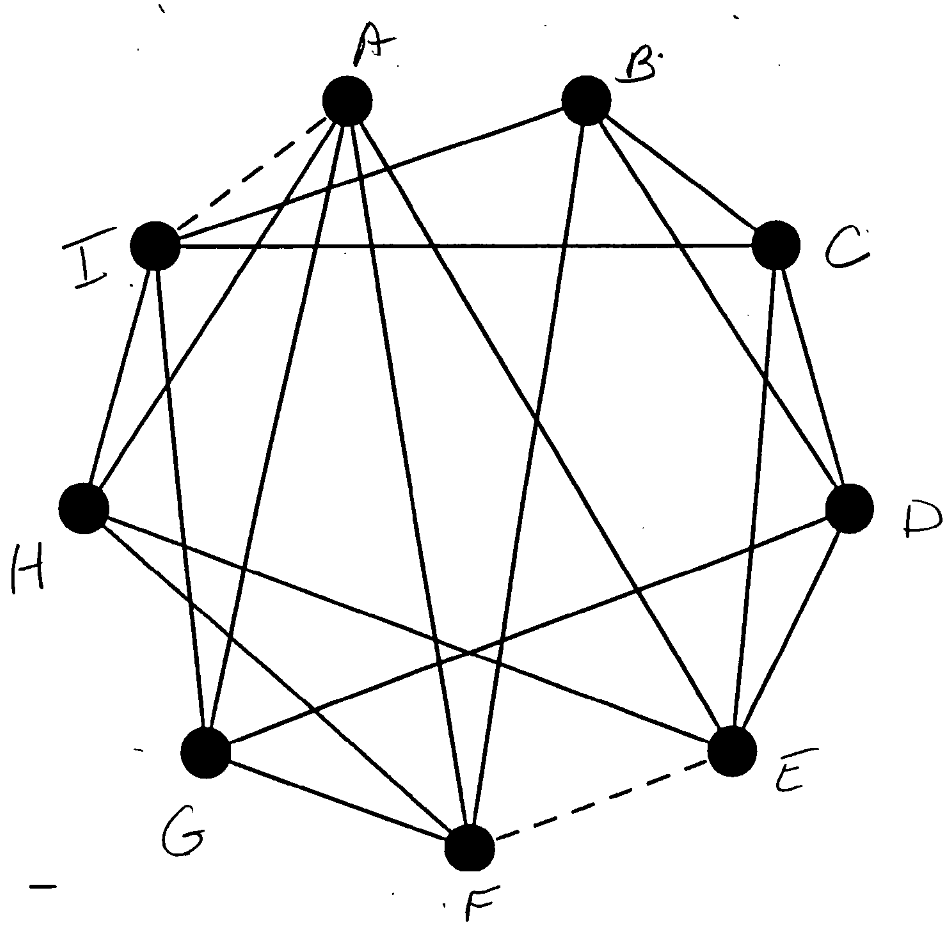


Fig 5A

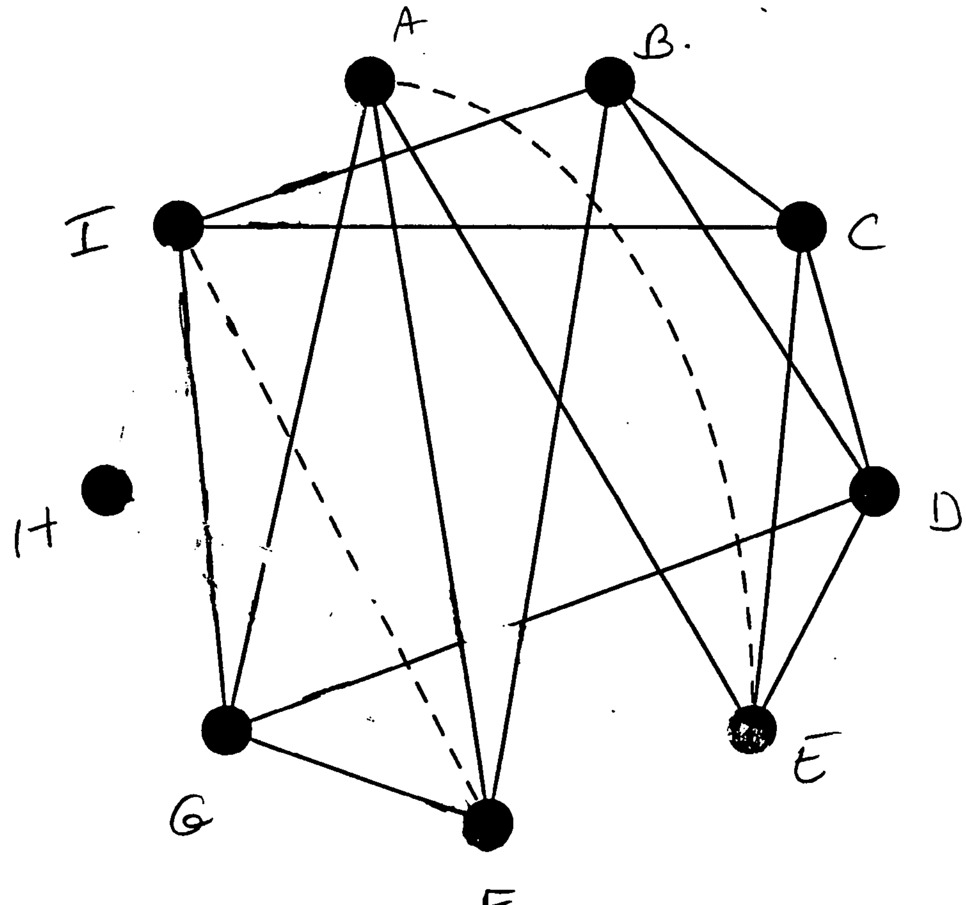


Fig 5B

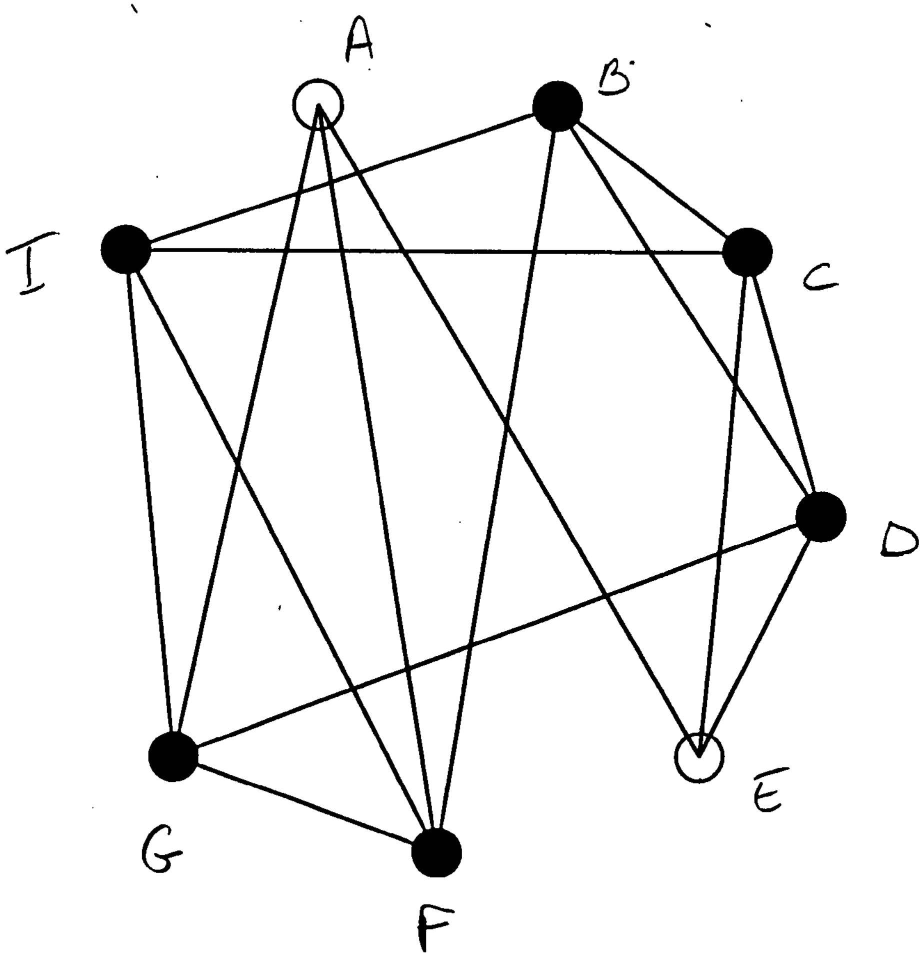


Fig 5C

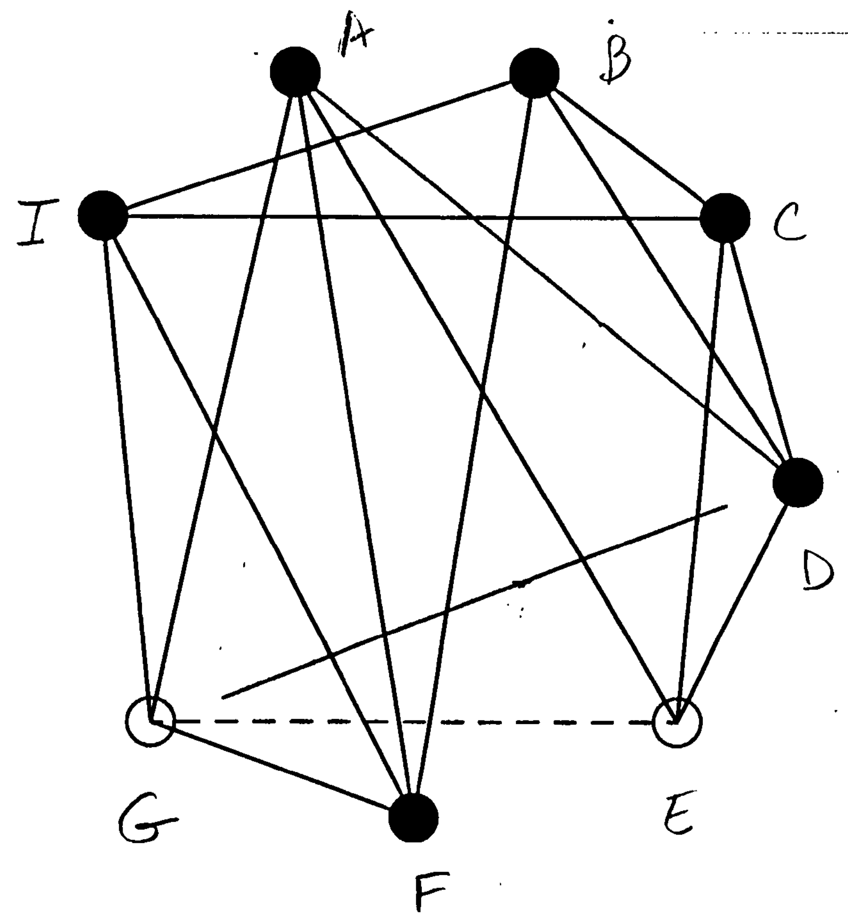


Fig 5D

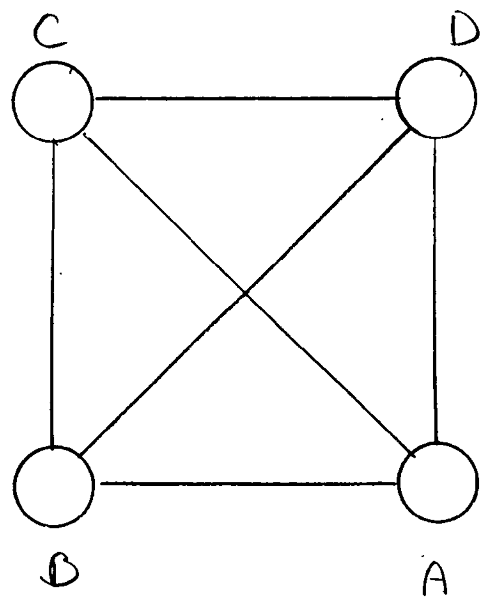


Fig 5E

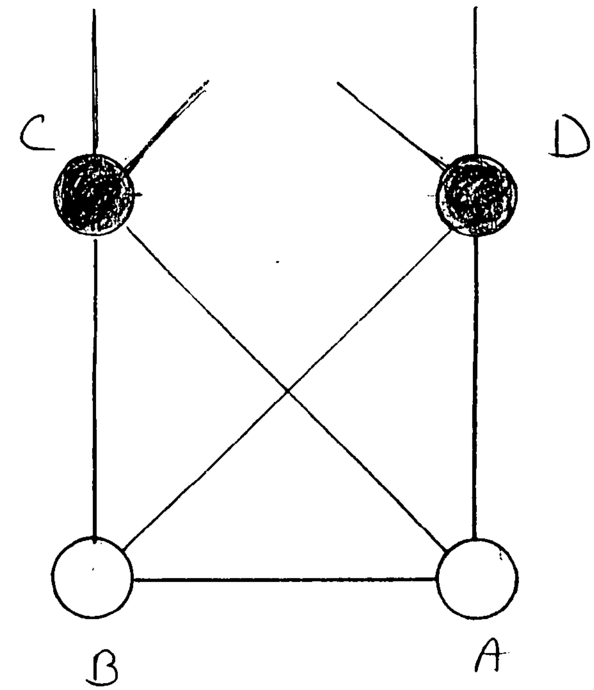


Fig 5F

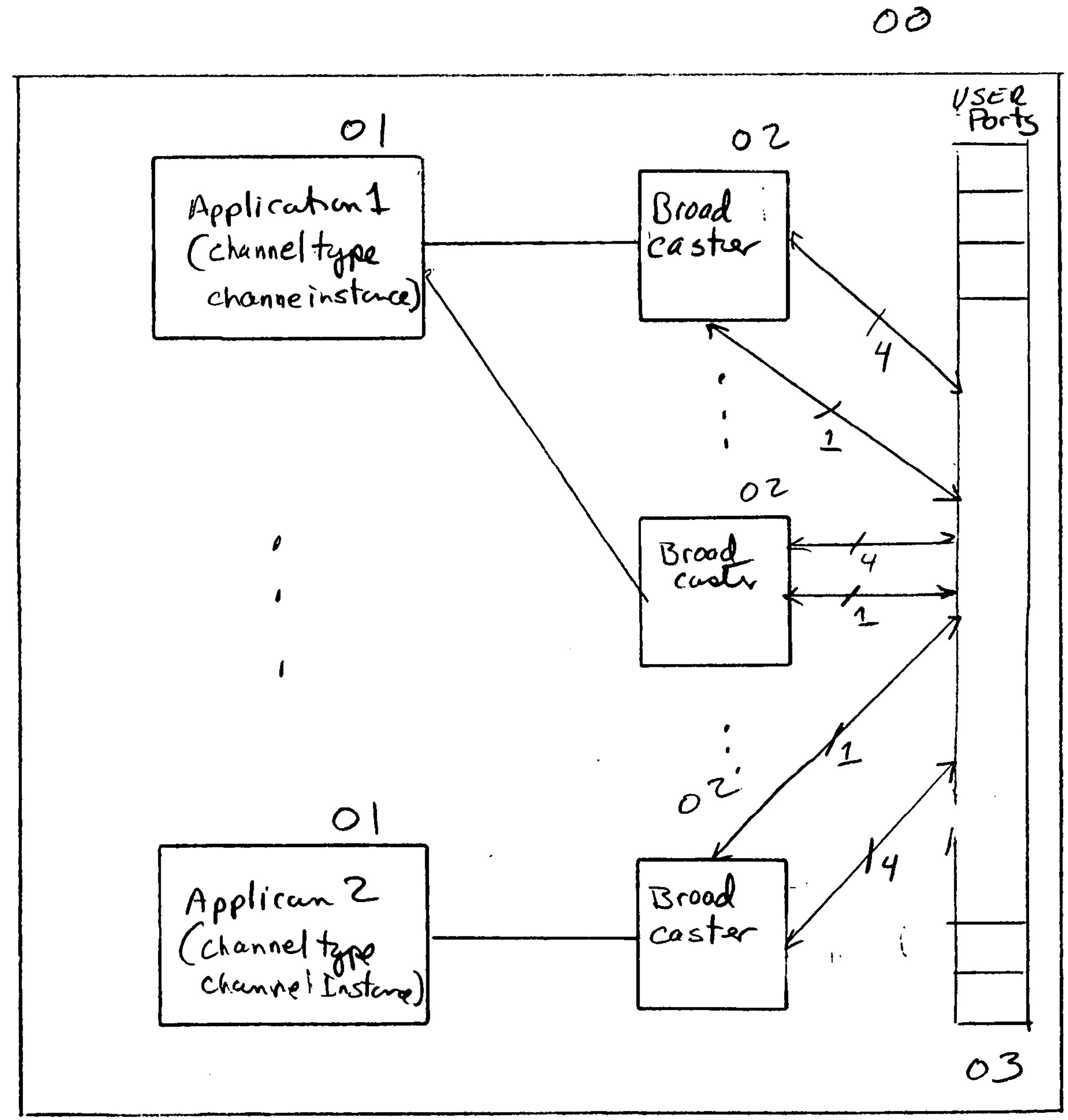

00
01
Application 1
(channel type
channe instance)
01
Applican 2
(channel type
channel Instance)
02
Broad caster
02
Broad caster
02
Broad caster
4
1
4
1
1
4
USER Ports
03

Fig 6

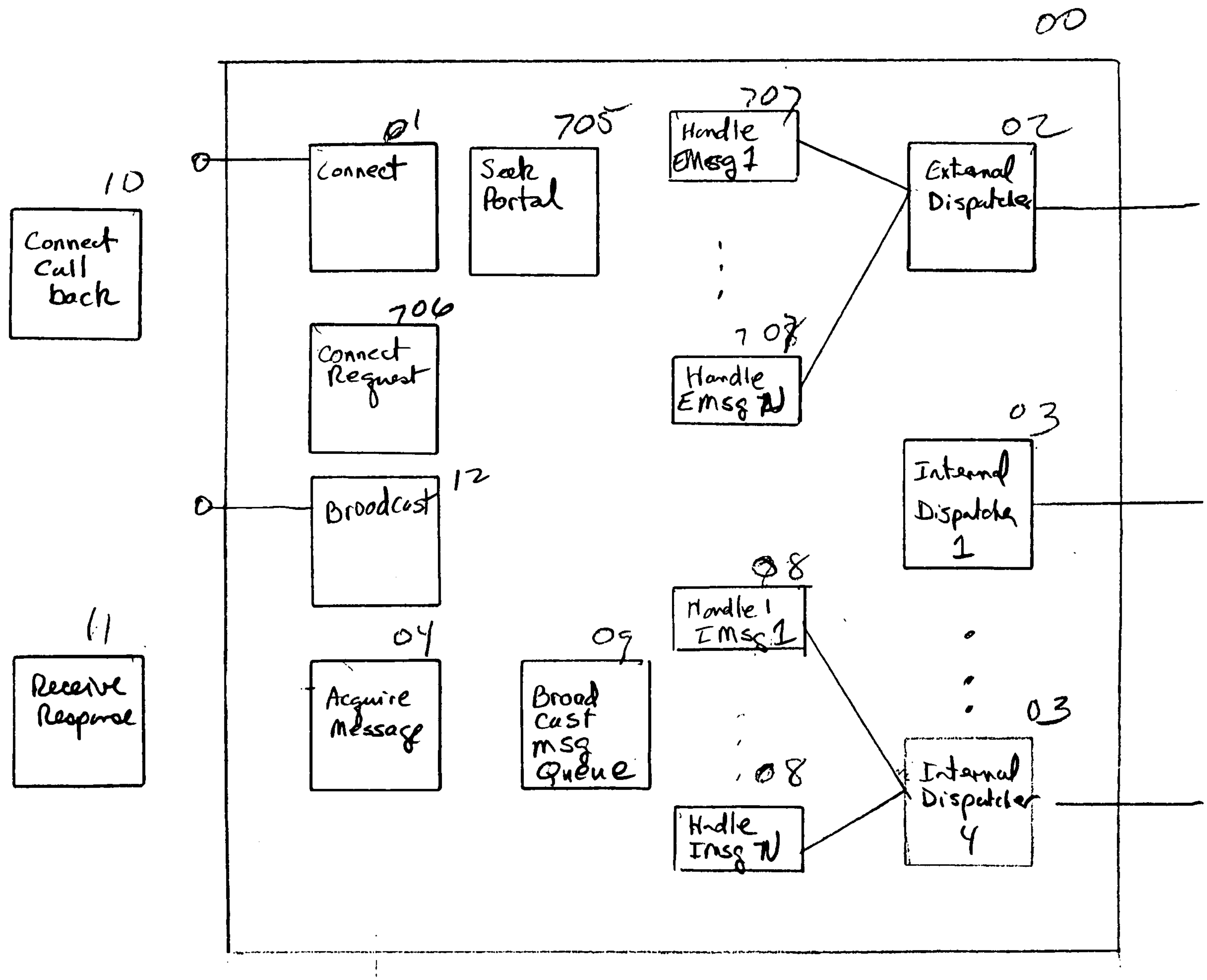

00
Connect Call back
10
Connect
01
Seek Portal
705
Handle EMsg 1
707
Handle EMsg N
708
External Dispatcher
02
Connect Request
706
Broadcast
12
Internal Dispatcher 1
03
Receive Response
11
Acquire Message
04
Broad cast msg Queue
09
Handle I IMsg 1
08
Handle IMsg N
08
Internal Dispatcher 4
03

Figure 7

p2

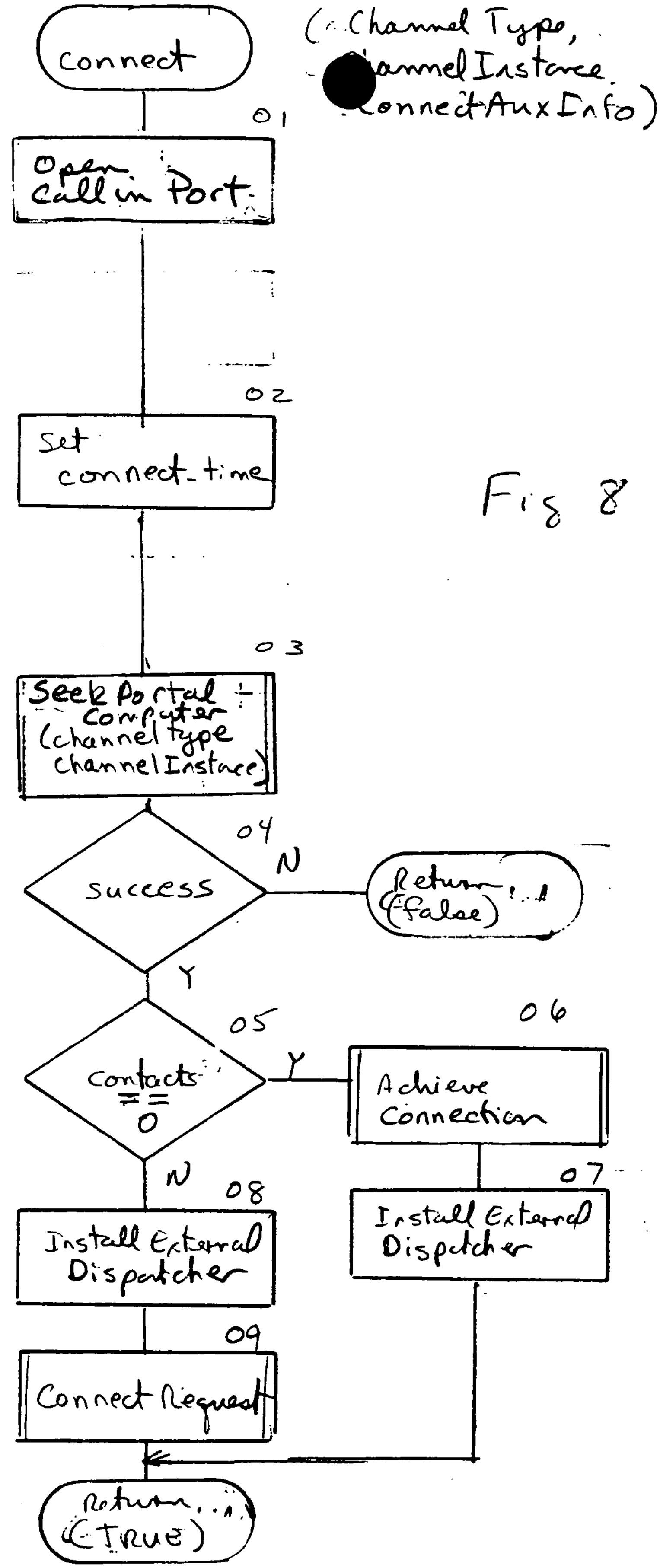
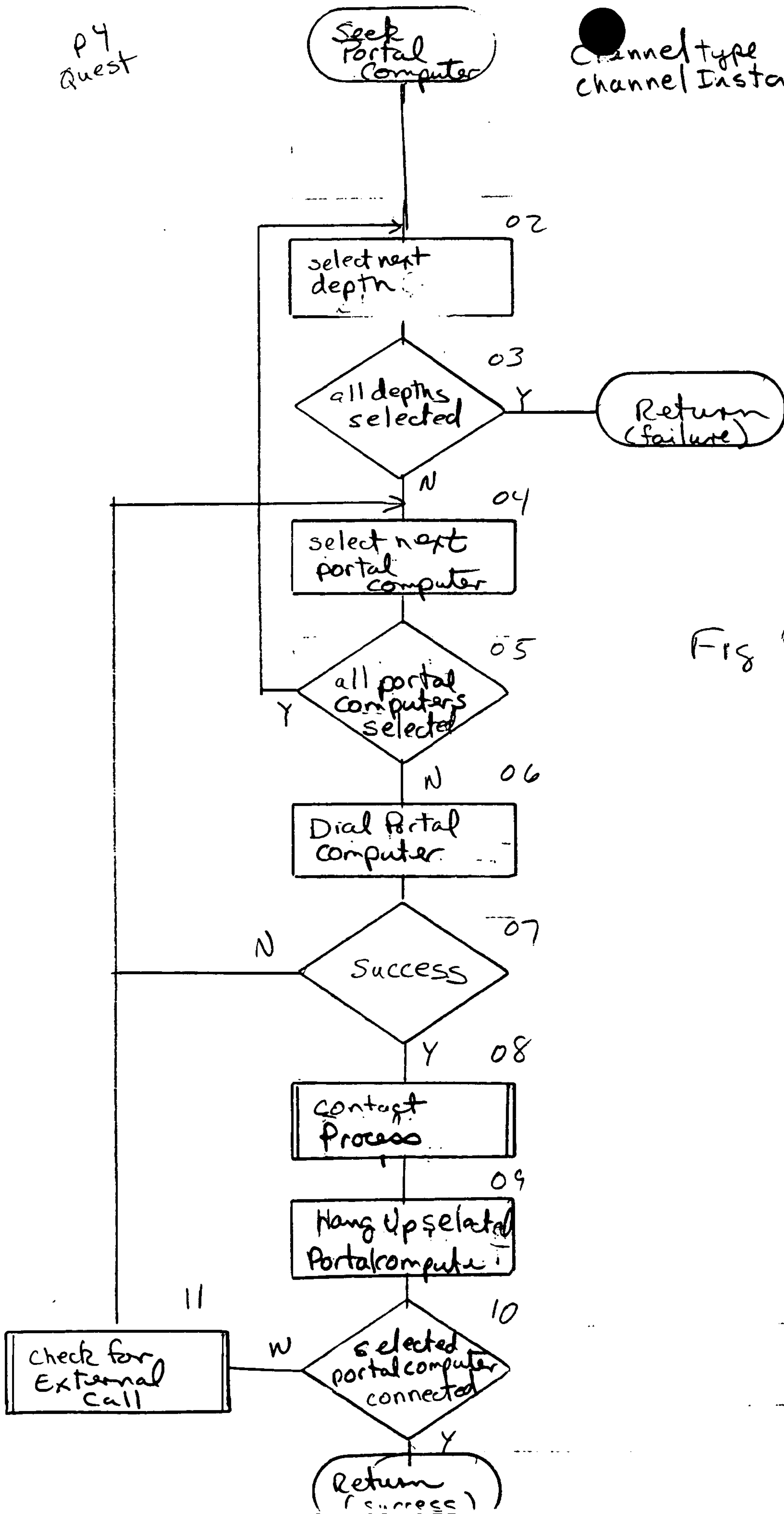

Connect
(ChannelType, ChannelInstance, ConnectAuxInfo)
01
Open Call in Port
02
Set connect_time
03
Seek Portal Computer (channelType ChannelInstance)
04
success
N
Return (false)
Y
05
contacts == 0
Y
06
Achieve Connection
07
Install External Dispatcher
N
08
Install External Dispatcher
09
Connect Request
Return (TRUE)

Fig 8

P4
Quest



Channel type
channel Instance

Fig 9

09625670.073001

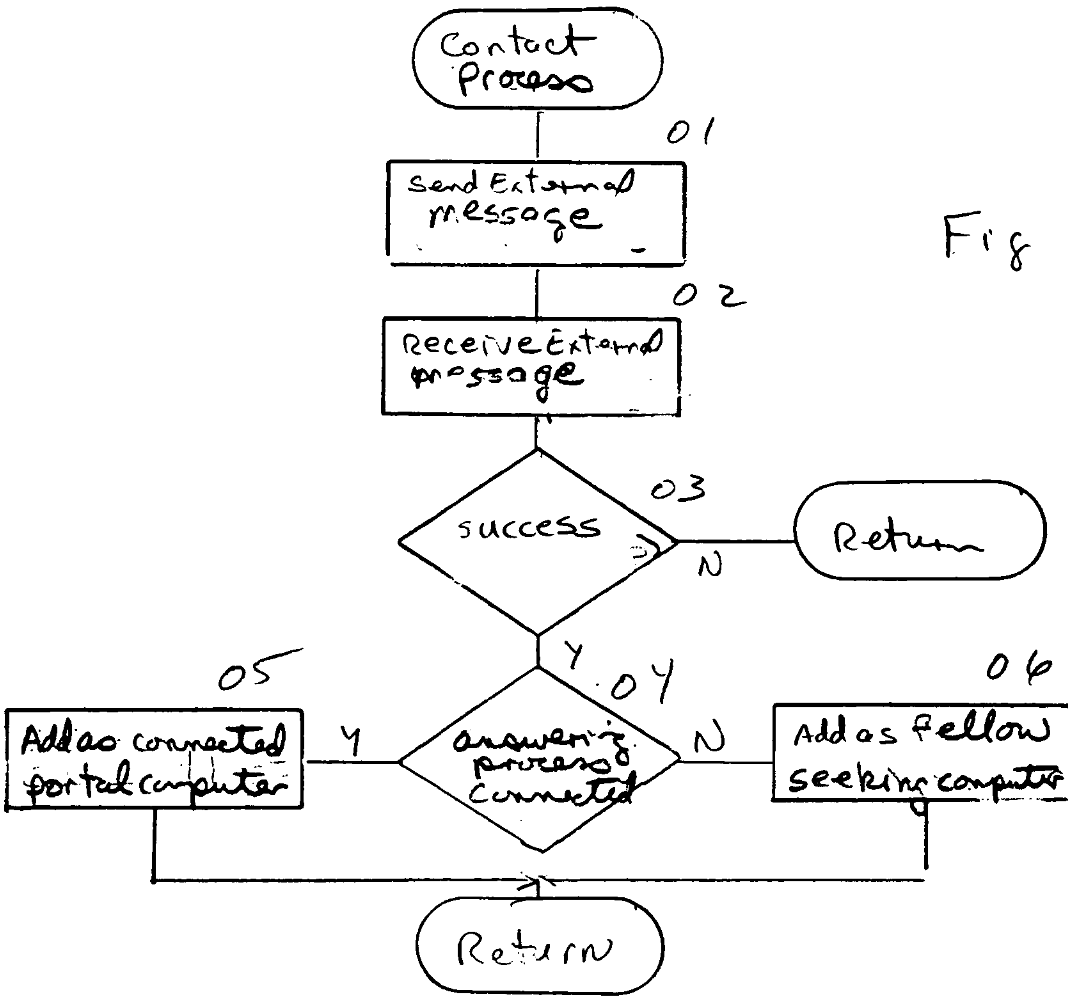

Contact Process
01
Send External message
02
Receive External message
03
success
N
Return
Y
04
answering process connected
05
Y
Add as connected portal computer
N
06
Add as fellow seeking computer
Return

Fig 10

p23

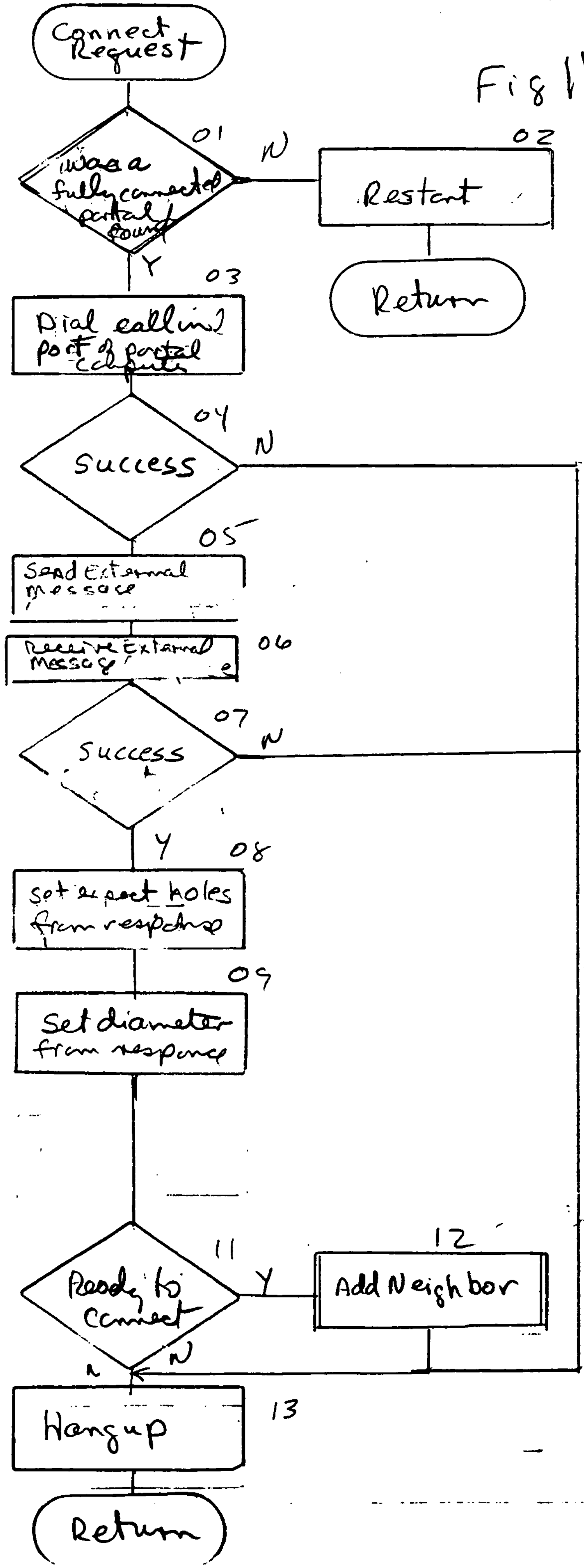
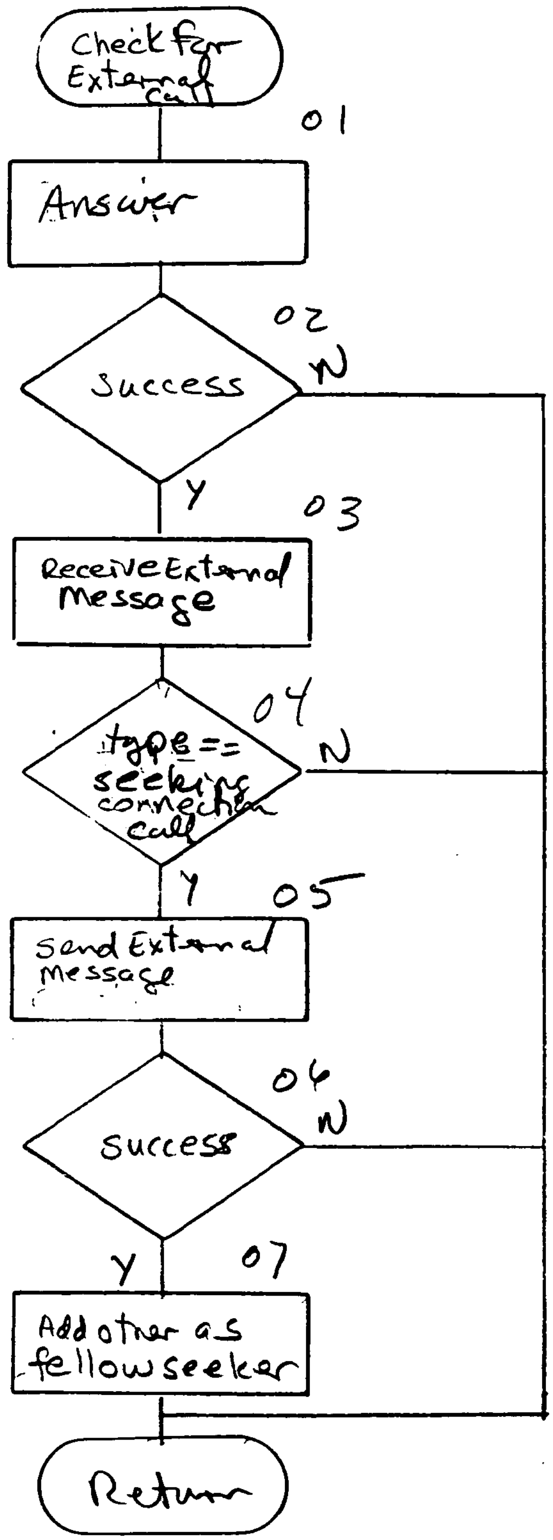


Fig 11

DOTE70"0Z562960



Check for External call
01
Answer
02
Success
N
Y
03
Receive External Message
04
type == seeking connection call
N
Y
05
Send External Message
06
Success
N
Y
07
Add other as fellow seeker
Return

Fig 12

Achieve Nirvana 22

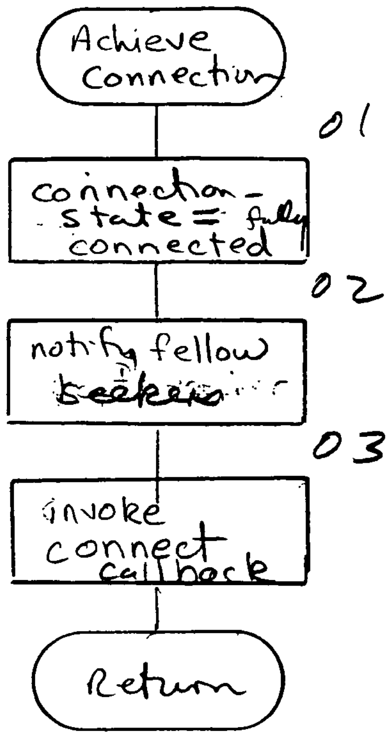
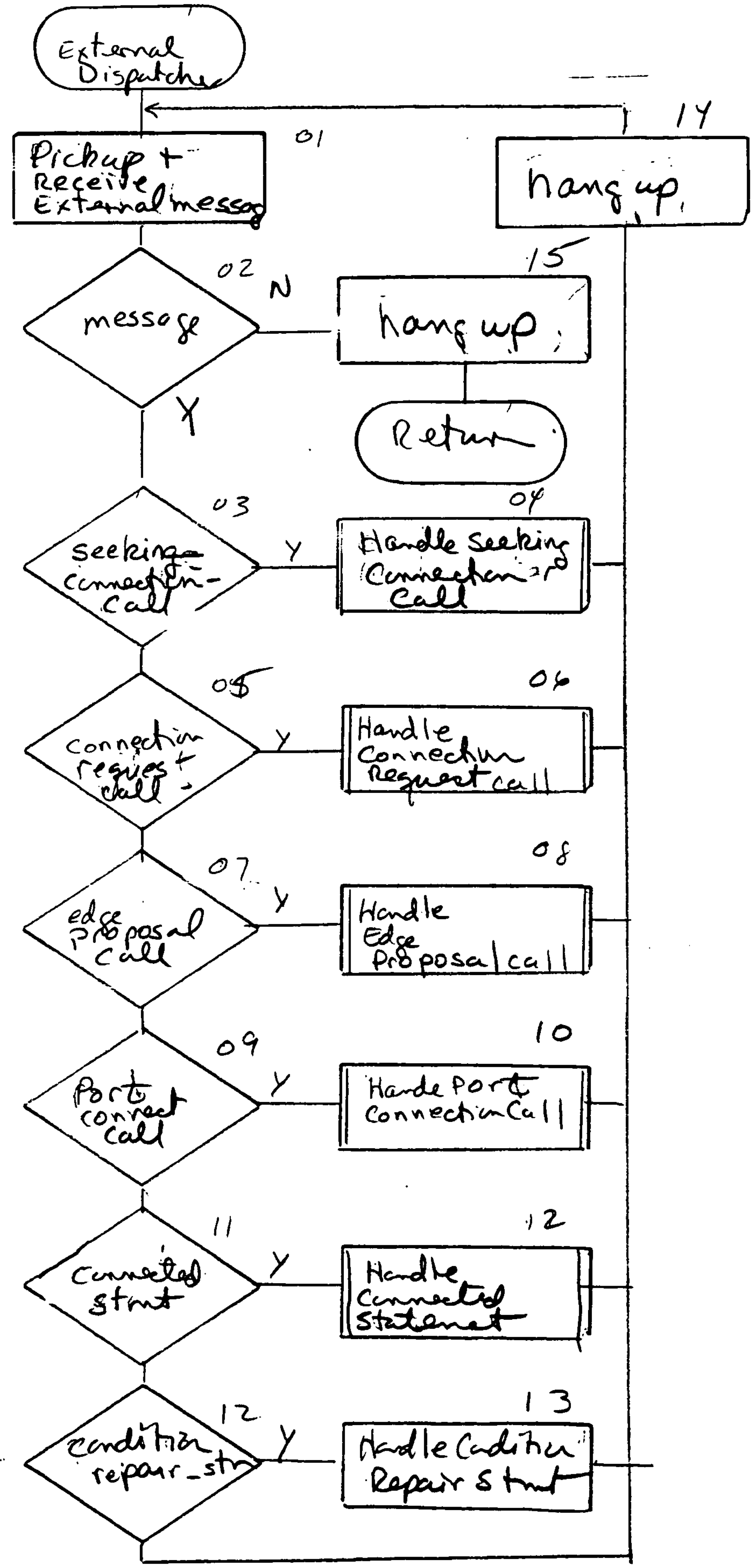


Fig 13

p7

Fig 14


External Dispatcher
01
Pickup + Receive External message
14
hang up
02
message
N
15
hang up
Y
Return
03
seeking connection call
Y
04
Handle Seeking Connection Call
05
connection request call
Y
06
Handle Connection Request call
07
edge proposal call
Y
08
Handle Edge Proposal call
09
Port connect call
Y
10
Handle Port Connection Call
11
Connected stmt
Y
12
Handle Connected Statement
12
condition repair_stmt
Y
13
Handle Condition Repair Stmt

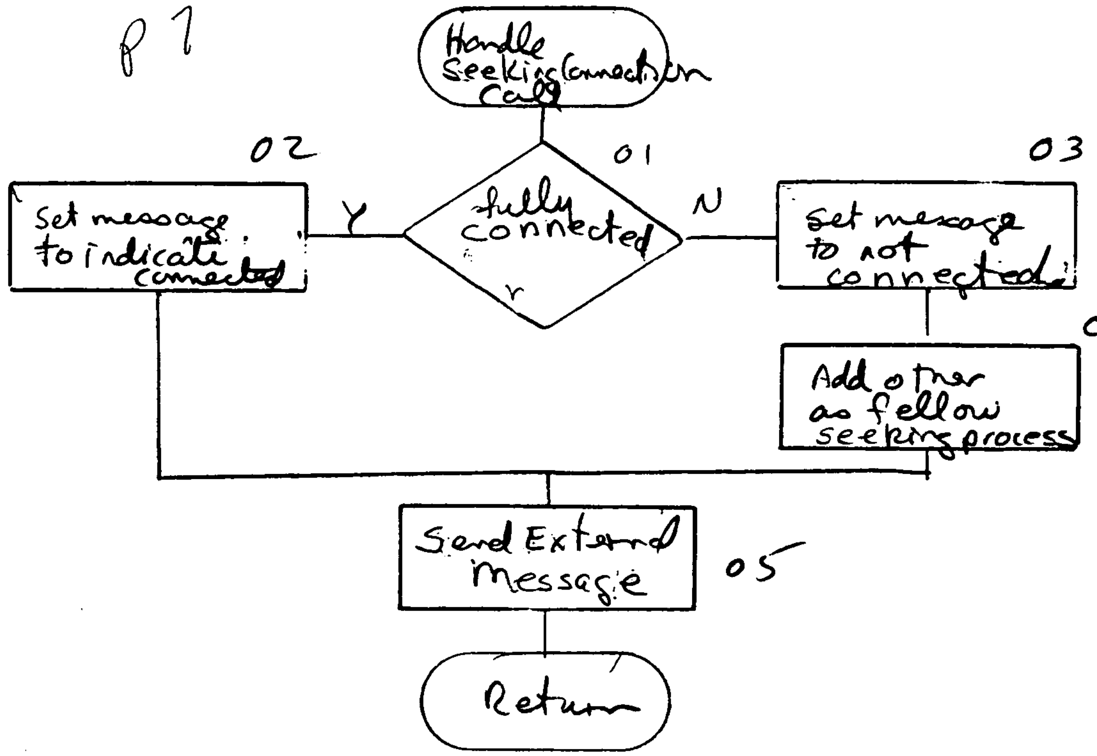
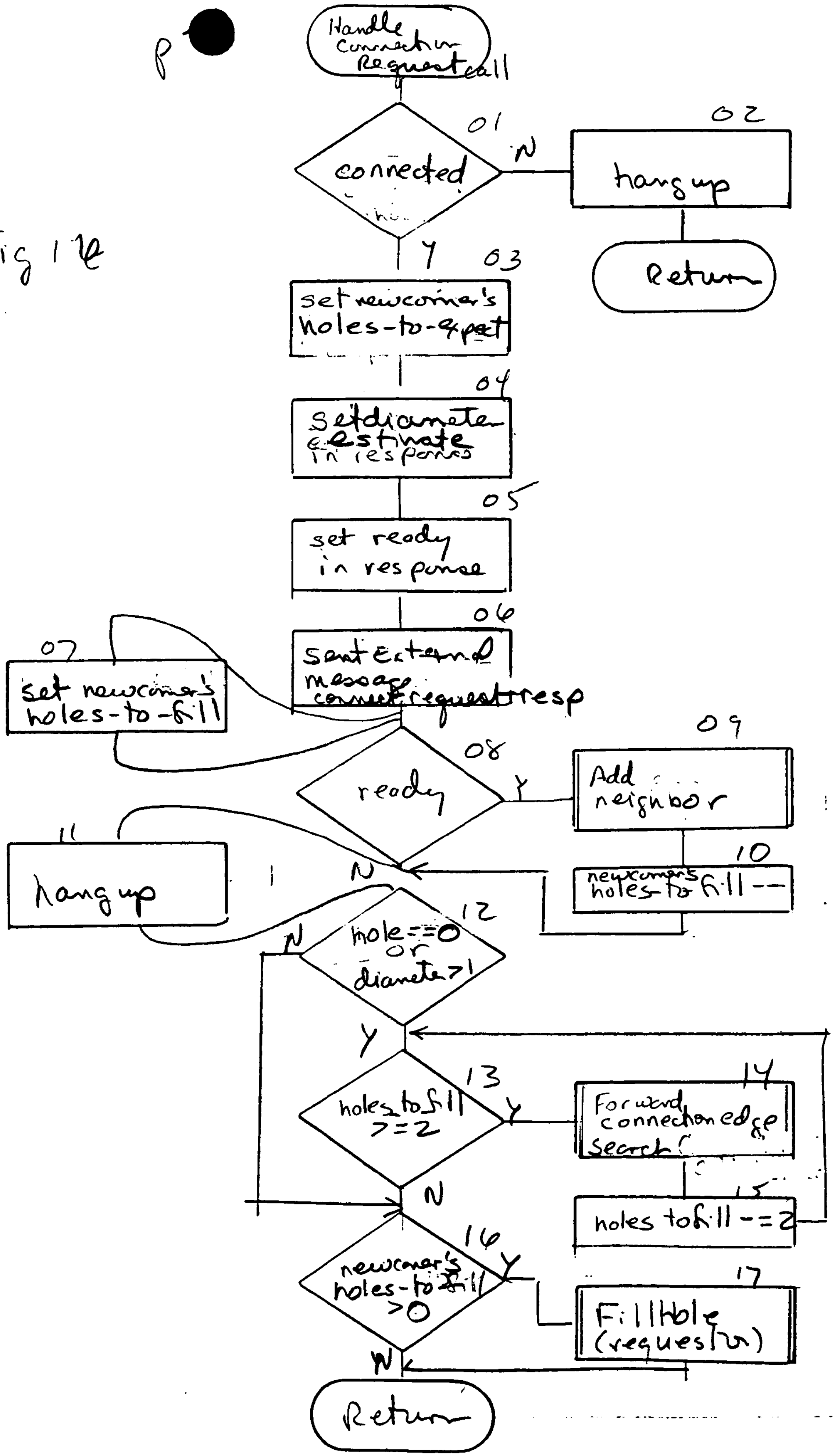


Fig 15

0996295700.073100

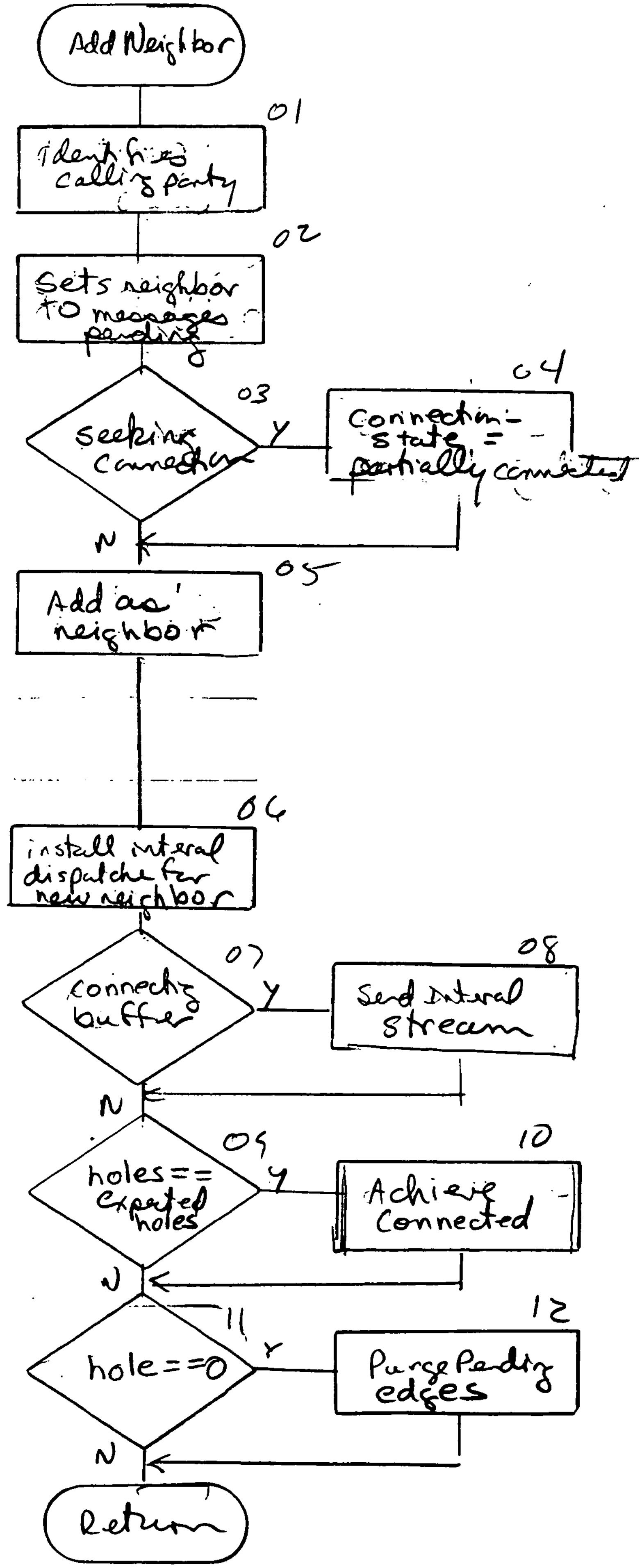
Fig 1e



Handle Connection Request call
01
connected
N
02
hang up
Return
Y
03
set newcomer's holes-to-expect
04
set diameter estimate in response
05
set ready in response
06
send Extend message connect request resp
07
set newcomer's holes-to-fill
08
ready
Y
09
Add neighbor
10
newcomer's holes-to-fill --
N
11
hang up
12
hole==0 or diameter>1
N
Y
13
holes to fill >=2
Y
14
Forward connection edge search
15
holes to fill -=2
N
16
newcomer's holes-to-fill >0
Y
17
Fill hole (requestor)
N
Return

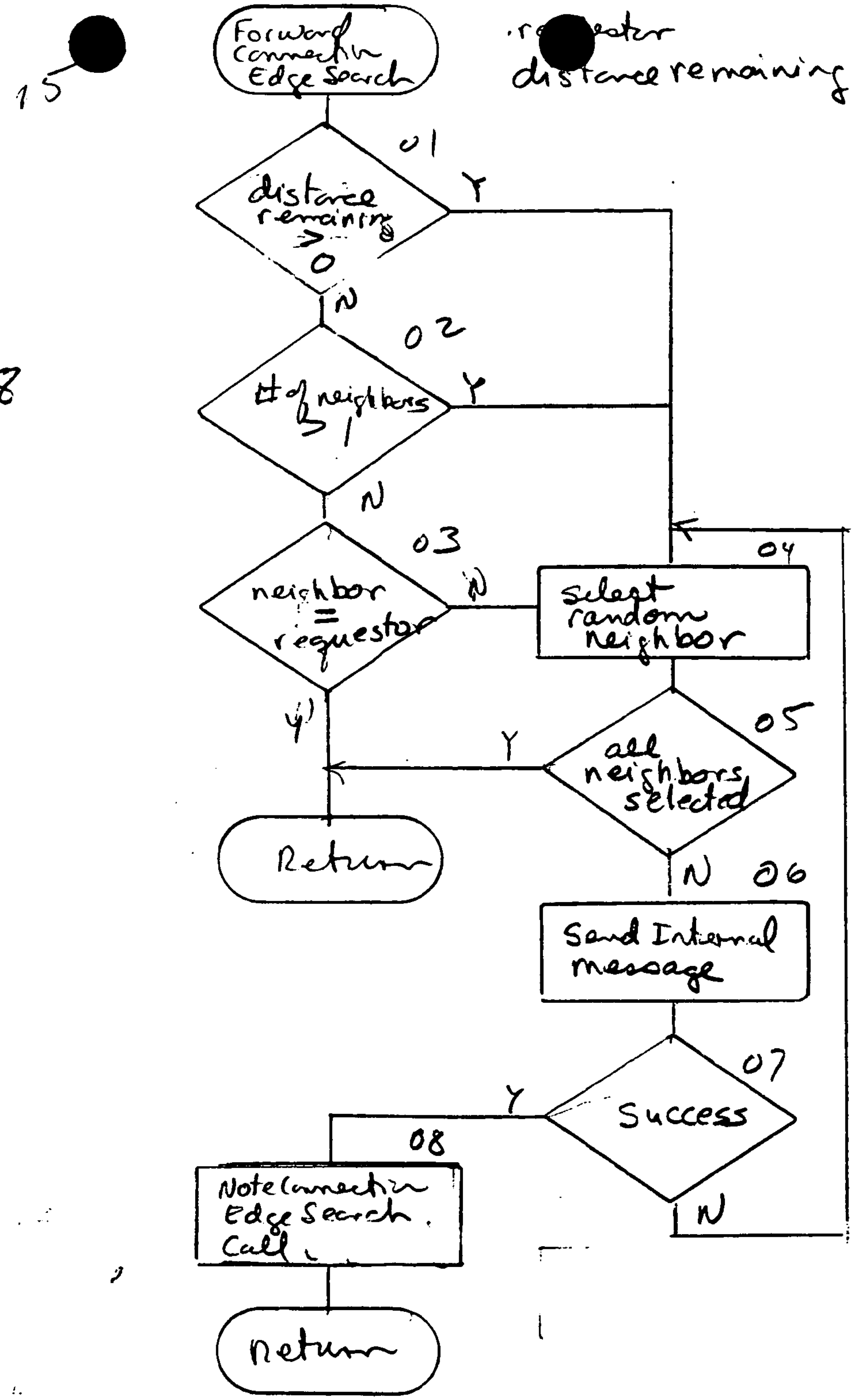
p9

Fig 17

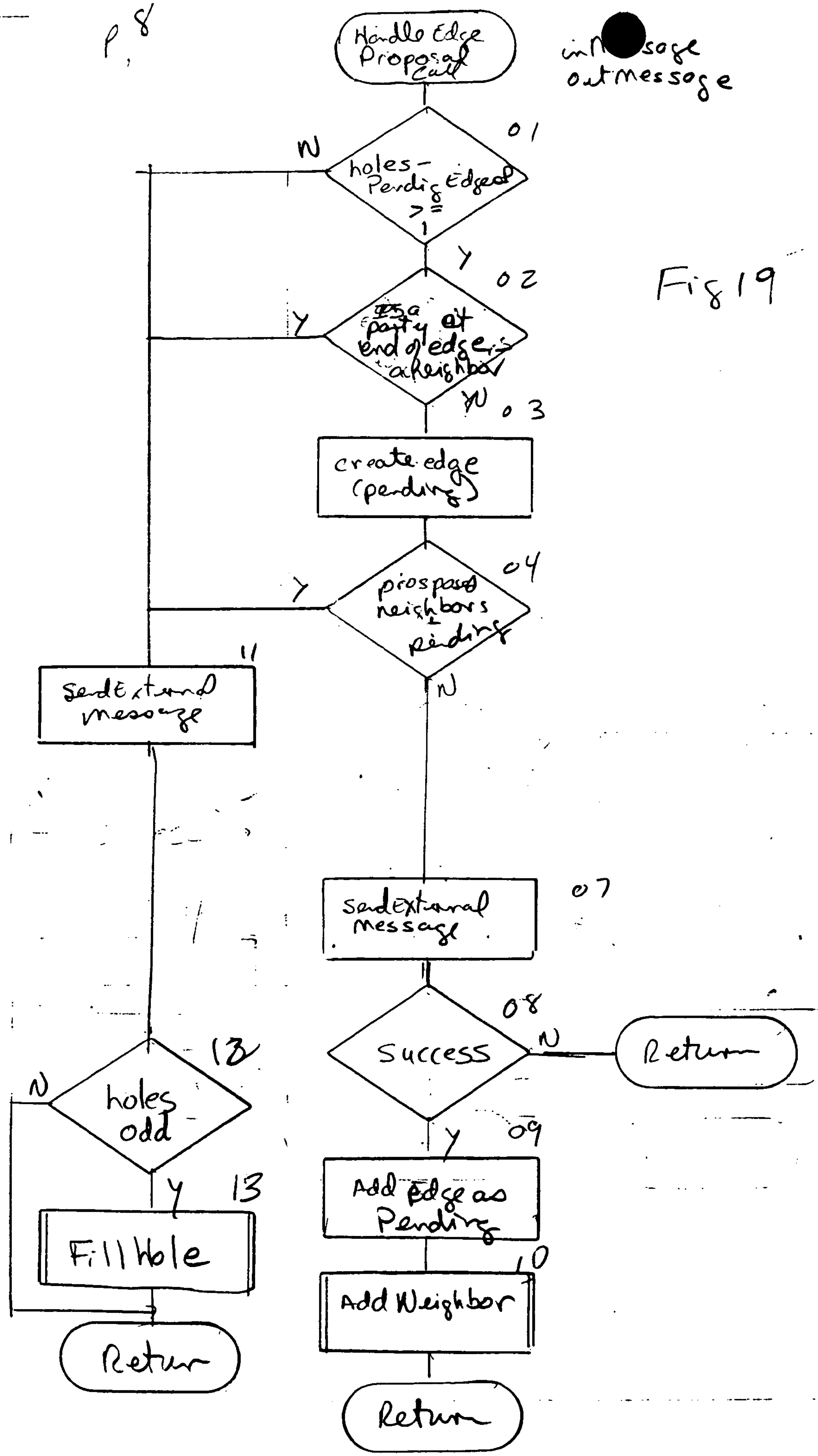


09625570.073100

Fig 18


15
Forward Connection Edge Search
requester
distance remaining
01
distance remaining > 0
Y
N
02
# of neighbors > 1
Y
N
03
neighbor = requestor
N
Y
04
select random neighbor
05
all neighbors selected
Y
N
06
Send Internal message
07
Success
Y
N
08
Note Connection Edge Search Call
Return
Return

p. 8



Handle Edge Proposal Call
outMessage
01
holes - Pending Edges >= 1
N
Y
02
Is a party at end of edge is a neighbor
Y
N
03
create edge (pending)
04
prospose neighbors = pending
Y
N
11
sendExternal message
07
sendExternal message
08
Success
N
Return
Y
09
Add Edge as Pending
10
Add Neighbor
Return
12
holes odd
N
Y
13
Fill Hole
Return

Fig 19

Fig 2D

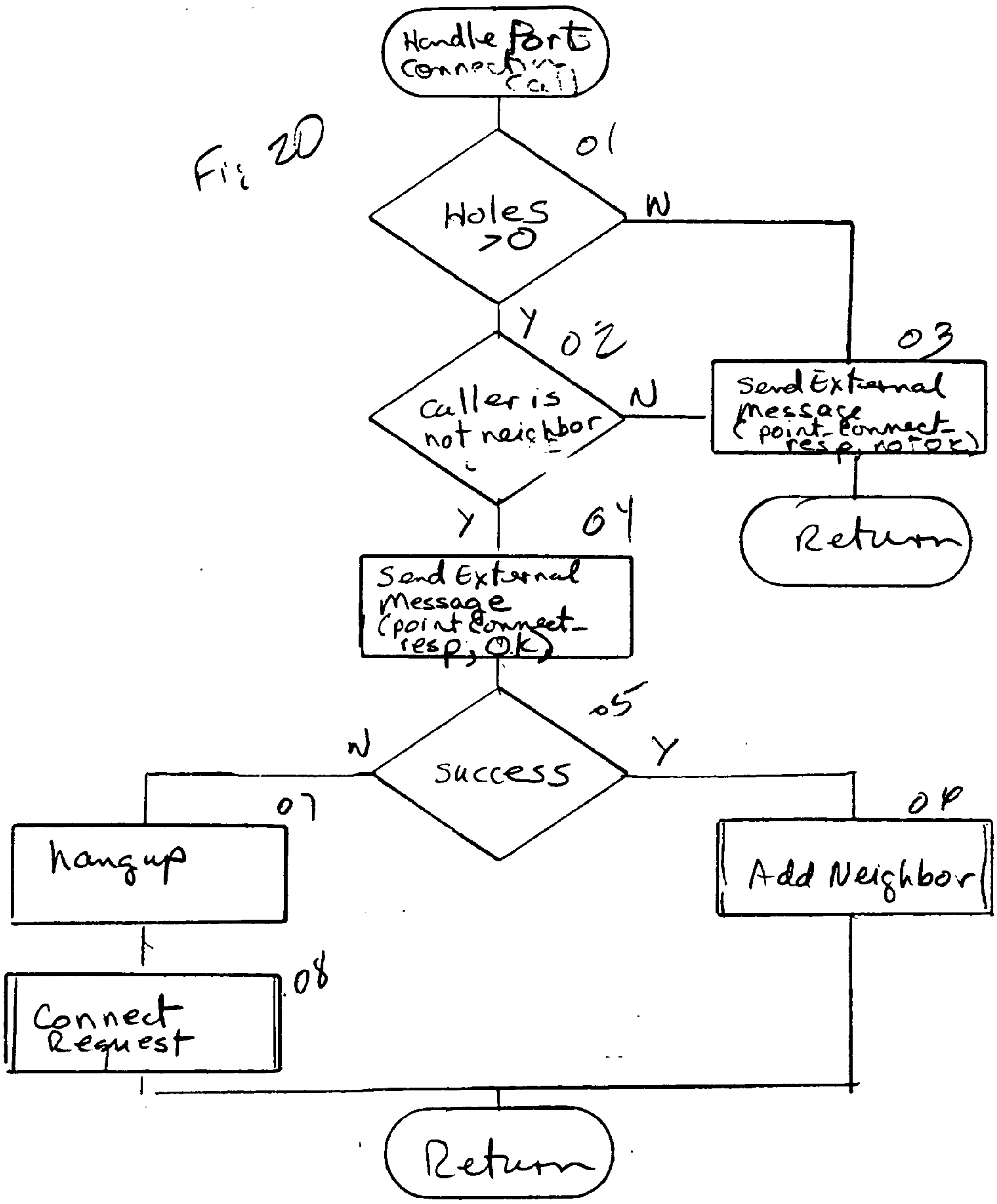
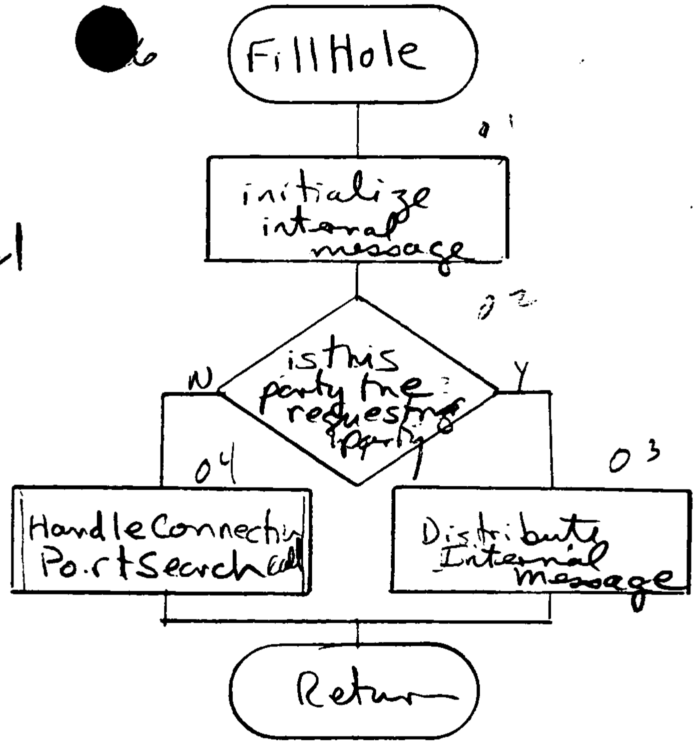

Handle Port Connection Call
01
Holes >0
N
Y
02
Caller is not neighbor
N
03
Send External message (point_connect_resp, notOK)
Return
Y
04
Send External Message (point connect_resp, OK)
05
Success
N
Y
07
hangup
06
Add Neighbor
08
Connect Request
Return

Fig 21



09629557 072400

9

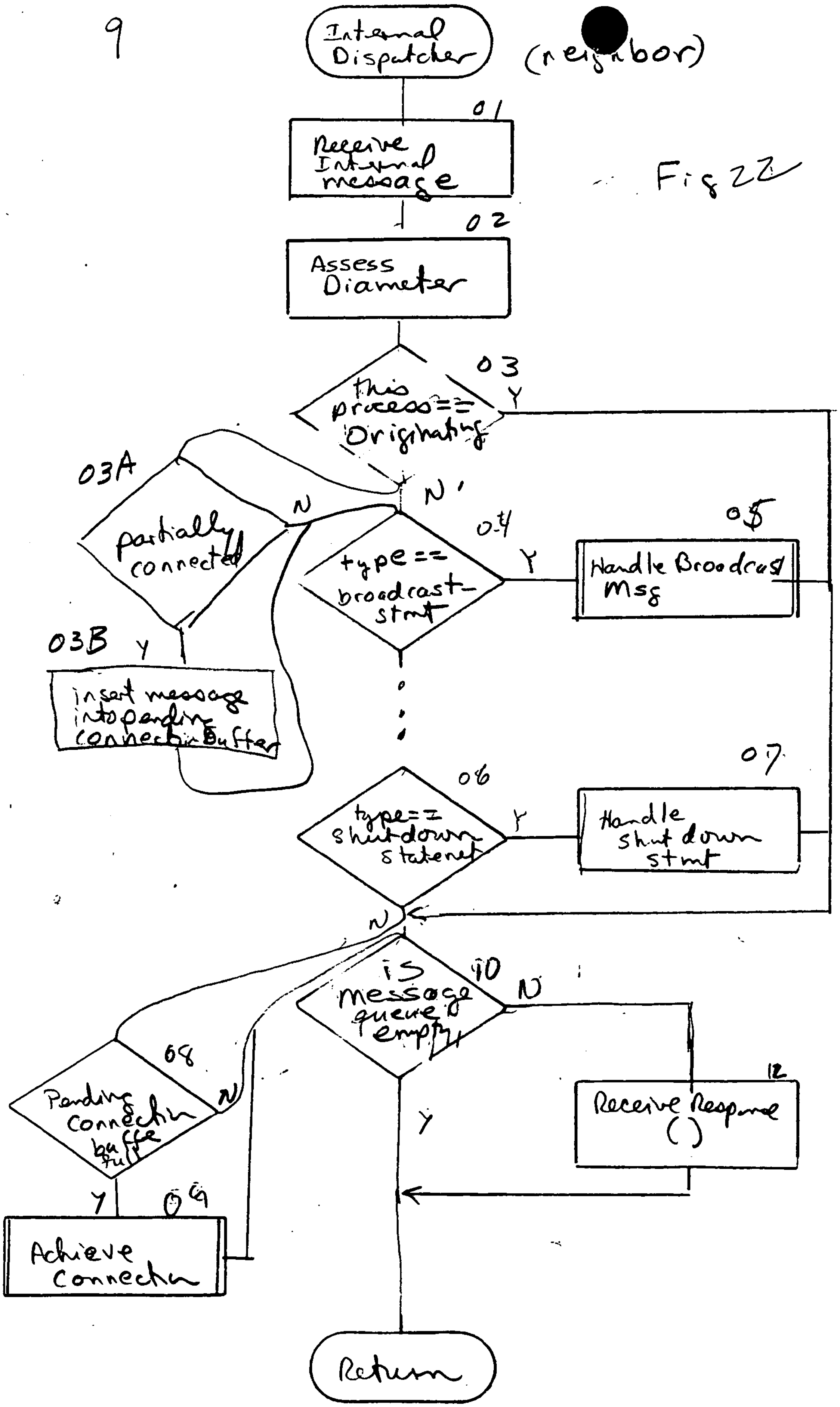

Internal Dispatcher
(neighbor)
01
Receive Internal message
02
Assess Diameter
03
this process == Originating
Y
N
03A
partially connected
N
Y
03B
insert message into pending connection Buffer
04
type == broadcast stmt
Y
05
Handle Broadcast Msg
06
type == shutdown statement
Y
07
Handle shut down stmt
N
10
is message queue empty
N
Y
12
Receive Response ( )
08
Pending connection buffer full
N
Y
09
Achieve connection
Return

Fig 22

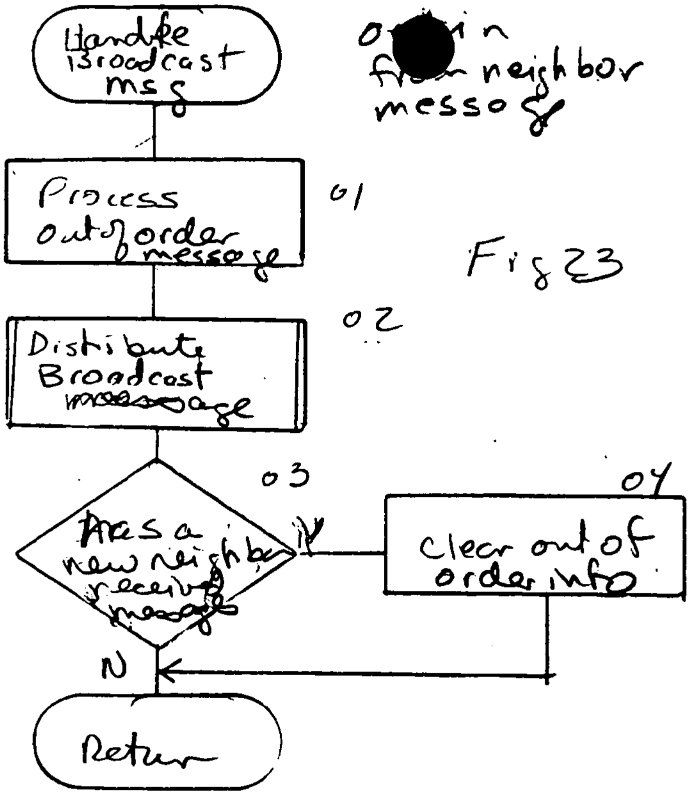

Handle Broadcast msg
Process out of order message
01
Distribute Broadcast message
02
Has a new neighbor receive message
03
Y
Clear out of order info
04
N
Return

Fig 23

20

message from Neighbor

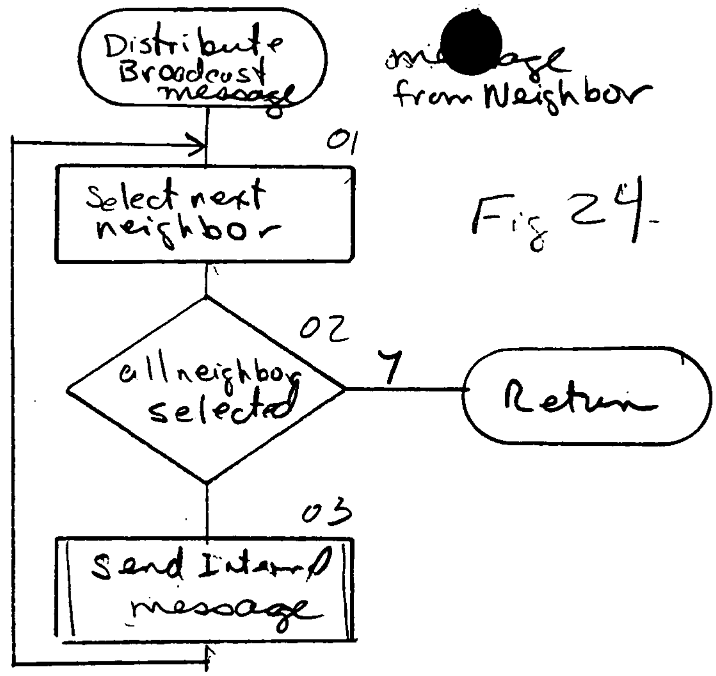
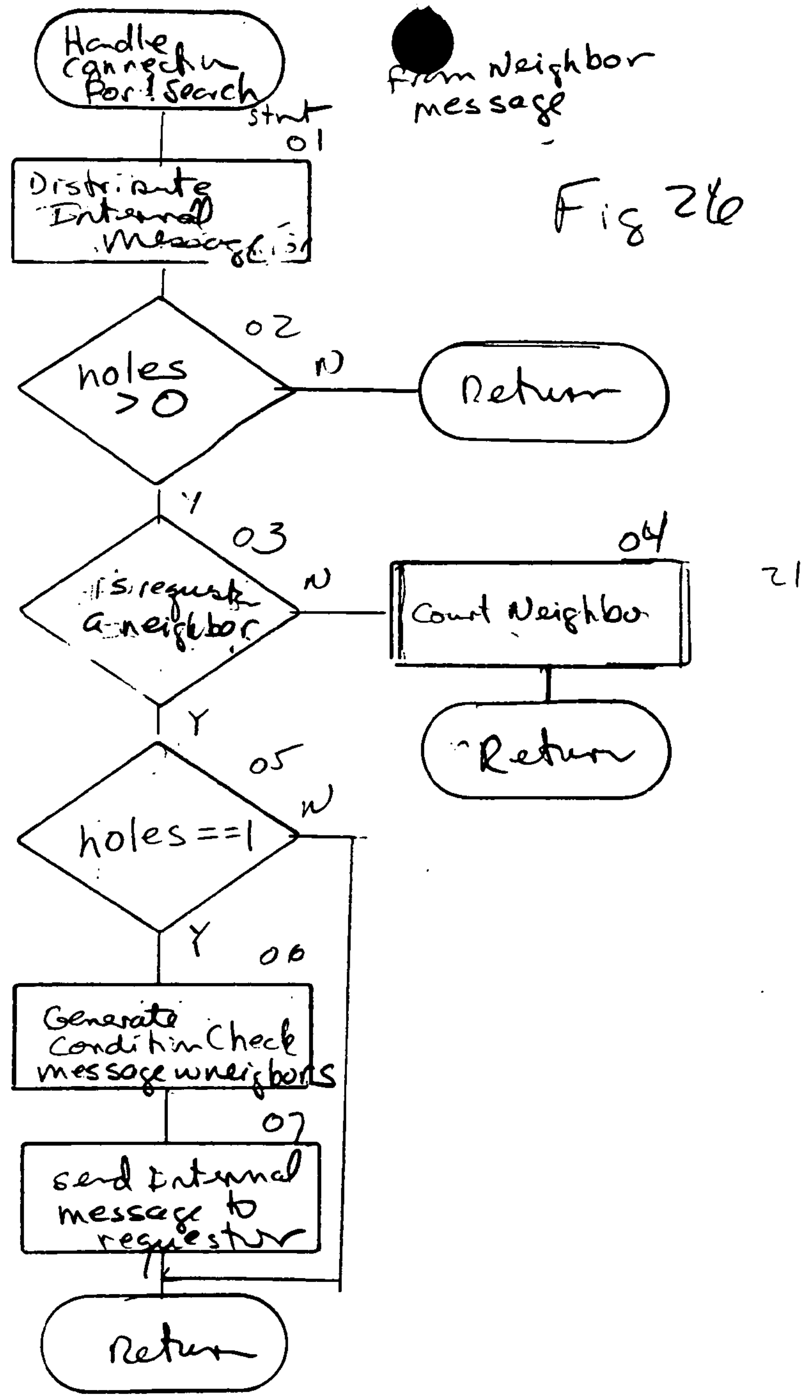


Fig 24

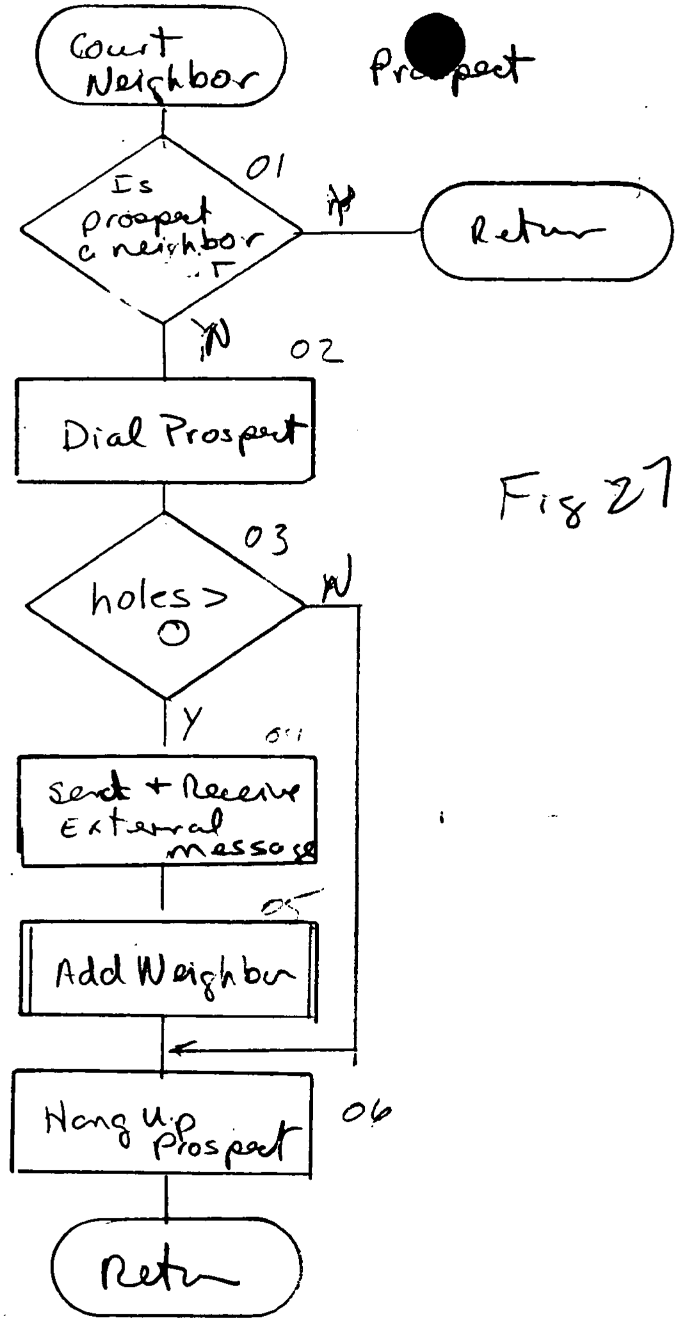
13

Fig 26



0013102 "07570 025296

21



Court Neighbor
Prospect
01
Is prospect a neighbor
N
Return
N
02
Dial Prospect
03
holes > 0
N
Y
Send + Receive External message
Add Neighbor
06
Hang up Prospect
Return

Fig 27

p14

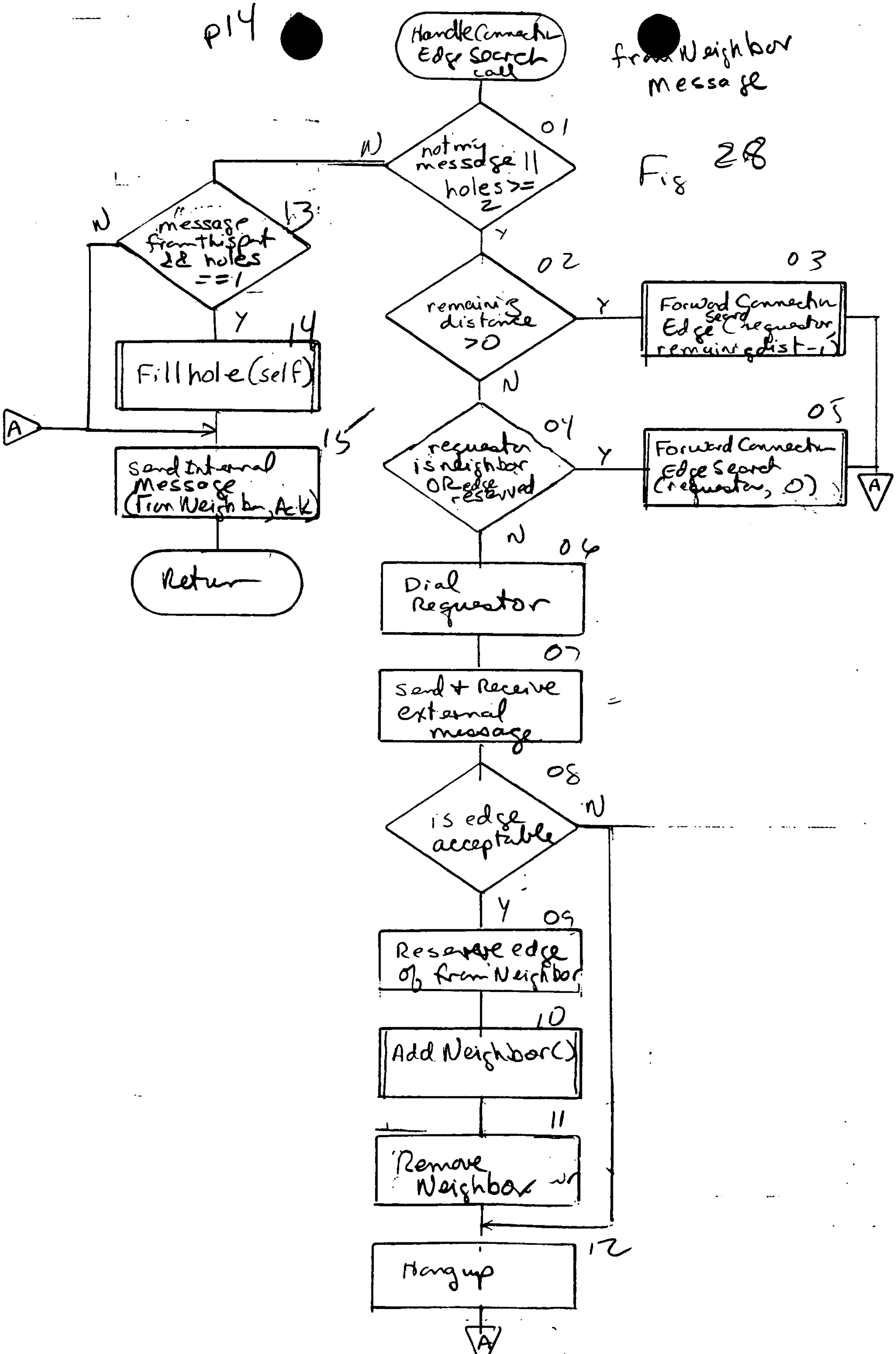
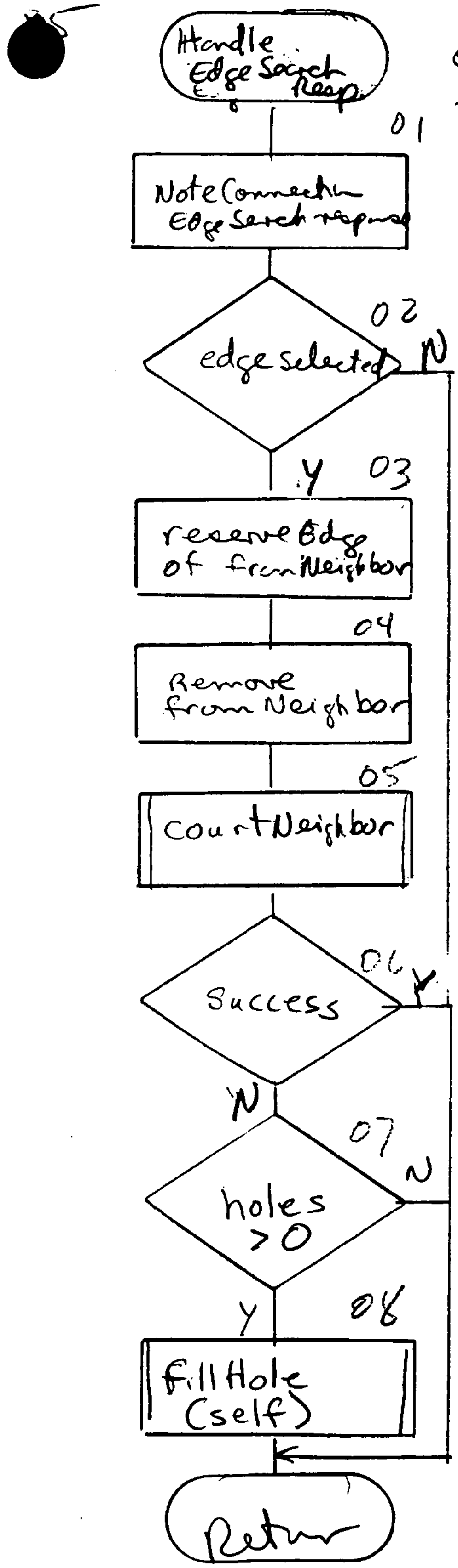


Fig 28

0096255 70 . 073100



on fromNeighbor message

Fig 29

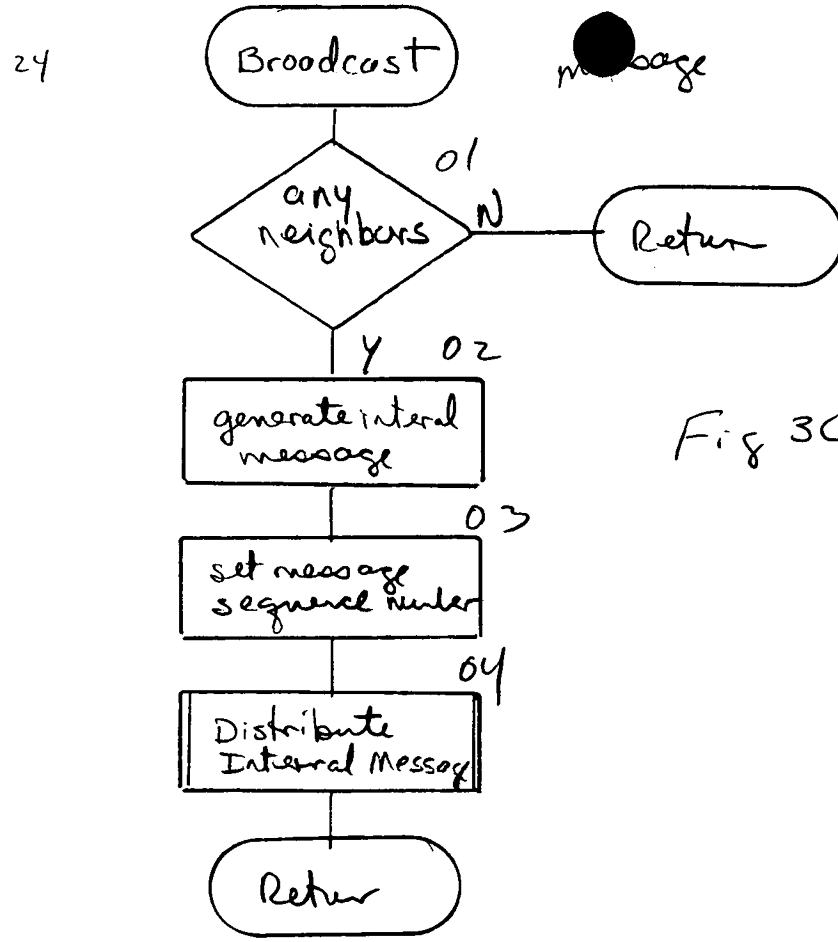

24
Broadcast
message
01
any neighbors
N
Return
Y
02
generate interal message
03
set message sequence number
04
Distribute Interral Message
Return

Fig 30

3

message

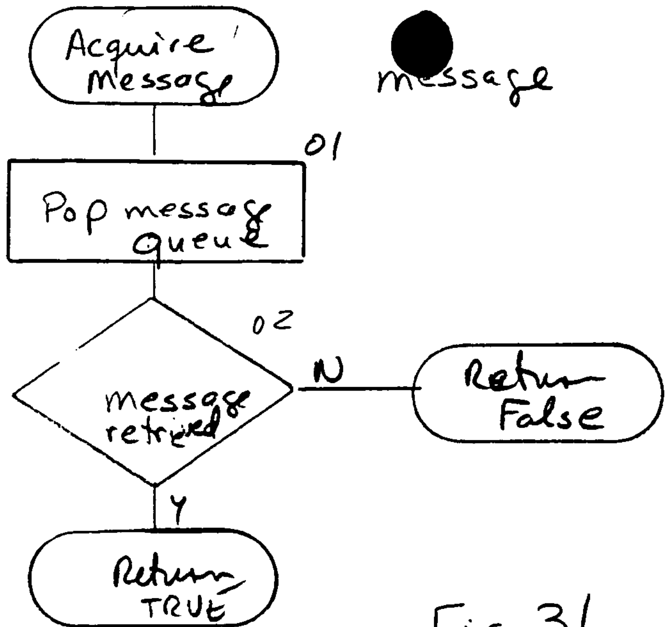
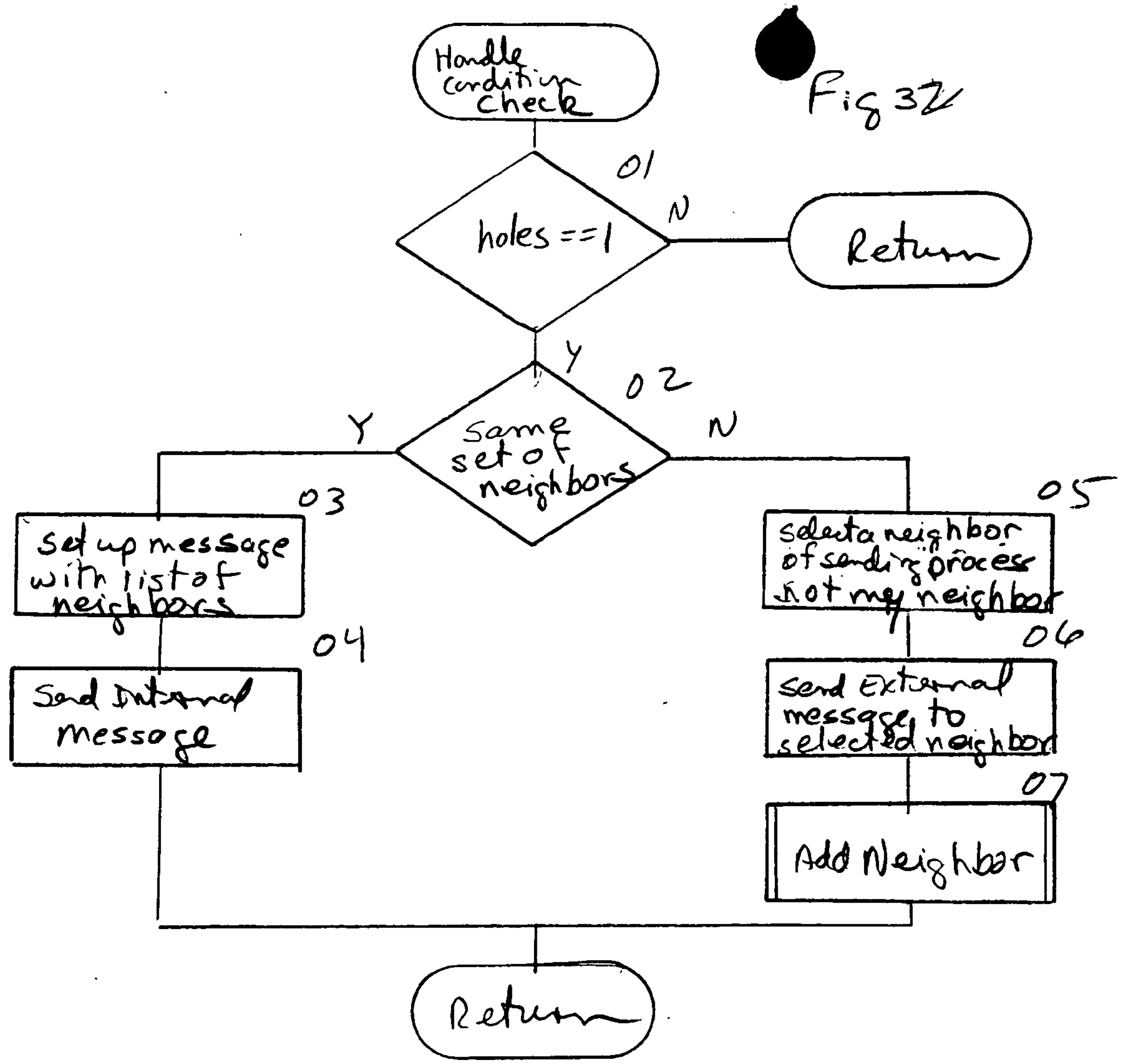


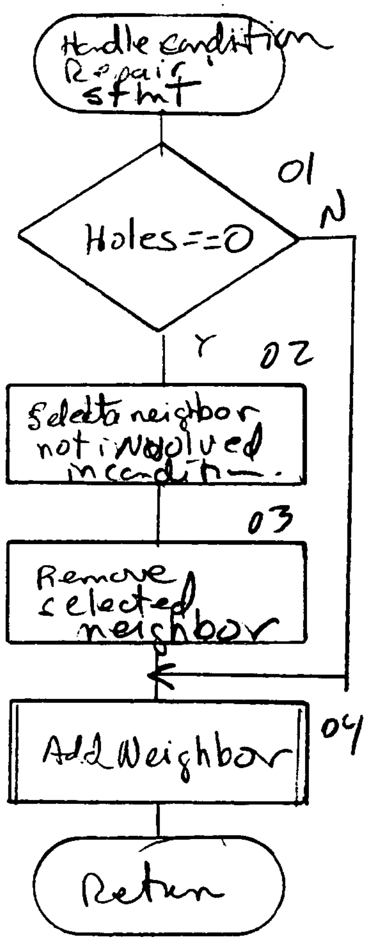
Fig 31



00310.07/2020960

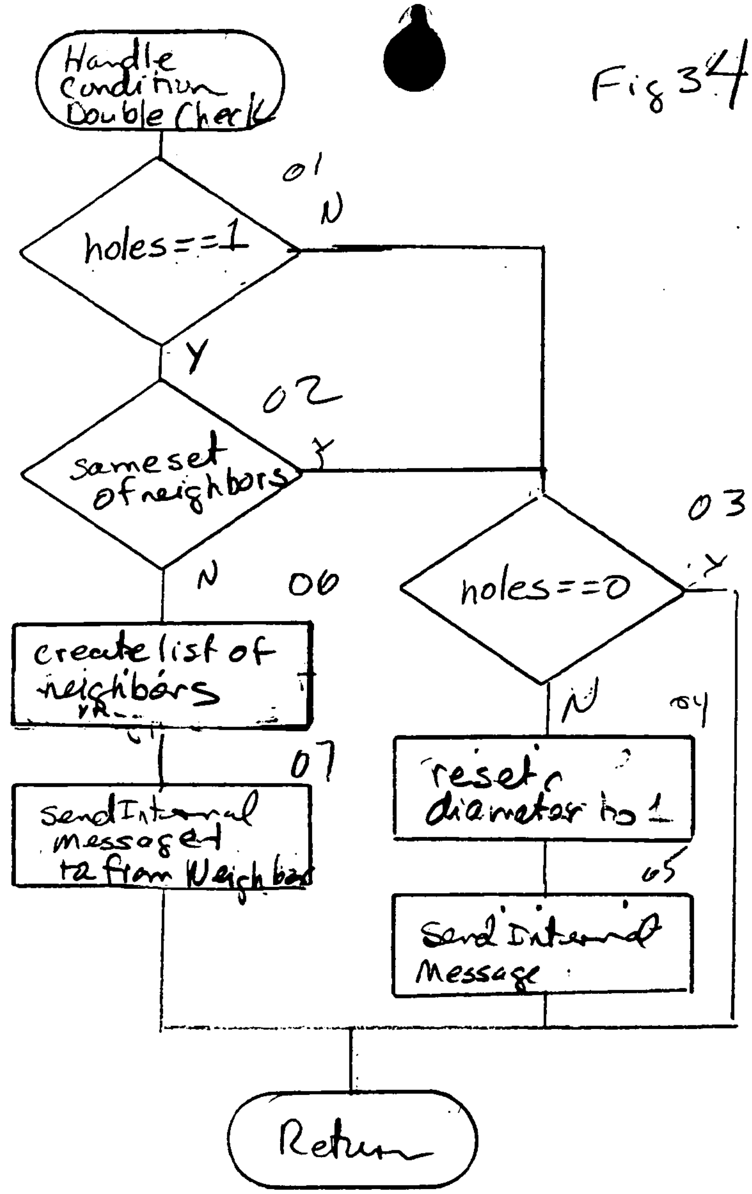
23

Fig 33



Handle condition Repair stmt
01
N
Holes==0
Y
02
Select a neighbor not involved in condition
03
Remove selected neighbor
04
Add Neighbor
Return

Fig 34



0096295470 .022301

JOINING A BROADCAST CHANNEL

CROSS-REFERENCE TO RELATED APPLICATIONS

This application is related to U.S. Patent Application No. ______________, entitled "BROADCASTING NETWORK," filed on July 31, 2000 (Attorney Docket No. 030048001 US); U.S. Patent Application No.______________, entitled "JOINING A BROADCAST CHANNEL," filed on July 31, 2000 (Attorney Docket No. 030048002 US); U.S. Patent Application No.______________, "LEAVING A BROADCAST CHANNEL," filed on July 31, 2000 (Attorney Docket No. 030048003 US); U.S. Patent Application No.______________, entitled "BROADCASTING ON A BROADCAST CHANNEL," filed on July 31, 2000 (Attorney Docket No. 030048004 US); U.S. Patent Application No.______________, entitled "CONTACTING A BROADCAST CHANNEL," filed on July 31, 2000 (Attorney Docket No. 030048005 US); U.S. Patent Application No.______________, entitled "DISTRIBUTED AUCTION SYSTEM," filed on July 31, 2000 (Attorney Docket No. 030048006 US); U.S. Patent Application No.______________, entitled "AN INFORMATION DELIVERY SERVICE," filed on July 31, 2000 (Attorney Docket No. 030048007 US); U.S. Patent Application No.______________, entitled "DISTRIBUTED CONFERENCING SYSTEM," filed on July 31, 2000 (Attorney Docket No. 030048008 US); and U.S. Patent Application No.______________, entitled "DISTRIBUTED GAME ENVIRONMENT," filed on July 31, 2000 (Attorney Docket No. 030048009 US), the disclosures of which are incorporated herein by reference.

TECHNICAL FIELD

The described technology relates generally to a computer network and more particularly, to a broadcast channel for a subset of a computers of an underlying network.

BACKGROUND

There are a wide variety of computer network communications techniques such as point-to-point network protocols, client/server middleware, multicasting network

protocols, and peer-to-peer middleware. Each of these communications techniques have their advantages and disadvantages, but none is particularly well suited to the simultaneous sharing of information among computers that are widely distributed. For example, collaborative processing applications, such as a network meeting programs, have a need to distribute information in a timely manner to all participants who may be geographically distributed.

The point-to-point network protocols, such as UNIX pipes, TCP/IP, and UDP, allow processes on different computers to communicate via point-to-point connections. The interconnection of all participants using point-to-point connections, while theoretically possible, does not scale well as a number of participants grows. For example, each participating process would need to manage its direct connections to all other participating processes. Programmers, however, find it very difficult to manage single connections, and management of multiple connections is much more complex. In addition, participating processes may be limited to the number of direct connections that they can support. This limits the number of possible participants in the sharing of information.

The client/server middleware systems provide a server that coordinates the communications between the various clients who are sharing the information. The server functions as a central authority for controlling access to shared resources. Examples of client/server middleware systems include remote procedure calls ("RPC"), database servers, and the common object request broker architecture ("CORBA"). Client/server middleware systems are not particularly well suited to sharing of information among many participants. In particular, when a client stores information to be shared at the server, each other client would need to poll the server to determine that new information is being shared. Such polling places a very high overhead on the communications network. Alternatively, each client may register a callback with the server, which the server then invokes when new information is available to be shared. Such a callback technique presents a performance bottleneck because a single server needs to call back to each client whenever new information is to be shared. In addition, the reliability of the entire sharing of information depends upon the reliability of the single server. Thus, a failure at a single computer (*i.e.*, the server) would prevent communications between any of the clients.

The multicasting network protocols allow the sending of broadcast messages to multiple recipients of a network. The current implementations of such multicasting network

protocols tend to place an unacceptable overhead on the underlying network. For example, UDP multicasting would swamp the Internet when trying to locate all possible participants. IP multicasting has other problems that include needing special-purpose infrastructure (*e.g.*, routers) to support the sharing of information efficiently.

The peer-to-peer middleware communications systems rely on a multicasting network protocol or a graph of point-to-point network protocols. Such peer-to-peer middleware is provided by the T.120 Internet standard, which is used in such products as Data Connection's D.C.-share and Microsoft's NetMeeting. These peer-to-peer middleware systems rely upon a user to assemble a point-to-point graph of the connections used for sharing the information. Thus, it is neither suitable nor desirable to use peer-to-peer middleware systems when more than a small number of participants is desired. In addition, the underlying architecture of the T.120 Internet standard is a tree structure, which relies on the root node of the tree for reliability of the entire network. That is, each message must pass through the root node in order to be received by all participants.

It would be desirable to have a reliable communications network that is suitable for the simultaneous sharing of information among a large number of the processes that are widely distributed.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 illustrates a graph that is 4-regular and 4-connected which represents a broadcast channel.

Figure 2 illustrates a graph representing 20 computers connected to a broadcast channel.

Figures 3A and 3B illustrate the process of connecting a new computer Z to the broadcast channel.

Figure 4A illustrates the broadcast channel of Figure 1 with an added computer.

Figure 4B illustrates the broadcast channel of Figure 4A with an added computer.

Figure 4C also illustrates the broadcast channel of Figure 4A with an added computer.

Figure 5A illustrates the disconnecting of a computer from the broadcast channel in a planned manner.

Figure 5B illustrates the disconnecting of a computer from the broadcast channel in an unplanned manner.

Figure 5C illustrates the neighbors with empty ports condition.

Figure 5D illustrates two computers that are not neighbors who now have empty ports.

Figure 5E illustrates the neighbors with empty ports condition in the small regime.

Figure 5F illustrates the situation of Figure 5E when in the large regime.

Figure 6 is a block diagram illustrating components of a computer that is connected to a broadcast channel.

Figure 7 is a block diagram illustrating the sub-components of the broadcaster component in one embodiment.

Figure 8 is a flow diagram illustrating the processing of the connect routine in one embodiment.

Figure 9 is a flow diagram illustrating the processing of the seek portal computer routine in one embodiment.

Figure 10 is a flow diagram illustrating the processing of the contact process routine in one embodiment.

Figure 11 is a flow diagram illustrating the processing of the connect request routine in one embodiment.

Figure 12 is a flow diagram of the processing of the check for external call routine in one embodiment.

Figure 13 is a flow diagram of the processing of the achieve connection routine in one embodiment.

Figure 14 is a flow diagram illustrating the processing of the external dispatcher routine in one embodiment.

Figure 15 is a flow diagram illustrating the processing of the handle seeking connection call routine in one embodiment.

Figure 16 is a flow diagram illustrating processing of the handle connection request call routine in one embodiment.

Figure 17 is a flow diagram illustrating the processing of the add neighbor routine in one embodiment.

Figure 18 is a flow diagram illustrating the processing of the forward connection edge search routine in one embodiment.

Figure 19 is a flow diagram illustrating the processing of the handle edge proposal call routine.

Figure 20 is a flow diagram illustrating the processing of the handle port connection call routine in one embodiment.

Figure 21 is a flow diagram illustrating the processing of the fill hole routine in one embodiment.

Figure 22 is a flow diagram illustrating the processing of the internal dispatcher routine in one embodiment.

Figure 23 is a flow diagram illustrating the processing of the handle broadcast message routine in one embodiment.

Figure 24 is a flow diagram illustrating the processing of the distribute broadcast message routine in one embodiment.

Figure 26 is a flow diagram illustrating the processing of the handle connection port search statement routine in one embodiment.

Figure 27 is a flow diagram illustrating the processing of the court neighbor routine in one embodiment.

Figure 28 is a flow diagram illustrating the processing of the handle connection edge search call routine in one embodiment.

Figure 29 is a flow diagram illustrating the processing of the handle connection edge search response routine in one embodiment.

Figure 30 is a flow diagram illustrating the processing of the broadcast routine in one embodiment.

Figure 31 is a flow diagram illustrating the processing of the acquire message routine in one embodiment.

Figure 32 is a flow diagram illustrating processing of the handle condition check message in one embodiment.

Figure 33 is a flow diagram illustrating processing of the handle condition repair statement routine in one embodiment.

Figure 34 is a flow diagram illustrating the processing of the handle condition double check routine.

DETAILED DESCRIPTION

A broadcast technique in which a broadcast channel overlays a point-to-point communications network is provided. The broadcasting of a message over the broadcast channel is effectively a multicast to those computers of the network that are currently connected to the broadcast channel. In one embodiment, the broadcast technique provides a logical broadcast channel to which host computers through their executing processes can be connected. Each computer that is connected to the broadcast channel can broadcast messages onto and receive messages off of the broadcast channel. Each computer that is connected to the broadcast channel receives all messages that are broadcast while it is connected. The logical broadcast channel is implemented using an underlying network system (*e.g.*, the Internet) that allows each computer connected to the underlying network system to send messages to each other connected computer using each computer's address. Thus, the broadcast technique effectively provides a broadcast channel using an underlying network system that sends messages on a point-to-point basis.

The broadcast technique overlays the underlying network system with a graph of point-to-point connections (*i.e.*, edges) between host computers (*i.e.*, nodes) through which the broadcast channel is implemented. In one embodiment, each computer is connected to four other computers, referred to as neighbors. (Actually, a process executing on a computer is connected to four other processes executing on this or four other computers.) To broadcast a message, the originating computer sends the message to each of its neighbors using its point-to-point connections. Each computer that receives the message then sends the message to its three other neighbors using the point-to-point connections. In this way, the message is propagated to each computer using the underlying network to effect the broadcasting of the message to each computer over a logical broadcast channel. A graph in which each node is connected to four other nodes is referred to as a 4-regular graph. The use of a 4-regular graph means that a computer would become disconnected from the broadcast channel only if all four of the connections to its neighbors fail. The graph used by the broadcast technique also has the property that it would take a failure of four computers to

divide the graph into disjoint sub-graphs, that is two separate broadcast channels. This property is referred to as being 4-connected. Thus, the graph is both 4-regular and 4-connected.

Figure 1 illustrates a graph that is 4-regular and 4-connected which represents the broadcast channel. Each of the nine nodes A-I represents a computer that is connected to the broadcast channel, and each of the edges represents an "edge" connection between two computers of the broadcast channel. The time it takes to broadcast a message to each computer on the broadcast channel depends on the speed of the connections between the computers and the number of connections between the originating computer and each other computer on the broadcast channel. The minimum number of connections that a message would need to traverse between each pair of computers is the "distance" between the computers (*i.e.*, the shortest path between the two nodes of the graph). For example, the distance between computers A and F is one because computer A is directly connected to computer F. The distance between computers A and B is two because there is no direct connection between computers A and B, but computer F is directly connected to computer B. Thus, a message originating at computer A would be sent directly to computer F, and then sent from computer F to computer B. The maximum of the distances between the computers is the "diameter" of broadcast channel. The diameter of the broadcast channel represented by Figure 1 is two. That is, a message sent by any computer would traverse no more than two connections to reach every other computer. Figure 2 illustrates a graph representing 20 computers connected to a broadcast channel. The diameter of this broadcast channel is 4. In particular, the shortest path between computers 1 and 3 contains four connections (1-12, 12-15, 15-18, and 18-3).

The broadcast technique includes (1) the connecting of computers to the broadcast channel (*i.e.*, composing the graph), (2) the broadcasting of messages over the broadcast channel (*i.e.*, broadcasting through the graph), and (3) the disconnecting of computers from the broadcast channel (*i.e.*, decomposing the graph) composing the graph.

<u>Composing the Graph</u>

To connect to the broadcast channel, the computer seeking the connection first locates a computer that is currently fully connected to the broadcast channel and then

establishes a connection with four of the computers that are already connected to the broadcast channel. (This assumes that there are at least four computers already connected to the broadcast channel. When there are fewer than five computers connected, the broadcast channel cannot be a 4-regular graph. In such a case, the broadcast channel is considered to be in a "small regime." The broadcast technique for the small regime is described below in detail. When five or more computers are connected, the broadcast channel is considered to be in the "large regime." This description assumes that the broadcast channel is in the large regime, unless specified otherwise.) Thus, the process of connecting to the broadcast channel includes locating the broadcast channel, identifying the neighbors for the connecting computer, and then connecting to each identified neighbor. Each computer is aware of one or more "portal computers" through which that computer may locate the broadcast channel. A seeking computer locates the broadcast channel by contacting the portal computers until it finds one that is currently fully connected to the broadcast channel. The found portal computer then directs the identifying of four computers (*i.e.*, to be the seeking computer's neighbors) to which the seeking computer is to connect. Each of these four computers then cooperates with the seeking computer to effect the connecting of the seeking computer to the broadcast channel. A computer that has started the process of locating a portal computer, but does not yet have a neighbor, is in the "seeking connection state." A computer that is connected to at least one neighbor, but not yet four neighbors, is in the "partially connected state." A computer that is currently, or has been, previously connected to four neighbors is in the "fully connected state."

Since the broadcast channel is a 4-regular graph, each of the identified computers is already connected to four computers. Thus, some connections between computers need to be broken so that the seeking computer can connect to four computers. In one embodiment, the broadcast technique identifies two pairs of computers that are currently connected to each other. Each of these pairs of computers breaks the connection between them, and then each of the four computers (two from each pair) connects to the seeking computer. Figures 3A and 3B illustrate the process of a new computer Z connecting to the broadcast channel. Figure 3A illustrates the broadcast channel before computer Z is connected. The pairs of computers B and E and computers C and D are the two pairs that are identified as the neighbors for the new computer Z. The connections between each of these pairs is broken, and a connection between computer Z and each of computers B, C, D, and E

is established as indicated by Figure 3B. The process of breaking the connection between two neighbors and reconnecting each of the former neighbors to another computer is referred to as "edge pinning" as the edge between two nodes may be considered to be stretched and pinned to a new node.

Each computer connected to the broadcast channel allocates five communications ports for communicating with other computers. Four of the ports are referred to as "internal" ports because they are the ports through which the messages of the broadcast channels are sent. The connections between internal ports of neighbors are referred to as "internal" connections. Thus, the internal connections of the broadcast channel form the 4-regular and 4-connected graph. The fifth port is referred to as an "external" port because it is used for sending non-broadcast messages between two computers. Neighbors can send non-broadcast messages either through their internal ports of their connection or through their external ports. A seeking computer uses external ports when locating a portal computer.

In one embodiment, the broadcast technique establishes the computer connections using the TCP/IP communications protocol, which is a point-to-point protocol, as the underlying network. The TCP/IP protocol provides for reliable and ordered delivery of messages between computers. The TCP/IP protocol provides each computer with a "port space" that is shared among all the processes that may execute on that computer. The ports are identified by numbers from 0 to 65,535. The first 2056 ports are reserved for specific applications (*e.g.*, port 80 for HTTP messages). The remainder of the ports are user ports that are available to any process. In one embodiment, a set of port numbers can be reserved for use by the computer connected to the broadcast channel. In an alternative embodiment, the port numbers used are dynamically identified by each computer. Each computer dynamically identifies an available port to be used as its call-in port. This call-in port is used to establish connections with the external port and the internal ports. Each computer that is connected to the broadcast channel can receive non-broadcast messages through its external port. A seeking computer tries "dialing" the port numbers of the portal computers until a portal computer "answers," a call on its call-in port. A portal computer answers when it is connected to or attempting to connect to the broadcast channel and its call-in port is dialed. (In this description, a telephone metaphor is used to describe the connections.) When a computer receives a call on its call-in port, it transfers the call to another port. Thus, the

seeking computer actually communicates through that transfer-to port, which is the external port. The call is transferred so that other computers can place calls to that computer via the call-in port. The seeking computer then communicates via that external port to request the portal computer to assist in connecting the seeking computer to the broadcast channel. The seeking computer could identify the call-in port number of a portal computer by successively dialing each port in port number order. As discussed below in detail, the broadcast technique uses a hashing algorithm to select the port number order, which may result in improved performance.

A seeking computer could connect to the broadcast channel by connecting to computers either directly connected to the found portal computer or directly connected to one of its neighbors. A possible problem with such a scheme for identifying the neighbors for the seeking computer is that the diameter of the broadcast channel may increase when each seeking computer uses the same found portal computer and establishes a connection to the broadcast channel directly through that found portal computer. Conceptually, the graph becomes elongated in the direction of where the new nodes are added. Figures 4A-4C illustrate that possible problem. Figure 4A illustrates the broadcast channel of Figure 1 with an added computer. Computer J was connected to the broadcast channel by edge pinning edges C-D and E-H to computer J. The diameter of this broadcast channel is still two. Figure 4B illustrates the broadcast channel of Figure 4A with an added computer. Computer K was connected to the broadcast channel by edge pinning edges E-J and B-C to computer K. The diameter of this broadcast channel is three, because the shortest path from computer G to computer K is through edges G-A, A-E, and E-K. Figure 4C also illustrates the broadcast channel of Figure 4A with an added computer. Computer K was connected to the broadcast channel by edge pinning edges D-G and E-J to computer K. The diameter of this broadcast channel is, however, still two. Thus, the selection of neighbors impacts the diameter of the broadcast channel. To help minimize the diameter, the broadcast technique uses a random selection technique to identify the four neighbors of a computer in the seeking connection state. The random selection technique tends to distribute the connections to new seeking computers throughout the computers of the broadcast channel which may result in smaller overall diameters.

Broadcasting Through the Graph

As described above, each computer that is connected to the broadcast channel can broadcast messages onto the broadcast channel and does receive all messages that are broadcast on the broadcast channel. The computer that originates a message to be broadcast sends that message to each of its four neighbors using the internal connections. When a computer receives a broadcast message from a neighbor, it sends the message to its three other neighbors. Each computer on the broadcast channel, except the originating computer, will thus receive a copy of each broadcast message from each of its four neighbors. Each computer, however, only sends the first copy of the message that it receives to its neighbors and disregards subsequently received copies. Thus, the total number of copies of a message that is sent between the computers is 3N+1, where N is the number of computers connected to the broadcast channel. Each computer sends three copies of the message, except for the originating computer, which sends four copies of the message.

The redundancy of the message sending helps to ensure the overall reliability of the broadcast channel. Since each computer has four connections to the broadcast channel, if one computer fails during the broadcast of a message, its neighbors have three other connections through which they will receive copies of the broadcast message. Also, if the internal connection between two computers is slow, each computer has three other connections through which it may receive a copy of each message sooner.

Each computer that originates a message numbers its own messages sequentially. Because of the dynamic nature of the broadcast channel and because there are many possible connection paths between computers, the messages may be received out of order. For example, the distance between an originating computer and a certain receiving computer may be four. After sending the first message, the originating computer and receiving computer may become neighbors and thus the distance between them changes to one. The first message may have to travel a distance of four to reach the receiving computer. The second message only has to travel a distance of one. Thus, it is possible for the second message to reach the receiving computer before the first message.

When the broadcast channel is in a steady state (*i.e.*, no computers connecting or disconnecting from the broadcast channel), out-of-order messages are not a problem because each computer will eventually receive both messages and can queue messages until all earlier ordered messages are received. If, however, the broadcast channel is not in a

steady state, then problems can occur. In particular, a computer may connect to the broadcast channel after the second message has already been received and forwarded on by its new neighbors. When a new neighbor eventually receives the first message, it sends the message to the newly connected computer. Thus, the newly connected computer will receive the first message, but will not receive the second message. If the newly connected computer needs to process the messages in order, it would wait indefinitely for the second message.

One solution to this problem is to have each computer queue all the messages that it receives until it can send them in their proper order to its neighbors. This solution, however, may tend to slow down the propagation of messages through the computers of the broadcast channel. Another solution that may have less impact on the propagation speed is to queue messages only at computers who are neighbors of the newly connected computers. Each already connected neighbor would forward messages as it receives them to its other neighbors who are not newly connected, but not to the newly connected neighbor. The already connected neighbor would only forward messages from each originating computer to the newly connected computer when it can ensure that no gaps in the messages from that originating computer will occur. In one embodiment, the already connected neighbor may track the highest sequence number of the messages already received and forwarded on from each originating computer. The already connected computer will send only higher numbered messages from the originating computers to the newly connected computer. Once all lower numbered messages have been received from all originating computers, then the already connected computer can treat the newly connected computer as its other neighbors and simply forward each message as it is received. In another embodiment, each computer may queue messages and only forwards to the newly connected computer those messages as the gaps are filled in. For example, a computer might receive messages 4 and 5 and then receive message 3. In such a case, the already connected computer would forward queue messages 4 and 5. When message 3 is finally received, the already connected computer will send messages 3, 4, and 5 to the newly connected computer. If messages 4 and 5 were sent to the newly connected computer before message 3, then the newly connected computer would process messages 4 and 5 and disregard message 3. Because the already connected computer queues messages 4 and 5, the newly connected computer will be able to process message 3. It is possible that a newly connected computer will receive a set of messages from an originating computer through one neighbor and then receive another set of message from the

same originating computer through another neighbor. If the second set of messages contains a message that is ordered earlier than the messages of the first set received, then the newly connected computer may ignore that earlier ordered message if the computer already processed those later ordered messages.

<u>Decomposing the Graph</u>

A connected computer disconnects from the broadcast channel either in a planned or unplanned manner. When a computer disconnects in a planned manner, it sends a disconnect message to each of its four neighbors. The disconnect message includes a list that identifies the four neighbors of the disconnecting computer. When a neighbor receives the disconnect message, it tries to connect to one of the computers on the list. In one embodiment, the first computer in the list will try to connect to the second computer in the list, and the third computer in the list will try to connect to the fourth computer in the list. If a computer cannot connect (*e.g.*, the first and second computers are already connected), then the computers may try connecting in various other combinations. If connections cannot be established, each computer broadcasts a message that it needs to establish a connection with another computer. When a computer with an available internal port receives the message, it can then establish a connection with the computer that broadcast the message. Figures 5A-5D illustrate the disconnecting of a computer from the broadcast channel. Figure 5A illustrates the disconnecting of a computer from the broadcast channel in a planned manner. When computer H decides to disconnect, it sends its list of neighbors to each of its neighbors (computers A, E, F and I) and then disconnects from each of its neighbors. When computers A and I receive the message they establish a connection between them as indicated by the dashed line, and similarly for computers E and F.

When a computer disconnects in an unplanned manner, such as resulting from a power failure, the neighbors connected to the disconnected computer recognize the disconnection when each attempts to send its next message to the now disconnected computer. Each former neighbor of the disconnected computer recognizes that it is short one connection (*i.e.*, it has a hole or empty port). When a connected computer detects that one of its neighbors is now disconnected, it broadcasts a port connection request on the broadcast channel, which indicates that it has one internal port that needs a connection. The port connection request identifies the call-in port of the requesting computer. When a connected

computer that is also seeking a connection receives the connection request, it communicates with the requesting computer through its external port to establish a connection between the two computers. Figure 5B illustrates the disconnecting of a computer from the broadcast channel in an unplanned manner. In this illustration, computer H has disconnected in an unplanned manner. When each of its neighbors, computers A, E, F, and I, recognizes the disconnection, each neighbor broadcasts a port connection request indicating that it needs to fill an empty port. As shown by the dashed lines, computers F and I and computers A and E respond to each other's requests and establish a connection.

It is possible that a planned or unplanned disconnection may result in two neighbors each having an empty internal port. In such a case, since they are neighbors, they are already connected and cannot fill their empty ports by connecting to each other. Such a condition is referred to as the "neighbors with empty ports" condition. Each neighbor broadcasts a port connection request when it detects that it has an empty port as described above. When a neighbor receives the port connection request from the other neighbor, it will recognize the condition that its neighbor also has an empty port. Such a condition may also occur when the broadcast channel is in the small regime. The condition can only be corrected when in the large regime. When in the small regime, each computer will have less than four neighbors. To detect this condition in the large regime, which would be a problem if not repaired, the first neighbor to receive the port connection request recognizes the condition and sends a condition check message to the other neighbor. The condition check message includes a list of the neighbors of the sending computer. When the receiving computer receives the list, it compares the list to its own list of neighbors. If the lists are different, then this condition has occurred in the large regime and repair is needed. To repair this condition, the receiving computer will send a condition repair request to one of the neighbors of the sending computer which is not already a neighbor of the receiving computer. When the computer receives the condition repair request, it disconnects from one of its neighbors (other than the neighbor that is involved with the condition) and connects to the computer that sent the condition repair request. Thus, one of the original neighbors involved in the condition will have had a port filled. However, two computers are still in need of a connection, the other original neighbor and the computer that is now disconnected from the computer that received the condition repair request. Those two computers send out port connection requests. If those two computers are not neighbors, then they will connect to

each other when they receive the requests. If, however, the two computers are neighbors, then they repeat the condition repair process until two non-neighbors are in need of connections.

It is possible that the two original neighbors with the condition may have the same set of neighbors. When the neighbor that receives the condition check message determines that the sets of neighbors are the same, it sends a condition double check message to one of its neighbors other than the neighbor who also has the condition. When the computer receives the condition double check message, it determines whether it has the same set of neighbors as the sending computer. If so, the broadcast channel is in the small regime and the condition is not a problem. If the set of neighbors are different, then the computer that received the condition double check message sends a condition check message to the original neighbors with the condition. The computer that receives that condition check message directs one of it neighbors to connect to one of the original neighbors with the condition by sending a condition repair message. Thus, one of the original neighbors with the condition will have its port filled.

Figure 5C illustrates the neighbors with empty ports condition. In this illustration, computer H disconnected in an unplanned manner, but computers F and I responded to the port connection request of the other and are now connected together. The other former neighbors of computer H, computers A and E, are already neighbors, which gives rise to the neighbors with empty ports condition. In this example, computer E received the port connection request from computer A, recognized the possible condition, and sent (since they are neighbors via the internal connection) a condition check message with a list of its neighbors to computer A. When computer A received the list, it recognized that computer E has a different set of neighbor (*i.e.*, the broadcast channel is in the large regime). Computer A selected computer D, which is a neighbor of computer E and sent it a condition repair request. When computer D received the condition repair request, it disconnected from one of its neighbors (other than computer E), which is computer G in this example. Computer D then connected to computer A. Figure 5D illustrates two computers that are not neighbors who now have empty ports. Computers E and G now have empty ports and are not currently neighbors. Therefore, computers E and G can connect to each other.

Figures 5E and 5F further illustrate the neighbors with empty ports condition. Figure 5E illustrates the neighbors with empty ports condition in the small regime. In this

example, if computer [illegible] disconnected in an unplanned manner, then each computer broadcasts a port connection request when it detects the disconnect. When computer A receives the port connection request form computer B, it detects the neighbors with empty ports condition and sends a condition check message to computer B. Computer B recognizes that it has the same set of neighbors (computer C and D) as computer A and then sends a condition double check message to computer C. Computer C recognizes that the broadcast channel is in the small regime because is also has the same set of neighbors as computers A and B, computer C may then broadcast a message indicating that the broadcast channel is in the small regime.

Figure 5F illustrates the situation of Figure 5E when in the large regime. As discussed above, computer C receives the condition double check message from computer B. In this case, computer C recognizes that the broadcast channel is in the large regime because it has a set of neighbors that is different from computer B. The edges extending up from computer C and D indicate connections to other computers. Computer C then sends a condition check message to computer B. When computer B receives the condition check message, it sends a condition repair message to one of the neighbors of computer C. The computer that receives the condition repair message disconnects from one of its neighbors, other than computer C, and tries to connect to computer B and the neighbor from which it disconnected tries to connect to computer A.

<u>Port Selection</u>

As described above, the TCP/IP protocol designates ports above number 2056 as user ports. The broadcast technique uses five user port numbers on each computer: one external port and four internal ports. Generally, user ports cannot be statically allocated to an application program because other applications programs executing on the same computer may use conflicting port numbers. As a result, in one embodiment, the computers connected to the broadcast channel dynamically allocate their port numbers. Each computer could simply try to locate the lowest number unused port on that computer and use that port as the call-in port. A seeking computer, however, does not know in advance the call-in port number of the portal computers when the port numbers are dynamically allocated. Thus, a seeking computer needs to dial ports of a portal computer starting with the lowest port number when locating the call-in port of a portal computer. If the portal computer is

connected to (or attempting to connect to) the broadcast channel, then the seeking computer would eventually find the call-in port. If the portal computer is not connected, then the seeking computer would eventually dial every user port. In addition, if each application program on a computer tried to allocate low-ordered port numbers, then a portal computer may end up with a high-numbered port for its call-in port because many of the low-ordered port numbers would be used by other application programs. Since the dialing of a port is a relatively slow process, it would take the seeking computer a long time to locate the call-in port of a portal computer. To minimize this time, the broadcast technique uses a port ordering algorithm to identify the port number order that a portal computer should use when finding an available port for its call-in port. In one embodiment, the broadcast technique uses a hashing algorithm to identify the port order. The algorithm preferably distributes the ordering of the port numbers randomly through out the user port number space and only selects each port number once. In addition, every time the algorithm is executed on any computer for a given channel type and channel instance, it generates the same port ordering. As described below, it is possible for a computer to be connected to multiple broadcast channels that are uniquely identified by channel type and channel instance. The algorithm may be "seeded" with channel type and channel instance in order to generate a unique ordering of port numbers for each broadcast channel. Thus, a seeking computer will dial the ports of a portal computer in the same order as the portal computer used when allocating its call-in port.

If many computers are at the same time seeking connection to a broadcast channel through a single portal computer, then the ports of the portal computer may be busy when called by seeking computers. The seeking computers would typically need to keep on redialing a busy port. The process of locating a call-in port may be significantly slowed by such redialing. In one embodiment, each seeking computer may each reorder the first few port numbers generated by the hashing algorithm. For example, each seeking computer could randomly reorder the first eight port numbers generated by the hashing algorithm. The random ordering could also be weighted where the first port number generated by the hashing algorithm would have a 50⁄ chance of being first in the reordering, the second port number would have a 25⁄ chance of being first in the reordering, and so on. Because the seeking computers would use different orderings, the likelihood of finding a busy port is reduced. For example, if the first eight port numbers are randomly selected, then it is

possible that eight seeking computers could be simultaneously dialing ports in different sequences which would reduce the chances of dialing a busy port.

Locating a Portal Computer

Each computer that can connect to the broadcast channel has a list of one or more portal computers through which it can connect to the broadcast channel. In one embodiment, each computer has the same set of portal computers. A seeking computer locates a portal computer that is connected to the broadcast channel by successively dialing the ports of each portal computer in the order specified by an algorithm. A seeking computer could select the first portal computer and then dial all its ports until a call-in port of a computer that is fully connected to the broadcast channel is found. If no call-in port is found, then the seeking computer would select the next portal computer and repeat the process until a portal computer with such a call-in port is found. A problem with such a seeking technique is that all user ports of each portal computer are dialed until a portal computer fully connected to the broadcast channel is found. In an alternate embodiment, the seeking computer selects a port number according to the algorithm and then dials each portal computer at that port number. If no acceptable call-in port to the broadcast channel is found, then the seeking computer selects the next port number and repeats the process. Since the call-in ports are likely allocated at lower-ordered port numbers, the seeking computer first dials the port numbers that are most likely to be call-in ports of the broadcast channel. The seeking computers may have a maximum search depth, that is the number of ports that it will dial when seeking a portal computer that is fully connected. If the seeking computer exhausts its search depth, then either the broadcast channel has not yet been established or, if the seeking computer is also a portal computer, it can then establish the broadcast channel with itself as the first fully connected computer.

When a seeking computer locates a portal computer that is itself not fully connected, the two computers do not connect when they first locate each other because the broadcast channel may already be established and accessible through a higher-ordered port number on another portal computer. If the two seeking computers were to connect to each other, then two disjoint broadcast channels would be formed. Each seeking computer can share its experience in trying to locate a portal computer with the other seeking computer. In particular, if one seeking computer has searched all the portal computers to a depth of eight,

then the one seeking computer can share that it has searched to a depth of eight with another seeking computer. If that other seeking computer has searched to a depth of, for example, only four, it can skip searching through depths five through eight and that other seeking computer can advance its searching to a depth of nine.

In one embodiment, each computer may have a different set of portal computers and a different maximum search depth. In such a situation, it may be possible that two disjoint broadcast channels are formed because a seeking computer cannot locate a fully connected port computer at a higher depth. Similarly, if the set of portal computers are disjoint, then two separate broadcast channels would be formed.

Identifying Neighbors for a Seeking Computer

As described above, the neighbors of a newly connecting computer are preferably selected randomly from the set of currently connected computers. One advantage of the broadcast channel, however, is that no computer has global knowledge of the broadcast channel. Rather, each computer has local knowledge of itself and its neighbors. This limited local knowledge has the advantage that all the connected computers are peers (as far as the broadcasting is concerned) and the failure of any one computer (actually any three computers when in the 4-regular and 4-connect form) will not cause the broadcast channel to fail. This local knowledge makes it difficult for a portal computer to randomly select four neighbors for a seeking computer.

To select the four computers, a portal computer sends an edge connection request message through one of its internal connections that is randomly selected. The receiving computer again sends the edge connection request message through one of its internal connections that is randomly selected. This sending of the message corresponds to a random walk through the graph that represents the broadcast channel. Eventually, a receiving computer will decide that the message has traveled far enough to represent a randomly selected computer. That receiving computer will offer the internal connection upon which it received the edge connection request message to the seeking computer for edge pinning. Of course, if either of the computers at the end of the offered internal connection are already neighbors of the seeking computer, then the seeking computer cannot connect through that internal connection. The computer that decided that the message has

traveled far enough will detect this condition of already being a neighbor and send the message to a randomly selected neighbor.

In one embodiment, the distance that the edge connection request message travels is established by the portal computer to be approximately twice the estimated diameter of the broadcast channel. The message includes an indication of the distance that it is to travel. Each receiving computer decrements that distance to travel before sending the message on. The computer that receives a message with a distance to travel that is zero is considered to be the randomly selected computer. If that randomly selected computer cannot connect to the seeking computer (*e.g.*, because it is already connected to it), then that randomly selected computer forwards the edge connection request to one of its neighbors with a new distance to travel. In one embodiment, the forwarding computer toggles the new distance to travel between zero and one to help prevent two computers from sending the message back and forth between each other.

Because of the local nature of the information maintained by each computer connected to the broadcast channel, the computers need not generally be aware of the diameter of the broadcast channel. In one embodiment, each message sent through the broadcast channel has a distance traveled field. Each computer that forwards a message increments the distance traveled field. Each computer also maintains an estimated diameter of the broadcast channel. When a computer receives a message that has traveled a distance that indicates that the estimated diameter is too small, it updates its estimated diameter and broadcasts an estimated diameter message. When a computer receives an estimated diameter message that indicates a diameter that is larger than its own estimated diameter, it updates its own estimated diameter. This estimated diameter is used to establish the distance that an edge connection request message should travel.

External Data Representation

The computers connected to the broadcast channel may internally store their data in different formats. For example, one computer may use 32-bit integers, and another computer may use 64-bit integers. As another example, one computer may use ASCII to represent text and another computer may use Unicode. To allow communications between heterogeneous computers, the messages sent over the broadcast channel may use the XDR ("eXternal Data Representation") format.

The underlying peer-to-peer communications protocol may send multiple messages in a single message stream. The traditional technique for retrieving messages from a stream has been to repeatedly invoke an operating system routine to retrieve the next message in the stream. The retrieval of each message may require two calls to the operating system: one to retrieve the size of the next message and the other to retrieve the number of bytes indicated by the retrieved size. Such calls to the operating system can, however, be very slow in comparison to the invocations of local routines. To overcome the inefficiencies of such repeated calls, the broadcast technique in one embodiment, uses XDR to identify the message boundaries in a stream of messages. The broadcast technique may request the operating system to provide the next, for example, 1,024 bytes from the stream. The broadcast technique can then repeatedly invoke the XDR routines to retrieve the messages and use the success or failure of each invocation to determine whether another block of 1,024 bytes needs to be retrieved from the operating system. The invocation of XDR routines do not involve system calls and are thus more efficient than repeated system calls.

M-Regular

In the embodiment described above, each fully connected computer has four internal connections. The broadcast technique can be used with other numbers of internal connections. For example, each computer could have 6, 8, or any even number of internal connections. As the number of internal connections increase, the diameter of the broadcast channel tends to decrease, and thus propagation time for a message tends to decrease. The time that it takes to connect a seeking computer to the broadcast channel may, however, increase as the number of internal connections increases. When the number of internal connectors is even, then the broadcast channel can be maintained as m-regular and m-connected (in the steady state). If the number of internal connections is odd, then when the broadcast channel has an odd number of computers connected, one of the computers will have less than that odd number of internal connections. In such a situation, the broadcast network is neither m-regular nor m-connected. When the next computer connects to the broadcast channel, it can again become m-regular and m-connected. Thus, with an odd number of internal connections, the broadcast channel toggles between being and not being m-regular and m-connected.

Components

Figure 6 is a block diagram illustrating components of a computer that is connected to a broadcast channel. The above description generally assumed that there was only one broadcast channel and that each computer had only one connection to that broadcast channel. More generally, a network of computers may have multiple broadcast channels, each computer may be connected to more than one broadcast channel, and each computer can have multiple connections to the same broadcast channel. The broadcast channel is well suited for computer processes (*e.g.*, application programs) that execute collaboratively, such as network meeting programs. Each computer process can connect to one or more broadcast channels. The broadcast channels can be identified by channel type (*e.g.*, application program name) and channel instance that represents separate broadcast channels for that channel type. When a process attempts to connect to a broadcast channel, it seeks a process currently connected to that broadcast channel that is executing on a portal computer. The seeking process identifies the broadcast channel by channel type and channel instance.

Computer 600 includes multiple application programs 601 executing as separate processes. Each application program interfaces with a broadcaster component 602 for each broadcast channel to which it is connected. The broadcaster component may be implement as an object that is instantiated within the process space of the application program. Alternatively, the broadcaster component may execute as a separate process or thread from the application program. In one embodiment, the broadcaster component provides functions (*e.g.*, methods of class) that can be invoked by the application programs. The primary functions provided may include a connect function that an application program invokes passing an indication of the broadcast channel to which the application program wants to connect. The application program may provide a callback routine that the broadcaster component invokes to notify the application program that the connection has been completed, that is the process enters the fully connected state. The broadcaster component may also provide an acquire message function that the application program can invoke to retrieve the next message that is broadcast on the broadcast channel. Alternatively, the application program may provide a callback routine (which may be a virtual function provided by the application program) that the broadcaster component invokes to notify the application program that a broadcast message has been received. Each broadcaster component allocates a call-in port using the hashing algorithm. When calls are answered at the call-in port, they are transferred to other ports that serve as the external and internal ports.

The computers connecting to the broadcast channel may include a central processing unit, memory, input devices (*e.g.*, keyboard and pointing device), output devices (*e.g.*, display devices), and storage devices (*e.g.*, disk drives). The memory and storage devices are computer-readable medium that may contain computer instructions that implement the broadcaster component. In addition, the data structures and message structures may be stored or transmitted via a signal transmitted on a computer-readable media, such as a communications link.

Figure 7 is a block diagram illustrating the sub-components of the broadcaster component in one embodiment. The broadcaster component includes a connect component 701, an external dispatcher 702, an internal dispatcher 703 for each internal connection, an acquire message component 704 and a broadcast component 712. The application program may provide a connect callback component 710 and a receive response component 711 that are invoked by the broadcaster component. The application program invokes the connect component to establish a connection to a designated broadcast channel. The connect component identifies the external port and installs the external dispatcher for handling messages that are received on the external port. The connect component invokes the seek portal computer component 705 to identify a portal computer that is connected to the broadcast channel and invokes the connect request component 706 to ask the portal computer (if fully connected) to select neighbor processes for the newly connecting process. The external dispatcher receives external messages, identifies the type of message, and invokes the appropriate handling routine 707. The internal dispatcher receives the internal messages, identifies the type of message, and invokes the appropriate handling routine 708. The received broadcast messages are stored in the broadcast message queue 709. The acquire message component is invoked to retrieve messages from the broadcast queue. The broadcast component is invoked by the application program to broadcast messages in the broadcast channel.

The following tables list messages sent by the broadcaster components.

EXTERNAL MESSAGES

Message Type	Description
seeking_connection_call	Indicates that a seeking process would like to know whether the receiving process is fully connected to the broadcast channel
connection_request_call	Indicates that the sending process would like the receiving process to initiate a connection of the sending process to the broadcast channel
edge_proposal_call	Indicates that the sending process is proposing an edge through which the receiving process can connect to the broadcast channel (*i.e.*, edge pinning)
port_connection_call	Indicates that the sending process is proposing a port through which the receiving process can connect to the broadcast channel
connected_stmt	Indicates that the sending process is connected to the broadcast channel
condition_repair_stmt	Indicates that the receiving process should disconnect from one of its neighbors and connect to one of the processes involved in the neighbors with empty port condition

INTERNAL MESSAGES

Message Type	Description
broadcast_stmt	Indicates a message that is being broadcast through the broadcast channel for the application programs
connection_port_search_stmt	Indicates that the designated process is looking for a port through which it can connect to the broadcast channel
connection_edge_search_call	Indicates that the requesting process is looking for an edge through which it can connect to the broadcast channel
connection_edge_search_resp	Indicates whether the edge between this process and the sending neighbor has been accepted by the requesting party
diameter_estimate_stmt	Indicates an estimated diameter of the broadcast channel
diameter_reset_stmt	Indicates to reset the estimated diameter to indicated diameter
disconnect_stmt	Indicates that the sending neighbor is disconnecting from the broadcast channel
condition_check_stmt	Indicates that neighbors with empty port condition have

	been detected
condition_double_check_stmt	Indicates that the neighbors with empty ports have the same set of neighbors
shutdown_stmt	Indicates that the broadcast channel is being shutdown

Flow Diagrams

Figures 8-34 are flow diagrams illustrating the processing of the broadcaster component in one embodiment. Figure 8 is a flow diagram illustrating the processing of the connect routine in one embodiment. This routine is passed a channel type (*e.g.*, application name) and channel instance (*e.g.*, session identifier), that identifies the broadcast channel to which this process wants to connect. The routine is also passed auxiliary information that includes the list of portal computers and a connection callback routine. When the connection is established, the connection callback routine is invoked to notify the application program. When this process invokes this routine, it is in the seeking connection state. When a portal computer is located that is connected and this routine connects to at least one neighbor, this process enters the partially connected state, and when the process eventually connects to four neighbors, it enters the fully connected state. When in the small regime, a fully connected process may have less than four neighbors. In block 801, the routine opens the call-in port through which the process is to communicate with other processes when establishing external and internal connections. The port is selected as the first available port using the hashing algorithm described above. In block 802, the routine sets the connect time to the current time. The connect time is used to identify the instance of the process that is connected through this external port. One process may connect to a broadcast channel of a certain channel type and channel instance using one call-in port and then disconnects, and another process may then connect to that same broadcast channel using the same call-in port. Before the other process becomes fully connected, another process may try to communicate with it thinking it is the fully connected old process. In such a case, the connect time can be used to identify this situation. In block 803, the routine invokes the seek portal computer routine passing the channel type and channel instance. The seek portal computer routine attempts to locate a portal computer through which this process can connect to the broadcast channel for the passed type and instance. In decision block 804, if the seek portal computer routine is successful in locating a fully connected process on that portal computer, then the routine continues at block 805, else the routine returns an unsuccessful indication. In decision block 805, if no portal computer other than the portal computer on which the process is executing was located, then this is the first process to fully connect to broadcast channel and the routine continues at block 806, else the routine continues at block 808. In block 806, the routine invokes the achieve connection routine to change the state of this process to fully connected. In block 807, the routine installs the external dispatcher for processing messages received through this process' external port for the passed channel type and channel instance. When a message is received through that external port, the external dispatcher is invoked. The routine then returns. In block 808, the routine installs an external dispatcher. In block 809, the routine invokes the connect request routine to initiate the process of identifying neighbors for the seeking computer. The routine then returns.

Figure 9 is a flow diagram illustrating the processing of the seek portal computer routine in one embodiment. This routine is passed the channel type and channel instance of the broadcast channel to which this process wishes to connect. This routine, for each search depth (*e.g.*, port number), checks the portal computers at that search depth. If a portal computer is located at that search depth with a process that is fully connected to the broadcast channel, then the routine returns an indication of success. In blocks 902-911, the routine loops selecting each search depth until a process is located. In block 902, the routine selects the next search depth using a port number ordering algorithm. In decision block 903, if all the search depths have already been selected during this execution of the loop, that is for the currently selected depth, then the routine returns a failure indication, else the routine continues at block 904. In blocks 904-911, the routine loops selecting each portal computer and determining whether a process of that portal computer is connected to (or attempting to connect to) the broadcast channel with the passed channel type and channel instance. In block 904, the routine selects the next portal computer. In decision block 905, if all the portal computers have already been selected, then the routine loops to block 902 to select the next search depth, else the routine continues at block 906. In block 906, the routine dials the selected portal computer through the port represented by the search depth. In decision block 907, if the dialing was successful, then the routine continues at block 908, else the routine loops to block 904 to select the next portal computer. The dialing will be successful if the dialed port is the call-in port of the broadcast channel of the passed channel type and channel instance of a process executing on that portal computer. In block 908, the routine invokes a contact process routine, which contacts the answering process of the portal computer through the dialed port and determines whether that process is fully connected to the broadcast channel. In block 909, the routine hangs up on the selected portal computer. In decision block 910, if the answering process is fully connected to the broadcast channel, then the routine returns a success indicator, else the routine continues at block 911. In block 911, the routine invokes the check for external call routine to determine whether an external call has been made to this process as a portal computer and processes that call. The routine then loops to block 904 to select the next portal computer.

Figure 10 is a flow diagram illustrating the processing of the contact process routine in one embodiment. This routine determines whether the process of the selected portal computer that answered the call-in to the selected port is fully connected to the broadcast channel. In block 1001, the routine sends an external message (*i.e.*, seeking_connection_call) to the answering process indicating that a seeking process wants to know whether the answering process is fully connected to the broadcast channel. In block 1002, the routine receives the external response message from the answering process. In decision block 1003, if the external response message is successfully received (*i.e.*, seeking_connection_resp), then the routine continues at block 1004, else the routine returns. Wherever the broadcast component requests to receive an external message, it sets a time out period. If the external message is not received within that time out period, the broadcaster component checks its own call-in port to see if another process is calling it. In particular, the dialed process may be calling the dialing process, which may result in a deadlock situation. The broadcaster component may repeat the receive request several times. If the expected message is not received, then the broadcaster component handles the error as appropriate. In decision block 1004, if the answering process indicates in its response message that it is fully connected to the broadcast channel, then the routine continues at block 1005, else the routine continues at block 1006. In block 1005, the routine adds the selected portal computer to a list of connected portal computers and then returns. In block 1006, the routine adds the answering process to a list of fellow seeking processes and then returns.

Figure 11 is a flow diagram illustrating the processing of the connect request routine in one embodiment. This routine requests a process of a portal computer that was identified as being fully connected to the broadcast channel to initiate the connection of this process to the broadcast channel. In decision block 1101, if at least one process of a portal computer was located that is fully connected to the broadcast channel, then the routine continues at block 1103, else the routine continues at block 1102. A process of the portal computer may no longer be in the list if it recently disconnected from the broadcast channel. In one embodiment, a seeking computer may always search its entire search depth and find multiple portal computers through which it can connect to the broadcast channel. In block 1102, the routine restarts the process of connecting to the broadcast channel and returns. In block 1103, the routine dials the process of one of the found portal computers through the call-in port. In decision block 1104, if the dialing is successful, then the routine continues at block 1105, else the routine continues at block 1113. The dialing may be unsuccessful if, for example, the dialed process recently disconnected from the broadcast channel. In block 1105, the routine sends an external message to the dialed process requesting a connection to the broadcast channel (*i.e.*, connection_request_call). In block 1106, the routine receives the response message (*i.e.*, connection_request_resp). In decision block 1107, if the response message is successfully received, then the routine continues at block 1108, else the routine continues at block 1113. In block 1108, the routine sets the expected number of holes (*i.e.*, empty internal connections) for this process based on the received response. When in the large regime, the expected number of holes is zero. When in the small regime, the expected number of holes varies from one to three. In block 1109, the routine sets the estimated diameter of the broadcast channel based on the received response. In decision block 1111, if the dialed process is ready to connect to this process as indicated by the response message, then the routine continues at block 1112, else the routine continues at block 1113. In block 1112, the routine invokes the add neighbor routine to add the answering process as a neighbor to this process. This adding of the answering process typically occurs when the broadcast channel is in the small regime. When in the large regime, the random walk search for a neighbor is performed. In block 1113, the routine hangs up the external connection with the answering process computer and then returns.

Figure 12 is a flow diagram of the processing of the check for external call routine in one embodiment. This routine is invoked to identify whether a fellow seeking process is attempting to establish a connection to the broadcast channel through this process. In block 1201, the routine attempts to answer a call on the call-in port. In decision block 1202, if the answer is successful, then the routine continues at block 1203, else the routine returns. In block 1203, the routine receives the external message from the external port. In decision block 1204, if the type of the message indicates that a seeking process is calling (*i.e.*, seeking_connection_call), then the routine continues at block 1205, else the routine returns. In block 1205, the routine sends an external message (*i.e.*, seeking_connection_resp) to the other seeking process indicating that this process is also is seeking a connection. In decision block 1206, if the sending of the external message is successful, then the routine continues at block 1207, else the routine returns. In block 1207, the routine adds the other seeking process to a list of fellow seeking processes and then returns. This list may be used if this process can find no process that is fully connected to the broadcast channel. In which case, this process may check to see if any fellow seeking process were successful in connecting to the broadcast channel. For example, a fellow seeking process may become the first process fully connected to the broadcast channel.

Figure 13 is a flow diagram of the processing of the achieve connection routine in one embodiment. This routine sets the state of this process to fully connected to the broadcast channel and invokes a callback routine to notify the application program that the process is now fully connected to the requested broadcast channel. In block 1301, the routine sets the connection state of this process to fully connected. In block 1302, the routine notifies fellow seeking processes that it is fully connected by sending a connected external message to them (*i.e.*, connected_stmt). In block 1303, the routine invokes the connect callback routine to notify the application program and then returns.

Figure 14 is a flow diagram illustrating the processing of the external dispatcher routine in one embodiment. This routine is invoked when the external port receives a message. This routine retrieves the message, identifies the external message type, and invokes the appropriate routine to handle that message. This routine loops processing each message until all the received messages have been handled. In block 1401, the routine answers (*e.g.*, picks up) the external port and retrieves an external message. In decision block 1402, if a message was retrieved, then the routine continues at block 1403, else the routine hangs up on the external port in block 1415 and returns. In decision block 1403, if the message type is for a process seeking a connection (*i.e.*, seeking_connection_call), then the routine invokes the handle seeking connection call routine in block 1404, else the routine continues at block 1405. In decision block 1405, if the message type is for a connection request call (*i.e.*, connection_request_call), then the routine invokes the handle connection request call routine in block 1406, else the routine continues at block 1407. In decision block 1407, if the message type is edge proposal call (*i.e.*, edge_proposal_call), then the routine invokes the handle edge proposal call routine in block 1408, else the routine continues at block 1409. In decision block 1409, if the message type is port connect call (*i.e.*, port_connect_call), then the routine invokes the handle port connection call routine in block 1410, else the routine continues at block 1411. In decision block 1411, if the message type is a connected statement (*i.e.*, connected_stmt), the routine invokes the handle connected statement in block 1112, else the routine continues at block 1212. In decision block 1412, if the message type is a condition repair statement (*i.e.*, condition_repair_stmt), then the routine invokes the handle condition repair routine in block 1413, else the routine loops to block 1414 to process the next message. After each handling routine is invoked, the routine loops to block 1414. In block 1414, the routine hangs up on the external port and continues at block 1401 to receive the next message.

Figure 15 is a flow diagram illustrating the processing of the handle seeking connection call routine in one embodiment. This routine is invoked when a seeking process is calling to identify a portal computer through which it can connect to the broadcast channel. In decision block 1501, if this process is currently fully connected to the broadcast channel identified in the message, then the routine continues at block 1502, else the routine continues at block 1503. In block 1502, the routine sets a message to indicate that this process is fully connected to the broadcast channel and continues at block 1505. In block 1503, the routine sets a message to indicate that this process is not fully connected. In block 1504, the routine adds the identification of the seeking process to a list of fellow seeking processes. If this process is not fully connected, then it is attempting to connect to the broadcast channel. In block 1505, the routine sends the external message response (*i.e.*, seeking_connection_resp) to the seeking process and then returns.

Figure 16 is a flow diagram illustrating processing of the handle connection request call routine in one embodiment. This routine is invoked when the calling process wants this process to initiate the connection of the process to the broadcast channel. This routine either allows the calling process to establish an internal connection with this process (*e.g.*, if in the small regime) or starts the process of identifying a process to which the calling process can connect. In decision block 1601, if this process is currently fully connected to the broadcast channel, then the routine continues at block 1603, else the routine hangs up on the external port in block 1602 and returns. In block 1603, routine sets the number of holes that the calling process should expect in the response message. In block 1604, the routine sets the estimated diameter in the response message. In block 1605, the routine indicates whether this process is ready to connect to the calling process. This process is ready to connect when the number of its holes is greater than zero and the calling process is not a neighbor of this process. In block 1606, the routine sends to the calling process an external message that is responsive to the connection request call (*i.e.*, connection_request_resp). In block 1607, the routine notes the number of holes that the calling process needs to fill as indicated in the request message. In decision block 1608, if this process is ready to connect to the calling process, then the routine continues at block 1609, else the routine continues at block 1611. In block 1609, the routine invokes the add neighbor routine to add the calling process as a neighbor. In block 1610, the routine decrements the number of holes that the calling process needs to fill and continues at block 1611. In block 1611, the routine hangs up on the external port. In decision block 1612, if this process has no holes or the estimated diameter is greater than one (*i.e.*, in the large regime), then the routine continues at block 1613, else the routine continues at block 1616. In blocks 1613-1615, the routine loops forwarding a request for an edge through which to connect to the calling process to the broadcast channel. One request is forwarded for each pair of holes of the calling process that needs to be filled. In decision block 1613, if the number of holes of the calling process to be filled is greater than or equal to two, then the routine continues at block 1614, else the routine continues at block 1616. In block 1614, the routine invokes the forward connection edge search routine. The invoked routine is passed to an indication of the calling process and the random walk distance. In one embodiment, the distance is twice in the estimated diameter of the broadcast channel. In block 1614, the routine decrements the holes left to fill by two and loops to block 1613. In decision block 1616, if there is still a hole to fill, then the routine continues at block 1617, else the routine returns. In block 1617, the routine invokes the fill hole routine passing the identification of the calling process. The fill hole routine broadcasts a connection port search statement (*i.e.*, connection_port_search_stmt) for a hole of a connected process through which the calling process can connect to the broadcast channel. The routine then returns.

Figure 17 is a flow diagram illustrating the processing of the add neighbor routine in one embodiment. This routine adds the process calling on the external port as a neighbor to this process. In block 1701, the routine identifies the calling process on the external port. In block 1702, the routine sets a flag to indicate that the neighbor has not yet received the broadcast messages from this process. This flag is used to ensure that there are no gaps in the messages initially sent to the new neighbor. The external port becomes the internal port for this connection. In decision block 1703, if this process is in the seeking connection state, then this process is connecting to its first neighbor and the routine continues at block 1704, else the routine continues at block 1705. In block 1704, the routine sets the connection state of this process to partially connected. In block 1705, the routine adds the calling process to the list of neighbors of this process. In block 1706, the routine installs an internal dispatcher for the new neighbor. The internal dispatcher is invoked when a message is received from that new neighbor through the internal port of that new neighbor. In decision block 1707, if this process buffered up messages while not fully connected, then the routine continues at block 1708, else the routine continues at block 1709. In one embodiment, a process that is partially connected may buffer the messages that it receives through an internal connection so that it can send these messages as it connects to new neighbors. In block 1708, the routine sends the buffered messages to the new neighbor through the internal port. In decision block 1709, if the number of holes of this process equals the expected number of holes, then this process is fully connected and the routine continues at block 1710, else the routine continues at block 1711. In block 1710, the routine invokes the achieve connected routine to indicate that this process is fully connected. In decision block 1711, if the number of holes for this process is zero, then the routine continues at block 1712, else the routine returns. In block 1712, the routine deletes any pending edges and then returns. A pending edge is an edge that has been proposed to this process for edge pinning, which in this case is no longer needed.

Figure 18 is a flow diagram illustrating the processing of the forward connection edge search routine in one embodiment. This routine is responsible for passing along a request to connect a requesting process to a randomly selected neighbor of this process through the internal port of the selected neighbor, that is part of the random walk. In decision block 1801, if the forwarding distance remaining is greater than zero, then the routine continues at block 1804, else the routine continues at block 1802. In decision block 1802, if the number of neighbors of this process is greater than one, then the routine continues at block 1804, else this broadcast channel is in the small regime and the routine continues at block 1803. In decision block 1803, if the requesting process is a neighbor of this process, then the routine returns, else the routine continues at block 1804. In blocks 1804-1807, the routine loops attempting to send a connection edge search call internal message (*i.e.*, connection_edge_search_call) to a randomly selected neighbor. In block 1804, the routine randomly selects a neighbor of this process. In decision block 1805, if all the neighbors of this process have already been selected, then the routine cannot forward the message and the routine returns, else the routine continues at block 1806. In block 1806, the routine sends a connection edge search call internal message to the selected neighbor. In decision block 1807, if the sending of the message is successful, then the routine continues at block 1808, else the routine loops to block 1804 to select the next neighbor. When the sending of an internal message is unsuccessful, then the neighbor may have disconnected from the broadcast channel in an unplanned manner. Whenever such a situation is detected by the broadcaster component, it attempts to find another neighbor by invoking the fill holes routine to fill a single hole or the forward connecting edge search routine to fill two holes. In block 1808, the routine notes that the recently sent connection edge search call has not yet been acknowledged and indicates that the edge to this neighbor is reserved if the remaining forwarding distance is less than or equal to one. It is reserved because the selected neighbor may offer this edge to the requesting process for edge pinning. The routine then returns.

Figure 19 is a flow diagram illustrating the processing of the handle edge proposal call routine. This routine is invoked when a message is received from a proposing process that proposes to connect an edge between the proposing process and one of its neighbors to this process for edge pinning. In decision block 1901, if the number of holes of this process minus the number of pending edges is greater than or equal to one, then this process still has holes to be filled and the routine continues at block 1902, else the routine continues at block 1911. In decision block 1902, if the proposing process or its neighbor is a neighbor of this process, then the routine continues at block 1911, else the routine continues at block 1903. In block 1903, the routine indicates that the edge is pending between this process and the proposing process. In decision block 1904, if a proposed neighbor is already pending as a proposed neighbor, then the routine continues at block 1911, else the routine continues at block 1907. In block 1907, the routine sends an edge proposal response as an external message to the proposing process (*i.e.*, edge_proposal_resp) indicating that the proposed edge is accepted. In decision block 1908, if the sending of the message was successful, then the routine continues at block 1909, else the routine returns. In block 1909, the routine adds the edge as a pending edge. In block 1910, the routine invokes the add neighbor routine to add the proposing process on the external port as a neighbor. The routine then returns. In block 1911, the routine sends an external message (*i.e.*, edge_proposal_resp) indicating that this proposed edge is not accepted. In decision block 1912, if the number of holes is odd, then the routine continues at block 1913, else the routine returns. In block 1913, the routine invokes the fill hole routine and then returns.

Figure 20 is a flow diagram illustrating the processing of the handle port connection call routine in one embodiment. This routine is invoked when an external message is received then indicates that the sending process wants to connect to one hole of this process. In decision block 2001, if the number of holes of this process is greater than zero, then the routine continues at block 2002, else the routine continues at block 2003. In decision block 2002, if the sending process is not a neighbor, then the routine continues at block 2004, else the routine continues to block 2003. In block 2003, the routine sends a port connection response external message (*i.e.*, port_connection_resp) to the sending process that indicates that it is not okay to connect to this process. The routine then returns. In block 2004, the routine sends a port connection response external message to the sending process that indicates that is okay to connect this process. In decision block 2005, if the sending of the message was successful, then the routine continues at block 2006, else the routine continues at block 2007. In block 2006, the routine invokes the add neighbor routine to add the sending process as a neighbor of this process and then returns. In block 2007, the routine hangs up the external connection. In block 2008, the routine invokes the connect request routine to request that a process connect to one of the holes of this process. The routine then returns.

Figure 21 is a flow diagram illustrating the processing of the fill hole routine in one embodiment. This routine is passed an indication of the requesting process. If this process is requesting to fill a hole, then this routine sends an internal message to other processes. If another process is requesting to fill a hole, then this routine invokes the routine to handle a connection port search request. In block 2101, the routine initializes a connection port search statement internal message (*i.e.*, connection_port_search_stmt). In decision block 2102, if this process is the requesting process, then the routine continues at block 2103, else the routine continues at block 2104. In block 2103, the routine distributes the message to the neighbors of this process through the internal ports and then returns. In block 2104, the routine invokes the handle connection port search routine and then returns.

Figure 22 is a flow diagram illustrating the processing of the internal dispatcher routine in one embodiment. This routine is passed an indication of the neighbor who sent the internal message. In block 2201, the routine receives the internal message. This routine identifies the message type and invokes the appropriate routine to handle the message. In block 2202, the routine assesses whether to change the estimated diameter of the broadcast channel based on the information in the received message. In decision block 2203, if this process is the originating process of the message or the message has already been received (*i.e.*, a duplicate), then the routine ignores the message and continues at block 2208, else the routine continues at block 2203A. In decision block 2203A, if the process is partially connected, then the routine continues at block 2203B, else the routine continues at block 2204. In block 2203B, the routine adds the message to the pending connection buffer and continues at block 2204. In decision blocks 2204-2207, the routine decodes the message type and invokes the appropriate routine to handle the message. For example, in decision block 2204, if the type of the message is broadcast statement (*i.e.*, broadcast_stmt), then the routine invokes the handle broadcast message routine in block 2205. After invoking the appropriate handling routine, the routine continues at block 2208. In decision block 2208, if the partially connected buffer is full, then the routine continues at block 2209, else the routine continues at block 2210. The broadcaster component collects all its internal messages in a buffer while partially connected so that it can forward the messages as it connects to new neighbors. If, however, that buffer becomes full, then the process assumes that it is now fully connected and that the expected number of connections was too high, because the broadcast channel is now in the small regime. In block 2209, the routine invokes the achieve connection routine and then continues in block 2210. In decision block 2210, if the application program message queue is empty, then the routine returns, else the routine continues at block 2212. In block 2212, the routine invokes the receive response routine passing the acquired message and then returns. The received response routine is a callback routine of the application program.

Figure 23 is a flow diagram illustrating the processing of the handle broadcast message routine in one embodiment. This routine is passed an indication of the originating process, an indication of the neighbor who sent the broadcast message, and the broadcast message itself. In block 2301, the routine performs the [illegible] of order processing for this message. The broadcaster component queues messages from each originating process until it can send them in sequence number order to the application program. In block 2302, the routine invokes the distribute broadcast message routine to forward the message to the neighbors of this process. In decision block 2303, if a newly connected neighbor is waiting to receive messages, then the routine continues at block 2304, else the routine returns. In block 2304, the routine sends the messages in the correct order if possible for each originating process and then returns.

Figure 24 is a flow diagram illustrating the processing of the distribute broadcast message routine in one embodiment. This routine sends the broadcast message to each of the neighbors of this process, except for the neighbor who sent the message to this process. In block 2401, the routine selects the next neighbor other than the neighbor who sent the message. In decision block 2402, if all such neighbors have already been selected, then the routine returns. In block 2403, the routine sends the message to the selected neighbor and then loops to block 2401 to select the next neighbor.

Figure 26 is a flow diagram illustrating the processing of the handle connection port search statement routine in one embodiment. This routine is passed an indication of the neighbor that sent the message and the message itself. In block 2601, the routine invokes the distribute internal message which sends the message to each of its neighbors other than the sending neighbor. In decision block 2602, if the number of holes of this process is greater than zero, then the routine continues at block 2603, else the routine returns. In decision block 2603, if the requesting process is a neighbor, then the routine continues at block 2605, else the routine continues at block 2604. In block 2604, the routine invokes the court neighbor routine and then returns. The court neighbor routine connects this process to the requesting process if possible. In block 2605, if this process has one hole, then the neighbors with empty ports condition exists and the routine continues at block 2606, else the routine returns. In block 2606, the routine generates a condition check message (*i.e.*, condition_check) that includes a list of this process' neighbors. In block 2607, the routine sends the message to the requesting neighbor.

Figure 27 is a flow diagram illustrating the processing of the court neighbor routine in one embodiment. This routine is passed an indication of the prospective neighbor for this process. If this process can connect to the prospective neighbor, then it sends a port connection call external message to the prospective neighbor and adds the prospective neighbor as a neighbor. In decision block 2701, if the prospective neighbor is already a neighbor, then the routine returns, else the routine continues at block 2702. In block 2702, the routine dials the prospective neighbor. In decision block 2703, if the number of holes of this process is greater than zero, then the routine continues at block 2704, else the routine continues at block 2706. In block 2704, the routine sends a port connection call external message (*i.e.*, port_connection_call) to the prospective neighbor and receives its response (*i.e.*, port_connection_resp). Assuming the response is successfully received, in block 2705, the routine adds the prospective neighbor as a neighbor of this process by invoking the add neighbor routine. In block 2706, the routine hangs up with the prospect and then returns.

Figure 28 is a flow diagram illustrating the processing of the handle connection edge search call routine in one embodiment. This routine is passed a indication of the neighbor who sent the message and the message itself. This routine either forwards the message to a neighbor or proposes the edge between this process and the sending neighbor to the requesting process for edge pinning. In decision block 2801, if this process is not the requesting process or the number of holes of the requesting process is still greater than or equal to two, then the routine continues at block 2802, else the routine continues at block 2813. In decision block 2802, if the forwarding distance is greater than zero, then the random walk is not complete and the routine continues at block 2803, else the routine continues at block 2804. In block 2803, the routine invokes the forward connection edge search routine passing the identification of the requesting process and the decremented forwarding distance. The routine then continues at block 2815. In decision block 2804, if the requesting process is a neighbor or the edge between this process and the sending neighbor is reserved because it has already been offered to a process, then the routine continues at block 2805, else the routine continues at block 2806. In block 2805, the routine invokes the forward connection edge search routine passing an indication of the requesting party and a toggle indicator that alternatively indicates to continue the random walk for one or two more computers. The routine then continues at block 2815. In block 2806, the routine dials the requesting process via the call-in port. In block 2807, the routine sends an edge proposal call external message (*i.e.*, edge_proposal_call) and receives the response (*i.e.*, edge_proposal_resp). Assuming that the response is successfully received, the routine continues at block 2808. In decision block 2808, if the response indicates that the edge is acceptable to the requesting process, then the routine continues at block 2809, else the routine continues at block 2812. In block 2809, the routine reserves the edge between this process and the sending neighbor. In block 2810, the routine adds the requesting process as a neighbor by invoking the add neighbor routine. In block 2811, the routine removes the sending neighbor as a neighbor. In block 2812, the routine hangs up the external port and continues at block 2815. In decision block 2813, if this process is the requesting process and the number of holes of this process equals one, then the routine continues at block 2814, else the routine continues at block 2815. In block 2814, the routine invokes the fill hole routine. In block 2815, the routine sends an connection edge search response message (*i.e.*, connection_edge_search_response) to the sending neighbor indicating acknowledgement and then returns. The graphs are sensitive to parity. That is, all possible paths starting from a node and ending at that node will have an even length unless the graph has a cycle whose length is odd. The broadcaster component uses a toggle indicator to vary the random walk distance between even and odd distances.

Figure 29 is a flow diagram illustrating the processing of the handle connection edge search response routine in one embodiment. This routine is passed as indication of the requesting process, the sending neighbor, and the message. In block 2901, the routine notes that the connection edge search response (*i.e.*, connection_edge_search_resp) has been received and if the forwarding distance is less than or equal to one unreserves the edge between this process and the sending neighbor. In decision block 2902, if the requesting process indicates that the edge is acceptable as indicated in the message, then the routine continues at block 2903, else the routine returns. In block 2903, the routine reserves the edge between this process and the sending neighbor. In block 2904, the routine removes the sending neighbor as a neighbor. In block 2905, the routine invokes the court neighbor routine to connect to the requesting process. In decision block 2906, if the invoked routine was unsuccessful, then the routine continues at block 2907, else the routine returns. In decision block 2907, if the number of holes of this process is greater than zero, then the routine continues at block 2908, else the routine returns. In block 2908, the routine invokes the fill hole routine and then returns.

Figure 30 is a flow diagram illustrating the processing of the broadcast routine in one embodiment. This routine is invoked by the application program to broadcast a message on the broadcast channel. This routine is passed the message to be broadcast. In decision block 3001, if this process has at least one neighbor, then the routine continues at block 3002, else the routine returns since it is the only process connected to be broadcast channel. In block 3002, the routine generates an internal message of the broadcast statement type (*i.e.*, broadcast _stmt). In block 3003, the routine sets the sequence number of the message. In block 3004, the routine invokes the distribute internal message routine to broadcast the message on the broadcast channel. The routine returns.

Figure 31 is a flow diagram illustrating the processing of the acquire message routine in one embodiment. The acquire message routine may be invoked by the application program or by a callback routine provided by the application program. This routine returns a message. In block 3101, the routine pops the message from the message queue of the broadcast channel. In decision block 3102, if a message was retrieved, then the routine returns an indication of success, else the routine returns indication of failure.

Figures 32-34 are flow diagrams illustrating the processing of messages associated with the neighbors with empty ports condition. Figure 32 is a flow diagram illustrating processing of the handle condition check message in one embodiment. This message is sent by a neighbor process that has one hole and has received a request to connect to a hole of this process. In decision block 3201, if the number of holes of this process is equal to one, then the routine continues at block 3202, else the neighbors with empty ports condition does not exist any more and the routine returns. In decision block 3202, if the sending neighbor and this process have the same set of neighbors, the routine continues at block 3203, else the routine continues at block 3205. In block 3203, the routine initializes a condition double check message (*i.e.*, condition_double_check) with the list of neighbors of this process. In block 3204, the routine sends the message internally to a neighbor other than sending neighbor. The routine then returns. In block 3205, the routine selects a neighbor of the sending process that is not also a neighbor of this process. In block 3206, the routine sends a condition repair message (*i.e.*, condition_repair_stmt) externally to the selected process. In block 3207, the routine invokes the add neighbor routine to add the selected neighbor as a neighbor of this process and then returns.

Figure 33 is a flow diagram illustrating processing of the handle condition repair statement routine in one embodiment. This routine removes an existing neighbor and connects to the process that sent the message. In decision block 3301, if this process has no holes, then the routine continues at block 3302, else the routine continues at block 3304. In block 3302, the routine selects a neighbor that is not involved in the neighbors with empty ports condition. In block 3303, the routine removes the selected neighbor as a neighbor of this process. Thus, this process that is executing the routine now has at least one hole. In block 3304, the routine invokes the add neighbor routine to add the process that sent the message as a neighbor of this process. The routine then returns.

Figure 34 is a flow diagram illustrating the processing of the handle condition double check routine. This routine determines whether the neighbors with empty ports condition really is a problem or whether the broadcast channel is in the small regime. In decision block 3401, if this process has one hole, then the routine continues at block 3402, else the routine continues at block 3403. If this process does not have one hole, then the set of neighbors of this process is not the same as the set of neighbors of the sending process. In decision block 3402, if this process and the sending process have the same set of neighbors, then the broadcast channel is not in the small regime and the routine continues at block 3403, else the routine continues at block 3406. In decision block 3403, if this process has no holes, then the routine returns, else the routine continues at block 3404. In block 3404, the routine sets the estimated diameter for this process to one. In block 3405, the routine broadcasts a diameter reset internal message (*i.e.*, diameter_reset) indicating that the estimated diameter is one and then returns. In block 3406, the routine creates a list of neighbors of this process. In block 3407, the routine sends the condition check message (*i.e.*, condition_check_stmt) with the list of neighbors to the neighbor who sent the condition double check message and then returns.

From the above description, it will be appreciated that although specific embodiments of the technology have been described, various modifications may be made without deviating from the spirit and scope of the invention. For example, the communications on the broadcast channel may be encrypted. Also, the channel instance or session identifier may be a very large number (*e.g.*, 128 bits) to help prevent an unauthorized user to maliciously tap into a broadcast channel. The portal computer may also enforce security and not allow an unauthorized user to connect to the broadcast channel. Accordingly, the invention is not limited except by the claims.

CLAIMS

1. A computer-based method for adding a participant to a network of participants, each participant being connected to three or more other participants, the method comprising:

identifying pair of participants of the network that are connected;

disconnecting the participants of the identified pair from each other; and

connecting each participant of the identified pair of participants to the added participant.

2. The method of claim 1 wherein each participant is connected to 4 participants.

3. The method of claim 1 wherein the identifying of a pair includes randomly selecting a pair of participants that are connected.

4. The method of claim 3 wherein the randomly selecting of a pair includes sending a message through the network on a randomly selected path.

5. The method of claim 4 wherein when a participant receives the message, the participant sends the message to a randomly selected participant to which it is connected.

6. The method of claim 4 wherein the randomly selected path is approximately proportional to the diameter of the network.

7. The method of claim 1 wherein the participant to be added requests a portal computer to initiate the identifying of the pair of participants.

8. The method of claim 7 wherein the initiating of the identifying of the pair of participants includes the portal computer sending a message to a connected participant requesting an edge connection.

9. The method of claim 8 wherein the portal computer indicates that the message is to travel a certain distance and wherein the participant that receives the message after the message has traveled that certain distance is one of the participants of the identified pair of participants.

10. The method of claim 9 wherein the certain distance is approximately twice the diameter of the network.

11. The method of claim 1 wherein the participants are connected via the Internet.

12. The method of claim 1 wherein the participants are connected via TCP/IP connections.

13. The method of claim 1 wherein the participants are computer processes.

14. A computer-based method for adding nodes to a graph that is m-regular and m-connected to maintain the graph as m-regular, where m is four or greater, the method comprising:

identifying p pairs of nodes of the graph that are connected, where p is one half of m;

disconnecting the nodes of each identified pair from each other; and

connecting each node of the identified pairs of nodes to the added node.

15. The method of claim 14 wherein identifying of the p pairs of nodes includes randomly selecting a pair of connected nodes.

16. The method of claim 14 wherein the nodes are computers and the connections are point-to-point communications connections.

17. The method of claim 14 wherein m is even.

18. A method of initiating adding of a participant to a network, the method comprising:

receiving a connection message from the participant to be added; and

sending a connection edge search message to a neighbor participant of the participant that received the message wherein the connection edge search message is forwarded to neighbor participants until a participant that receives the connection edge search message decides to connect to the participant to be added.

19. The method of claim 18 wherein the sent connection edge search message includes an indication of the number of participants to which the connection edge search message should be forwarded.

20. The method of claim 19 wherein the number of participants is based on the diameter of the network.

21. The method of claim 19 wherein the number of participants is approximately twice the diameter.

22. The method of claim 18 wherein when a participant decides to connect to the participant to be added, the neighbor participant that sent the connection edge search message to the participant that decided to connect also decides to connect to the participant to be added.

23. The method of claim 18 wherein participants that receive the connection edge search message forward the connection edge search message to a randomly selected neighbor.

24. A method in a computer system for connecting to a new participant of a network, the method comprising:

receiving at a participant a connection edge search message;

identifying a neighbor participant of the participant that received the connection edge search message;

notifying the neighbor participant to connect to the new participant;

disconnecting the participant from the identified neighbor participant; and

connecting the participant to the new participant.

25. The method of claim 24 including determining whether the participant is the last participant in a path of participants through which the connection edge search message was sent.

26. The method of claim 25 wherein when the participant is not the last participant in the path, sending the connection edge search message to a neighbor of the participant.

27. The method of claim 26 including randomly selecting the neighbor participant to which the connection edge search message is to be sent.

28. The method of claim 24 wherein the received connection edge search message includes an indication of the number of participants through which the connection edge search message is to be sent.

29. The method of claim 24 including when the participant is already a neighbor of the new participant, sending the connection edge search message to a neighbor participant of the participant.

30. The method of claim 24 wherein the participants are computer processes.

31. The method of claim 24 wherein the connections are point-to-point connections.

32. A computer-readable medium containing instructions for controlling a computer system to connect a participant to a network of participants, each participant being connected to three or more other participants, the network representing a broadcast channel wherein each participant forwards broadcast messages that it receives to its neighbor participants, by a method comprising:

identifying a pair of participants of the network that are connected;

disconnecting the participants of the identified pair from each other; and

connecting each participant of the identified pair of participants to the added participant.

33. The computer-readable medium of claim 32 wherein each participant is connected to 4 participants.

34. The computer-readable medium of claim 32 wherein the identifying of a pair includes randomly selecting a pair of participants that are connected.

35. The computer-readable medium of claim 34 wherein the randomly selecting of a pair includes sending a message through the network on a randomly selected path.

36. The computer-readable medium of claim 35 wherein when a participant receives the message, the participant sends the message to a randomly selected participant to which it is connected.

37. The computer-readable medium of claim 35 wherein the randomly selected path is approximately twice a diameter of the network.

38. The computer-readable medium of claim 32 wherein the participant to be added requests a portal computer to initiate the identifying of the pair of participants.

39. The computer-readable medium of claim 38 wherein the initiating of the identifying of the pair of participants includes the portal computer sending a message to a connected participant requesting an edge connection.

40. The computer-readable medium of claim 38 wherein the portal computer indicates that the message is to travel a certain distance and wherein the participant that receives the message after the message has traveled that certain distance is one of the identified pair of participants.

41. A method in a computer system for connecting to a participant of a network, the method comprising:

receiving at a participant a connection port search message sent by a requesting participant; and

when the participant has a port that is available through which it can connect to the requesting participant,

sending a port connection message to the requesting participant proposing that the requesting participant connect to the available port of the participant; and

when the participant receives a port proposal response message that indicates the requesting participant accepts to connect to the available port, connecting the participant to the requesting participant.

42. The method of claim 41 including:

when the participant does not have a port that is available through which it can connect to the requesting participant, sending the connection port search message to a neighbor participant.

43. The method of claim 41 wherein a port is available when the requesting participant is not already connected to the participant and the participant has an empty port.

44. A method in a computer system of detecting neighbors with empty ports condition in a network, the method comprising:

receiving at a first participant a connection port search message indicating that a second participant has an empty port; and

when the first participant is already connected to the second participant and the first participant has an empty port, sending a condition check message from the first participant to the second participant wherein the condition check message identifies neighbors of the first participant.

45. The method of claim 44 including:

when the second participant receives the condition check message,

when the second participant does not have the same neighbors as the first participant, sending a condition repair message to third participant that is a neighbor of the first participant but is not a neighbor of the second participant.

46. The method of claim 45 including:

when the third participant receives the condition repair message,

disconnecting from a neighbor of the third participant other than the first participant; and

connecting to the second participant.

47. The method of claim 44 including:

when the second participant receives the condition check message,

when the second participant has the same neighbors as the first participant, sending a condition double check message to a third participant that is a neighbor of the second participant.

48. The method of claim 47 including:

when the third participant receives the condition double check message,

when the third participant does not have the same neighbors as the first participant, sending a condition check message to a fourth participant that is not the first participant or the second participant.

49. The method of claim 48 ~~including~~:

when the fourth participant receives the condition check message,

sending a condition repair message to a fifth participant directing the fifth participant to connect to the first participant or the second participant.

This Form is for INTERNAL PTO USE ONLY
It does NOT get mailed to the applicant.

NOTICE OF FILING / CLAIM FEE(S) DUE (CALCULATION SHEET)

APPLICATION NUMBER: 09/629570

Total Fee Calculation

	Fee Code Sm./Lg.	Total # Claims		Number Extra	X	Fee Sm. Entry	Fee Lg. Entry	=	Total
Basic Filing Fee	201/101						690.00	=	690.00
Total Claims >20	203/103	48	-20 =	28	X		18.00	=	504.00
Independent Claims >3	202/102	7	-3 =	4	X		78.00	=	312.00
Mult. Dep Claim Present	204/104							=	
Surcharge	205/105						130.00	=	130.00
English Translation	139								

TOTAL FEE CALCULATION 1636.00

Fees due upon filing the application.

Total Filing Fees Due = $ 1636.00

Less Filing Fees Submitted - $ ______

BALANCE DUE = $ 1636.00

D. Thomas
Office of Initial Patent Examination

FORM OIPE-RAM-01 (Rev. 12/97)

PATENT APPLICATION FEE DETERMINATION RECORD
Effective December 29, 1999

Application or Docket Number: 09/629570

CLAIMS AS FILED - PART I

FOR	NUMBER FILED (Column 1)	NUMBER EXTRA (Column 2)	SMALL ENTITY RATE	SMALL ENTITY FEE	OR	OTHER THAN SMALL ENTITY RATE	OTHER THAN SMALL ENTITY FEE
BASIC FEE				345.00	OR		690.00
TOTAL CLAIMS	48 minus 20=	* 28	X$ 9=		OR	X$18=	504.00
INDEPENDENT CLAIMS	7 minus 3 =	* 4	X39=		OR	X78=	312.00
MULTIPLE DEPENDENT CLAIM PRESENT			+130=		OR	+260=	
			TOTAL		OR	TOTAL	1506.00

SMALL ENTITY TYPE ☐

* If the difference in column 1 is less than zero, enter "0" in column 2

CLAIMS AS AMENDED - PART II

AMENDMENT A

9	CLAIMS REMAINING AFTER AMENDMENT (Column 1)		HIGHEST NUMBER PREVIOUSLY PAID FOR (Column 2)	PRESENT EXTRA (Column 3)	SMALL ENTITY RATE	ADDITIONAL FEE	OR	OTHER THAN SMALL ENTITY RATE	ADDITIONAL FEE
Total	* 26	Minus	** 48	= —	X$ 9=		OR	X$18=	
Independent	* 3	Minus	*** 7	= —	X39=		OR	X78=	
FIRST PRESENTATION OF MULTIPLE DEPENDENT CLAIM					+130=		OR	+260=	
					TOTAL ADDIT. FEE		OR	TOTAL ADDIT. FEE	

AMENDMENT B

	CLAIMS REMAINING AFTER AMENDMENT (Column 1)		HIGHEST NUMBER PREVIOUSLY PAID FOR (Column 2)	PRESENT EXTRA (Column 3)	SMALL ENTITY RATE	ADDITIONAL FEE	OR	OTHER THAN SMALL ENTITY RATE	ADDITIONAL FEE
Total	*	Minus	**	=	X$ 9=		OR	X$18=	
Independent	*	Minus	***	=	X39=		OR	X78=	
FIRST PRESENTATION OF MULTIPLE DEPENDENT CLAIM					+130=		OR	+260=	
					TOTAL ADDIT. FEE		OR	TOTAL ADDIT. FEE	

AMENDMENT C

	CLAIMS REMAINING AFTER AMENDMENT (Column 1)		HIGHEST NUMBER PREVIOUSLY PAID FOR (Column 2)	PRESENT EXTRA (Column 3)	SMALL ENTITY RATE	ADDITIONAL FEE	OR	OTHER THAN SMALL ENTITY RATE	ADDITIONAL FEE
Total	*	Minus	**	=	X$ 9=		OR	X$18=	
Independent	*	Minus	***	=	X39=		OR	X78=	
FIRST PRESENTATION OF MULTIPLE DEPENDENT CLAIM					+130=		OR	+260=	
					TOTAL ADDIT. FEE		OR	TOTAL ADDIT. FEE	

* If the entry in column 1 is less than the entry in column 2, write "0" in column 3.
** If the "Highest Number Previously Paid For" IN THIS SPACE is less than 20, enter "20."
***If the "Highest Number Previously Paid For" IN THIS SPACE is less than 3, enter "3."
The "Highest Number Previously Paid For" (Total or Independent) is the highest number found in the appropriate box in column 1.

FORM PTO-875
(Rev. 12/99)

Patent and Trademark Office, U.S. DEPARTMENT OF COMMERCE

UNITED STATES PATENT AND TRADEMARK OFFICE

COMMISSIONER FOR PATENTS
UNITED STATES PATENT AND TRADEMARK OFFICE
WASHINGTON, D.C. 20231
www.uspto.gov

APPLICATION NUMBER	FILING/RECEIPT DATE	FIRST NAMED APPLICANT	ATTORNEY DOCKET NUMBER
09/629,570	07/31/2000	Virgil E. Bourassa	PTOSB/05 (4/98)

25096
PERKINS COIE LLP
PATENT-SEA
PO BOX 1247
SEATTLE, WA 98111-1247

FORMALITIES LETTER

OC000000005422946

Date Mailed: 09/25/2000

NOTICE TO FILE MISSING PARTS OF NONPROVISIONAL APPLICATION

FILED UNDER 37 CFR 1.53(b)

Filing Date Granted

An application number and filing date have been accorded to this application. The item(s) indicated below, however, are missing. Applicant is given TWO MONTHS from the date of this Notice within which to file all required items and pay any fees required below to avoid abandonment. Extensions of time may be obtained by filing a petition accompanied by the extension fee under the provisions of 37 CFR 1.136(a).

- The statutory basic filing fee is missing.
  Applicant must submit **$ 690** *to complete the basic filing fee and/or file a small entity statement claiming such status (37 CFR 1.27).*
- Total additional claim fee(s) for this application is $816.
  - $**504** for **28** total claims over 20.
  - $**312** for **4** independent claims over 3 .
- The oath or declaration is missing.
  A properly signed oath or declaration in compliance with 37 CFR 1.63, identifying the application by the above Application Number and Filing Date, is required.
- To avoid abandonment, a late filing fee or oath or declaration surcharge as set forth in 37 CFR 1.16(e) of $130 for a non-small entity, must be submitted with the missing items identified in this letter.

- **The balance due by applicant is $ 1636.**

A copy of this notice MUST be returned with the reply.

M. Jackson

Customer Service Center
Initial Patent Examination Division (703) 308-1202

PART 3 - OFFICE COPY

SECOND #3

PATENT

I hereby certify that on the date specified below, this correspondence is being deposited with the United States Postal Service as first-class mail in an envelope addressed to Box Missing Parts, Commissioner for Patents, Washington, DC 20231.

10/30/00
Date

Jeanne Connelly

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicants	:	Fred B. Holt and Virgil E. Bourassa
Application No.	:	09/629,570
Filed	:	July 31, 2000
For	:	JOINING A BROADCAST CHANNEL

Docket No.	:	030048002US
Date	:	October 30, 2000

Box Missing Parts
Commissioner for Patents
Washington, DC 20231

RESPONSE TO NOTICE TO FILE MISSING PARTS OF APPLICATION

Sir:

In response to the Notice to File Missing Parts dated September 25, 2000, please find enclosed a Declaration, Power of Attorney, Authorization for Extensions of Time Under 37 CFR § 1.136(a)(3), and a copy of the Notice to File Missing Parts for the above-identified application.

The fees have been calculated as follows:

Basic Fee	$	710.00
Total Claims (49, 29 extra)		522.00
Independent Claims (7, 4 extra)		320.00
Missing Parts Surcharge		130.00
Total	$	1682.00

The Commissioner is hereby authorized to charge the fees of $1,682.00 and any additional filing fees or to credit any overpayment to Deposit Account No. 50-0665. A duplicate copy of this response is enclosed.

Respectfully submitted,

Perkins Coie LLP

Maurice J. Pirio

Maurice J. Pirio
Registration No. 33,273

MJP:jc

Enclosures:
- Postcard
- Copy of this Response
- Declaration
- Power of Attorney
- Authorization for Extensions of Time Under 37 CFR § 1.136(a)(3)
- Copy of Notice to File Missing Parts

PERKINS COIE LLP
P.O. Box 1247
Seattle, Washington 98111-1247
(206) 583-8888
FAX: (206) 583-8500

#3

UNITED STATES PATENT AND TRADEMARK OFFICE

COMMISSIONER FOR PATENTS
UNITED STATES PATENT AND TRADEMARK OFFICE
WASHINGTON, D.C. 20231
www.uspto.gov

APPLICATION NUMBER	FILING/RECEIPT DATE	FIRST NAMED APPLICANT	ATTORNEY DOCKET NUMBER
09/629,570	07/31/2000	Virgil E. Bourassa	PTOSB/05 (4/98)

25096
PERKINS COIE LLP
PATENT-SEA
PO BOX 1247
SEATTLE, WA 98111-1247

FORMALITIES LETTER

OC000000005422946

Date Mailed: 09/25/2000

NOTICE TO FILE MISSING PARTS OF NONPROVISIONAL APPLICATION

FILED UNDER 37 CFR 1.53(b)

Filing Date Granted

An application number and filing date have been accorded to this application. The item(s) indicated below, however, are missing. Applicant is given TWO MONTHS from the date of this Notice within which to file all required items and pay any fees required below to avoid abandonment. Extensions of time may be obtained by filing a petition accompanied by the extension fee under the provisions of 37 CFR 1.136(a).

- The statutory basic filing fee is missing.
  Applicant must submit **$ 690** *to complete the basic filing fee and/or file a small entity statement claiming such status (37 CFR 1.27).*
- Total additional claim fee(s) for this application is $816.
  - **$504** for **28** total claims over 20.
  - **$312** for **4** independent claims over 3 .
- The oath or declaration is missing.
  A properly signed oath or declaration in compliance with 37 CFR 1.63, identifying the application by the above Application Number and Filing Date, is required.
- To avoid abandonment, a late filing fee or oath or declaration surcharge as set forth in 37 CFR 1.16(e) of $130 for a non-small entity, must be submitted with the missing items identified in this letter.

- **The balance due by applicant is $ 1636.**

11/09/2000 KZEWDIE 00000022 500665 09629570
01 FC:101 710.00 CH
02 FC:105 130.00 CH
03 FC:103 522.00 CH
04 FC:102 320.00 CH

A copy of this notice <u>MUST</u> be returned with the reply.

M Jackson

Customer Service Center
Initial Patent Examination Division (703) 308-1202

PART 2 - COPY TO BE RETURNED WITH RESPONSE

#3

DECLARATION

As the below-named inventors, we declare that:

Our residences, post office addresses, and citizenships are as stated below under our names.

We believe we are the original, first, and joint inventors of the subject matter claimed and for which a patent is sought on the invention entitled "JOINING A BROADCAST CHANNEL," the specification of which was filed in the U.S. Patent and Trademark Office on July 31, 2000 and assigned application number 09/629,570.

We have reviewed and understand the contents of the above-identified specification, including the claims, as amended by any amendment specifically referred to above.

We acknowledge our duty to disclose information which is material to the patentability of this application in accordance with 37 C.F.R. § 1.56(a).

We further declare that all statements made herein of our own knowledge are true and that all statements made on information and belief are believed to be true; and further, that these statements were made with the knowledge that the making of willfully false statements and the like is punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code, and may jeopardize the validity of any patent issuing from this patent application.

Fred B. Holt

Date 25 Oct 2000

Residence	:	City of Seattle State of Washington
Citizenship	:	United States of America
P.O. Address	:	5520 31st Avenue NE Seattle, Washington 98105

Q:\Clients\Boeing IP (03004)\8002 (Joining)\Declaration.doc

Virgil E. Bourassa

Date 10/26/2000

Residence	:	City of Bellevue
		State of Washington
Citizenship	:	United States of America
P.O. Address	:	16110 SE 24th Street Bellevue, Washington 98008

PATENT

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicants	:	Fred B. Holt and Virgil E. Bourassa
Application No.	:	09/629,570
Filed	:	July 31, 2000
For	:	JOINING A BROADCAST CHANNEL

Art Unit	:	2744
Docket No.	:	030048002US

Commissioner for Patents
Washington, DC 20231

ELECTION UNDER 37 C.F.R. §§ 3.71 AND 3.73

AND POWER OF ATTORNEY

Sir:

The undersigned, being Assignee of the entire interest in the above-identified application by virtue of an Assignment filed concurrently herewith, a copy of which is enclosed, hereby elects under 37 C.F.R. § 3.71, to prosecute the application to the exclusion of the inventors.

Assignee hereby appoints JERRY A. RIEDINGER, Registration No. 30,582; MAURICE J. PIRIO, Registration No. 33,273; JOHN C. STEWART, Registration No. 40,188; MICHAEL D. BROADDUS, Registration No. 41,637; BRIAN P. MCQUILLEN, Registration No. 41,989; CATHERINE HONG TRAN, Registration No. 43,960; ROBERT G. WOOLSTON, Registration No. 37,263; PAUL T. PARKER, Registration No. 38,264; JOHN M. WECHKIN, Registration No. 42,216; CHRISTOPHER DALEY-WATSON, Registration No. 34,807; STEVEN D. LAWRENZ, Registration No. 37,376; JAMES A.D. WHITE, Registration No. 43,985; and FRANK ABRAMONTE, Registration No. 38,066, of the firm of Perkins Coie LLP and ROBERT

L. GULLETTE, Registration No. 26,899, PAUL C. CULLOM, JR., Registration No. 25,580, ANN K. GALBRAITH, Registration No. 33,530, JAMES P. HAMLEY, Registration No. 28,081, JOHN C. HAMMAR, Registration No. 29,928, LAWRENCE W. NELSON, Registration No. 34,684 and ROBERT R. RICHARDSON, Registration No. 40,143 of The Boeing Company, as the principal attorneys with full power of substitution, association, and revocation to prosecute said application, to transact all business in the Patent and Trademark Office connected therewith, and to receive the letters patent therefor. Please direct all telephone calls to Maurice J. Pirio at (206) 583-8888 and telecopies to (206) 583-8500.

Please direct all correspondence to:

Patent-SEA
Perkins Coie LLP
P.O. Box 1247
Seattle, Washington 98111-1247
Attn: Maurice J. Pirio

Pursuant to 37 C.F.R. § 3.73, the undersigned duly authorized designee of Assignee certifies that the evidentiary documents have been reviewed, specifically the Assignment to The Boeing Company filed concurrently herewith for recording, a copy of which is attached hereto, and certifies that to the best of my knowledge and belief, title remains in the name of the Assignee.

The Boeing Company

10/27/00 — Robert R. Richardson

Date — Name of Person Signing

Counsel

Tile of Person Signing

MJP:jc

Enclosure:
Copy of Assignment

Q:\Clients\Boeing IP (03004)\8002 (Joining)\Election & POA.doc

ASSIGNMENT

WHEREAS, we, Fred B. Holt and Virgil E. Bourassa ("ASSIGNORS"), having post office addresses of 5520 31st Avenue NE, Seattle, Washington 98105 and 16110 SE 24th Street, Bellevue, Washington 98008, respectively, are the joint inventors of an invention entitled "JOINING A BROADCAST CHANNEL," as described and claimed in the specification for which an application for United States letters patent was filed on July 31, 2000 and assigned Application No. 09/629,570.

WHEREAS, The Boeing Company ("ASSIGNEE"), a corporation of the State of Delaware having its principal place of business at Seattle, Washington, is desirous of acquiring the entire right, title, and interest in and to the invention and in and to any patents that may be granted therefor in the United States and in any and all foreign countries;

NOW, THEREFORE, in exchange for good and valuable consideration, the receipt and sufficiency of which is hereby acknowledged, ASSIGNORS hereby sell, assign, and transfer unto ASSIGNEE, its legal representatives, successors, and assigns, the entire right, title and interest in and to the invention as set forth in the above-mentioned application, including any continuations, continuations-in-part, divisions, reissues, re-examinations, or extensions thereof, any other inventions described in the application, and any and all patents of the United States of America and all foreign countries that may be issued for the invention, including the right to file foreign applications directly in the name of ASSIGNEE and to claim priority rights deriving from the United States application to which foreign applications are entitled by virtue of international convention, treaty or otherwise, the invention, application and all patents on the invention to be held and enjoyed by ASSIGNEE and its successors and assigns for their use and benefit and of their successors and assigns as fully and entirely as the same would have been held and enjoyed by ASSIGNORS had this assignment, transfer, and sale not been made.

UPON THE ABOVE-STATED CONSIDERATIONS, ASSIGNORS agree to not execute any writing or do any act whatsoever conflicting with this assignment, and at any time upon request, without further or additional consideration but at the expense of ASSIGNEE, execute all instruments and documents and do such additional acts as ASSIGNEE may deem necessary or desirable to perfect ASSIGNEE's enjoyment of this grant, and render all necessary assistance required for the making and prosecution of applications for United States and foreign patents on the invention, for litigation regarding the patents, or for the purpose of protecting title to the invention or patents therefor.

ASSIGNORS authorize and request the Commissioner of Patents and Trademarks to issue any Patent of the United States that may be issued for the invention to ASSIGNEE.

26 Oct 2000 ____________________ [signature] ____________________

Date — Fred B. Holt

State of Washington)

) ss.

County of King)

I certify that I know or have satisfactory evidence that Fred B. Holt is the person who appeared before me, and the person acknowledged that he signed this instrument and acknowledged it to be his free and voluntary act for the uses and purposes mentioned in the instrument.

DEANNE R. FLANAGAN COMMISSION EXPIRES NOTARY PUBLIC AUG 24, 2003 STATE OF WASHINGTON

Dated 10·26·2000

Signature of Notary Public [signature]

Printed Name DeAnne R. Flanagan

My appointment expires 8·24·03

10/26/2000 — Date

Virgil E. Bourassa

State of Washington)
) ss.
County of King)

I certify that I know or have satisfactory evidence that Virgil E. Bourassa is the person who appeared before me, and the person acknowledged that he signed this instrument and acknowledged it to be his free and voluntary act for the uses and purposes mentioned in the instrument.

DEANNE R. FLANAGAN
COMMISSION EXPIRES
NOTARY
PUBLIC
AUG 24 2003
STATE OF WASHINGTON

Dated 10·26·00

Signature of Notary Public

Printed Name DeAnne R. Flanagan

My appointment expires 8·24·03

Q:\Clients\Boeing IP (03004)\8002 (Joining)\Assignment.doc

PATENT

#10

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicants : Fred B. Holt and Virgil E. Bourassa
Application No. : 09/629,570
Filed : July 31, 2000
For : JOINING A BROADCAST CHANNEL

Art Unit : 2744
Docket No. : 030048002US
Date : October 30, 2000

Commissioner for Patents
Washington, DC 20231

AUTHORIZATION FOR EXTENSIONS OF TIME UNDER 37 C.F.R. § 1.136(A)(3)

Sir:

With respect to the above-identified application, the Commissioner is authorized to treat any concurrent or future reply requiring a petition for an extension of time under 37 C.F.R. § 1.136(a)(3) for its timely submission as incorporating a petition therefor for the appropriate length of time. The Commissioner is also authorized to charge any fees which may be required, or credit any overpayment, to Deposit Account No. 50-0665.

Date October 30, 2000

Maurice J. Pirio
Maurice J. Pirio
Registration No. 33,273

PERKINS COIE LLP
P.O. Box 1247
Seattle, Washington 98111-1247
(206) 583-8888
FAX: (206) 583-8500

UNITED STATES DEPARTMENT OF COMMERCE
Patent and Trademark Office
ASSISTANT SECRETARY AND COMMISSIONER
OF PATENTS AND TRADEMARKS
Washington, D.C. 20231

#4

RECEIVED
MAY 3 - 2001
Technology Center 2100

CHANGE OF ADDRESS/POWER OF ATTORNEY

FILE LOCATION 21C1 SERIAL NUMBER 09629570 PATENT NUMBER

THE CORRESPONDENCE ADDRESS HAS BEEN CHANGED TO CUSTOMER # 25096

THE PRACTITIONERS OF RECORD HAVE BEEN CHANGED TO CUSTOMER # 25096

THE FEE ADDRESS HAS BEEN CHANGED TO CUSTOMER # 25096

ON 04/12/01 THE ADDRESS OF RECORD FOR CUSTOMER NUMBER 25096 IS:

PERKINS COIE LLP
1201 3RD AVENUE , SUITE 4800
SEATTLE WA 98101-3099

AND THE PRACTITIONERS OF RECORD FOR CUSTOMER NUMBER 25096 ARE:

30582 33273 33904 34807 37263 37376 38264 40188 41637 41989
42216 43960 43985 46140

PTO INSTRUCTIONS: PLEASE TAKE THE FOLLOWING ACTION WHEN THE CORRESPONDENCE ADDRESS HAS BEEN CHANGED TO CUSTOMER NUMBER: RECORD, ON THE NEXT AVAILABLE CONTENTS LINE OF THE FILE JACKET, 'ADDRESS CHANGE TO CUSTOMER NUMBER'. LINE THROUGH THE OLD ADDRESS ON THE FILE JACKET LABEL AND ENTER ONLY THE 'CUSTOMER NUMBER' AS THE NEW ADDRESS. FILE THIS LETTER IN THE FILE JACKET. WHEN ABOVE CHANGES ARE ONLY TO FEE ADDRESS AND/OR PRACTITIONERS OF RECORD, FILE LETTER IN THE FILE JACKET.
THIS FILE IS ASSIGNED TO GAU 2154.

PTO-FMD
TALBOT-1/97

GP/2154 #5

Attorney Docket No. 030048002US

I hereby certify that this correspondence is being deposited with the U.S. Postal Service with sufficient postage as First Class Mail in an envelope addressed to: Commissioner for Patents, Washington, D.C., 20231, on:

Date: 4/18/02 By: [signature]
Jeanne Connelly

PATENT

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

IN RE APPLICATION OF:

VIRGIL E. BOURASSA AND FRED B. HOLT

APPLICATION NO.: 09/629,570

FILED: JULY 31, 2000

FOR: JOINING A BROADCAST CHANNEL

Information Disclosure Statement Within Three Months of Application Filing or Before First Action – 37 CFR 1.97(b)

Commissioner for Patents
Washington, D.C. 20231

Sir:

1. Timing of Submission

This information disclosure is being filed within three months of the filing date of this application or date of entry into the national stage of an international application or before the mailing date of a first Office action on the merits, whichever occurs last [37 CFR 1.97(b)]. The references listed on the enclosed Form PTO/SB/08A (modified) may be material to the examination of this application; the Examiner is requested to make them of record in the application.

2. Cited Information

☒ Copies of the following references are enclosed:

- ☒ All cited references
- ☐ References marked by asterisks
- ☐ The following:

☐ Copies of the following references can be found in parent application Ser. No.

- ☐ All cited references
- ☐ References marked by asterisks
- ☐ The following:

☐ The following references are not in English. For each such reference, the undersigned has enclosed (i) a translation of the reference; (ii) a copy of a communication from a foreign patent office or International Searching Authority citing the reference, (iii) a copy of a reference which appears to be an English-language counterpart, or (iv) an English-language abstract for the reference prepared by a third party. Applicant has not verified that the translation, English-language counterpart or third-party abstract is an accurate representation of the teachings of the non-English reference, though, and reserves the right to demonstrate otherwise.

- ☐ All cited references
- ☐ References marked by ampersands
- ☐ The following:

3. Effect of Information Disclosure Statement (37 CFR 1.97(h))

This Information Disclosure Statement is not to be construed as a representation that: (i) a search has been made; (ii) additional information material to the examination of this application does not exist; (iii) the information, protocols, results and the like reported by third parties are accurate or enabling; or (iv) the cited information is, or is considered to be, material to patentability. In addition, applicant does not admit that any enclosed item of information constitutes prior art to the subject invention and specifically reserves the right to demonstrate that any such reference is not prior art.

4. Fee Payment

No fees are believed due. However, should the Commissioner determine that fees are due in order for this Information Disclosure Statement to be considered, the Commissioner is hereby authorized to charge such fees to Deposit Account No. 50-0665.

5. Patent Term Adjustment (37 CFR 1.704(d))

☐ The undersigned states that each item of information submitted herewith was cited in a communication from a foreign patent office in a counterpart application and that this communication was not received by any individual designated in 37 C.F.R. § 1.56(c) more than thirty days prior to the filing of this statement. 37 C.F.R. § 1.704(d).

Respectfully submitted,
Perkins Coie LLP

Date: 4-18-02

Maurice J. Pirio
Registration No. 33,273

Correspondence Address:
Customer No. 25096
Perkins Coie LLP
P.O. Box 1247
Seattle, Washington 98111-1247
Phone: (206) 583-8888

GRAPH THEORY WITH APPLICATIONS

J. A. Bondy and U. S. R. Murty

Department of Combinatorics and Optimization,
University of Waterloo,
Ontario, Canada

AMERICAN ELSEVIER PUBLISHING CO., INC.

First published in Great Britain 1976 by
The Macmillan Press Ltd

Reprinted 1977

First published in the U.S.A. 1976 by
AMERICAN ELSEVIER PUBLISHING CO., INC.
52 Vanderbilt Avenue
New York, N.Y. 10017

ISBN 0-444-19451-7
LCCCN 75-29826

Printed in Great Britain

To our parents

Contents

1 Graphs and Subgraphs

1.1 GRAPHS AND SIMPLE GRAPHS

Many real-world situations can conveniently be described by means of a diagram consisting of a set of points together with lines joining certain pairs of these points. For example, the points could represent people, with lines joining pairs of friends; or the points might be communication centres, with lines representing communication links. Notice that in such diagrams one is mainly interested in whether or not two given points are joined by a line; the manner in which they are joined is immaterial. A mathematical abstraction of situations of this type gives rise to the concept of a graph.

A *graph* G is an ordered triple $(V(G), E(G), \psi_G)$ consisting of a nonempty set $V(G)$ of *vertices*, a set $E(G)$, disjoint from $V(G)$, of *edges*, and an *incidence function* ψ_G that associates with each edge of G an unordered pair of (not necessarily distinct) vertices of G. If e is an edge and u and v are vertices such that $\psi_G(e) = uv$, then e is said to *join* u and v; the vertices u and v are called the *ends* of e.

Two examples of graphs should serve to clarify the definition.

Example 1

$$G = (V(G), E(G), \psi_G)$$

where

$$V(G) = \{v_1, v_2, v_3, v_4, v_5\}$$
$$E(G) = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}$$

and ψ_G is defined by

$$\psi_G(e_1) = v_1v_2,\ \psi_G(e_2) = v_2v_3,\ \psi_G(e_3) = v_3v_3,\ \psi_G(e_4) = v_3v_4$$
$$\psi_G(e_5) = v_2v_4,\ \psi_G(e_6) = v_4v_5,\ \psi_G(e_7) = v_2v_5,\ \psi_G(e_8) = v_2v_5$$

Example 2

$$H = (V(H), E(H), \psi_H)$$

where

$$V(H) = \{u, v, w, x, y\}$$
$$E(H) = \{a, b, c, d, e, f, g, h\}$$

and ψ_H is defined by

$$\psi_H(a) = uv,\quad \psi_H(b) = uu,\quad \psi_H(c) = vw,\quad \psi_H(d) = wx$$
$$\psi_H(e) = vx,\quad \psi_H(f) = wx,\quad \psi_H(g) = ux,\quad \psi_H(h) = xy$$

1 Graphs and Subgraphs

1.1 GRAPHS AND SIMPLE GRAPHS

Many real-world situations can conveniently be described by means of a diagram consisting of a set of points together with lines joining certain pairs of these points. For example, the points could represent people, with lines joining pairs of friends; or the points might be communication centres, with lines representing communication links. Notice that in such diagrams one is mainly interested in whether or not two given points are joined by a line; the manner in which they are joined is immaterial. A mathematical abstraction of situations of this type gives rise to the concept of a graph.

A *graph* G is an ordered triple $(V(G), E(G), \psi_G)$ consisting of a nonempty set $V(G)$ of *vertices*, a set $E(G)$, disjoint from $V(G)$, of *edges*, and an *incidence function* ψ_G that associates with each edge of G an unordered pair of (not necessarily distinct) vertices of G. If e is an edge and u and v are vertices such that $\psi_G(e) = uv$, then e is said to *join* u and v; the vertices u and v are called the *ends* of e.

Two examples of graphs should serve to clarify the definition.

Example 1

$$G = (V(G), E(G), \psi_G)$$

where

$$V(G) = \{v_1, v_2, v_3, v_4, v_5\}$$
$$E(G) = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}$$

and ψ_G is defined by

$$\psi_G(e_1) = v_1v_2,\ \psi_G(e_2) = v_2v_3,\ \psi_G(e_3) = v_3v_3,\ \psi_G(e_4) = v_3v_4$$
$$\psi_G(e_5) = v_2v_4,\ \psi_G(e_6) = v_4v_5,\ \psi_G(e_7) = v_2v_5,\ \psi_G(e_8) = v_2v_5$$

Example 2

$$H = (V(H), E(H), \psi_H)$$

where

$$V(H) = \{u, v, w, x, y\}$$
$$E(H) = \{a, b, c, d, e, f, g, h\}$$

and ψ_H is defined by

$$\psi_H(a) = uv,\quad \psi_H(b) = uu,\quad \psi_H(c) = vw,\quad \psi_H(d) = wx$$
$$\psi_H(e) = vx,\quad \psi_H(f) = wx,\quad \psi_H(g) = ux,\quad \psi_H(h) = xy$$

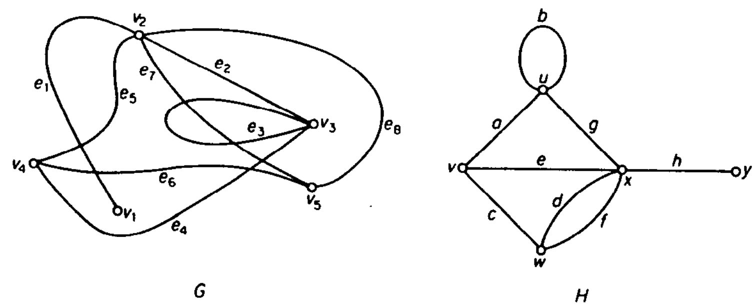


Figure 1.1. Diagrams of graphs G and H

Graphs are so named because they can be represented graphically, and it is this graphical representation which helps us understand many of their properties. Each vertex is indicated by a point, and each edge by a line joining the points which represent its ends.† Diagrams of G and H are shown in figure 1.1. (For clarity, vertices are depicted here as small circles.)

There is no unique way of drawing a graph; the relative positions of points representing vertices and lines representing edges have no significance. Another diagram of G, for example, is given in figure 1.2. A diagram of a graph merely depicts the incidence relation holding between its vertices and edges. We shall, however, often draw a diagram of a graph and refer to it as the graph itself; in the same spirit, we shall call its points 'vertices' and its lines 'edges'.

Note that two edges in a diagram of a graph may intersect at a point that

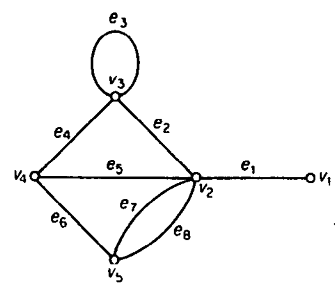


Figure 1.2. Another diagram of G

† In such a drawing it is understood that no line intersects itself or passes through a point representing a vertex which is not an end of the corresponding edge—this is clearly always possible.

is not a vertex (for example e_1 and e_6 of graph G in figure 1.1). Those graphs that have a diagram whose edges intersect only at their ends are called *planar*, since such graphs can be represented in the plane in a simple manner. The graph of figure 1.3*a* is planar, even though this is not immediately clear from the particular representation shown (see exercise 1.1.2). The graph of figure 1.3*b*, on the other hand, is nonplanar. (This will be proved in chapter 9.)

Most of the definitions and concepts in graph theory are suggested by the graphical representation. The ends of an edge are said to be *incident* with the edge, and vice versa. Two vertices which are incident with a common edge are *adjacent*, as are two edges which are incident with a common vertex. An edge with identical ends is called a *loop*, and an edge with distinct ends a *link*. For example, the edge e_3 of G (figure 1.2) is a loop; all other edges of G are links.

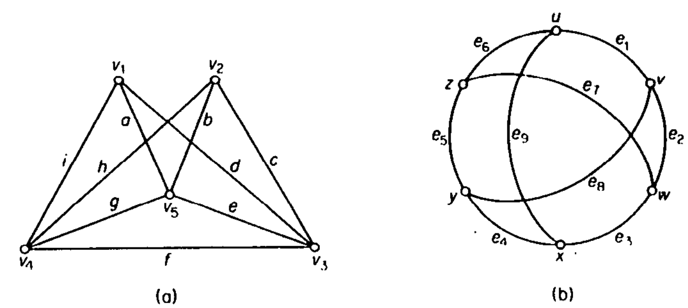


Figure 1.3. Planar and nonplanar graphs

A graph is *finite* if both its vertex set and edge set are finite. In this book we study only finite graphs, and so the term 'graph' always means 'finite graph'. We call a graph with just one vertex *trivial* and all other graphs *nontrivial*.

A graph is *simple* if it has no loops and no two of its links join the same pair of vertices. The graphs of figure 1.1 are not simple, whereas the graphs of figure 1.3 are. Much of graph theory is concerned with the study of simple graphs.

We use the symbols $\nu(G)$ and $\varepsilon(G)$ to denote the numbers of vertices and edges in graph G. Throughout the book the letter G denotes a graph. Moreover, when just one graph is under discussion, we usually denote this graph by G. We then omit the letter G from graph-theoretic symbols and write, for instance, V, E, ν and ε instead of $V(G)$, $E(G)$, $\nu(G)$ and $\varepsilon(G)$.

Exercises

1.1.1 List five situations from everyday life in which graphs arise naturally.

1.1.2 Draw a different diagram of the graph of figure 1.3*a* to show that it is indeed planar.

1.1.3 Show that if G is simple, then $\varepsilon \leq \binom{\nu}{2}$.

1.2 GRAPH ISOMORPHISM

Two graphs G and H are *identical* (written $G = H$) if $V(G) = V(H)$, $E(G) = E(H)$, and $\psi_G = \psi_H$. If two graphs are identical then they can clearly be represented by identical diagrams. However, it is also possible for graphs that are not identical to have essentially the same diagram. For example, the diagrams of G in figure 1.2 and H in figure 1.1 look exactly the same, with the exception that their vertices and edges have different labels. The graphs G and H are not identical, but isomorphic. In general, two graphs G and H are said to be *isomorphic* (written $G \cong H$) if there are bijections $\theta: V(G) \to V(H)$ and $\phi: E(G) \to E(H)$ such that $\psi_G(e) = uv$ if and only if $\psi_H(\phi(e)) = \theta(u)\theta(v)$; such a pair (θ, ϕ) of mappings is called an *isomorphism* between G and H.

To show that two graphs are isomorphic, one must indicate an isomorphism between them. The pair of mappings (θ, ϕ) defined by

$$\theta(v_1) = y, \quad \theta(v_2) = x, \quad \theta(v_3) = u, \quad \theta(v_4) = v, \quad \theta(v_5) = w$$

and

$$\phi(e_1) = h, \quad \phi(e_2) = g, \quad \phi(e_3) = b, \quad \phi(e_4) = a$$
$$\phi(e_5) = e, \quad \phi(e_6) = c, \quad \phi(e_7) = d, \quad \phi(e_8) = f$$

is an isomorphism between the graphs G and H of examples 1 and 2; G and H clearly have the same structure, and differ only in the names of vertices and edges. Since it is in structural properties that we shall primarily be interested, we shall often omit labels when drawing graphs; an unlabelled graph can be thought of as a representative of an equivalence class of isomorphic graphs. We assign labels to vertices and edges in a graph mainly for the purpose of referring to them. For instance, when dealing with simple graphs, it is often convenient to refer to the edge with ends u and v as 'the edge uv'. (This convention results in no ambiguity since, in a simple graph, at most one edge joins any pair of vertices.)

We conclude this section by introducing some special classes of graphs. A simple graph in which each pair of distinct vertices is joined by an edge is called a *complete graph*. Up to isomorphism, there is just one complete graph on n vertices; it is denoted by K_n. A drawing of K_5 is shown in figure 1.4*a*. An *empty graph*, on the other hand, is one with no edges. A *bipartite graph* is one whose vertex set can be partitioned into two subsets X and Y, so that each edge has one end in X and one end in Y; such a partition (X, Y) is called a *bipartition* of the graph. A *complete bipartite graph* is a simple bipartite graph with bipartition (X, Y) in which each vertex of X is joined to each vertex of Y; if $|X| = m$ and $|Y| = n$, such a graph is denoted by $K_{m,n}$. The graph defined by the vertices and edges of a cube (figure 1.4*b*) is bipartite; the graph in figure 1.4*c* is the complete bipartite graph $K_{3,3}$.

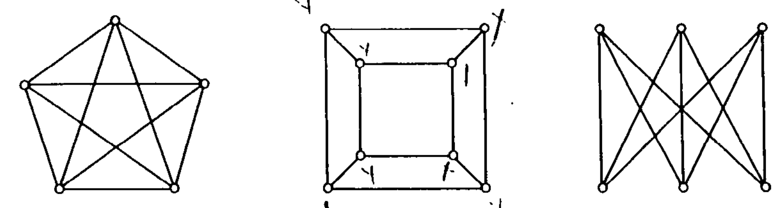


Figure 1.4. (*a*) K_5; (*b*) the cube; (*c*) $K_{3,3}$

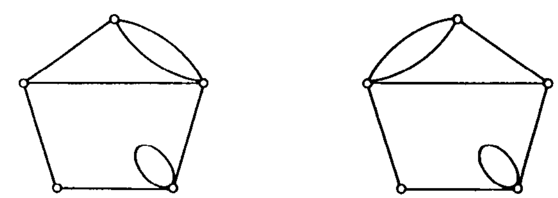
There are many other graphs whose structures are of special interest. Appendix III includes a selection of such graphs.

Exercises

1.2.1 Find an isomorphism between the graphs G and H of examples 1 and 2 different from the one given.

1.2.2 (*a*) Show that if $G \cong H$, then $\nu(G) = \nu(H)$ and $\varepsilon(G) = \varepsilon(H)$.
(*b*) Give an example to show that the converse is false.

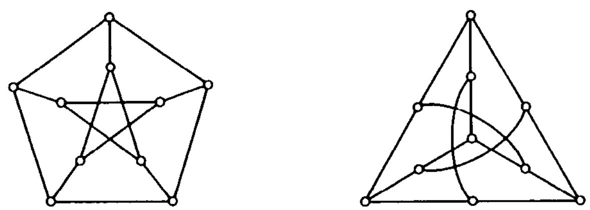
1.2.3 Show that the following graphs are not isomorphic:



1.2.4 Show that there are eleven nonisomorphic simple graphs on four vertices.

1.2.5 Show that two simple graphs G and H are isomorphic if and only if there is a bijection $\theta: V(G) \to V(H)$ such that $uv \in E(G)$ if and only if $\theta(u)\theta(v) \in E(H)$.

1.2.6 Show that the following graphs are isomorphic:



1.2.7 Let G be simple. Show that $\varepsilon = \binom{\nu}{2}$ if and only if G is complete.

1.2.8 Show that

(*a*) $\varepsilon(K_{m,n}) = mn$;
(*b*) if G is simple and bipartite, then $\varepsilon \leq \nu^2/4$.

1.2.9 A *k-partite graph* is one whose vertex set can be partitioned into k subsets so that no edge has both ends in any one subset; a *complete k-partite graph* is one that is simple and in which each vertex is joined to every vertex that is not in the same subset. The complete m-partite graph on n vertices in which each part has either $[n/m]$ or $\{n/m\}$ vertices is denoted by $T_{m,n}$. Show that

(*a*) $\varepsilon(T_{m,n}) = \binom{n-k}{2} + (m-1)\binom{k+1}{2}$, where $k = [n/m]$;

(*b*)* if G is a complete m-partite graph on n vertices, then $\varepsilon(G) \leq \varepsilon(T_{m,n})$, with equality only if $G \cong T_{m,n}$.

1.2.10 The *k-cube* is the graph whose vertices are the ordered k-tuples of 0's and 1's, two vertices being joined if and only if they differ in exactly one coordinate. (The graph shown in figure 1.4*b* is just the 3-cube.) Show that the k-cube has 2^k vertices, $k2^{k-1}$ edges and is bipartite.

1.2.11 (*a*) The *complement* G^c of a simple graph G is the simple graph with vertex set V, two vertices being adjacent in G^c if and only if they are not adjacent in G. Describe the graphs K_n^c and $K_{m,n}^c$.
(*b*) A simple graph G is *self-complementary* if $G \cong G^c$. Show that if G is self-complementary, then $\nu \equiv 0, 1 \pmod 4$.

1.2.12 An *automorphism* of a graph is an isomorphism of the graph onto itself.

(*a*) Show, using exercise 1.2.5, that an automorphism of a simple graph G can be regarded as a permutation on V which preserves adjacency, and that the set of such permutations form a

group $\Gamma(G)$ (the *automorphism group* of G) under the usual operation of composition.
(*b*) Find $\Gamma(K_n)$ and $\Gamma(K_{m,n})$.
(*c*) Find a nontrivial simple graph whose automorphism group is the identity.
(*d*) Show that for any simple graph G, $\Gamma(G) = \Gamma(G^c)$.
(*e*) Consider the permutation group A with elements (1)(2)(3), (1, 2, 3) and (1, 3, 2). Show that there is no simple graph G with vertex set {1, 2, 3} such that $\Gamma(G) = A$.
(*f*) Find a simple graph G such that $\Gamma(G) \cong A$. (Frucht, 1939 has shown that every abstract group is isomorphic to the automorphism group of some graph.)

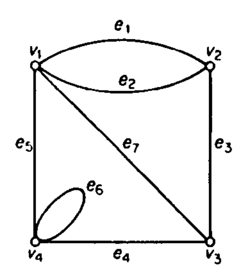
1.2.13 A simple graph G is *vertex-transitive* if, for any two vertices u and v, there is an element g in $\Gamma(G)$ such that $g(u) = g(v)$; G is *edge-transitive* if, for any two edges u_1v_1 and u_2v_2, there is an element h in $\Gamma(G)$ such that $h(\{u_1, v_1\}) = \{u_2, v_2\}$. Find

(*a*) a graph which is vertex-transitive but not edge-transitive;
(*b*) a graph which is edge-transitive but not vertex-transitive.

1.3 THE INCIDENCE AND ADJACENCY MATRICES

To any graph G there corresponds a $\nu \times \varepsilon$ matrix called the incidence matrix of G. Let us denote the vertices of G by $v_1, v_2, \ldots, v_\nu$ and the edges by $e_1, e_2, \ldots, e_\varepsilon$. Then the *incidence matrix* of G is the matrix $\mathbf{M}(G) = [m_{ij}]$, where m_{ij} is the number of times (0, 1 or 2) that v_i and e_j are incident. The incidence matrix of a graph is just a different way of specifying the graph.

Another matrix associated with G is the *adjacency matrix*; this is the $\nu \times \nu$ matrix $\mathbf{A}(G) = [a_{ij}]$, in which a_{ij} is the number of edges joining v_i and v_j. A graph, its incidence matrix, and its adjacency matrix are shown in figure 1.5.



G

	e_1	e_2	e_3	e_4	e_5	e_6	e_7
v_1	1	1	0	0	1	0	1
v_2	1	1	1	0	0	0	0
v_3	0	0	1	1	0	0	1
v_4	0	0	0	1	1	2	0

$\mathbf{M}(G)$

	v_1	v_2	v_3	v_4
v_1	0	2	1	1
v_2	2	0	1	0
v_3	1	1	0	1
v_4	1	0	1	1

$\mathbf{A}(G)$

Figure 1.5

The adjacency matrix of a graph is generally considerably smaller than its incidence matrix, and it is in this form that graphs are commonly stored in computers.

Exercises

1.3.1 Let $\mathbf{M}$ be the incidence matrix and $\mathbf{A}$ the adjacency matrix of a graph G.

(*a*) Show that every column sum of $\mathbf{M}$ is 2.
(*b*) What are the column sums of $\mathbf{A}$?

1.3.2 Let G be bipartite. Show that the vertices of G can be enumerated so that the adjacency matrix of G has the form

$$\begin{bmatrix} \mathbf{0} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{0} \end{bmatrix}$$

where $\mathbf{A}_{21}$ is the transpose of $\mathbf{A}_{12}$.

1.3.3* Show that if G is simple and the eigenvalues of $\mathbf{A}$ are distinct, then the automorphism group of G is abelian

1.4 SUBGRAPHS

A graph H is a *subgraph* of G (written $H \subseteq G$) if $V(H) \subseteq V(G)$, $E(H) \subseteq E(G)$, and ψ_H is the restriction of ψ_G to $E(H)$. When $H \subseteq G$ but $H \neq G$, we write $H \subset G$ and call H a *proper subgraph* of G. If H is a subgraph of G, G is a *supergraph* of H. A *spanning subgraph* (or *spanning supergraph*) of G is a subgraph (or supergraph) H with $V(H) = V(G)$.

By deleting from G all loops and, for every pair of adjacent vertices, all but one link joining them, we obtain a simple spanning subgraph of G, called the *underlying simple graph* of G. Figure 1.6 shows a graph and its underlying simple graph.

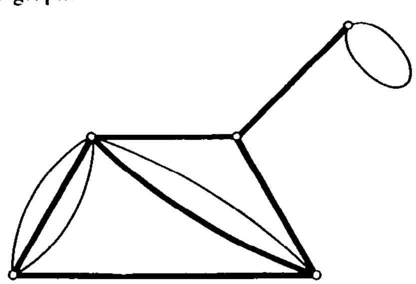


Figure 1.6. A graph and its underlying simple graph

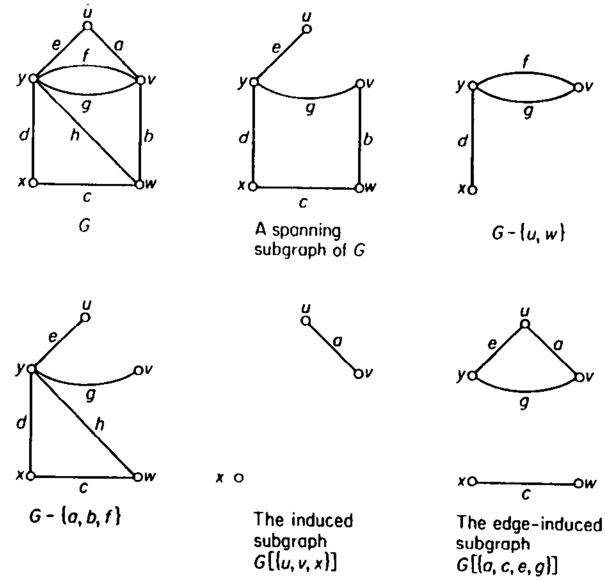


Figure 1.7

Suppose that V' is a nonempty subset of V. The subgraph of G whose vertex set is V' and whose edge set is the set of those edges of G that have both ends in V' is called the subgraph of G *induced by* V' and is denoted by $G[V']$; we say that $G[V']$ is an *induced subgraph* of G. The induced subgraph $G[V\backslash V']$ is denoted by $G - V'$; it is the subgraph obtained from G by deleting the vertices in V' together with their incident edges. If $V' = \{v\}$ we write $G - v$ for $G - \{v\}$.

Now suppose that E' is a nonempty subset of E. The subgraph of G whose vertex set is the set of ends of edges in E' and whose edge set is E' is called the subgraph of G *induced* by E' and is denoted by $G[E']$; $G[E']$ is an *edge-induced subgraph* of G. The spanning subgraph of G with edge set $E\backslash E'$ is written simply as $G - E'$; it is the subgraph obtained from G by deleting the edges in E'. Similarly, the graph obtained from G by adding a set of edges E' is denoted by $G + E'$. If $E' = \{e\}$ we write $G - e$ and $G + e$ instead of $G - \{e\}$ and $G + \{e\}$.

Subgraphs of these various types are depicted in figure 1.7.

Let G_1 and G_2 be subgraphs of G. We say that G_1 and G_2 are *disjoint* if they have no vertex in common, and *edge-disjoint* if they have no edge in common. The *union* $G_1 \cup G_2$ of G_1 and G_2 is the subgraph with vertex set

$V(G_1) \cup V(G_2)$ and edge set $E(G_1) \cup E(G_2)$; if G_1 and G_2 are disjoint, we sometimes denote their union by $G_1 + G_2$. The *intersection* $G_1 \cap G_2$ of G_1 and G_2 is defined similarly, but in this case G_1 and G_2 must have at least one vertex in common.

Exercises

1.4.1 Show that every simple graph on n vertices is isomorphic to a subgraph of K_n.

1.4.2 Show that
(*a*) every induced subgraph of a complete graph is complete;
(*b*) every subgraph of a bipartite graph is bipartite.

1.4.3 Describe how $\mathbf{M}(G - E')$ and $\mathbf{M}(G - V')$ can be obtained from $\mathbf{M}(G)$, and how $\mathbf{A}(G - V')$ can be obtained from $\mathbf{A}(G)$.

1.4.4 Find a bipartite graph that is not isomorphic to a subgraph of any k-cube.

1.4.5* Let G be simple and let n be an integer with $1 < n < \nu - 1$. Show that if $\nu \geq 4$ and all induced subgraphs of G on n vertices have the same number of edges, then either $G \cong K_\nu$ or $G \cong K_\nu^c$.

1.5 VERTEX DEGREES

The *degree* $d_G(v)$ of a vertex v in G is the number of edges of G incident with v, each loop counting as two edges. We denote by $\delta(G)$ and $\Delta(G)$ the minimum and maximum degrees, respectively, of vertices of G.

Theorem 1.1

$$\sum_{v \in V} d(v) = 2\varepsilon$$

Proof Consider the incidence matrix $\mathbf{M}$. The sum of the entries in the row corresponding to vertex v is precisely $d(v)$, and therefore $\sum_{v \in V} d(v)$ is just the sum of all entries in $\mathbf{M}$. But this sum is also 2ε, since (exercise 1.3.1*a*) each of the ε column sums of $\mathbf{M}$ is 2 □

Corollary 1.1 In any graph, the number of vertices of odd degree is even.

Proof Let V_1 and V_2 be the sets of vertices of odd and even degree in G, respectively. Then

$$\sum_{v \in V_1} d(v) + \sum_{v \in V_2} d(v) = \sum_{v \in V} d(v)$$

is even, by theorem 1.1. Since $\sum_{v \in V_2} d(v)$ is also even, it follows that $\sum_{v \in V_1} d(v)$ is even. Thus $|V_1|$ is even □

A graph G is *k-regular* if $d(v) = k$ for all $v \in V$; a *regular graph* is one that is k-regular for some k. Complete graphs and complete bipartite graphs $K_{n,n}$ are regular; so, also, are the k-cubes.

Exercises

1.5.1 Show that $\delta \leq 2\varepsilon/\nu \leq \Delta$.

1.5.2 Show that if G is simple, the entries on the diagonals of both $\mathbf{MM}'$ and $\mathbf{A}^2$ are the degrees of the vertices of G.

1.5.3 Show that if a k-regular bipartite graph with $k > 0$ has bipartition (X, Y), then $|X| = |Y|$.

1.5.4 Show that, in any group of two or more people, there are always two with exactly the same number of friends inside the group.

1.5.5 If G has vertices $v_1, v_2, \ldots, v_\nu$, the sequence $(d(v_1), d(v_2), \ldots, d(v_\nu))$ is called a *degree sequence* of G. Show that a sequence $(d_1, d_2, \ldots, d_n)$ of non-negative integers is a degree sequence of some graph if and only if $\sum_{i=1}^{n} d_i$ is even.

1.5.6 A sequence $\mathbf{d} = (d_1, d_2, \ldots, d_n)$ is *graphic* if there is a simple graph with degree sequence $\mathbf{d}$. Show that
(*a*) the sequences (7, 6, 5, 4, 3, 3, 2) and (6, 6, 5, 4, 3, 3, 1) are not graphic;
(*b*) if $\mathbf{d}$ is graphic and $d_1 \geq d_2 \geq \ldots \geq d_n$, then $\sum_{i=1}^{n} d_i$ is even and

$$\sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{i=k+1}^{n} \min\{k, d_i\} \quad \text{for} \quad 1 \leq k \leq n$$

(Erdös and Gallai, 1960 have shown that this necessary condition is also sufficient for $\mathbf{d}$ to be graphic.)

1.5.7 Let $\mathbf{d} = (d_1, d_2, \ldots, d_n)$ be a nonincreasing sequence of non-negative integers, and denote the sequence $(d_2 - 1, d_3 - 1, \ldots, d_{d_1+1} - 1, d_{d_1+2}, \ldots, d_n)$ by $\mathbf{d}'$.
(*a*)* Show that $\mathbf{d}$ is graphic if and only if $\mathbf{d}'$ is graphic.
(*b*) Using (*a*), describe an algorithm for constructing a simple graph with degree sequence $\mathbf{d}$, if such a graph exists.
(V. Havel, S. Hakimi)

1.5.8* Show that a loopless graph G contains a bipartite spanning subgraph H such that $d_H(v) \geq \frac{1}{2} d_G(v)$ for all $v \in V$.

1.5.9* Let $S = \{x_1, x_2, \ldots, x_n\}$ be a set of points in the plane such that the distance between any two points is at least one. Show that there are at most $3n$ pairs of points at distance exactly one.

1.5.10 The *edge graph* of a graph G is the graph with vertex set $E(G)$ in which two vertices are joined if and only if they are adjacent edges in

G. Show that, if *G* is simple

(*a*) the edge graph of *G* has $\varepsilon(G)$ vertices and $\sum_{v \in V(G)} \binom{d_G(v)}{2}$ edges;

(*b*) the edge graph of K_5 is isomorphic to the complement of the graph featured in exercise 1.2.6.

1.6 PATHS AND CONNECTION

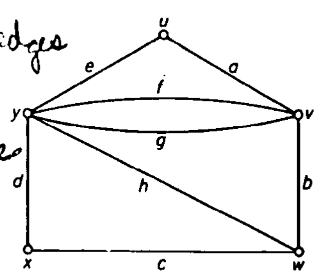
A *walk* in *G* is a finite non-null sequence $W = v_0 e_1 v_1 e_2 v_2 \ldots e_k v_k$, whose terms are alternately vertices and edges, such that, for $1 \le i \le k$, the ends of e_i are v_{i-1} and v_i. We say that *W* is a walk *from* v_0 *to* v_k, or a (v_0, v_k)-*walk*. The vertices v_0 and v_k are called the *origin* and *terminus* of *W*, respectively, and $v_1, v_2, \ldots, v_{k-1}$ its *internal vertices*. The integer *k* is the *length* of *W*.

If $W = v_0 e_1 v_1 \ldots e_k v_k$ and $W' = v_k e_{k+1} v_{k+1} \ldots e_l v_l$ are walks, the walk $v_k e_k v_{k-1} \ldots e_1 v_0$, obtained by reversing *W*, is denoted by W^{-1} and the walk $v_0 e_1 v_1 \ldots e_l v_l$, obtained by concatenating *W* and *W'* at v_k, is denoted by *WW'*. A *section* of a walk $W = v_0 e_1 v_1 \ldots e_k v_k$ is a walk that is a subsequence $v_i e_{i+1} v_{i+1} \ldots e_j v_j$ of consecutive terms of *W*; we refer to this subsequence as the (v_i, v_j)-*section* of *W*.

In a simple graph, a walk $v_0 e_1 v_1 \ldots e_k v_k$ is determined by the sequence $v_0 v_1 \ldots v_k$ of its vertices; hence a walk in a simple graph can be specified simply by its vertex sequence. Moreover, even in graphs that are not simple, we shall sometimes refer to a sequence of vertices in which consecutive terms are adjacent as a 'walk'. In such cases it should be understood that the discussion is valid for every walk with that vertex sequence.

If the edges $e_1, e_2, \ldots, e_k$ of a walk *W* are distinct, *W* is called a *trail*; in this case the length of *W* is just $\varepsilon(W)$. If, in addition, the vertices $v_0, v_1, \ldots, v_k$ are distinct, *W* is called a *path*. Figure 1.8 illustrates a walk, a trail and a path in a graph. We shall also use the word 'path' to denote a graph or subgraph whose vertices and edges are the terms of a path.

trail distinct edges
path distinct vertices, edges



Walk: *uavfyfvgyhwbv*
Trail: *wcxdyhwbvgy*
Path: *xcwhyeuav*

Figure 1.8

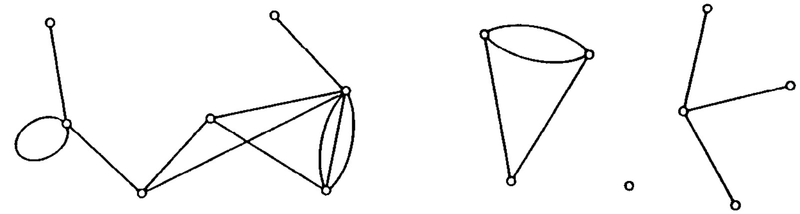


Figure 1.9. (*a*) A connected graph; (*b*) a disconnected graph with three components

Two vertices *u* and *v* of *G* are said to be *connected* if there is a (u, v)-path in *G*. Connection is an equivalence relation on the vertex set *V*. Thus there is a partition of *V* into nonempty subsets $V_1, V_2, \ldots, V_\omega$ such that two vertices *u* and *v* are connected if and only if both *u* and *v* belong to the same set V_i. The subgraphs $G[V_1], G[V_2], \ldots, G[V_\omega]$ are called the *components* of *G*. If *G* has exactly one component, *G* is *connected*; otherwise *G* is *disconnected*. We denote the number of components of *G* by $\omega(G)$. Connected and disconnected graphs are depicted in figure 1.9.

Exercises

1.6.1 Show that if there is a (u, v)-walk in *G*, then there is also a (u, v)-path in *G*.

1.6.2 Show that the number of (v_i, v_j)-walks of length *k* in *G* is the (i, j)th entry of $\mathbf{A}^k$.

1.6.3 Show that if *G* is simple and $\delta \ge k$, then *G* has a path of length *k*.

1.6.4 Show that *G* is connected if and only if, for every partition of *V* into two nonempty sets V_1 and V_2, there is an edge with one end in V_1 and one end in V_2.

1.6.5 (*a*) Show that if *G* is simple and $\varepsilon > \binom{\nu-1}{2}$, then *G* is connected.

(*b*) For $\nu > 1$, find a disconnected simple graph *G* with $\varepsilon = \binom{\nu-1}{2}$.

1.6.6 (*a*) Show that if *G* is simple and $\delta > [\nu/2] - 1$, then *G* is connected.
(*b*) Find a disconnected $([\nu/2] - 1)$-regular simple graph for ν even.

1.6.7 Show that if *G* is disconnected, then G^c is connected.

1.6.8 (*a*) Show that if $e \in E$, then $\omega(G) \le \omega(G - e) \le \omega(G) + 1$.
(*b*) Let $v \in V$. Show that $G - e$ cannot, in general, be replaced by $G - v$ in the above inequality.

1.6.9 Show that if *G* is connected and each degree in *G* is even, then, for any $v \in V$, $\omega(G - v) \le \frac{1}{2} d(v)$.

1.6.10 Show that any two longest paths in a connected graph have a vertex in common.

1.6.11 If vertices u and v are connected in G, the *distance* between u and v in G, denoted by $d_G(u, v)$, is the length of a shortest (u, v)-path in G; if there is no path connecting u and v we define $d_G(u, v)$ to be infinite. Show that, for any three vertices u, v and w, $d(u, v) + d(v, w) \geq d(u, w)$.

1.6.12 The *diameter* of G is the maximum distance between two vertices of G. Show that if G has diameter greater than three, then G^c has diameter less than three.

1.6.13 Show that if G is simple with diameter two and $\Delta = \nu - 2$, then $\varepsilon \geq 2\nu - 4$.

1.6.14 Show that if G is simple and connected but not complete, then G has three vertices u, v and w such that $uv, vw \in E$ and $uw \notin E$.

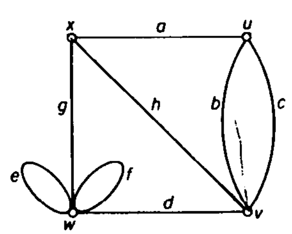
1.7 CYCLES

A walk is *closed* if it has positive length and its origin and terminus are the same. A closed trail whose origin and internal vertices are distinct is a *cycle*. Just as with paths we sometimes use the term 'cycle' to denote a graph corresponding to a cycle. A cycle of length k is called a *k-cycle*; a k-cycle is *odd* or *even* according as k is odd or even. A 3-cycle is often called a *triangle*. Examples of a closed trail and a cycle are given in figure 1.10.

Using the concept of a cycle, we can now present a characterisation of bipartite graphs.

Theorem 1.2 A graph is bipartite if and only if it contains no odd cycle.

Proof Suppose that G is bipartite with bipartition (X, Y), and let $C = v_0v_1 \ldots v_kv_0$ be a cycle of G. Without loss of generality we may assume that $v_0 \in X$. Then, since $v_0v_1 \in E$ and G is bipartite, $v_1 \in Y$. Similarly $v_2 \in X$ and, in general, $v_{2i} \in X$ and $v_{2i+1} \in Y$. Since $v_0 \in X$, $v_k \in Y$. Thus $k = 2i + 1$, for some i, and it follows that C is even.



Closed trail: *ucvhxgwfwdvbu*
Cycle: *xaubvhx*

Figure 1.10

It clearly suffices to prove the converse for connected graphs. Let G be a connected graph that contains no odd cycles. We choose an arbitrary vertex u and define a partition (X, Y) of V by setting

$$X = \{x \in V \mid d(u, x) \text{ is even}\}$$
$$Y = \{y \in V \mid d(u, y) \text{ is odd}\}$$

We shall show that (X, Y) is a bipartition of G. Suppose that v and w are two vertices of X. Let P be a shortest (u, v)-path and Q be a shortest (u, w)-path. Denote by u_1 the last vertex common to P and Q. Since P and Q are shortest paths, the (u, u_1)-sections of both P and Q are shortest (u, u_1)-paths and, therefore, have the same length. Now, since the lengths of both P and Q are even, the lengths of the (u_1, v)-section P_1 of P and the (u_1, w)-section Q_1 of Q must have the same parity. It follows that the (v, w)-path $P_1^{-1}Q_1$ is of even length. If v were joined to w, $P_1^{-1}Q_1wv$ would be a cycle of odd length, contrary to the hypothesis. Therefore no two vertices in X are adjacent; similarly, no two vertices in Y are adjacent □

Exercises

1.7.1 Show that if an edge e is in a closed trail of G, then e is in a cycle of G.

1.7.2 Show that if $\delta \geq 2$, then G contains a cycle.

1.7.3* Show that if G is simple and $\delta \geq 2$, then G contains a cycle of length at least $\delta + 1$.

1.7.4 The *girth* of G is the length of a shortest cycle in G; if G has no cycles we define the girth of G to be infinite. Show that

(*a*) a k-regular graph of girth four has at least $2k$ vertices, and (up to isomorphism) there exists exactly one such graph on $2k$ vertices;

(*b*) a k-regular graph of girth five has at least $k^2 + 1$ vertices.

1.7.5 Show that a k-regular graph of girth five and diameter two has exactly $k^2 + 1$ vertices, and find such a graph for $k = 2, 3$. (Hoffman and Singleton, 1960 have shown that such a graph can exist only if $k = 2, 3, 7$ and, possibly, 57.)

1.7.6 Show that

(*a*) if $\varepsilon \geq \nu$, G contains a cycle;

(*b*)* if $\varepsilon \geq \nu + 4$, G contains two edge-disjoint cycles. (L. Pósa)

APPLICATIONS

1.8 THE SHORTEST PATH PROBLEM

With each edge e of G let there be associated a real number $w(e)$, called its *weight*. Then G, together with these weights on its edges, is called a *weighted*

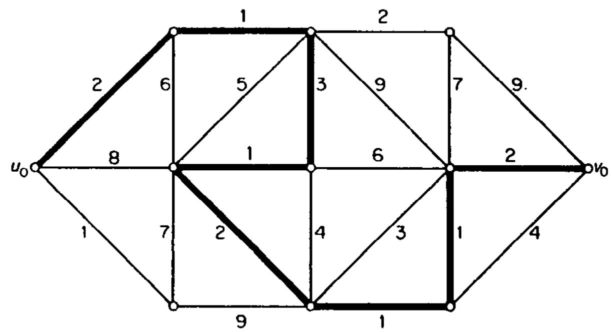


Figure 1.11. A (u_0, v_0)-path of minimum weight

graph. Weighted graphs occur frequently in applications of graph theory. In the friendship graph, for example, weights might indicate intensity of friendship; in the communications graph, they could represent the construction or maintenance costs of the various communication links.

If H is a subgraph of a weighted graph, the *weight* $w(H)$ of H is the sum of the weights $\sum_{e \in E(H)} w(e)$ on its edges. Many optimisation problems amount to finding, in a weighted graph, a subgraph of a certain type with minimum (or maximum) weight. One such is the *shortest path problem:* given a railway network connecting various towns, determine a shortest route between two specified towns in the network.

Here one must find, in a weighted graph, a path of minimum weight connecting two specified vertices u_0 and v_0; the weights represent distances by rail between directly-linked towns, and are therefore non-negative. The path indicated in the graph of figure 1.11 is a (u_0, v_0)-path of minimum weight (exercise 1.8.1).

We now present an algorithm for solving the shortest path problem. For clarity of exposition, we shall refer to the weight of a path in a weighted graph as its *length;* similarly the minimum weight of a (u, v)-path will be called the *distance* between u and v and denoted by $d(u, v)$. These definitions coincide with the usual notions of length and distance, as defined in section 1.6, when all the weights are equal to one.

It clearly suffices to deal with the shortest path problem for simple graphs; so we shall assume here that G is simple. We shall also assume that all the weights are positive. This, again, is not a serious restriction because, if the weight of an edge is zero, then its ends can be identified. We adopt the convention that $w(uv) = \infty$ if $uv \notin E$.

The algorithm to be described was discovered by Dijkstra (1959) and, independently, by Whiting and Hillier (1960). It finds not only a shortest (u_0, v_0)-path, but shortest paths from u_0 to all other vertices of G. The basic idea is as follows.

Suppose that S is a proper subset of V such that $u_0 \in S$, and let $\bar{S}$ denote $V \backslash S$. If $P = u_0 \ldots \bar{u}\bar{v}$ is a shortest path from u_0 to $\bar{S}$ then clearly $\bar{u} \in S$ and the $(u_0, \bar{u})$-section of P must be a shortest $(u_0, \bar{u})$-path. Therefore

$$d(u_0, \bar{v}) = d(u_0, \bar{u}) + w(\bar{u}\bar{v})$$

and the distance from u_0 to $\bar{S}$ is given by the formula

$$d(u_0, \bar{S}) = \min_{\substack{u \in S \\ v \in \bar{S}}}\{d(u_0, u) + w(uv)\} \quad (1.1)$$

This formula is the basis of Dijkstra's algorithm. Starting with the set $S_0 = \{u_0\}$, an increasing sequence $S_0, S_1, \ldots, S_{\nu-1}$ of subsets of V is constructed, in such a way that, at the end of stage i, shortest paths from u_0 to all vertices in S_i are known.

The first step is to determine a vertex nearest to u_0. This is achieved by computing $d(u_0, \bar{S}_0)$ and selecting a vertex $u_1 \in \bar{S}_0$ such that $d(u_0, u_1) = d(u_0, \bar{S}_0)$; by (1.1)

$$d(u_0, \bar{S}_0) = \min_{\substack{u \in S_0 \\ v \in \bar{S}_0}}\{d(u_0, u) + w(uv)\} = \min_{v \in \bar{S}_0}\{w(u_0 v)\}$$

and so $d(u_0, \bar{S}_0)$ is easily computed. We now set $S_1 = \{u_0, u_1\}$ and let P_1 denote the path u_0u_1; this is clearly a shortest (u_0, u_1)-path. In general, if the set $S_k = \{u_0, u_1, \ldots, u_k\}$ and corresponding shortest paths $P_1, P_2, \ldots, P_k$ have already been determined, we compute $d(u_0, \bar{S}_k)$ using (1.1) and select a vertex $u_{k+1} \in \bar{S}_k$ such that $d(u_0, u_{k+1}) = d(u_0, \bar{S}_k)$. By (1.1), $d(u_0, u_{k+1}) = d(u_0, u_j) + w(u_j u_{k+1})$ for some $j \leq k$; we get a shortest (u_0, u_{k+1})-path by adjoining the edge $u_j u_{k+1}$ to the path P_j.

We illustrate this procedure by considering the weighted graph depicted in figure 1.12*a*. Shortest paths from u_0 to the remaining vertices are determined in seven stages. At each stage, the vertices to which shortest paths have been found are indicated by solid dots, and each is labelled by its distance from u_0; initially u_0 is labelled 0. The actual shortest paths are indicated by solid lines. Notice that, at each stage, these shortest paths together form a connected graph without cycles; such a graph is called a *tree*, and we can think of the algorithm as a 'tree-growing' procedure. The final tree, in figure 1.12*h*, has the property that, for each vertex v, the path connecting u_0 and v is a shortest (u_0, v)-path.

Dijkstra's algorithm is a refinement of the above procedure. This refinement is motivated by the consideration that, if the minimum in (1.1) were to be computed from scratch at each stage, many comparisons would be

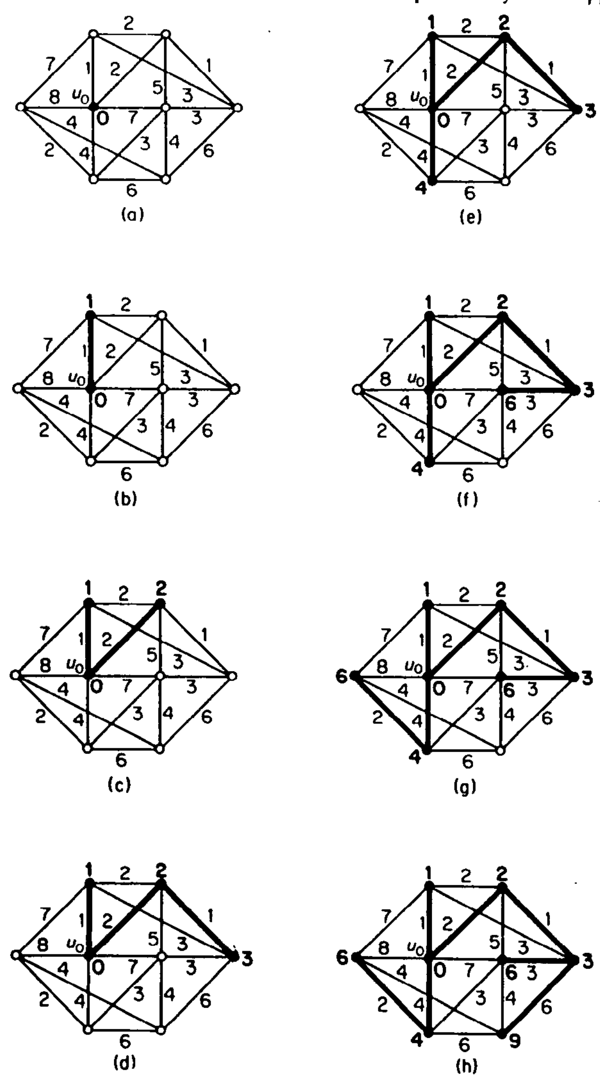


Figure 1.12. Shortest path algorithm

repeated unnecessarily. To avoid such repetitions, and to retain computational information from one stage to the next, we adopt the following labelling procedure. Throughout the algorithm, each vertex v carries a label $l(v)$ which is an upper bound on $d(u_0, v)$. Initially $l(u_0) = 0$ and $l(v) = \infty$ for $v \neq u_0$. (In actual computations ∞ is replaced by any sufficiently large number.) As the algorithm proceeds, these labels are modified so that, at the end of stage i,

$$l(u) = d(u_0, u) \quad \text{for} \quad u \in S_i$$

and

$$l(v) = \min_{u \in S_{i-1}}\{d(u_0, u) + w(uv)\} \quad \text{for} \quad v \in \bar{S}_i$$

Dijkstra's Algorithm

1. Set $l(u_0) = 0$, $l(v) = \infty$ for $v \neq u_0$, $S_0 = \{u_0\}$ and $i = 0$.
2. For each $v \in \bar{S}_i$, replace $l(v)$ by $\min\{l(v), l(u_i) + w(u_i v)\}$. Compute $\min_{v \in \bar{S}_i}\{l(v)\}$ and let u_{i+1} denote a vertex for which this minimum is attained. Set $S_{i+1} = S_i \cup \{u_{i+1}\}$.
3. If $i = \nu - 1$, stop. If $i < \nu - 1$, replace i by $i + 1$ and go to step 2.

When the algorithm terminates, the distance from u_0 to v is given by the final value of the label $l(v)$. (If our interest is in determining the distance to one specific vertex v_0, we stop as soon as some u_j equals v_0.) A flow diagram summarising this algorithm is shown in figure 1.13.

As described above, Dijkstra's algorithm determines only the distances from u_0 to all the other vertices, and not the actual shortest paths. These shortest paths can, however, be easily determined by keeping track of the predecessors of vertices in the tree (exercise 1.8.2).

Dijkstra's algorithm is an example of what Edmonds (1965) calls a good algorithm. A graph-theoretic algorithm is *good* if the number of computational steps required for its implementation on any graph G is bounded above by a polynomial in ν and ε (such as $3\nu^2\varepsilon$). An algorithm whose implementation may require an exponential number of steps (such as 2^ν) might be very inefficient for some large graphs.

To see that Dijkstra's algorithm is good, note that the computations involved in boxes 2 and 3 of the flow diagram, totalled over all iterations, require $\nu(\nu-1)/2$ additions and $\nu(\nu-1)$ comparisons. One of the questions that is not elaborated upon in the flow diagram is the matter of deciding whether a vertex belongs to $\bar{S}$ or not (box 1). Dreyfus (1969) reports a technique for doing this that requires a total of $(\nu-1)^2$ comparisons. Hence, if we regard either a comparison or an addition as a basic computational unit, the total number of computations required for this algorithm is approximately $5\nu^2/2$, and thus of order ν^2. (A function $f(\nu, \varepsilon)$ is of *order*

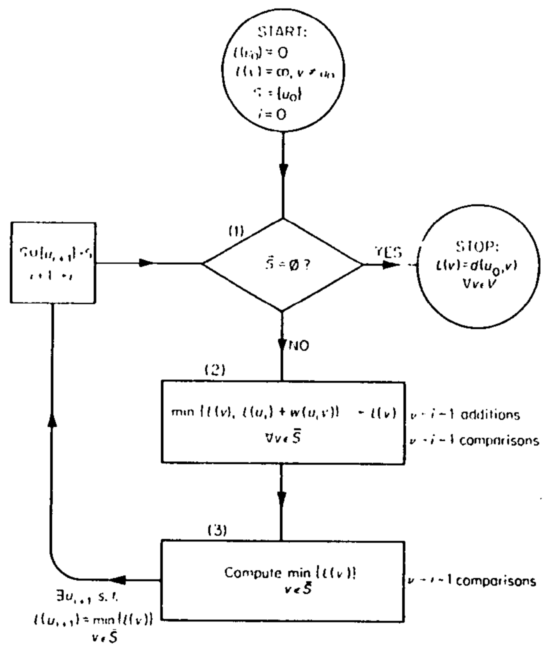


Figure 1.13. Dijkstra's algorithm

$g(\nu, \varepsilon)$ if there exists a positive constant c such that $f(\nu, \varepsilon)/g(\nu, \varepsilon) \leq c$ for all ν and ε.)

Although the shortest path problem can be solved by a good algorithm, there are many problems in graph theory for which no good algorithm is known. We refer the reader to Aho, Hopcroft and Ullman (1974) for further details.

Exercises

1.8.1 Find shortest paths from u_0 to all other vertices in the weighted graph of figure 1.11.

1.8.2 What additional instructions are needed in order that Dijkstra's algorithm determine shortest paths rather than merely distances?

1.8.3 A company has branches in each of six cities $C_1, C_2, \ldots, C_6$. The fare for a direct flight from C_i to C_j is given by the (i, j)th entry in the following matrix (∞ indicates that there is no direct flight):

$$\begin{bmatrix} 0 & 50 & \infty & 40 & 25 & 10 \\ 50 & 0 & 15 & 20 & \infty & 25 \\ \infty & 15 & 0 & 10 & 20 & \infty \\ 40 & 20 & 10 & 0 & 10 & 25 \\ 25 & \infty & 20 & 10 & 0 & 55 \\ 10 & 25 & \infty & 25 & 55 & 0 \end{bmatrix}$$

The company is interested in computing a table of cheapest routes between pairs of cities. Prepare such a table.

1.8.4 A wolf, a goat and a cabbage are on one bank of a river. A ferryman wants to take them across, but, since his boat is small, he can take only one of them at a time. For obvious reasons, neither the wolf and the goat nor the goat and the cabbage can be left unguarded. How is the ferryman going to get them across the river?

1.8.5 Two men have a full eight-gallon jug of wine, and also two empty jugs of five and three gallons capacity, respectively. What is the simplest way for them to divide the wine equally?

1.8.6 Describe a good algorithm for determining
(*a*) the components of a graph;
(*b*) the girth of a graph.
How good are your algorithms?

1.9 SPERNER'S LEMMA

Every continuous mapping f of a closed n-disc to itself has a fixed point (that is, a point x such that $f(x) = x$). This powerful theorem, known as *Brouwer's fixed-point theorem*, has a wide range of applications in modern mathematics. Somewhat surprisingly, it is an easy consequence of a simple combinatorial lemma due to Sperner (1928). And, as we shall see in this section, Sperner's lemma is, in turn, an immediate consequence of corollary 1.1.

Sperner's lemma concerns the decomposition of a simplex (line segment, triangle, tetrahedron and so on) into smaller simplices. For the sake of simplicity we shall deal with the two-dimensional case.

Let T be a closed triangle in the plane. A subdivision of T into a finite number of smaller triangles is said to be *simplicial* if any two intersecting triangles have either a vertex or a whole side in common (see figure 1.14*a*).

Suppose that a simplicial subdivision of T is given. Then a labelling of the vertices of triangles in the subdivision in three symbols 0, 1 and 2 is said to be *proper* if

(i) the three vertices of T are labelled 0, 1 and 2 (in any order), and
(ii) for $0 \leq i < j \leq 2$, each vertex on the side of T joining vertices labelled i and j is labelled either i or j.

0213

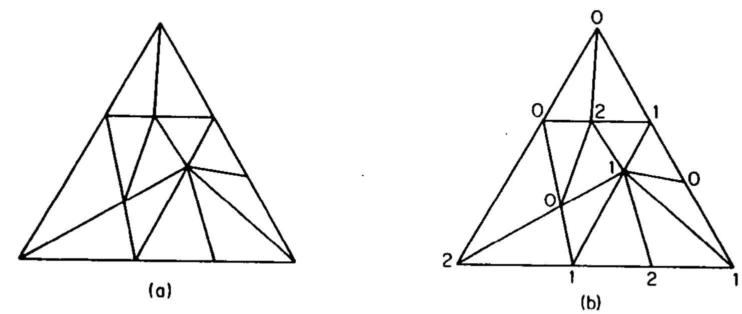


Figure 1.14. (*a*) A simplicial subdivision of a triangle; (*b*) a proper labelling of the subdivision

We call a triangle in the subdivision whose vertices receive all three labels a *distinguished* triangle. The proper labelling in figure 1.14*b* has three distinguished triangles.

Theorem 1.3 (Sperner's lemma) Every properly labelled simplicial subdivision of a triangle has an odd number of distinguished triangles.

Proof Let T_0 denote the region outside T, and let $T_1, T_2, \ldots, T_n$ be the triangles of the subdivision. Construct a graph on the vertex set $\{v_0, v_1, \ldots, v_n\}$ by joining v_i and v_j whenever the common boundary of T_i and T_j is an edge with labels 0 and 1 (see figure 1.15).

In this graph, v_0 is clearly of odd degree (exercise 1.9.1). It follows from corollary 1.1 that an odd number of the vertices $v_1, v_2, \ldots, v_n$ are of odd degree. Now it is easily seen that none of these vertices can have degree three, and so those with odd degree must have degree one. But a vertex v_i is of degree one if and only if the triangle T_i is distinguished □

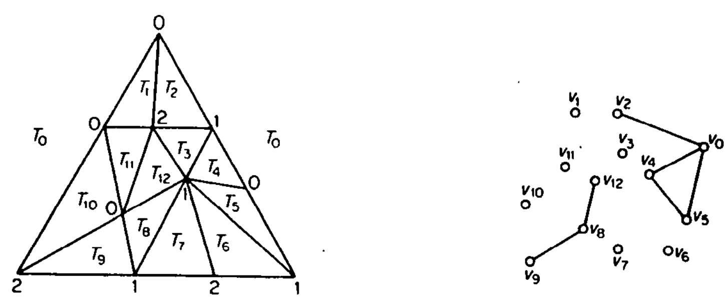


Figure 1.15

We shall now briefly indicate how Sperner's lemma can be used to deduce Brouwer's fixed-point theorem. Again, for simplicity, we shall only deal with the two-dimensional case. Since a closed 2-disc is homeomorphic to a closed triangle, it suffices to prove that a continuous mapping of a closed triangle to itself has a fixed point.

Let T be a given closed triangle with vertices x_0, x_1 and x_2. Then each point x of T can be written uniquely as $x = a_0x_0 + a_1x_1 + a_2x_2$, where each $a_i \geq 0$ and $\sum a_i = 1$, and we can represent x by the vector (a_0, a_1, a_2); the real numbers a_0, a_1 and a_2 are called the *barycentric coordinates* of x.

Now let f be any continuous mapping of T to itself, and suppose that

$$f(a_0, a_1, a_2) = (a'_0, a'_1, a'_2)$$

Define S_i as the set of points (a_0, a_1, a_2) in T for which $a'_i \leq a_i$. To show that f has a fixed point, it is enough to show that $S_0 \cap S_1 \cap S_2 \neq \emptyset$. For suppose that $(a_0, a_1, a_2) \in S_0 \cap S_1 \cap S_2$. Then, by the definition of S_i, we have that $a'_i \leq a_i$ for each i, and this, coupled with the fact that $\sum a'_i = \sum a_i$, yields

$$(a'_0, a'_1, a'_2) = (a_0, a_1, a_2)$$

In other words, (a_0, a_1, a_2) is a fixed point of f.

So consider an arbitrary subdivision of T and a proper labelling such that each vertex labelled i belongs to S_i; the existence of such a labelling is easily seen (exercise 1.9.2*a*). It follows from Sperner's lemma that there is a triangle in the subdivision whose three vertices belong to S_0, S_1 and S_2. Now this holds for any subdivision of T and, since it is possible to choose subdivisions in which each of the smaller triangles are of arbitrarily small diameter, we conclude that there exist three points of S_0, S_1 and S_2 which are arbitrarily close to one another. Because the sets S_i are closed (exercise 1.9.2*b*), one may deduce that $S_0 \cap S_1 \cap S_2 \neq \emptyset$.

For details of the above proof and other applications of Sperner's lemma, the reader is referred to Tompkins (1964).

Exercises

1.9.1 In the proof of Sperner's lemma, show that the vertex v_0 is of odd degree.

1.9.2 In the proof of Brouwer's fixed-point theorem, show that

(*a*) there exists a proper labelling such that each vertex labelled i belongs to S_i;

(*b*) the sets S_i are closed.

1.9.3 State and prove Sperner's lemma for higher dimensional simplices.

REFERENCES

Aho, A. V., Hopcroft, J. E. and Ullman, J. D. (1974). *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numer. Math.*, **1**, 269–71

Dreyfus, S. E. (1969). An appraisal of some shortest-path algorithms. *Operations Res.*, **17**, 395–412

Edmonds, J. (1965). Paths, trees and flowers. *Canad. J. Math.*, **17**, 449–67

Erdös, P. and Gallai, T. (1960). Graphs with prescribed degrees of vertices (Hungarian). *Mat. Lapok*, **11**, 264–74

Frucht, R. (1939). Herstellung von Graphen mit vorgegebener abstrakter Gruppe. *Compositio Math.*, **6**, 239–50

Hoffman, A. J. and Singleton, R. R. (1960). On Moore graphs with diameters 2 and 3. *IBM J. Res. Develop.*, **4**, 497–504

Sperner, E. (1928). Neuer Beweis für die Invarianz der Dimensionszahl und des Gebietes. *Hamburger Abhand.*, **6**, 265–72

Tompkins, C. B. (1964). Sperner's lemma and some extensions, in *Applied Combinatorial Mathematics*, ch. 15 (ed. E. F. Beckenbach), Wiley, New York, pp. 416–55

Whiting, P. D. and Hillier, J. A. (1960). A method for finding the shortest route through a road network. *Operational Res. Quart.*, **11**, 37–40

2 Trees

2.1 TREES

An *acyclic* graph is one that contains no cycles. A *tree* is a connected acyclic graph. The trees on six vertices are shown in figure 2.1.

Theorem 2.1 In a tree, any two vertices are connected by a unique path.

Proof By contradiction. Let G be a tree, and assume that there are two distinct (u, v)-paths P_1 and P_2 in G. Since $P_1 \neq P_2$, there is an edge $e = xy$ of P_1 that is not an edge of P_2. Clearly the graph $(P_1 \cup P_2) - e$ is connected. It therefore contains an (x, y)-path P. But then $P + e$ is a cycle in the acyclic graph G, a contradiction □

The converse of this theorem holds for graphs without loops (exercise 2.1.1).

Observe that all the trees on six vertices (figure 2.1) have five edges. In general we have:

Theorem 2.2 If G is a tree, then $\varepsilon = \nu - 1$.

Proof By induction on ν. When $\nu = 1$, $G \cong K_1$ and $\varepsilon = 0 = \nu - 1$.

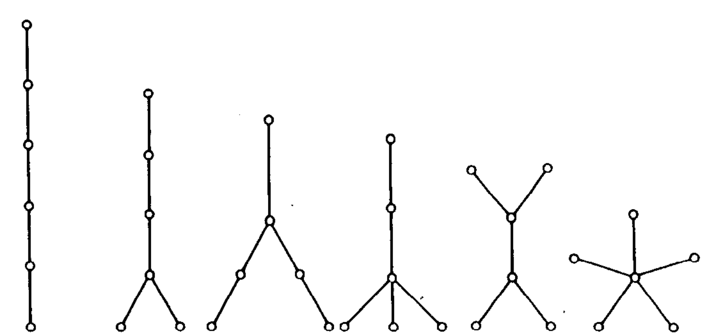


Figure 2.1. The trees on six vertices

Suppose the theorem true for all trees on fewer than ν vertices, and let G be a tree on $\nu \geq 2$ vertices. Let $uv \in E$. Then $G - uv$ contains no (u, v)-path, since uv is the unique (u, v)-path in G. Thus $G - uv$ is disconnected and so (exercise 1.6.8a) $\omega(G - uv) = 2$. The components G_1 and G_2 of $G - uv$, being acyclic, are trees. Moreover, each has fewer than ν vertices. Therefore, by the induction hypothesis

$$\varepsilon(G_i) = \nu(G_i) - 1 \quad \text{for} \quad i = 1, 2$$

Thus

$$\varepsilon(G) = \varepsilon(G_1) + \varepsilon(G_2) + 1 = \nu(G_1) + \nu(G_2) - 1 = \nu(G) - 1 \quad \square$$

Corollary 2.2 Every nontrivial tree has at least two vertices of degree one.

Proof Let G be a nontrivial tree. Then

$$d(v) \geq 1 \quad \text{for all} \quad v \in V$$

Also, by theorems 1.1 and 2.2, we have

$$\sum_{v \in V} d(v) = 2\varepsilon = 2\nu - 2$$

It now follows that $d(v) = 1$ for at least two vertices v $\square$

Another, perhaps more illuminating, way of proving corollary 2.2 is to show that the origin and terminus of a longest path in a nontrivial tree both have degree one (see exercise 2.1.2).

Exercises

2.1.1 Show that if any two vertices of a loopless graph G are connected by a unique path, then G is a tree.

2.1.2 Prove corollary 2.2 by showing that the origin and terminus of a longest path in a nontrivial tree both have degree one.

2.1.3 Prove corollary 2.2 by using exercise 1.7.2.

2.1.4 Show that every tree with exactly two vertices of degree one is a path.

2.1.5 Let G be a graph with $\nu - 1$ edges. Show that the following three statements are equivalent:

(*a*) G is connected;
(*b*) G is acyclic;
(*c*) G is a tree.

2.1.6 Show that if G is a tree with $\Delta \geq k$, then G has at least k vertices of degree one.

2.1.7 An acyclic graph is also called a *forest*. Show that

(*a*) each component of a forest is a tree;
(*b*) G is a forest if and only if $\varepsilon = \nu - \omega$.

2.1.8 A *centre* of G is a vertex u such that $\max_{v \in V} d(u, v)$ is as small as possible. Show that a tree has either exactly one centre or two, adjacent, centres.

2.1.9 Show that if G is a forest with exactly $2k$ vertices of odd degree, then there are k edge-disjoint paths $P_1, P_2, \ldots, P_k$ in G such that $E(G) = E(P_1) \cup E(P_2) \cup \ldots \cup E(P_k)$.

2.1.10* Show that a sequence $(d_1, d_2, \ldots, d_\nu)$ of positive integers is a degree sequence of a tree if and only if $\sum_{i=1}^{\nu} d_i = 2(\nu - 1)$.

2.1.11 Let T be an arbitrary tree on $k+1$ vertices. Show that if G is simple and $\delta \geq k$ then G has a subgraph isomorphic to T.

2.1.12 A saturated hydrocarbon is a molecule C_mH_n in which every carbon atom has four bonds, every hydrogen atom has one bond, and no sequence of bonds forms a cycle. Show that, for every positive integer m, C_mH_n can exist only if $n = 2m + 2$.

2.2 CUT EDGES AND BONDS

A *cut edge* of G is an edge e such that $\omega(G - e) > \omega(G)$. The graph of figure 2.2 has the three cut edges indicated.

Theorem 2.3 An edge e of G is a cut edge of G if and only if e is contained in no cycle of G.

Proof Let e be a cut edge of G. Since $\omega(G - e) > \omega(G)$, there exist vertices u and v of G that are connected in G but not in $G - e$. There is therefore some (u, v)-path P in G which, necessarily, traverses e. Suppose that x and y are the ends of e, and that x precedes y on P. In $G - e$, u is connected to x by a section of P and y is connected to v by a section of P. If e were in a cycle C, x and y would be connected in $G - e$ by the path $C - e$. Thus, u and v would be connected in $G - e$, a contradiction.

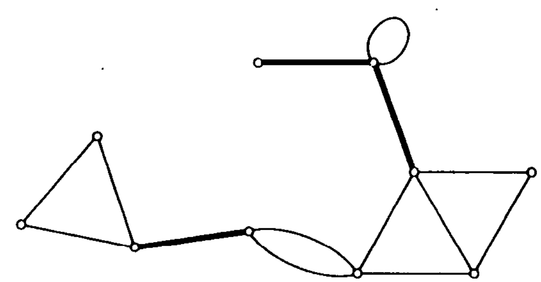


Figure 2.2. The cut edges of a graph

Conversely, suppose that $e = xy$ is not a cut edge of G; thus, $\omega(G-e) = \omega(G)$. Since there is an (x, y)-path (namely xy) in G, x and y are in the same component of G. It follows that x and y are in the same component of $G-e$, and hence that there is an (x, y)-path P in $G-e$. But then e is in the cycle $P+e$ of G □

Theorem 2.4 A connected graph is a tree if and only if every edge is a cut edge.

Proof Let G be a tree and let e be an edge of G. Since G is acyclic, e is contained in no cycle of G and is therefore, by theorem 2.3, a cut edge of G.

Conversely, suppose that G is connected but is not a tree. Then G contains a cycle C. By theorem 2.3, no edge of C can be a cut edge of G □

A *spanning tree* of G is a spanning subgraph of G that is a tree.

Corollary 2.4.1 Every connected graph contains a spanning tree.

Proof Let G be connected and let T be a minimal connected spanning subgraph of G. By definition $\omega(T) = 1$ and $\omega(T-e) > 1$ for each edge e of T. It follows that each edge of T is a cut edge and therefore, by theorem 2.4, that T, being connected, is a tree □

Figure 2.3 depicts a connected graph and one of its spanning trees.

Corollary 2.4.2 If G is connected, then $\varepsilon \geq \nu - 1$.

Proof Let G be connected. By corollary 2.4.1, G contains a spanning tree T. Therefore

$$\varepsilon(G) \geq \varepsilon(T) = \nu(T) - 1 = \nu(G) - 1 \quad \square$$

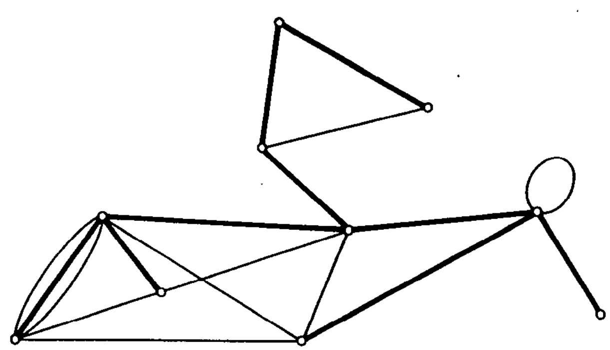


Figure 2.3. A spanning tree in a connected graph

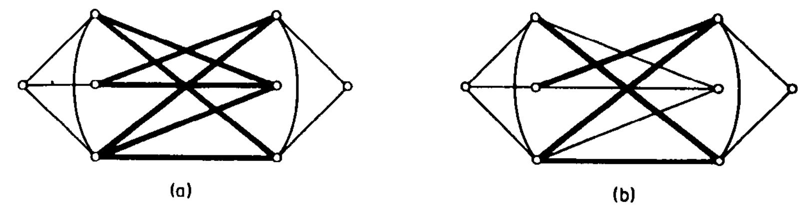


Figure 2.4. (*a*) An edge cut; (*b*) a bond

Theorem 2.5 Let T be a spanning tree of a connected graph G and let e be an edge of G not in T. Then $T+e$ contains a unique cycle.

Proof Since T is acyclic, each cycle of $T+e$ contains e. Moreover, C is a cycle of $T+e$ if and only if $C-e$ is a path in T connecting the ends of e. By theorem 2.1, T has a unique such path; therefore $T+e$ contains a unique cycle □

For subsets S and S' of V, we denote by $[S, S']$ the set of edges with one end in S and the other in S'. An *edge cut* of G is a subset of E of the form $[S, \bar{S}]$ where S is a nonempty proper subset of V and $\bar{S} = V \backslash S$. A minimal nonempty edge cut of G is called a *bond*; each cut edge e, for instance, gives rise to a bond $\{e\}$. If G is connected, then a bond B of G is a minimal subset of E such that $G-B$ is disconnected. Figure 2.4 indicates an edge cut and a bond in a graph.

If H is a subgraph of G, the *complement of H in G*, denoted by $\bar{H}(G)$, is the subgraph $G-E(H)$. If G is connected, a subgraph of the form $\bar{T}$, where T is a spanning tree, is called a *cotree* of G.

Theorem 2.6 Let T be a spanning tree of a connected graph G, and let e be any edge of T. Then

(i) the cotree $\bar{T}$ contains no bond of G;
(ii) $\bar{T}+e$ contains a unique bond of G.

Proof (i) Let B be a bond of G. Then $G-B$ is disconnected, and so cannot contain the spanning tree T. Therefore B is not contained in $\bar{T}$. (ii) Denote by S the vertex set of one of the two components of $T-e$. The edge cut $B = [S, \bar{S}]$ is clearly a bond of G, and is contained in $\bar{T}+e$. Now, for any $b \in B$, $T-e+b$ is a spanning tree of G. Therefore every bond of G contained in $\bar{T}+e$ must include every such element b. It follows that B is the only bond of G contained in $\bar{T}+e$ □

The relationship between bonds and cotrees is analogous to that between cycles and spanning trees. Statement (i) of theorem 2.6 is the analogue for bonds of the simple fact that a spanning tree is acyclic, and (ii) is the analogue of theorem 2.5. This 'duality' between cycles and bonds will be further explored in chapter 12 (see also exercise 2.2.10).

Exercises

2.2.1 Show that G is a forest if and only if every edge of G is a cut edge.

2.2.2 Let G be connected and let $e \in E$. Show that

(*a*) e is in every spanning tree of G if and only if e is a cut edge of G;

(*b*) e is in no spanning tree of G if and only if e is a loop of G.

2.2.3 Show that if G is loopless and has exactly one spanning tree T, then $G = T$.

2.2.4 Let F be a maximal forest of G. Show that

(*a*) for every component H of G, $F \cap H$ is a spanning tree of H;

(*b*) $\varepsilon(F) = \nu(G) - \omega(G)$.

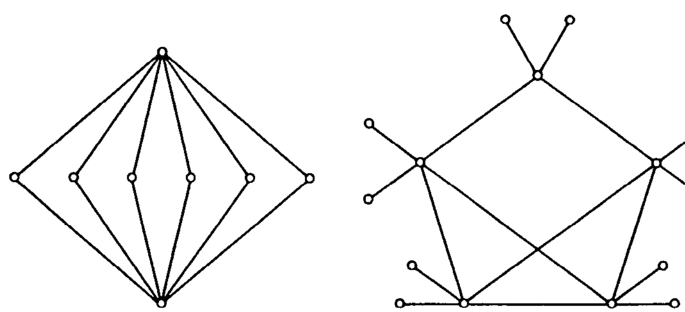
2.2.5 Show that G contains at least $\varepsilon - \nu + \omega$ distinct cycles.

2.2.6 Show that

(*a*) if each degree in G is even, then G has no cut edge;

(*b*) if G is a k-regular bipartite graph with $k \geq 2$, then G has no cut edge.

2.2.7 Find the number of nonisomorphic spanning trees in the following graphs:



2.2.8 Let G be connected and let S be a nonempty proper subset of V. Show that the edge cut $[S, \bar{S}]$ is a bond of G if and only if both $G[S]$ and $G[\bar{S}]$ are connected.

2.2.9 Show that every edge cut is a disjoint union of bonds.

2.2.10 Let B_1 and B_2 be bonds and let C_1 and C_2 be cycles (regarded as sets of edges) in a graph. Show that

(*a*) $B_1 \Delta B_2$ is a disjoint union of bonds;

(*b*) $C_1 \Delta C_2$ is a disjoint union of cycles,

where Δ denotes symmetric difference;

(*c*) for any edge e, $(B_1 \cup B_2)\backslash\{e\}$ contains a bond;

(*d*) for any edge e, $(C_1 \cup C_2)\backslash\{e\}$ contains a cycle.

2.2.11 Show that if a graph G contains k edge-disjoint spanning trees then, for each partition $(V_1, V_2, \ldots, V_n)$ of V, the number of edges which have ends in different parts of the partition is at least $k(n-1)$.

(Tutte, 1961 and Nash-Williams, 1961 have shown that this necessary condition for G to contain k edge-disjoint spanning trees is also sufficient.)

2.2.12* Let S be an n-element set, and let $\mathscr{A} = \{A_1, A_2, \ldots, A_n\}$ be a family of n distinct subsets of S. Show that there is an element $x \in S$ such that the sets $A_1 \cup \{x\}, A_2 \cup \{x\}, \ldots, A_n \cup \{x\}$ are all distinct.

2.3 CUT VERTICES

A vertex v of G is a *cut vertex* if E can be partitioned into two nonempty subsets E_1 and E_2 such that $G[E_1]$ and $G[E_2]$ have just the vertex v in common. If G is loopless and nontrivial, then v is a cut vertex of G if and only if $\omega(G - v) > \omega(G)$. The graph of figure 2.5 has the five cut vertices indicated.

Theorem 2.7 A vertex v of a tree G is a cut vertex of G if and only if $d(v) > 1$.

Proof If $d(v) = 0$, $G \cong K_1$ and, clearly, v is not a cut vertex.

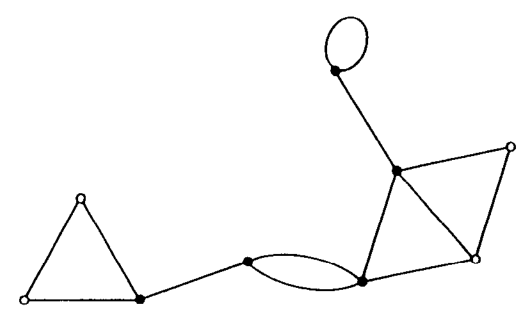


Figure 2.5. The cut vertices of a graph

If $d(v)=1$, $G-v$ is an acyclic graph with $\nu(G-v)-1$ edges, and thus (exercise 2.1.5) a tree. Hence $\omega(G-v)=1=\omega(G)$, and v is not a cut vertex of G.

If $d(v)>1$, there are distinct vertices u and w adjacent to v. The path uvw is a (u, w)-path in G. By theorem 2.1 uvw is the unique (u, w)-path in G. It follows that there is no (u, w)-path in $G-v$, and therefore that $\omega(G-v)>1=\omega(G)$. Thus v is a cut vertex of G □

Corollary 2.7 Every nontrivial loopless connected graph has at least two vertices that are not cut vertices.

Proof Let G be a nontrivial loopless connected graph. By corollary 2.4.1, G contains a spanning tree T. By corollary 2.2 and theorem 2.7, T has at least two vertices that are not cut vertices. Let v be any such vertex. Then

$$\omega(T-v)=1$$

Since T is a spanning subgraph of G, $T-v$ is a spanning subgraph of $G-v$ and therefore

$$\omega(G-v)\leq\omega(T-v)$$

It follows that $\omega(G-v)=1$, and hence that v is not a cut vertex of G. Since there are at least two such vertices v, the proof is complete □

Exercises

2.3.1 Let G be connected with $\nu\geq 3$. Show that

(*a*) if G has a cut edge, then G has a vertex v such that $\omega(G-v)>\omega(G)$;

(*b*) the converse of (*a*) is not necessarily true.

2.3.2 Show that a simple connected graph that has exactly two vertices which are not cut vertices is a path.

2.4 CAYLEY'S FORMULA

There is a simple and elegant recursive formula for the number of spanning trees in a graph. It involves the operation of contraction of an edge, which we now introduce. An edge e of G is said to be *contracted* if it is deleted and its ends are identified; the resulting graph is denoted by $G\cdot e$. Figure 2.6 illustrates the effect of contracting an edge.

It is clear that if e is a link of G, then

$$\nu(G\cdot e)=\nu(G)-1 \quad \varepsilon(G\cdot e)=\varepsilon(G)-1 \quad \text{and} \quad \omega(G\cdot e)=\omega(G)$$

Therefore, if T is a tree, so too is $T\cdot e$.

We denote the number of spanning trees of G by $\tau(G)$.

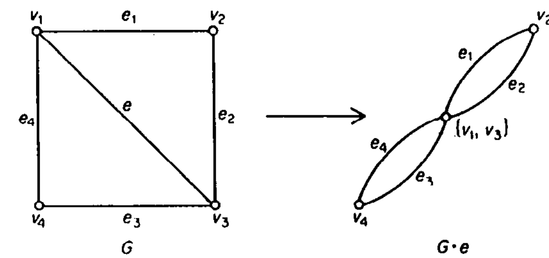


Figure 2.6. Contraction of an edge

Theorem 2.8 If e is a link of G, then $\tau(G)=\tau(G-e)+\tau(G\cdot e)$.

Proof Since every spanning tree of G that does not contain e is also a spanning tree of $G-e$, and conversely, $\tau(G-e)$ is the number of spanning trees of G that do not contain e.

Now to each spanning tree T of G that contains e, there corresponds a spanning tree $T\cdot e$ of $G\cdot e$. This correspondence is clearly a bijection (see figure 2.7). Therefore $\tau(G\cdot e)$ is precisely the number of spanning trees of G that contain e. It follows that $\tau(G)=\tau(G-e)+\tau(G\cdot e)$ □

Figure 2.8 illustrates the recursive calculation of $\tau(G)$ by means of theorem 2.8; the number of spanning trees in a graph is represented symbolically by the graph itself.

Although theorem 2.8 provides a method of calculating the number of spanning trees in a graph, this method is not suitable for large graphs. Fortunately, and rather surprisingly, there is a closed formula for $\tau(G)$ which expresses $\tau(G)$ as a determinant; we shall present this result in chapter 12. In the special case when G is complete, a simple formula for $\tau(G)$ was discovered by Cayley (1889). The proof we give is due to Prüfer (1918).

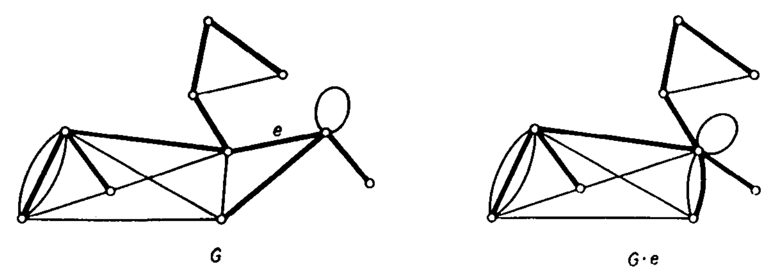


Figure 2.7

Theorem 2.9 $\tau(K_n) = n^{n-2}$.

Proof Let the vertex set of K_n be $N = \{1, 2, \ldots, n\}$. We note that n^{n-2} is the number of sequences of length $n-2$ that can be formed from N. Thus, to prove the theorem, it suffices to establish a one–one correspondence between the set of spanning trees of K_n and the set of such sequences.

With each spanning tree T of K_n, we associate a unique sequence $(t_1, t_2, \ldots, t_{n-2})$ as follows. Regarding N as an ordered set, let s_1 be the first vertex of degree one in T; the vertex adjacent to s_1 is taken as t_1. We now delete s_1 from T, denote by s_2 the first vertex of degree one in $T - s_1$, and take the vertex adjacent to s_2 as t_2. This operation is repeated until t_{n-2} has been defined and a tree with just two vertices remains; the tree in figure 2.9, for instance, gives rise to the sequence (4, 3, 5, 3, 4, 5). It can be seen that different spanning trees of K_n determine difference sequences.

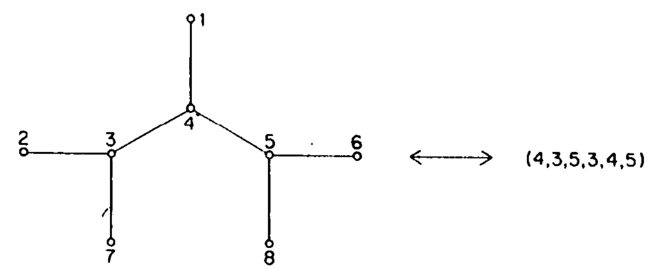


Figure 2.9

The reverse procedure is equally straightforward. Observe, first, that any vertex v of T occurs $d_T(v) - 1$ times in $(t_1, t_2, \ldots, t_{n-2})$. Thus the vertices of degree one in T are precisely those that do not appear in this sequence. To reconstruct T from $(t_1, t_2, \ldots, t_{n-2})$, we therefore proceed as follows. Let s_1 be the first vertex of N not in $(t_1, t_2, \ldots, t_{n-2})$; join s_1 to t_1. Next, let s_2 be the first vertex of $N\backslash\{s_1\}$ not in $(t_2, \ldots, t_{n-2})$, and join s_2 to t_2. Continue in this way until the $n-2$ edges $s_1t_1, s_2t_2, \ldots, s_{n-2}t_{n-2}$ have been determined. T is now obtained by adding the edge joining the two remaining vertices of $N\backslash\{s_1, s_2, \ldots, s_{n-2}\}$. It is easily verified that different sequences give rise to different spanning trees of K_n. We have thus established the desired one–one correspondence □

Note that n^{n-2} is not the number of nonisomorphic spanning trees of K_n, but the number of distinct spanning trees of K_n; there are just six nonisomorphic spanning trees of K_6 (see figure 2.1), whereas there are $6^4 = 1296$ distinct spanning trees of K_6.

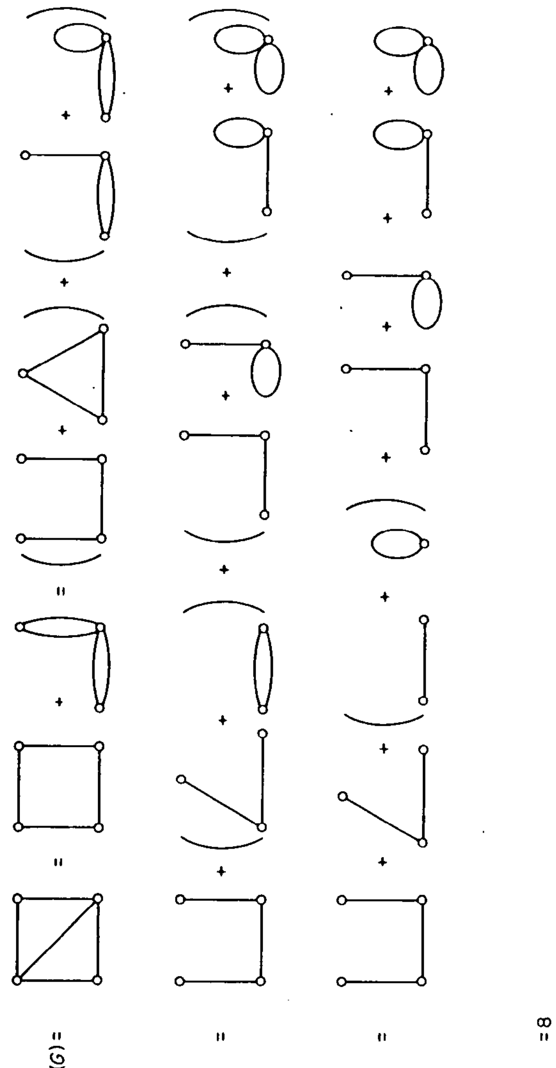


Figure 2.8. Recursive calculation of $\tau(G)$

Exercises

2.4.1 Using the recursion formula of theorem 2.8, evaluate the number of spanning trees in $K_{3,3}$.

2.4.2* A *wheel* is a graph obtained from a cycle by adding a new vertex and edges joining it to all the vertices of the cycle; the new edges are called the *spokes* of the wheel. Obtain an expression for the number of spanning trees in a wheel with n spokes.

2.4.3 Draw all sixteen spanning trees of K_4.

2.4.4 Show that if e is an edge of K_n, then $\tau(K_n - e) = (n-2)n^{n-3}$.

2.4.5 (*a*) Let H be a graph in which every two adjacent vertices are joined by k edges and let G be the underlying simple graph of H. Show that $\tau(H) = k^{\nu-1}\tau(G)$.

(*b*) Let H be the graph obtained from a graph G when each edge of G is replaced by a path of length k. Show that $\tau(H) = k^{\varepsilon-\nu+1}\tau(G)$.

(*c*) Deduce from (*b*) that $\tau(K_{2,n}) = n2^{n-1}$.

APPLICATIONS

2.5 THE CONNECTOR PROBLEM

A railway network connecting a number of towns is to be set up. Given the cost c_{ij} of constructing a direct link between towns v_i and v_j, design such a network to minimise the total cost of construction. This is known as the *connector problem*.

By regarding each town as a vertex in a weighted graph with weights $w(v_iv_j) = c_{ij}$, it is clear that this problem is just that of finding, in a weighted graph G, a connected spanning subgraph of minimum weight. Moreover, since the weights represent costs, they are certainly non-negative, and we may therefore assume that such a minimum-weight spanning subgraph is a spanning tree T of G. A minimum-weight spanning tree of a weighted graph will be called an *optimal tree*; the spanning tree indicated in the weighted graph of figure 2.10 is an optimal tree (exercise 2.5.1).

We shall now present a good algorithm for finding an optimal tree in a nontrivial weighted connected graph, thereby solving the connector problem.

Consider, first, the case when each weight $w(e) = 1$. An optimal tree is then a spanning tree with as few edges as possible. Since each spanning tree of a graph has the same number of edges (theorem 2.2), in this special case we merely need to construct some spanning tree of the graph. A simple

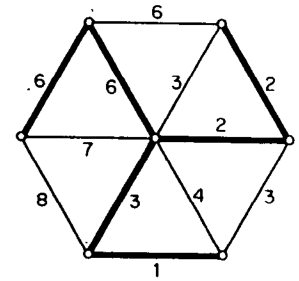


Figure 2.10. An optimal tree in a weighted graph

inductive algorithm for finding such a tree is the following:

1. Choose a link e_1.
2. If edges $e_1, e_2, \ldots, e_i$ have been chosen, then choose e_{i+1} from $E\backslash\{e_1, e_2, \ldots, e_i\}$ in such a way that $G[\{e_1, e_2, \ldots, e_{i+1}\}]$ is acyclic.
3. Stop when step 2 cannot be implemented further.

This algorithm works because a maximal acyclic subgraph of a connected graph is necessarily a spanning tree. It was extended by Kruskal (1956) to solve the general problem; his algorithm is valid for arbitrary real weights.

Kruskal's Algorithm

1. Choose a link e_1 such that $w(e_1)$ is as small as possible.
2. If edges $e_1, e_2, \ldots, e_i$ have been chosen, then choose an edge e_{i+1} from $E\backslash\{e_1, e_2, \ldots, e_i\}$ in such a way that
   (i) $G[\{e_1, e_2, \ldots, e_{i+1}\}]$ is acyclic;
   (ii) $w(e_{i+1})$ is as small as possible subject to (i).
3. Stop when step 2 cannot be implemented further.

As an example, consider the table of airline distances in miles between six of the largest cities in the world, London, Mexico City, New York, Paris, Peking and Tokyo:

	L	MC	NY	Pa	Pe	T
L	—	5558	3469	214	5074	5959
MC	5558	—	2090	5725	7753	7035
NY	3469	2090	—	3636	6844	6757
Pa	214	5725	3636	—	5120	6053
Pe	5074	7753	6844	5120	—	1307
T	5959	7035	6757	6053	1307	—

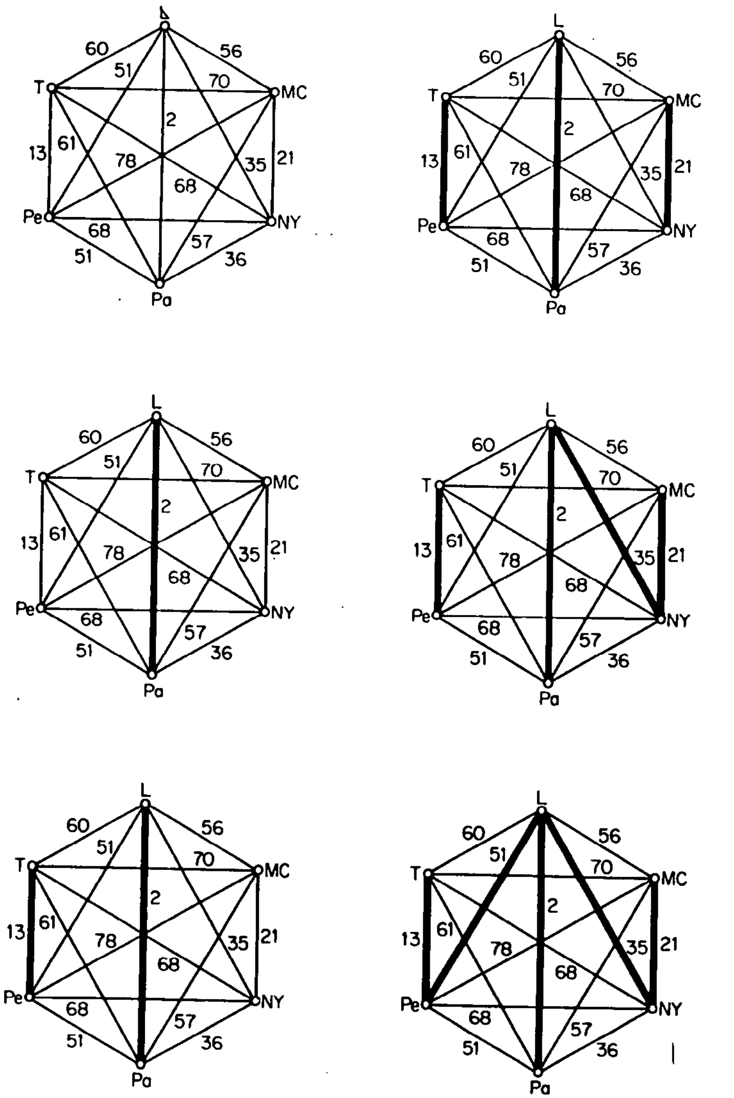


Figure 2.11

This table determines a weighted complete graph with vertices L, MC, NY, Pa, Pe and T. The construction of an optimal tree in this graph is shown in figure 2.11 (where, for convenience, distances are given in hundreds of miles).

Kruskal's algorithm clearly produces a spanning tree (for the same reason that the simpler algorithm above does). The following theorem ensures that such a tree will always be optimal.

Theorem 2.10 Any spanning tree $T^* = G[\{e_1, e_2, \ldots, e_{\nu-1}\}]$ constructed by Kruskal's algorithm is an optimal tree.

Proof By contradiction. For any spanning tree T of G other than T^*, denote by $f(T)$ the smallest value of i such that e_i is not in T. Now assume that T^* is not an optimal tree, and let T be an optimal tree such that $f(T)$ is as large as possible.

Suppose that $f(T) = k$; this means that $e_1, e_2, \ldots, e_{k-1}$ are in both T and T^*, but that e_k is not in T. By theorem 2.5, $T + e_k$ contains a unique cycle C. Let e_k' be an edge of C that is in T but not in T^*. By theorem 2.3, e_k' is not a cut edge of $T + e_k$. Hence $T' = (T + e_k) - e_k'$ is a connected graph with $\nu - 1$ edges, and therefore (exercise 2.1.5) is another spanning tree of G. Clearly

$$w(T') = w(T) + w(e_k) - w(e_k') \tag{2.1}$$

Now, in Kruskal's algorithm, e_k was chosen as an edge with the smallest weight such that $G[\{e_1, e_2, \ldots, e_k\}]$ was acyclic. Since $G[\{e_1, e_2, \ldots, e_{k-1}, e_k'\}]$ is a subgraph of T, it is also acyclic. We conclude that

$$w(e_k') \geq w(e_k) \tag{2.2}$$

Combining (2.1) and (2.2) we have

$$w(T') \leq w(T)$$

and so T', too, is an optimal tree. However

$$f(T') > k = f(T)$$

contradicting the choice of T. Therefore $T = T^*$, and T^* is indeed an optimal tree □

A flow diagram for Kruskal's algorithm is shown in figure 2.12. The edges are first sorted in order of increasing weight (box 1); this takes about $\varepsilon \log \varepsilon$ computations (see Knuth, 1973). Box 2 just checks to see how many edges have been chosen. (S is the set of edges already chosen and i is their number.) When $i = \nu - 1$, $S = \{e_1, e_2, \ldots, e_{\nu-1}\}$ is the edge set of an optimal tree T^* of G. In box 3, to check if $G[S \cup \{a_i\}]$ is acyclic, one must ascertain whether the ends of a_i are in different components of the forest $G[S]$ or not. This can be achieved in the following way. The vertices are labelled so that, at any stage, two vertices belong to the same component of $G[S]$ if and only

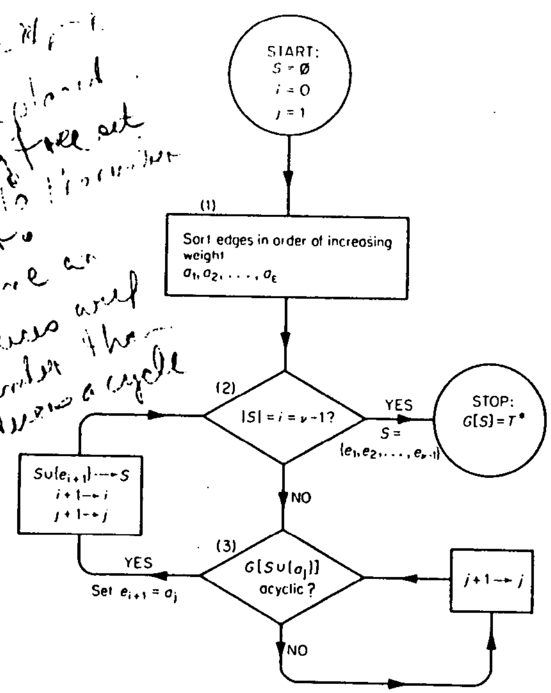


Figure 2.12. Kruskal's algorithm

if they have the same label; initially, vertex v_l is assigned the label l, $1 \le l \le \nu$. With this labelling scheme, $G[S \cup \{a_j\}]$ is acyclic if and only if the ends of a_j have different labels. If this is the case, a_j is taken as e_{i+1}; otherwise, a_j is discarded and a_{j+1}, the next candidate for e_{i+1}, is tested. Once e_{i+1} has been added to S, the vertices in the two components of $G[S]$ that contain the ends of e_{i+1} are relabelled with the smaller of their two labels. For each edge, one comparison suffices to check whether its ends have the same or different labels; this takes ε computations. After edge e_{i+1} has been added to S, the relabelling of vertices takes at most ν comparisons; hence, for all $\nu - 1$ edges $e_1, e_2, \ldots, e_{\nu-1}$ we need $\nu(\nu-1)$ computations. Kruskal's algorithm is therefore a good algorithm.

Exercises

2.5.1 Show, by applying Kruskal's algorithm, that the tree indicated in figure 2.10 is indeed optimal.

2.5.2 Adapt Kruskal's algorithm to solve the *connector problem with preassignments*: construct, at minimum cost, a network linking a number of towns, with the additional requirement that certain selected pairs of towns be directly linked.

2.5.3 Can Kruskal's algorithm be adapted to find

(*a*) a *maximum*-weight tree in a weighted connected graph?

(*b*) a minimum-weight maximal *forest* in a weighted graph?

If so, how?.

2.5.4 Show that the following Kruskal-type algorithm does not necessarily yield a minimum-weight spanning *path* in a weighted complete graph:

1. Choose a link e_1 such that $w(e_1)$ is as small as possible.
2. If edges $e_1, e_2, \ldots, e_i$ have been chosen, then choose an edge e_{i+1} from $E \backslash \{e_1, e_2, \ldots, e_i\}$ in such a way that
   (i) $G[\{e_1, e_2, \ldots, e_{i+1}\}]$ is a union of disjoint paths;
   (ii) $w(e_{i+1})$ is as small as possible subject to (i).
3. Stop when step 2 cannot be implemented further.

2.5.5 The *tree graph* of a connected graph G is the graph whose vertices are the spanning trees $T_1, T_2, \ldots, T_\tau$ of G, with T_i and T_j joined if and only if they have exactly $\nu - 2$ edges in common. Show that the tree graph of any connected graph is connected.

REFERENCES

Cayley, A. (1889). A theorem on trees. *Quart. J. Math.*, **23**, 376–78

Knuth, D. E. (1973). *The Art of Computer Programming*, vol. 3: Sorting and Searching, Addison-Wesley, Reading, Mass., p. 184

Kruskal, J. B. Jr. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. Amer. Math. Soc.*, **7**, 48–50

Nash-Williams, C. St. J. A. (1961). Edge-disjoint spanning trees of finite graphs. *J. London Math. Soc.*, **36**, 445–50

Prüfer, H. (1918). Neuer Beweis eines Satzes über Permutationen. *Arch. Math. Phys.*, **27**, 742–44

Tutte, W. T. (1961). On the problem of decomposing a graph into n connected factors. *J. London Math. Soc.*, **36**, 221–30

3 Connectivity

3.1 CONNECTIVITY

In section 1.6 we introduced the concept of connection in graphs. Consider, now, the four connected graphs of figure 3.1.

G_1 is a tree, a minimal connected graph; deleting any edge disconnects it. G_2 cannot be disconnected by the deletion of a single edge, but can be disconnected by the deletion of one vertex, its cut vertex. There are no cut edges or cut vertices in G_3, but even so G_3 is clearly not as well connected as G_4, the complete graph on five vertices. Thus, intuitively, each successive graph is more strongly connected than the previous one. We shall now define two parameters of a graph, its connectivity and edge connectivity, which measure the extent to which it is connected.

A *vertex cut* of G is a subset V' of V such that $G - V'$ is disconnected. A *k-vertex cut* is a vertex cut of k elements. A complete graph has no vertex cut; in fact, the only graphs which do not have vertex cuts are those that contain complete graphs as spanning subgraphs. If G has at least one pair of distinct nonadjacent vertices, the *connectivity* $\kappa(G)$ of G is the minimum k for which G has a k-vertex cut; otherwise, we define $\kappa(G)$ to be $\nu - 1$. Thus $\kappa(G) = 0$ if G is either trivial or disconnected. G is said to be *k-connected* if $\kappa(G) \geq k$. All nontrivial connected graphs are 1-connected.

Recall that an edge cut of G is a subset of E of the form $[S, \bar{S}]$, where S is a nonempty proper subset of V. A *k-edge cut* is an edge cut of k elements. If G is nontrivial and E' is an edge cut of G, then $G - E'$ is disconnected; we then define the *edge connectivity* $\kappa'(G)$ of G to be the minimum k for which G has a k-edge cut. If G is trivial, $\kappa'(G)$ is defined to be zero. Thus $\kappa'(G) = 0$ if G is either trivial or disconnected, and $\kappa'(G) = 1$ if G is a connected graph with a cut edge. G is said to be *k-edge-connected* if $\kappa'(G) \geq k$. All nontrivial connected graphs are 1-edge-connected.

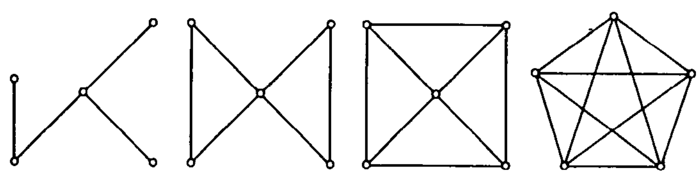


Figure 3.1

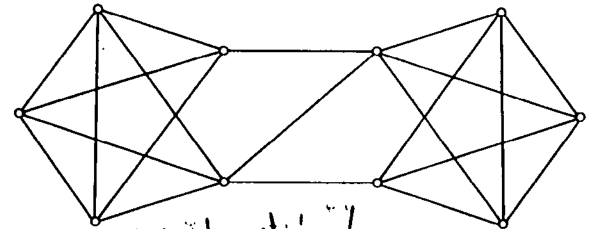


Figure 3.2

Theorem 3.1 $\kappa \leq \kappa' \leq \delta$.

Proof If G is trivial, then $\kappa' = 0 \leq \delta$. Otherwise, the set of links incident with a vertex of degree δ constitute a δ-edge cut of G. It follows that $\kappa' \leq \delta$.

We prove that $\kappa \leq \kappa'$ by induction on κ'. The result is true if $\kappa' = 0$, since then G must be either trivial or disconnected. Suppose that it holds for all graphs with edge connectivity less than k, let G be a graph with $\kappa'(G) = k > 0$, and let e be an edge in a k-edge cut of G. Setting $H = G - e$, we have $\kappa'(H) = k - 1$ and so, by the induction hypothesis, $\kappa(H) \leq k - 1$.

If H contains a complete graph as a spanning subgraph, then so does G and

$$\kappa(G) = \kappa(H) \leq k - 1$$

Otherwise, let S be a vertex cut of H with $\kappa(H)$ elements. Since $H - S$ is disconnected, either $G - S$ is disconnected, and then

$$\kappa(G) \leq \kappa(H) \leq k - 1$$

or else $G - S$ is connected and e is a cut edge of $G - S$. In this latter case, either $\nu(G - S) = 2$ and

$$\kappa(G) \leq \nu(G) - 1 = \kappa(H) + 1 \leq k$$

or (exercise 2.3.1a) $G - S$ has a 1-vertex cut $\{v\}$, implying that $S \cup \{v\}$ is a vertex cut of G and

$$\kappa(G) \leq \kappa(H) + 1 \leq k$$

Thus in each case we have $\kappa(G) \leq k = \kappa'(G)$. The result follows by the principle of induction □

The inequalities in theorem 3.1 are often strict. For example, the graph G of figure 3.2 has $\kappa = 2$, $\kappa' = 3$ and $\delta = 4$.

Exercises

3.1.1 (*a*) Show that if G is k-edge-connected, with $k>0$, and if E' is a set of k edges of G, then $\omega(G-E')\leq 2$.
(*b*) For $k>0$, find a k-connected graph G and a set V' of k vertices of G such that $\omega(G-V')>2$.

3.1.2 Show that if G is k-edge-connected, then $\varepsilon \geq k\nu/2$.

3.1.3 (*a*) Show that if G is simple and $\delta \geq \nu-2$, then $\kappa=\delta$.
(*b*) Find a simple graph G with $\delta=\nu-3$ and $\kappa<\delta$.

3.1.4 (*a*) Show that if G is simple and $\delta \geq \nu/2$, then $\kappa'=\delta$.
(*b*) Find a simple graph G with $\delta=[(\nu/2)-1]$ and $\kappa'<\delta$.

3.1.5 Show that if G is simple and $\delta \geq (\nu+k-2)/2$, then G is k-connected.

3.1.6 Show that if G is simple and 3-regular, then $\kappa=\kappa'$.

3.1.7 Show that if l, m and n are integers such that $0<l\leq m\leq n$, then there exists a simple graph G with $\kappa=l$, $\kappa'=m$, and $\delta=n$.
(G. Chartrand and F. Harary)

3.2 BLOCKS

2 vertex connected

A connected graph that has no cut vertices is called a *block*. Every block with at least three vertices is 2-connected. A *block of a graph* is a subgraph that is a block and is maximal with respect to this property. Every graph is the union of its blocks; this is illustrated in figure 3.3.

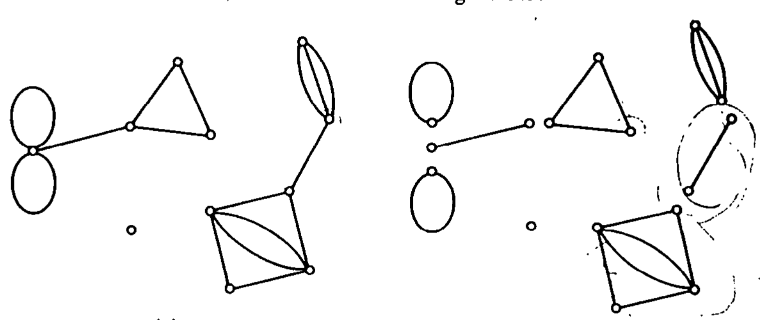


Figure 3.3. (*a*) G; (*b*) the blocks of G

A family of paths in G is said to be *internally-disjoint* if no vertex of G is an internal vertex of more than one path of the family. The following theorem is due to Whitney (1932).

Theorem 3.2 A graph G with $\nu \geq 3$ is 2-connected if and only if any two vertices of G are connected by at least two internally-disjoint paths.

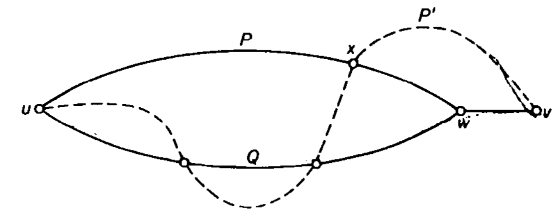


Figure 3.4

Proof If any two vertices of G are connected by at least two internally-disjoint paths then, clearly, G is connected and has no 1-vertex cut. Hence G is 2-connected.

Conversely, let G be a 2-connected graph. We shall prove, by induction on the distance $d(u, v)$ between u and v, that any two vertices u and v are connected by at least two internally-disjoint paths.

Suppose, first, that $d(u, v)=1$. Then, since G is 2-connected, the edge uv is not a cut edge and therefore, by theorem 2.3, it is contained in a cycle. It follows that u and v are connected by two internally-disjoint paths in G.

Now assume that the theorem holds for any two vertices at distance less than k, and let $d(u, v)=k\geq 2$. Consider a (u, v)-path of length k, and let w be the vertex that precedes v on this path. Since $d(u, w)=k-1$, it follows from the induction hypothesis that there are two internally-disjoint (u, w)-paths P and Q in G. Also, since G is 2-connected, $G-w$ is connected and so contains a (u, v)-path P'. Let x be the last vertex of P' that is also in $P\cup Q$ (see figure 3.4). Since u is in $P\cup Q$, there is such an x; we do not exclude the possibility that $x=v$.

We may assume, without loss of generality, that x is in P. Then G has two internally-disjoint (u, v)-paths, one composed of the section of P from u to x together with the section of P' from x to v, and the other composed of Q together with the path wv □

Corollary 3.2.1 If G is 2-connected, then any two vertices of G lie on a common cycle.

Proof This follows immediately from theorem 3.2 since two vertices lie on a common cycle if and only if they are connected by two internally-disjoint paths □

It is convenient, now, to introduce the operation of subdivision of an edge. An edge e is said to be *subdivided* when it is deleted and replaced by a path of length two connecting its ends, the internal vertex of this path being a new vertex. This is illustrated in figure 3.5.

0225

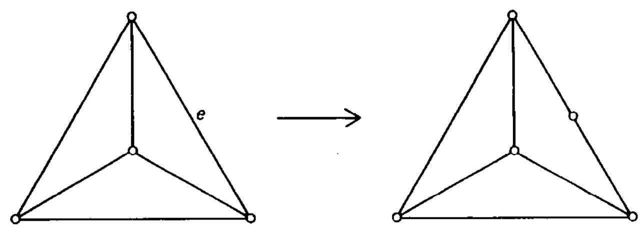


Figure 3.5. Subdivision of an edge

It can be seen that the class of blocks with at least three vertices is closed under the operation of subdivision. The proof of the next corollary uses this fact.

Corollary 3.2.2 If G is a block with $\nu \geq 3$, then any two edges of G lie on a common cycle.

Proof Let G be a block with $\nu \geq 3$, and let e_1 and e_2 be two edges of G. Form a new graph G' by subdividing e_1 and e_2, and denote the new vertices by v_1 and v_2. Clearly, G' is a block with at least five vertices, and hence is 2-connected. It follows from corollary 3.2.1 that v_1 and v_2 lie on a common cycle of G'. Thus e_1 and e_2 lie on a common cycle of G (see figure 3.6) □

Theorem 3.2 has a generalisation to k-connected graphs, known as *Menger's theorem*: a graph G with $\nu \geq k+1$ is k-connected if and only if any two distinct vertices of G are connected by at least k internally-disjoint paths. There is also an edge analogue of this theorem: a graph G is k-edge-connected if and only if any two distinct vertices of G are connected

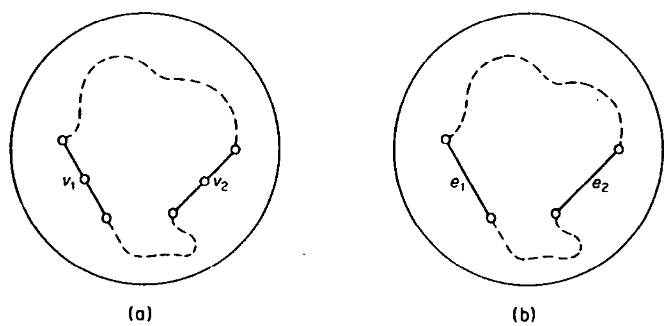


Figure 3.6. (*a*) G'; (*b*) G

by at least k edge-disjoint paths. Proofs of these theorems will be given in chapter 11.

Exercises

3.2.1 Show that a graph is 2-edge-connected if and only if any two vertices are connected by at least two edge-disjoint paths.

3.2.2 Give an example to show that if P is a (u, v)-path in a 2-connected graph G, then G does not necessarily contain a (u, v)-path Q internally-disjoint from P.

3.2.3 Show that if G has no even cycles, then each block of G is either K_1 or K_2, or an odd cycle.

3.2.4 Show that a connected graph which is not a block has at least two blocks that each contain exactly one cut vertex.

3.2.5 Show that the number of blocks in G is equal to $\omega + \sum_{v \in V} (b(v) - 1)$, where $b(v)$ denotes the number of blocks of G containing v.

3.2.6* Let G be a 2-connected graph and let X and Y be disjoint subsets of V, each containing at least two vertices. Show that G contains disjoint paths P and Q such that

(i) the origins of P and Q belong to X,
(ii) the termini of P and Q belong to Y, and
(iii) no internal vertex of P or Q belongs to $X \cup Y$.

3.2.7* A nonempty graph G is κ-*critical* if, for every edge e, $\kappa(G-e) < \kappa(G)$.

(*a*) Show that every κ-critical 2-connected graph has a vertex of degree two.
(Halin, 1969 has shown that, in general, every κ-critical k-connected graph has a vertex of degree k.)

(*b*) Show that if G is a κ-critical 2-connected graph with $\nu \geq 4$, then $\varepsilon \leq 2\nu - 4$. (G. A. Dirac)

3.2.8 Describe a good algorithm for finding the blocks of a graph.

APPLICATIONS

3.3 CONSTRUCTION OF RELIABLE COMMUNICATION NETWORKS

If we think of a graph as representing a communication network, the connectivity (or edge connectivity) becomes the smallest number of communication stations (or communication links) whose breakdown would jeopardise communication in the system. The higher the connectivity and edge connectivity, the more reliable the network. From this point of view, a tree network, such as the one obtained by Kruskal's algorithm, is not very reliable, and one is led to consider the following generalisation of the connector problem.

Let k be a given positive integer and let G be a weighted graph. Determine a minimum-weight k-connected spanning subgraph of G.

For $k = 1$, this problem reduces to the connector problem, which can be solved by Kruskal's algorithm. For values of k greater than one, the problem is unsolved and is known to be difficult. However, if G is a complete graph in which each edge is assigned unit weight, then the problem has a simple solution which we now present.

Observe that, for a weighted complete graph on n vertices in which each edge is assigned unit weight, a minimum-weight m-connected spanning subgraph is simply an m-connected graph on n vertices with as few edges as possible. We shall denote by $f(m, n)$ the least number of edges that an m-connected graph on n vertices can have. (It is, of course, assumed that $m < n$.) By theorems 3.1 and 1.1

$$f(m, n) \geq \{mn/2\} \tag{3.1}$$

We shall show that equality holds in (3.1) by constructing an m-connected graph $H_{m,n}$ on n vertices that has exactly $\{mn/2\}$ edges. The structure of $H_{m,n}$ depends on the parities of m and n; there are three cases.

Case 1 m even. Let $m = 2r$. Then $H_{2r,n}$ is constructed as follows. It has vertices $0, 1, \ldots, n-1$ and two vertices i and j are joined if $i - r \leq j \leq i + r$ (where addition is taken modulo n). $H_{4,8}$ is shown in figure 3.7*a*.

Case 2 m odd, n even. Let $m = 2r+1$. Then $H_{2r+1,n}$ is constructed by first drawing $H_{2r,n}$ and then adding edges joining vertex i to vertex $i+(n/2)$ for $1 \leq i \leq n/2$. $H_{5,8}$ is shown in figure 3.7*b*.

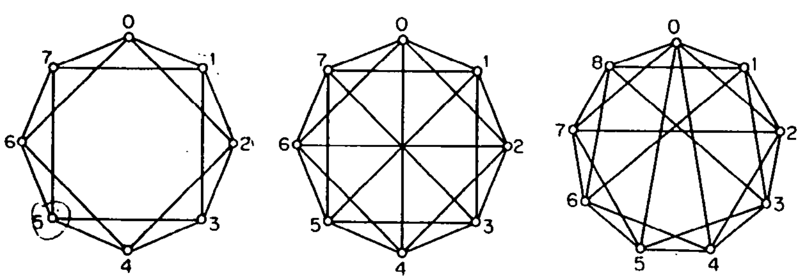


Figure 3.7. (*a*) $H_{4,8}$; (*b*) $H_{5,8}$; (*c*) $H_{5,9}$

Case 3 m odd, n odd. Let $m = 2r+1$. Then $H_{2r+1,n}$ is constructed by first drawing $H_{2r,n}$ and then adding edges joining vertex 0 to vertices $(n-1)/2$ and $(n+1)/2$ and vertex i to vertex $i+(n+1)/2$ for $1 \leq i < (n-1)/2$. $H_{5,9}$ is shown in figure 3.7*c*.

Theorem 3.3 (Harary, 1962) The graph $H_{m,n}$ is m-connected.

Proof Consider the case $m = 2r$. We shall show that $H_{2r,n}$ has no vertex cut of fewer than $2r$ vertices. If possible, let V' be a vertex cut with $|V'| < 2r$. Let i and j be vertices belonging to different components of $H_{2r,n} - V'$. Consider the two sets of vertices

$$S = \{i, i+1, \ldots, j-1, j\}$$

and

$$T = \{j, j+1, \ldots, i-1, i\}$$

where addition is taken modulo n. Since $|V'| < 2r$, we may assume, without loss of generality, that $|V' \cap S| < r$. Then there is clearly a sequence of distinct vertices in $S \backslash V'$ which starts with i, ends with j, and is such that the difference between any two consecutive terms is at most r. But such a sequence is an (i, j)-path in $H_{2r,n} - V'$, a contradiction. Hence $H_{2r,n}$ is $2r$-connected.

The case $m = 2r+1$ is left as an exercise (exercise 3.3.1) □

It is easy to see that $\varepsilon(H_{m,n}) = \{mn/2\}$. Thus, by theorem 3.3,

$$f(m, n) \leq \{mn/2\} \tag{3.2}$$

It now follows from (3.1) and (3.2) that

$$f(m, n) = \{mn/2\}$$

and that $H_{m,n}$ is an m-connected graph on n vertices with as few edges as possible.

We note that since, for any graph G, $\kappa \leq \kappa'$ (theorem 3.1), $H_{m,n}$ is also m-edge-connected. Thus, denoting by $g(m, n)$ the least possible number of edges in an m-edge-connected graph on n vertices, we have, for $1 < m < n$

$$g(m, n) = \{mn/2\} \tag{3.3}$$

Exercises

3.3.1 Show that $H_{2r+1,n}$ is $(2r+1)$-connected.

3.3.2 Show that $\kappa(H_{m,n}) = \kappa'(H_{m,n}) = m$.

3.3.3 Find a graph with nine vertices and 23 edges that is 5-connected but not isomorphic to the graph $H_{5,9}$ of figure 3.7*c*.

3.3.4 Show that (3.3) holds for all values of m and n with $m > 1$ and $n > 1$.

3.3.5 Find, for all $\nu \geq 5$, a 2-connected graph G of diameter two with $\varepsilon = 2\nu - 5$.
(Murty, 1969 has shown that every such graph has at least this number of edges.)

REFERENCES

Halin, R. (1969). A theorem on n-connected graphs. *J. Combinatorial Theory*, **7**, 150–54

Harary, F. (1962). The maximum connectivity of a graph. *Proc. Nat. Acad. Sci. U.S.A.*, **48**, 1142–46

Murty, U. S. R. (1969). Extremal nonseparable graphs of diameter 2, in *Proof Techniques in Graph Theory* (ed. F. Harary), Academic Press, New York, pp. 111–18

Whitney, H. (1932). Non-separable and planar graphs. *Trans. Amer. Math. Soc.*, **34**, 339–62

4 Euler Tours and Hamilton Cycles

4.1 EULER TOURS

A trail that traverses every edge of G is called an *Euler trail* of G because Euler was the first to investigate the existence of such trails in graphs. In the earliest known paper on graph theory (Euler, 1736), he showed that it was impossible to cross each of the seven bridges of Königsberg once and only once during a walk through the town. A plan of Königsberg and the river Pregel is shown in figure 4.1*a*. As can be seen, proving that such a walk is impossible amounts to showing that the graph of figure 4.1*b* contains no Euler trail.

A *tour* of G is a closed walk that traverses each edge of G at least once. An *Euler tour* is a tour which traverses each edge exactly once (in other words, a closed Euler trail). A graph is *eulerian* if it contains an Euler tour.

Theorem 4.1 A nonempty connected graph is eulerian if and only if it has no vertices of odd degree.

Proof Let G be eulerian, and let C be an Euler tour of G with origin (and terminus) u. Each time a vertex v occurs as an internal vertex of C, two of the edges incident with v are accounted for. Since an Euler tour contains

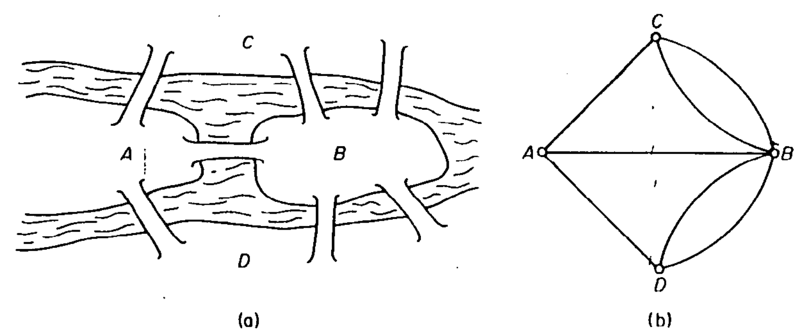


Figure 4.1. The bridges of Königsberg and their graph

Introduction to Algorithms

The MIT Electrical Engineering and Computer Science Series

Harold Abelson and Gerald Jay Sussman with Julie Sussman, *Structure and Interpretation of Computer Programs*, 1985.

William McC. Siebert, *Circuits, Signals, and Systems*, 1986.

Berthold Klaus Paul Horn, *Robot Vision*, 1986.

Barbara Liskov and John Guttag, *Abstraction and Specification in Program Development*, 1986.

Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms*, 1990.

Stephen A. Ward and Robert H. Halstead, Jr., *Computation Structures*, 1990.

Thomas H. Cormen
Charles E. Leiserson
Ronald L. Rivest

Introduction to Algorithms

The MIT Press
Cambridge, Massachusetts London, England

McGraw-Hill Book Company
New York St. Louis San Francisco Montreal Toronto

Third printing, 1991

This book is one of a series of texts written by faculty of the Electrical Engineering and Computer Science Department at the Massachusetts Institute of Technology. It was edited and produced by The MIT Press under a joint production-distribution agreement with the McGraw-Hill Book Company.

Ordering Information:

North America
Text orders should be addressed to the McGraw-Hill Book Company. All other orders should be addressed to The MIT Press.

Outside North America
All orders should be addressed to The MIT Press or its local distributor.

This book was printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Cormen, Thomas H.
Introduction to Algorithms / Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest.
p. cm.—(The MIT electrical engineering and computer science series)
Includes bibliographical references.
ISBN 0-262-03141-8 (MIT Press).—ISBN 0-07-013143-0 (McGraw-Hill)
1. Electronic digital computers—Programming. 2. Algorithms.
I. Leiserson, Charles Eric. II. Rivest, Ronald L. III. Title
IV. Series.
QA76.6.C662 1989
005.1–dc20 89-13027
CIP

Contents

II Sorting and Order Statistics

III Data Structures

IV Advanced Design and Analysis Techniques

0233

V *Advanced Data Structures*

VI *Graph Algorithms*

VII Selected Topics

$a\,R\,b$ implies $b\,R\,a$. Transitivity, therefore, implies $a\,R\,a$. Is the professor correct?

5.3 Functions

Given two sets A and B, a ***function*** f is a binary relation on $A \times B$ such that for all $a \in A$, there exists precisely one $b \in B$ such that $(a,b) \in f$. The set A is called the ***domain*** of f, and the set B is called the ***codomain*** of f. We sometimes write $f : A \to B$; and if $(a,b) \in f$, we write $b = f(a)$, since b is uniquely determined by the choice of a.

Intuitively, the function f assigns an element of B to each element of A. No element of A is assigned two different elements of B, but the same element of B can be assigned to two different elements of A. For example, the binary relation

$$f = \{(a,b) : a \in \mathrm{N} \text{ and } b = a \bmod 2\}$$

is a function $f : \mathrm{N} \to \{0,1\}$, since for each natural number a, there is exactly one value b in $\{0,1\}$ such that $b = a \bmod 2$. For this example, $0 = f(0)$, $1 = f(1)$, $0 = f(2)$, etc. In contrast, the binary relation

$$g = \{(a,b) : a \in \mathrm{N} \text{ and } a + b \text{ is even}\}$$

is not a function, since $(1,3)$ and $(1,5)$ are both in g, and thus for the choice $a = 1$, there is not precisely one b such that $(a,b) \in g$.

Given a function $f : A \to B$, if $b = f(a)$, we say that a is the ***argument*** of f and that b is the ***value*** of f at a. We can define a function by stating its value for every element of its domain. For example, we might define $f(n) = 2n$ for $n \in \mathrm{N}$, which means $f = \{(n, 2n) : n \in \mathrm{N}\}$. Two functions f and g are ***equal*** if they have the same domain and codomain and if, for all a in the domain, $f(a) = g(a)$.

A ***finite sequence*** of length n is a function f whose domain is the set $\{0, 1, \ldots, n-1\}$. We often denote a finite sequence by listing its values: $\langle f(0), f(1), \ldots, f(n-1)\rangle$. An ***infinite sequence*** is a function whose domain is the set N of natural numbers. For example, the Fibonacci sequence, defined by (2.13), is the infinite sequence $\langle 0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots\rangle$.

When the domain of a function f is a Cartesian product, we often omit the extra parentheses surrounding the argument of f. For example, if $f : A_1 \times A_2 \times \cdots \times A_n \to B$, we would write $b = f(a_1, a_2, \ldots, a_n)$ instead of $b = f((a_1, a_2, \ldots, a_n))$. We also call each a_i an ***argument*** to the function f, though technically the (single) argument to f is the n-tuple $(a_1, a_2, \ldots, a_n)$.

If $f : A \to B$ is a function and $b = f(a)$, then we sometimes say that b is the ***image*** of a under f. The image of a set $A' \subseteq A$ under f is defined by

$$f(A') = \{b \in B : b = f(a) \text{ for some } a \in A'\} .$$

The ***range*** of f is the image of its domain, that is, $f(A)$. For example, the range of the function $f : \mathrm{N} \to \mathrm{N}$ defined by $f(n) = 2n$ is $f(\mathrm{N}) = \{m : m = 2n \text{ for some } n \in \mathrm{N}\}$.

A function is a ***surjection*** if its range is its codomain. For example, the function $f(n) = \lfloor n/2 \rfloor$ is a surjective function from N to N, since every element in N appears as the value of f for some argument. In contrast, the function $f(n) = 2n$ is not a surjective function from N to N, since no argument to f can produce 3 as a value. The function $f(n) = 2n$ is, however, a surjective function from the natural numbers to the even numbers. A surjection $f : A \to B$ is sometimes described as mapping A ***onto*** B. When we say that f is onto, we mean that it is surjective.

A function $f : A \to B$ is an ***injection*** if distinct arguments to f produce distinct values, that is, if $a \neq a'$ implies $f(a) \neq f(a')$. For example, the function $f(n) = 2n$ is an injective function from N to N, since each even number b is the image under f of at most one element of the domain, namely $b/2$. The function $f(n) = \lfloor n/2 \rfloor$ is not injective, since the value 1 is produced by two arguments: 2 and 3. An injection is sometimes called a ***one-to-one*** function.

A function $f : A \to B$ is a ***bijection*** if it is injective and surjective. For example, the function $f(n) = (-1)^n \lceil n/2 \rceil$ is a bijection from N to Z:

$$\begin{array}{ccr} 0 & \to & 0\,, \\ 1 & \to & -1\,, \\ 2 & \to & 1\,, \\ 3 & \to & -2\,, \\ 4 & \to & 2\,, \\ & \vdots & \end{array}$$

The function is injective, since no element of Z is the image of more than one element of N. It is surjective, since every element of Z appears as the image of some element of N. Hence, the function is bijective. A bijection is sometimes called a ***one-to-one correspondence***, since it pairs elements in the domain and codomain. A bijection from a set A to itself is sometimes called a ***permutation***.

When a function f is bijective, its ***inverse*** f^{-1} is defined as

$$f^{-1}(b) = a \text{ if and only if } f(a) = b .$$

For example, the inverse of the function $f(n) = (-1)^n \lceil n/2 \rceil$ is

$$f^{-1}(m) = \begin{cases} 2m & \text{if } m \geq 0\,, \\ -2m - 1 & \text{if } m < 0\,. \end{cases}$$

Exercises

5.3-1
Let A and B be finite sets, and let $f : A \to B$ be a function. Show that

a. if f is injective, then $|A| \le |B|$;

b. if f is surjective, then $|A| \ge |B|$.

5.3-2
Is the function $f(x) = x + 1$ bijective when the domain and the codomain are **N**? Is it bijective when the domain and the codomain are **Z**?

5.3-3
Give a natural definition for the inverse of a binary relation such that if a relation is in fact a bijective function, its relational inverse is its functional inverse.

5.3-4 ★
Give a bijection from **Z** to **Z** × **Z**.

5.4 Graphs

This section presents two kinds of graphs: directed and undirected. The reader should be aware that certain definitions in the literature differ from those given here, but for the most part, the differences are slight. Section 23.1 shows how graphs can be represented in computer memory.

A ***directed graph*** (or ***digraph***) G is a pair (V, E), where V is a finite set and E is a binary relation on V. The set V is called the ***vertex set*** of G, and its elements are called ***vertices*** (singular: ***vertex***). The set E is called the ***edge set*** of G, and its elements are called ***edges***. Figure 5.2(a) is a pictorial representation of a directed graph on the vertex set $\{1, 2, 3, 4, 5, 6\}$. Vertices are represented by circles in the figure, and edges are represented by arrows. Note that ***self-loops***—edges from a vertex to itself—are possible.

In an ***undirected graph*** $G = (V, E)$, the edge set E consists of *unordered* pairs of vertices, rather than ordered pairs. That is, an edge is a set $\{u, v\}$, where $u, v \in V$ and $u \neq v$. By convention, we use the notation (u, v) for an edge, rather than the set notation $\{u, v\}$, and (u, v) and (v, u) are considered to be the same edge. In an undirected graph, self-loops are forbidden, and so every edge consists of exactly two distinct vertices. Figure 5.2(b) is a pictorial representation of an undirected graph on the vertex set $\{1, 2, 3, 4, 5, 6\}$.

Many definitions for directed and undirected graphs are the same, although certain terms have slightly different meanings in the two contexts. If (u, v) is an edge in a directed graph $G = (V, E)$, we say that (u, v)

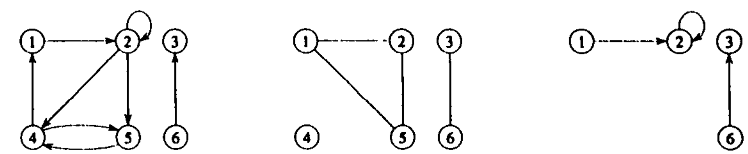


Figure 5.2 Directed and undirected graphs. **(a)** A directed graph $G = (V, E)$, where $V = \{1, 2, 3, 4, 5, 6\}$ and $E = \{(1,2), (2,2), (2,4), (2,5), (4,1), (4,5), (5,4), (6,3)\}$. The edge $(2,2)$ is a self-loop. **(b)** An undirected graph $G = (V, E)$, where $V = \{1, 2, 3, 4, 5, 6\}$ and $E = \{(1,2), (1,5), (2,5), (3,6)\}$. The vertex 4 is isolated. **(c)** The subgraph of the graph in part (a) induced by the vertex set $\{1, 2, 3, 6\}$.

is ***incident from*** or ***leaves*** vertex u and is ***incident to*** or ***enters*** vertex v. For example, the edges leaving vertex 2 in Figure 5.2(a) are $(2,2)$, $(2,4)$, and $(2,5)$. The edges entering vertex 2 are $(1,2)$ and $(2,2)$. If (u, v) is an edge in an undirected graph $G = (V, E)$, we say that (u, v) is ***incident on*** vertices u and v. In Figure 5.2(b), the edges incident on vertex 2 are $(1,2)$ and $(2,5)$.

If (u, v) is an edge in a graph $G = (V, E)$, we say that vertex v is ***adjacent*** to vertex u. When the graph is undirected, the adjacency relation is symmetric. When the graph is directed, the adjacency relation is not necessarily symmetric. If v is adjacent to u in a directed graph, we sometimes write $u \to v$. In parts (a) and (b) of Figure 5.2, vertex 2 is adjacent to vertex 1, since the edge $(1,2)$ belongs to both graphs. Vertex 1 is *not* adjacent to vertex 2 in Figure 5.2(a), since the edge $(2,1)$ does not belong to the graph.

The ***degree*** of a vertex in an undirected graph is the number of edges incident on it. For example, vertex 2 in Figure 5.2(b) has degree 2. In a directed graph, the ***out-degree*** of a vertex is the number of edges leaving it, and the ***in-degree*** of a vertex is the number of edges entering it. The ***degree*** of a vertex in a directed graph is its in-degree plus its out-degree. Vertex 2 in Figure 5.2(a) has in-degree 2, out-degree 3, and degree 5.

A ***path*** of ***length*** k from a vertex u to a vertex u' in a graph $G = (V, E)$ is a sequence $\langle v_0, v_1, v_2, \ldots, v_k \rangle$ of vertices such that $u = v_0$, $u' = v_k$, and $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \ldots, k$. The length of the path is the number of edges in the path. The path ***contains*** the vertices $v_0, v_1, \ldots, v_k$ and the edges $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$. If there is a path p from u to u', we say that u' is ***reachable*** from u via p, which we sometimes write as $u \overset{p}{\leadsto} u'$ if G is directed. A path is ***simple*** if all vertices in the path are distinct. In Figure 5.2(a), the path $\langle 1, 2, 5, 4 \rangle$ is a simple path of length 3. The path $\langle 2, 5, 4, 5 \rangle$ is not simple.

A ***subpath*** of path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a contiguous subsequence of its vertices. That is, for any $0 \le i \le j \le k$, the subsequence of vertices $\langle v_i, v_{i+1}, \ldots, v_j \rangle$ is a subpath of p.

In a directed graph, a path $\langle v_0, v_1, \ldots, v_k \rangle$ forms a ***cycle*** if $v_0 = v_k$ and the path contains at least one edge. The cycle is ***simple*** if, in addition, $v_1, v_2, \ldots, v_k$ are distinct. A self-loop is a cycle of length 1. Two paths $\langle v_0, v_1, v_2, \ldots, v_{k-1}, v_0 \rangle$ and $\langle v'_0, v'_1, v'_2, \ldots, v'_{k-1}, v'_0 \rangle$ form the same cycle if there exists an integer j such that $v'_i = v_{(i+j) \bmod k}$ for $i = 0, 1, \ldots, k-1$. In Figure 5.2(a), the path $\langle 1, 2, 4, 1 \rangle$ forms the same cycle as the paths $\langle 2, 4, 1, 2 \rangle$ and $\langle 4, 1, 2, 4 \rangle$. This cycle is simple, but the cycle $\langle 1, 2, 4, 5, 4, 1 \rangle$ is not. The cycle $\langle 2, 2 \rangle$ formed by the edge $(2, 2)$ is a self-loop. A directed graph with no self-loops is ***simple***. In an undirected graph, a path $\langle v_0, v_1, \ldots, v_k \rangle$ forms a ***cycle*** if $v_0 = v_k$ and $v_1, v_2, \ldots, v_k$ are distinct. For example, in Figure 5.2(b), the path $\langle 1, 2, 5, 1 \rangle$ is a cycle. A graph with no cycles is ***acyclic***.

An undirected graph is ***connected*** if every pair of vertices is connected by a path. The ***connected components*** of a graph are the equivalence classes of vertices under the "is reachable from" relation. The graph in Figure 5.2(b) has three connected components: $\{1, 2, 5\}$, $\{3, 6\}$, and $\{4\}$. Every vertex in $\{1, 2, 5\}$ is reachable from every other vertex in $\{1, 2, 5\}$. An undirected graph is connected if it has exactly one connected component, that is, if every vertex is reachable from every other vertex.

A directed graph is ***strongly connected*** if every two vertices are reachable from each other. The ***strongly connected components*** of a graph are the equivalence classes of vertices under the "are mutually reachable" relation. A directed graph is strongly connected if it has only one strongly connected component. The graph in Figure 5.2(a) has three strongly connected components: $\{1, 2, 4, 5\}$, $\{3\}$, and $\{6\}$. All pairs of vertices in $\{1, 2, 4, 5\}$ are mutually reachable. The vertices $\{3, 6\}$ do not form a strongly connected component, since vertex 6 cannot be reached from vertex 3.

Two graphs $G = (V, E)$ and $G' = (V', E')$ are ***isomorphic*** if there exists a bijection $f : V \to V'$ such that $(u, v) \in E$ if and only if $(f(u), f(v)) \in E'$. In other words, we can relabel the vertices of G to be vertices of G', maintaining the corresponding edges in G and G'. Figure 5.3(a) shows a pair of isomorphic graphs G and G' with respective vertex sets $V = \{1, 2, 3, 4, 5, 6\}$ and $V' = \{u, v, w, x, y, z\}$. The mapping from V to V' given by $f(1) = u$, $f(2) = v$, $f(3) = w$, $f(4) = x$, $f(5) = y$, $f(6) = z$ is the required bijective function. The graphs in Figure 5.3(b) are not isomorphic. Although both graphs have 5 vertices and 7 edges, the top graph has a vertex of degree 4 and the bottom graph does not.

We say that a graph $G' = (V', E')$ is a ***subgraph*** of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. Given a set $V' \subseteq V$, the subgraph of G ***induced*** by V' is the graph $G' = (V', E')$, where

$$E' = \{(u, v) \in E : u, v \in V'\} .$$

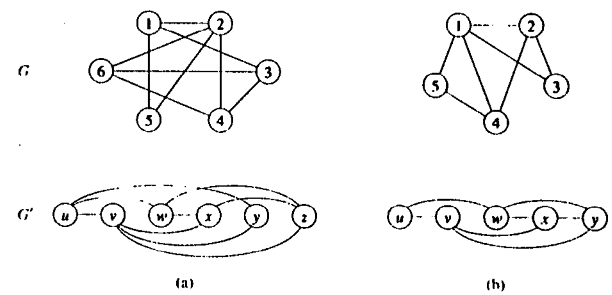


Figure 5.3 **(a)** A pair of isomorphic graphs. The vertices of the top graph are mapped to the vertices of the bottom graph by $f(1) = u, f(2) = v, f(3) = w, f(4) = x, f(5) = y, f(6) = z$. **(b)** Two graphs that are not isomorphic, since the top graph has a vertex of degree 4 and the bottom graph does not.

The subgraph induced by the vertex set $\{1, 2, 3, 6\}$ in Figure 5.2(a) appears in Figure 5.2(c) and has the edge set $\{(1, 2), (2, 2), (6, 3)\}$.

Given an undirected graph $G = (V, E)$, the ***directed version*** of G is the directed graph $G' = (V, E')$, where $(u, v) \in E'$ if and only if $(u, v) \in E$. That is, each undirected edge (u, v) in G is replaced in the directed version by the two directed edges (u, v) and (v, u). Given a directed graph $G = (V, E)$, the ***undirected version*** of G is the undirected graph $G' = (V, E')$, where $(u, v) \in E'$ if and only if $u \ne v$ and $(u, v) \in E$. That is, the undirected version contains the edges of G "with their directions removed" and with self-loops eliminated. (Since (u, v) and (v, u) are the same edge in an undirected graph, the undirected version of a directed graph contains it only once, even if the directed graph contains both edges (u, v) and (v, u).) In a directed graph $G = (V, E)$, a ***neighbor*** of a vertex u is any vertex that is adjacent to u in the undirected version of G. That is, v is a neighbor of u if either $(u, v) \in E$ or $(v, u) \in E$. In an undirected graph, u and v are neighbors if they are adjacent.

Several kinds of graphs are given special names. A ***complete graph*** is an undirected graph in which every pair of vertices is adjacent. A ***bipartite graph*** is an undirected graph $G = (V, E)$ in which V can be partitioned into two sets V_1 and V_2 such that $(u, v) \in E$ implies either $u \in V_1$ and $v \in V_2$ or $u \in V_2$ and $v \in V_1$. That is, all edges go between the two sets V_1 and V_2. An acyclic, undirected graph is a ***forest***, and a connected, acyclic, undirected graph is a ***(free) tree*** (see Section 5.5). We often take the first letters of "directed acyclic graph" and call such a graph a ***dag***.

There are two variants of graphs that you may occasionally encounter. A ***multigraph*** is like an undirected graph, but it can have both multiple edges between vertices and self-loops. A ***hypergraph*** is like an undirected graph, but each ***hyperedge***, rather than connecting two vertices, connects an arbitrary subset of vertices. Many algorithms written for ordinary directed and undirected graphs can be adapted to run on these graphlike structures.

Exercises

5.4-1
Attendees of a faculty party shake hands to greet each other, and each professor remembers how many times he or she shook hands. At the end of the party, the department head sums up the number of times that each professor shook hands. Show that the result is even by proving the ***handshaking lemma***: if $G = (V, E)$ is an undirected graph, then

$$\sum_{v \in V} \text{degree}(v) = 2|E| .$$

5.4-2
Show that in an undirected graph, the length of a cycle must be at least 3.

5.4-3
Show that if a directed or undirected graph contains a path between two vertices u and v, then it contains a simple path between u and v. Show that if a directed graph contains a cycle, then it contains a simple cycle.

5.4-4
Show that any connected, undirected graph $G = (V, E)$ satisfies $|E| \geq |V| - 1$.

5.4-5
Verify that in an undirected graph, the "is reachable from" relation is an equivalence relation on the vertices of the graph. Which of the three properties of an equivalence relation hold in general for the "is reachable from" relation on the vertices of a directed graph?

5.4-6
What is the undirected version of the directed graph in Figure 5.2(a)? What is the directed version of the undirected graph in Figure 5.2(b)?

5.4-7 ★
Show that a hypergraph can be represented by a bipartite graph if we let incidence in the hypergraph correspond to adjacency in the bipartite graph. (*Hint:* Let one set of vertices in the bipartite graph correspond to vertices of the hypergraph, and let the other set of vertices of the bipartite graph correspond to hyperedges.)

5.5 Trees

As with graphs, there are many related, but slightly different, notions of trees. This section presents definitions and mathematical properties of several kinds of trees. Sections 11.4 and 23.1 describe how trees can be represented in a computer memory.

5.5.1 Free trees

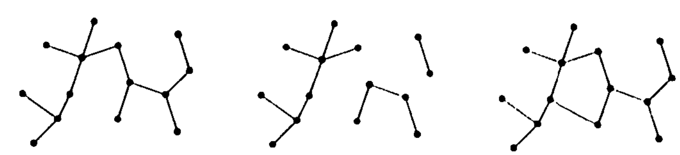
As defined in Section 5.4, a ***free tree*** is a connected, acyclic, undirected graph. We often omit the adjective "free" when we say that a graph is a tree. If an undirected graph is acyclic but possibly disconnected, it is a ***forest***. Many algorithms that work for trees also work for forests. Figure 5.4(a) shows a free tree, and Figure 5.4(b) shows a forest. The forest in Figure 5.4(b) is not a tree because it is not connected. The graph in Figure 5.4(c) is neither a tree nor a forest, because it contains a cycle.

The following theorem captures many important facts about free trees.

Theorem 5.2 (Properties of free trees)
Let $G = (V, E)$ be an undirected graph. The following statements are equivalent.

1. G is a free tree.
2. Any two vertices in G are connected by a unique simple path.
3. G is connected, but if any edge is removed from E, the resulting graph is disconnected.
4. G is connected, and $|E| = |V| - 1$.
5. G is acyclic, and $|E| = |V| - 1$.
6. G is acyclic, but if any edge is added to E, the resulting graph contains a cycle.

Proof (1) ⇒ (2): Since a tree is connected, any two vertices in G are connected by at least one simple path. Let u and v be vertices that are



(a) (b) (c)

Figure 5.4 (a) A free tree. (b) A forest. (c) A graph that contains a cycle and is therefore neither a tree nor a forest.

toward finding k that is accomplished by lines 7–9. That is, phase 1 consists of moving ahead in the list by random skips only. Likewise, phase 2 discounts progress accomplished by lines 4–6, and thus it operates like ordinary linear search.

Let X_t be the random variable that describes the distance in the linked list (that is, through the chain of *next* pointers) from position i to the desired key k after t iterations of phase 1.

b. Argue that the expected running time of COMPACT-LIST-SEARCH is $O(t + \mathrm{E}[X_t])$ for all $t \geq 0$.

c. Show that $\mathrm{E}[X_t] \leq \sum_{r=1}^{n}(1 - r/n)^t$. (*Hint:* Use equation (6.28).)

d. Show that $\sum_{r=0}^{n-1} r^t \leq n^{t+1}/(t+1)$.

e. Prove that $\mathrm{E}[X_t] \leq n/(t+1)$, and explain why this formula makes intuitive sense.

f. Show that COMPACT-LIST-SEARCH runs in $O(\sqrt{n})$ expected time.

Chapter notes

Aho, Hopcroft, and Ullman [5] and Knuth [121] are excellent references for elementary data structures. Gonnet [90] provides experimental data on the performance of many data structure operations.

The origin of stacks and queues as data structures in computer science is unclear, since corresponding notions already existed in mathematics and paper-based business practices before the introduction of digital computers. Knuth [121] cites A. M. Turing for the development of stacks for subroutine linkage in 1947.

Pointer-based data structures also seem to be a folk invention. According to Knuth, pointers were apparently used in early computers with drum memories. The A-1 language developed by G. M. Hopper in 1951 represented algebraic formulas as binary trees. Knuth credits the IPL-II language, developed in 1956 by A. Newell, J. C. Shaw, and H. A. Simon, for recognizing the importance and promoting the use of pointers. Their IPL-III language, developed in 1957, included explicit stack operations.

12 Hash Tables

Many applications require a dynamic set that supports only the dictionary operations INSERT, SEARCH, and DELETE. For example, a compiler for a computer language maintains a symbol table, in which the keys of elements are arbitrary character strings that correspond to identifiers in the language. A hash table is an effective data structure for implementing dictionaries. Although searching for an element in a hash table can take as long as searching for an element in a linked list—$\Theta(n)$ time in the worst case—in practice, hashing performs extremely well. Under reasonable assumptions, the expected time to search for an element in a hash table is $O(1)$.

A hash table is a generalization of the simpler notion of an ordinary array. Directly addressing into an ordinary array makes effective use of our ability to examine an arbitrary position in an array in $O(1)$ time. Section 12.1 discusses direct addressing in more detail. Direct addressing is applicable when we can afford to allocate an array that has one position for every possible key.

When the number of keys actually stored is small relative to the total number of possible keys, hash tables become an effective alternative to directly addressing an array, since a hash table typically uses an array of size proportional to the number of keys actually stored. Instead of using the key as an array index directly, the array index is *computed* from the key. Section 12.2 presents the main ideas, and Section 12.3 describes how array indices can be computed from keys using hash functions. Several variations on the basic theme are presented and analyzed; the "bottom line" is that hashing is an extremely effective and practical technique: the basic dictionary operations require only $O(1)$ time on the average.

12.1 Direct-address tables

Direct addressing is a simple technique that works well when the universe U of keys is reasonably small. Suppose that an application needs a dynamic set in which each element has a key drawn from the universe $U = \{0, 1, \ldots, m-1\}$, where m is not too large. We shall assume that no two elements have the same key.

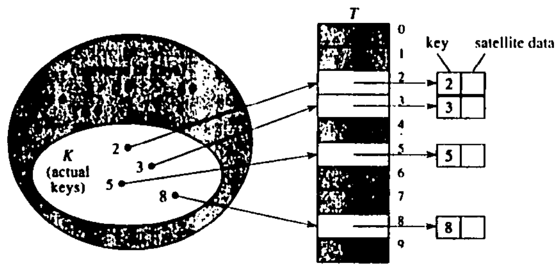


Figure 12.1 Implementing a dynamic set by a direct-address table T. Each key in the universe $U = \{0, 1, \ldots, 9\}$ corresponds to an index in the table. The set $K = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements. The other slots, heavily shaded, contain NIL.

To represent the dynamic set, we use an array, or ***direct-address table***, $T[0..m-1]$, in which each position, or ***slot***, corresponds to a key in the universe U. Figure 12.1 illustrates the approach; slot k points to an element in the set with key k. If the set contains no element with key k, then $T[k] = \text{NIL}$.

The dictionary operations are trivial to implement.

DIRECT-ADDRESS-SEARCH(T, k)
  return $T[k]$

DIRECT-ADDRESS-INSERT(T, x)
  $T[key[x]] \leftarrow x$

DIRECT-ADDRESS-DELETE(T, x)
  $T[key[x]] \leftarrow \text{NIL}$

Each of these operations is fast: only $O(1)$ time is required.

For some applications, the elements in the dynamic set can be stored in the direct-address table itself. That is, rather than storing an element's key and satellite data in an object external to the direct-address table, with a pointer from a slot in the table to the object, we can store the object in the slot itself, thus saving space. Moreover, it is often unnecessary to store the key field of the object, since if we have the index of an object in the table, we have its key. If keys are not stored, however, we must have some way to tell if the slot is empty.

Exercises

12.1-1
Consider a dynamic set S that is represented by a direct-address table T of length m. Describe a procedure that finds the maximum element of S. What is the worst-case performance of your procedure?

12.1-2
A ***bit vector*** is simply an array of bits (0's and 1's). A bit vector of length m takes much less space than an array of m pointers. Describe how to use a bit vector to represent a dynamic set of distinct elements with no satellite data. Dictionary operations should run in $O(1)$ time.

12.1-3
Suggest how to implement a direct-address table in which the keys of stored elements do not need to be distinct and the elements can have satellite data. All three dictionary operations (INSERT, DELETE, and SEARCH) should run in $O(1)$ time. (Don't forget that DELETE takes as an argument a pointer to an object to be deleted, not a key.)

12.1-4 ★
We wish to implement a dictionary by using direct addressing on a *huge* array. At the start, the array entries may contain garbage, and initializing the entire array is impractical because of its size. Describe a scheme for implementing a direct-address dictionary on a huge array. Each stored object should use $O(1)$ space; the operations SEARCH, INSERT, and DELETE should take $O(1)$ time each; and the initialization of the data structure should take $O(1)$ time. (*Hint:* Use an additional stack, whose size is the number of keys actually stored in the dictionary, to help determine whether a given entry in the huge array is valid or not.)

12.2 Hash tables

The difficulty with direct addressing is obvious: if the universe U is large, storing a table T of size $|U|$ may be impractical, or even impossible, given the memory available on a typical computer. Furthermore, the set K of keys *actually stored* may be so small relative to U that most of the space allocated for T would be wasted.

When the set K of keys stored in a dictionary is much smaller than the universe U of all possible keys, a hash table requires much less storage than a direct-address table. Specifically, the storage requirements can be reduced to $\Theta(|K|)$, even though searching for an element in the hash table still requires only $O(1)$ time. (The only catch is that this bound is for

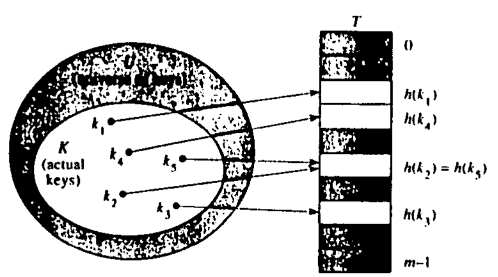


Figure 12.2 Using a hash function h to map keys to hash-table slots. Keys k_2 and k_5 map to the same slot, so they collide.

the *average time*, whereas for direct addressing it holds for the *worst-case time*.)

With direct addressing, an element with key k is stored in slot k. With hashing, this element is stored in slot $h(k)$; that is, a ***hash function*** h is used to compute the slot from the key k. Here h maps the universe U of keys into the slots of a ***hash table*** $T[0..m-1]$:

$$h : U \rightarrow \{0, 1, \ldots, m-1\} .$$

We say that an element with key k ***hashes*** to slot $h(k)$; we also say that $h(k)$ is the ***hash value*** of key k. Figure 12.2 illustrates the basic idea. The point of the hash function is to reduce the range of array indices that need to be handled. Instead of $|U|$ values, we need to handle only m values. Storage requirements are correspondingly reduced.

The fly in the ointment of this beautiful idea is that two keys may hash to the same slot—a ***collision***. Fortunately, there are effective techniques for resolving the conflict created by collisions.

Of course, the ideal solution would be to avoid collisions altogether. We might try to achieve this goal by choosing a suitable hash function h. One idea is to make h appear to be "random," thus avoiding collisions or at least minimizing their number. The very term "to hash," evoking images of random mixing and chopping, captures the spirit of this approach. (Of course, a hash function h must be deterministic in that a given input k should always produce the same output $h(k)$.) Since $|U| > m$, however, there must be two keys that have the same hash value; avoiding collisions altogether is therefore impossible. Thus, while a well-designed, "random"-looking hash function can minimize the number of collisions, we still need a method for resolving the collisions that do occur.

The remainder of this section presents the simplest collision resolution technique, called chaining. Section 12.4 introduces an alternative method for resolving collisions, called open addressing.

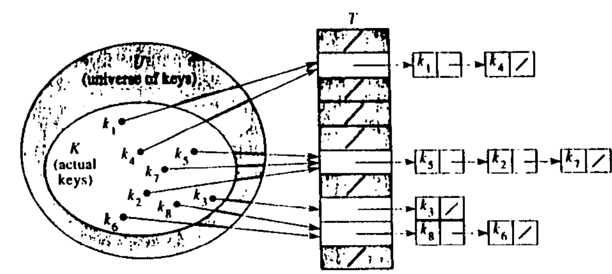


Figure 12.3 Collision resolution by chaining. Each hash-table slot $T[j]$ contains a linked list of all the keys whose hash value is j. For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_2) = h(k_7)$.

Collision resolution by chaining

In ***chaining***, we put all the elements that hash to the same slot in a linked list, as shown in Figure 12.3. Slot j contains a pointer to the head of the list of all stored elements that hash to j; if there are no such elements, slot j contains NIL.

The dictionary operations on a hash table T are easy to implement when collisions are resolved by chaining.

CHAINED-HASH-INSERT(T, x)

insert x at the head of list $T[h(key[x])]$

CHAINED-HASH-SEARCH(T, k)

search for an element with key k in list $T[h(k)]$

CHAINED-HASH-DELETE(T, x)

delete x from the list $T[h(key[x])]$

The worst-case running time for insertion is $O(1)$. For searching, the worst-case running time is proportional to the length of the list; we shall analyze this more closely below. Deletion of an element x can be accomplished in $O(1)$ time if the lists are doubly linked. (If the lists are singly linked, we must first find x in the list $T[h(key[x])]$, so that the *next* link of x's predecessor can be properly set to splice x out; in this case, deletion and searching have essentially the same running time.)

Analysis of hashing with chaining

How well does hashing with chaining perform? In particular, how long does it take to search for an element with a given key?

Given a hash table T with m slots that stores n elements, we define the ***load factor*** α for T as n/m, that is, the average number of elements stored in a chain. Our analysis will be in terms of α; that is, we imagine α staying fixed as n and m go to infinity. (Note that α can be less than, equal to, or greater than 1.)

The worst-case behavior of hashing with chaining is terrible: all n keys hash to the same slot, creating a list of length n. The worst-case time for searching is thus $\Theta(n)$ plus the time to compute the hash function—no better than if we used one linked list for all the elements. Clearly, hash tables are not used for their worst-case performance.

The average performance of hashing depends on how well the hash function h distributes the set of keys to be stored among the m slots, on the average. Section 12.3 discusses these issues, but for now we shall assume that any given element is equally likely to hash into any of the m slots, independently of where any other element has hashed to. We call this the assumption of ***simple uniform hashing***.

We assume that the hash value $h(k)$ can be computed in $O(1)$ time, so that the time required to search for an element with key k depends linearly on the length of the list $T[h(k)]$. Setting aside the $O(1)$ time required to compute the hash function and access slot $h(k)$, let us consider the expected number of elements examined by the search algorithm, that is, the number of elements in the list $T[h(k)]$ that are checked to see if their keys are equal to k. We shall consider two cases. In the first, the search is unsuccessful: no element in the table has key k. In the second, the search successfully finds an element with key k.

Theorem 12.1
In a hash table in which collisions are resolved by chaining, an unsuccessful search takes time $\Theta(1+\alpha)$, on the average, under the assumption of simple uniform hashing.

Proof Under the assumption of simple uniform hashing, any key k is equally likely to hash to any of the m slots. The average time to search unsuccessfully for a key k is thus the average time to search to the end of one of the m lists. The average length of such a list is the load factor $\alpha = n/m$. Thus, the expected number of elements examined in an unsuccessful search is α, and the total time required (including the time for computing $h(k)$) is $\Theta(1+\alpha)$. ■

Theorem 12.2
In a hash table in which collisions are resolved by chaining, a successful search takes time $\Theta(1+\alpha)$, on the average, under the assumption of simple uniform hashing.

Proof We assume that the key being searched for is equally likely to be any of the n keys stored in the table. We also assume that the CHAINED-HASH-INSERT procedure inserts a new element at the end of the list instead of the front. (By Exercise 12.2-3, the average successful search time is the same whether new elements are inserted at the front of the list or at the end.) The expected number of elements examined during a successful search is 1 more than the number of elements examined when the sought-for element was inserted (since every new element goes at the end of the list). To find the expected number of elements examined, we therefore take the average, over the n items in the table, of 1 plus the expected length of the list to which the ith element is added. The expected length of that list is $(i-1)/m$, and so the expected number of elements examined in a successful search is

$$\begin{aligned}\frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{i-1}{m}\right) &= 1+\frac{1}{nm}\sum_{i=1}^{n}(i-1)\\ &= 1+\left(\frac{1}{nm}\right)\left(\frac{(n-1)n}{2}\right)\\ &= 1+\frac{\alpha}{2}-\frac{1}{2m}\,.\end{aligned}$$

Thus, the total time required for a successful search (including the time for computing the hash function) is $\Theta(2+\alpha/2-1/2m)=\Theta(1+\alpha)$. ■

What does this analysis mean? If the number of hash-table slots is at least proportional to the number of elements in the table, we have $n = O(m)$ and, consequently, $\alpha = n/m = O(m)/m = O(1)$. Thus, searching takes constant time on average. Since insertion takes $O(1)$ worst-case time (see Exercise 12.2-3), and deletion takes $O(1)$ worst-case time when the lists are doubly linked, all dictionary operations can be supported in $O(1)$ time on average.

Exercises

12.2-1
Suppose we use a random hash function h to hash n distinct keys into an array T of length m. What is the expected number of collisions? More precisely, what is the expected cardinality of $\{(x,y) : h(x) = h(y)\}$?

12.2-2
Demonstrate the insertion of the keys 5, 28, 19, 15, 20, 33, 12, 17, 10 into a hash table with collisions resolved by chaining. Let the table have 9 slots, and let the hash function be $h(k) = k \bmod 9$.

12.2-3
Argue that the expected time for a successful search with chaining is the same whether new elements are inserted at the front or at the end of a list. (*Hint:* Show that the expected successful search time is the same for *any* two orderings of any list.)

12.2-4
Professor Marley hypothesizes that substantial performance gains can be obtained if we modify the chaining scheme so that each list is kept in sorted order. How does the professor's modification affect the running time for successful searches, unsuccessful searches, insertions, and deletions?

12.2-5
Suggest how storage for elements can be allocated and deallocated within the hash table itself by linking all unused slots into a free list. Assume that one slot can store a flag and either one element plus a pointer or two pointers. All dictionary and free-list operations should run in $O(1)$ expected time. Does the free list need to be doubly linked, or does a singly linked free list suffice?

12.2-6
Show that if $|U| > nm$, there is a subset of U of size n consisting of keys that all hash to the same slot, so that the worst-case searching time for hashing with chaining is $\Theta(n)$.

12.3 Hash functions

In this section, we discuss some issues regarding the design of good hash functions and then present three schemes for their creation: hashing by division, hashing by multiplication, and universal hashing.

What makes a good hash function?

A good hash function satisfies (approximately) the assumption of simple uniform hashing: each key is equally likely to hash to any of the m slots. More formally, let us assume that each key is drawn independently from U according to a probability distribution P; that is, $P(k)$ is the probability that k is drawn. Then the assumption of simple uniform hashing is that

$$\sum_{k:h(k)=j} P(k) = \frac{1}{m} \qquad \text{for } j = 0, 1, \ldots, m-1 . \tag{12.1}$$

Unfortunately, it is generally not possible to check this condition, since P is usually unknown.

Sometimes (rarely) we do know the distribution P. For example, suppose the keys are known to be random real numbers k independently and uniformly distributed in the range $0 \le k < 1$. In this case, the hash function

$$h(k) = \lfloor km \rfloor$$

can be shown to satisfy equation (12.1).

In practice, heuristic techniques can be used to create a hash function that is likely to perform well. Qualitative information about P is sometimes useful in this design process. For example, consider a compiler's symbol table, in which the keys are arbitrary character strings representing identifiers in a program. It is common for closely related symbols, such as `pt` and `pts`, to occur in the same program. A good hash function would minimize the chance that such variants hash to the same slot.

A common approach is to derive the hash value in a way that is expected to be independent of any patterns that might exist in the data. For example, the "division method" (discussed further below) computes the hash value as the remainder when the key is divided by a specified prime number. Unless that prime is somehow related to patterns in the probability distribution P, this method gives good results.

Finally, we note that some applications of hash functions might require stronger properties than are provided by simple uniform hashing. For example, we might want keys that are "close" in some sense to yield hash values that are far apart. (This property is especially desirable when we are using linear probing, defined in Section 12.4.)

Interpreting keys as natural numbers

Most hash functions assume that the universe of keys is the set $\mathbf{N} = \{0, 1, 2, \ldots\}$ of natural numbers. Thus, if the keys are not natural numbers, a way must be found to interpret them as natural numbers. For example, a key that is a character string can be interpreted as an integer expressed in suitable radix notation. Thus, the identifier `pt` might be interpreted as the pair of decimal integers $(112, 116)$, since `p` $= 112$ and `t` $= 116$ in the ASCII character set; then, expressed as a radix-128 integer, `pt` becomes $(112 \cdot 128) + 116 = 14452$. It is usually straightforward in any given application to devise some such simple method for interpreting each key as a (possibly large) natural number. In what follows, we shall assume that the keys are natural numbers.

12.3.1 The division method

In the ***division method*** for creating hash functions, we map a key k into one of m slots by taking the remainder of k divided by m. That is, the hash function is

$$h(k) = k \bmod m .$$

For example, if the hash table has size $m = 12$ and the key is $k = 100$, then $h(k) = 4$. Since it requires only a single division operation, hashing by division is quite fast.

When using the division method, we usually avoid certain values of m. For example, m should not be a power of 2, since if $m = 2^p$, then $h(k)$ is just the p lowest-order bits of k. Unless it is known a priori that the probability distribution on keys makes all low-order p-bit patterns equally likely, it is better to make the hash function depend on all the bits of the key. Powers of 10 should be avoided if the application deals with decimal numbers as keys, since then the hash function does not depend on all the decimal digits of k. Finally, it can be shown that when $m = 2^p - 1$ and k is a character string interpreted in radix 2^p, two strings that are identical except for a transposition of two adjacent characters will hash to the same value.

Good values for m are primes not too close to exact powers of 2. For example, suppose we wish to allocate a hash table, with collisions resolved by chaining, to hold roughly $n = 2000$ character strings, where a character has 8 bits. We don't mind examining an average of 3 elements in an unsuccessful search, so we allocate a hash table of size $m = 701$. The number 701 is chosen because it is a prime near $n = 2000/3$ but not near any power of 2. Treating each key k as an integer, our hash function would be

$$h(k) = k \bmod 701 .$$

As a precautionary measure, we could check how evenly this hash function distributes sets of keys among the slots, where the keys are chosen from "real" data.

12.3.2 The multiplication method

The ***multiplication method*** for creating hash functions operates in two steps. First, we multiply the key k by a constant A in the range $0 < A < 1$ and extract the fractional part of kA. Then, we multiply this value by m and take the floor of the result. In short, the hash function is

$$h(k) = \lfloor m (k A \bmod 1) \rfloor ,$$

where "$kA \bmod 1$" means the fractional part of kA, that is, $kA - \lfloor kA \rfloor$.

An advantage of the multiplication method is that the value of m is not critical. We typically choose it to be a power of 2—$m = 2^p$ for some

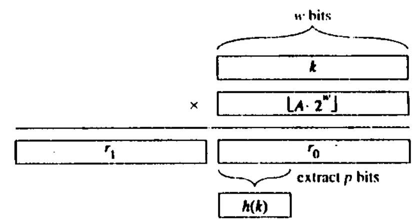


Figure 12.4 The multiplication method of hashing. The w-bit representation of the key k is multiplied by the w-bit value $\lfloor A \cdot 2^w \rfloor$, where $0 < A < 1$ is a suitable constant. The p highest-order bits of the lower w-bit half of the product form the desired hash value $h(k)$.

integer p—since we can then easily implement the function on most computers as follows. Suppose that the word size of the machine is w bits and that k fits into a single word. Referring to Figure 12.4, we first multiply k by the w-bit integer $\lfloor A \cdot 2^w \rfloor$. The result is a $2w$-bit value $r_1 2^w + r_0$, where r_1 is the high-order word of the product and r_0 is the low-order word of the product. The desired p-bit hash value consists of the p most significant bits of r_0.

Although this method works with any value of the constant A, it works better with some values than with others. The optimal choice depends on the characteristics of the data being hashed. Knuth [123] discusses the choice of A in some detail and suggests that

$$A \approx (\sqrt{5} - 1)/2 = 0.6180339887\ldots \qquad (12.2)$$

is likely to work reasonably well.

As an example, if we have $k = 123456$, $m = 10000$, and A as in equation (12.2), then

$$\begin{aligned} h(k) &= \lfloor 10000 \cdot (123456 \cdot 0.61803\ldots \bmod 1) \rfloor \\ &= \lfloor 10000 \cdot (76300.0041151\ldots \bmod 1) \rfloor \\ &= \lfloor 10000 \cdot 0.0041151\ldots \rfloor \\ &= \lfloor 41.151\ldots \rfloor \\ &= 41 . \end{aligned}$$

12.3.3 Universal hashing

If a malicious adversary chooses the keys to be hashed, then he can choose n keys that all hash to the same slot, yielding an average retrieval time of $\Theta(n)$. Any fixed hash function is vulnerable to this sort of worst-case behavior; the only effective way to improve the situation is to choose the hash function *randomly* in a way that is *independent* of the keys that are actually going to be stored. This approach, called ***universal hashing***, yields good performance on the average, no matter what keys are chosen by the adversary.

The main idea behind universal hashing is to select the hash function at random at run time from a carefully designed class of functions. As in the case of quicksort, randomization guarantees that no single input will always evoke worst-case behavior. Because of the randomization, the algorithm can behave differently on each execution, even for the same input. This approach guarantees good average-case performance, no matter what keys are provided as input. Returning to the example of a compiler's symbol table, we find that the programmer's choice of identifiers cannot now cause consistently poor hashing performance. Poor performance occurs only if the compiler chooses a random hash function that causes the set of identifiers to hash poorly, but the probability of this occurring is small and is the same for any set of identifiers of the same size.

Let $\mathcal{H}$ be a finite collection of hash functions that map a given universe U of keys into the range $\{0, 1, \ldots, m-1\}$. Such a collection is said to be ***universal*** if for each pair of distinct keys $x, y \in U$, the number of hash functions $h \in \mathcal{H}$ for which $h(x) = h(y)$ is precisely $|\mathcal{H}|/m$. In other words, with a hash function randomly chosen from $\mathcal{H}$, the chance of a collision between x and y when $x \neq y$ is exactly $1/m$, which is exactly the chance of a collision if $h(x)$ and $h(y)$ are randomly chosen from the set $\{0, 1, \ldots, m-1\}$.

The following theorem shows that a universal class of hash functions gives good average-case behavior.

Theorem 12.3
If h is chosen from a universal collection of hash functions and is used to hash n keys into a table of size m, where $n \leq m$, the expected number of collisions involving a particular key x is less than 1.

Proof For each pair y, z of distinct keys, let c_{yz} be a random variable that is 1 if $h(y) = h(z)$ (i.e., if y and z collide using h) and 0 otherwise. Since, by definition, a single pair of keys collides with probability $1/m$, we have

$$\mathrm{E}[c_{yz}] = 1/m .$$

Let C_x be the total number of collisions involving key x in a hash table T of size m containing n keys. Equation (6.24) gives

$$\mathrm{E}[C_x] = \sum_{\substack{y \in T \\ y \neq x}} \mathrm{E}[c_{xy}] = \frac{n-1}{m} .$$

Since $n \leq m$, we have $\mathrm{E}[C_x] < 1$. ■

But how easy is it to design a universal class of hash functions? It is quite easy, as a little number theory will help us prove. Let us choose our table size m to be prime (as in the division method). We decompose a key x into $r+1$ bytes (i.e., characters, or fixed-width binary substrings), so that $x = \langle x_0, x_1, \ldots, x_r \rangle$; the only requirement is that the maximum value of a byte should be less than m. Let $a = \langle a_0, a_1, \ldots, a_r \rangle$ denote a sequence of $r+1$ elements chosen randomly from the set $\{0, 1, \ldots, m-1\}$. We define a corresponding hash function $h_a \in \mathcal{H}$:

$$h_a(x) = \sum_{i=0}^{r} a_i x_i \bmod m . \qquad (12.3)$$

With this definition,

$$\mathcal{H} = \bigcup_a \{h_a\} \qquad (12.4)$$

has m^{r+1} members.

Theorem 12.4
The class $\mathcal{H}$ defined by equations (12.3) and (12.4) is a universal class of hash functions.

Proof Consider any pair of distinct keys x, y. Assume that $x_0 \neq y_0$. (A similar argument can be made for a difference in any other byte position.) For any fixed values of $a_1, a_2, \ldots, a_r$, there is exactly one value of a_0 that satisfies the equation $h(x) = h(y)$; this a_0 is the solution to

$$a_0(x_0 - y_0) \equiv -\sum_{i=1}^{r} a_i(x_i - y_i) \pmod{m} .$$

To see this property, note that since m is prime, the nonzero quantity $x_0 - y_0$ has a multiplicative inverse modulo m, and thus there is a unique solution for a_0 modulo m. (See Section 33.4.) Therefore, each pair of keys x and y collides for exactly m^r values of a, since they collide exactly once for each possible value of $\langle a_1, a_2, \ldots, a_r \rangle$ (i.e., for the unique value of a_0 noted above). Since there are m^{r+1} possible values for the sequence a, keys x and y collide with probability exactly $m^r/m^{r+1} = 1/m$. Therefore, $\mathcal{H}$ is universal. ■

Exercises

12.3-1
Suppose we wish to search a linked list of length n, where each element contains a key k along with a hash value $h(k)$. Each key is a long character

string. How might we take advantage of the hash values when searching the list for an element with a given key?

12.3-2
Suppose a string of r characters is hashed into m slots by treating it as a radix-128 number and then using the division method. The number m is easily represented as a 32-bit computer word, but the string of r characters, treated as a radix-128 number, takes many words. How can we apply the division method to compute the hash value of the character string without using more than a constant number of words of storage outside the string itself?

12.3-3
Consider a version of the division method in which $h(k) = k \bmod m$, where $m = 2^p - 1$ and k is a character string interpreted in radix 2^p. Show that if string x can be derived from string y by permuting its characters, then x and y hash to the same value. Give an example of an application in which this property would be undesirable in a hash function.

12.3-4
Consider a hash table of size $m = 1000$ and the hash function $h(k) = \lfloor m(kA \bmod 1)\rfloor$ for $A = (\sqrt{5} - 1)/2$. Compute the locations to which the keys 61, 62, 63, 64, and 65 are mapped.

12.3-5
Show that if we restrict each component a_i of a in equation (12.3) to be nonzero, then the set $\mathcal{H} = \{h_a\}$ as defined in equation (12.4) is not universal. (*Hint:* Consider the keys $x = 0$ and $y = 1$.)

12.4 Open addressing

In ***open addressing***, all elements are stored in the hash table itself. That is, each table entry contains either an element of the dynamic set or NIL. When searching for an element, we systematically examine table slots until the desired element is found or it is clear that the element is not in the table. There are no lists and no elements stored outside the table, as there are in chaining. Thus, in open addressing, the hash table can "fill up" so that no further insertions can be made; the load factor α can never exceed 1.

Of course, we could store the linked lists for chaining inside the hash table, in the otherwise unused hash-table slots (see Exercise 12.2-5), but the advantage of open addressing is that it avoids pointers altogether. Instead of following pointers, we *compute* the sequence of slots to be examined. The extra memory freed by not storing pointers provides the hash table with a larger number of slots for the same amount of memory, potentially yielding fewer collisions and faster retrieval.

To perform insertion using open addressing, we successively examine, or ***probe***, the hash table until we find an empty slot in which to put the key. Instead of being fixed in the order $0, 1, \ldots, m-1$ (which requires $\Theta(n)$ search time), the sequence of positions probed *depends upon the key being inserted*. To determine which slots to probe, we extend the hash function to include the probe number (starting from 0) as a second input. Thus, the hash function becomes

$$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\} .$$

With open addressing, we require that for every key k, the ***probe sequence***

$$\langle h(k,0), h(k,1), \ldots, h(k, m-1)\rangle$$

be a permutation of $\langle 0, 1, \ldots, m-1\rangle$, so that every hash-table position is eventually considered as a slot for a new key as the table fills up. In the following pseudocode, we assume that the elements in the hash table T are keys with no satellite information; the key k is identical to the element containing key k. Each slot contains either a key or NIL (if the slot is empty).

```
HASH-INSERT(T, k)
i ← 0
repeat j ← h(k, i)
       if T[j] = NIL
          then T[j] ← k
               return j
          else i ← i + 1
  until i = m
error "hash table overflow"
```

The algorithm for searching for key k probes the same sequence of slots that the insertion algorithm examined when key k was inserted. Therefore, the search can terminate (unsuccessfully) when it finds an empty slot, since k would have been inserted there and not later in its probe sequence. (Note that this argument assumes that keys are not deleted from the hash table.) The procedure HASH-SEARCH takes as input a hash table T and a key k, returning j if slot j is found to contain key k, or NIL if key k is not present in table T.

```
HASH-SEARCH(T, k)
i ← 0
repeat j ← h(k, i)
       if T[j] = k
          then return j
       i ← i + 1
  until T[j] = NIL or i = m
return NIL
```

Deletion from an open-address hash table is difficult. When we delete a key from slot i, we cannot simply mark that slot as empty by storing NIL in it. Doing so might make it impossible to retrieve any key k during whose insertion we had probed slot i and found it occupied. One solution is to mark the slot by storing in it the special value DELETED instead of NIL. We would then modify the procedure HASH-SEARCH so that it keeps on looking when it sees the value DELETED, while HASH-INSERT would treat such a slot as if it were empty so that a new key can be inserted. When we do this, though, the search times are no longer dependent on the load factor α, and for this reason chaining is more commonly selected as a collision resolution technique when keys must be deleted.

In our analysis, we make the assumption of ***uniform hashing***: we assume that each key considered is equally likely to have any of the $m!$ permutations of $\{0, 1, \ldots, m-1\}$ as its probe sequence. Uniform hashing generalizes the notion of simple uniform hashing defined earlier to the situation in which the hash function produces not just a single number, but a whole probe sequence. True uniform hashing is difficult to implement, however, and in practice suitable approximations (such as double hashing, defined below) are used.

Three techniques are commonly used to compute the probe sequences required for open addressing: linear probing, quadratic probing, and double hashing. These techniques all guarantee that $\langle h(k,1), h(k,2), \ldots, h(k,m)\rangle$ is a permutation of $\langle 0, 1, \ldots, m-1\rangle$ for each key k. None of these techniques fulfills the assumption of uniform hashing, however, since none of them is capable of generating more than m^2 different probe sequences (instead of the $m!$ that uniform hashing requires). Double hashing has the greatest number of probe sequences and, as one might expect, seems to give the best results.

Linear probing

Given an ordinary hash function $h' : U \rightarrow \{0, 1, \ldots, m-1\}$, the method of ***linear probing*** uses the hash function

$$h(k, i) = (h'(k) + i) \bmod m$$

for $i = 0, 1, \ldots, m-1$. Given key k, the first slot probed is $T[h'(k)]$. We next probe slot $T[h'(k)+1]$, and so on up to slot $T[m-1]$. Then we wrap around to slots $T[0], T[1], \ldots$, until we finally probe slot $T[h'(k)-1]$. Since the initial probe position determines the entire probe sequence, only m distinct probe sequences are used with linear probing.

Linear probing is easy to implement, but it suffers from a problem known as ***primary clustering***. Long runs of occupied slots build up, increasing the average search time. For example, if we have $n = m/2$ keys in the table, where every even-indexed slot is occupied and every odd-indexed slot is empty, then the average unsuccessful search takes 1.5 probes. If the first $n = m/2$ locations are the ones occupied, however, the average number of probes increases to about $n/4 = m/8$. Clusters are likely to arise, since if an empty slot is preceded by i full slots, then the probability that the empty slot is the next one filled is $(i+1)/m$, compared with a probability of $1/m$ if the preceding slot was empty. Thus, runs of occupied slots tend to get longer, and linear probing is not a very good approximation to uniform hashing.

Quadratic probing

Quadratic probing uses a hash function of the form

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m \,, \tag{12.5}$$

where (as in linear probing) h' is an auxiliary hash function, c_1 and $c_2 \neq 0$ are auxiliary constants, and $i = 0, 1, \ldots, m-1$. The initial position probed is $T[h'(k)]$; later positions probed are offset by amounts that depend in a quadratic manner on the probe number i. This method works much better than linear probing, but to make full use of the hash table, the values of c_1, c_2, and m are constrained. Problem 12-4 shows one way to select these parameters. Also, if two keys have the same initial probe position, then their probe sequences are the same, since $h(k_1, 0) = h(k_2, 0)$ implies $h(k_1, i) = h(k_2, i)$. This leads to a milder form of clustering, called ***secondary clustering***. As in linear probing, the initial probe determines the entire sequence, so only m distinct probe sequences are used.

Double hashing

Double hashing is one of the best methods available for open addressing because the permutations produced have many of the characteristics of randomly chosen permutations. ***Double hashing*** uses a hash function of the form

$$h(k, i) = (h_1(k) + i h_2(k)) \bmod m \,,$$

where h_1 and h_2 are auxiliary hash functions. The initial position probed is $T[h_1(k)]$; successive probe positions are offset from previous positions by

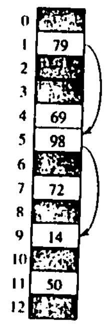


Figure 12.5 Insertion by double hashing. Here we have a hash table of size 13 with $h_1(k) = k \bmod 13$ and $h_2(k) = 1 + (k \bmod 11)$. Since $14 \equiv 1 \bmod 13$ and $14 \equiv 3 \bmod 11$, the key 14 will be inserted into empty slot 9, after slots 1 and 5 have been examined and found to be already occupied.

the amount $h_2(k)$, modulo m. Thus, unlike the case of linear or quadratic probing, the probe sequence here depends in two ways upon the key k, since the initial probe position, the offset, or both, may vary. Figure 12.5 gives an example of insertion by double hashing.

The value $h_2(k)$ must be relatively prime to the hash-table size m for the entire hash table to be searched. Otherwise, if m and $h_2(k)$ have greatest common divisor $d > 1$ for some key k, then a search for key k would examine only $(1/d)$th of the hash table. (See Chapter 33.) A convenient way to ensure this condition is to let m be a power of 2 and to design h_2 so that it always produces an odd number. Another way is to let m be prime and to design h_2 so that it always returns a positive integer less than m. For example, we could choose m prime and let

$$h_1(k) = k \bmod m ,$$

$$h_2(k) = 1 + (k \bmod m') ,$$

where m' is chosen to be slightly less than m (say, $m - 1$ or $m - 2$). For example, if $k = 123456$ and $m = 701$, we have $h_1(k) = 80$ and $h_2(k) = 257$, so the first probe is to position 80, and then every 257th slot (modulo m) is examined until the key is found or every slot is examined.

Double hashing represents an improvement over linear or quadratic probing in that $\Theta(m^2)$ probe sequences are used, rather than $\Theta(m)$, since each possible $(h_1(k), h_2(k))$ pair yields a distinct probe sequence, and as we vary the key, the initial probe position $h_1(k)$ and the offset $h_2(k)$ may vary independently. As a result, the performance of double hashing appears to be very close to the performance of the "ideal" scheme of uniform hashing.

Analysis of open-address hashing

Our analysis of open addressing, like our analysis of chaining, is expressed in terms of the load factor α of the hash table, as n and m go to infinity. Recall that if n elements are stored in a table with m slots, the average number of elements per slot is $\alpha = n/m$. Of course, with open addressing, we have at most one element per slot, and thus $n \le m$, which implies $\alpha \le 1$.

We assume that uniform hashing is used. In this idealized scheme, the probe sequence $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ for each key k is equally likely to be any permutation on $\langle 0, 1, \ldots, m-1\rangle$. That is, each possible probe sequence is equally likely to be used as the probe sequence for an insertion or a search. Of course, a given key has a unique fixed probe sequence associated with it; what is meant here is that, considering the probability distribution on the space of keys and the operation of the hash function on the keys, each possible probe sequence is equally likely.

We now analyze the expected number of probes for hashing with open addressing under the assumption of uniform hashing, beginning with an analysis of the number of probes made in an unsuccessful search.

Theorem 12.5
Given an open-address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$, assuming uniform hashing.

Proof In an unsuccessful search, every probe but the last accesses an occupied slot that does not contain the desired key, and the last slot probed is empty. Let us define

$$p_i = \Pr\{\text{exactly } i \text{ probes access occupied slots}\}$$

for $i = 0, 1, 2, \ldots$. For $i > n$, we have $p_i = 0$, since we can find at most n slots already occupied. Thus, the expected number of probes is

$$1 + \sum_{i=0}^{\infty} i p_i . \tag{12.6}$$

To evaluate equation (12.6), we define

$$q_i = \Pr\{\text{at least } i \text{ probes access occupied slots}\}$$

for $i = 0, 1, 2, \ldots$. We can then use identity (6.28):

$$\sum_{i=0}^{\infty} i p_i = \sum_{i=1}^{\infty} q_i .$$

What is the value of q_i for $i \geq 1$? The probability that the first probe accesses an occupied slot is n/m; thus,

$$q_1 = \frac{n}{m} .$$

With uniform hashing, a second probe, if necessary, is to one of the remaining $m - 1$ unprobed slots, $n - 1$ of which are occupied. We make a second probe only if the first probe accesses an occupied slot; thus,

$$q_2 = \left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right) .$$

In general, the ith probe is made only if the first $i-1$ probes access occupied slots, and the slot probed is equally likely to be any of the remaining $m - i + 1$ slots, $n - i + 1$ of which are occupied. Thus,

$$\begin{aligned} q_i &= \left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right)\cdots\left(\frac{n-i+1}{m-i+1}\right) \\ &\leq \left(\frac{n}{m}\right)^i \\ &= \alpha^i \end{aligned}$$

for $i = 1, 2, \ldots, n$, since $(n-j)/(m-j) \leq n/m$ if $n \leq m$ and $j \geq 0$. After n probes, all n occupied slots have been seen and will not be probed again, and thus $q_i = 0$ for $i > n$.

We are now ready to evaluate equation (12.6). Given the assumption that $\alpha < 1$, the average number of probes in an unsuccessful search is

$$\begin{aligned} 1 + \sum_{i=0}^{\infty} i p_i &= 1 + \sum_{i=1}^{\infty} q_i \\ &\leq 1 + \alpha + \alpha^2 + \alpha^3 + \cdots \qquad (12.7) \\ &= \frac{1}{1-\alpha} . \end{aligned}$$

Equation (12.7) has an intuitive interpretation: one probe is always made, with probability approximately α a second probe is needed, with probability approximately α^2 a third probe is needed, and so on. ■

If α is a constant, Theorem 12.5 predicts that an unsuccessful search runs in $O(1)$ time. For example, if the hash table is half full, the average number of probes in an unsuccessful search is $1/(1 - .5) = 2$. If it is 90 percent full, the average number of probes is $1/(1 - .9) = 10$.

Theorem 12.5 gives us the performance of the HASH-INSERT procedure almost immediately.

Corollary 12.6
Inserting an element into an open-address hash table with load factor α requires at most $1/(1 - \alpha)$ probes on average, assuming uniform hashing.

Proof An element is inserted only if there is room in the table, and thus $\alpha < 1$. Inserting a key requires an unsuccessful search followed by placement of the key in the first empty slot found. Thus, the expected number of probes is $1/(1 - \alpha)$. ■

Computing the expected number of probes for a successful search requires a little more work.

Theorem 12.7
Given an open-address hash table with load factor $\alpha < 1$, the expected number of probes in a successful search is at most

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha} + \frac{1}{\alpha} ,$$

assuming uniform hashing and assuming that each key in the table is equally likely to be searched for.

Proof A search for a key k follows the same probe sequence as was followed when the element with key k was inserted. By Corollary 12.6, if k was the $(i + 1)$st key inserted into the hash table, the expected number of probes made in a search for k is at most $1/(1 - i/m) = m/(m - i)$. Averaging over all n keys in the hash table gives us the average number of probes in a successful search:

$$\begin{aligned} \frac{1}{n}\sum_{i=0}^{n-1} \frac{m}{m-i} &= \frac{m}{n}\sum_{i=0}^{n-1} \frac{1}{m-i} \\ &= \frac{1}{\alpha}(H_m - H_{m-n}) , \end{aligned}$$

where $H_i = \sum_{j=1}^{i} 1/j$ is the ith harmonic number (as defined in equation (3.5)). Using the bounds $\ln i \leq H_i \leq \ln i + 1$ from equations (3.11) and (3.12), we obtain

$$\begin{aligned} \frac{1}{\alpha}(H_m - H_{m-n}) &\leq \frac{1}{\alpha}(\ln m + 1 - \ln(m - n)) \\ &= \frac{1}{\alpha} \ln \frac{m}{m-n} + \frac{1}{\alpha} \\ &= \frac{1}{\alpha} \ln \frac{1}{1-\alpha} + \frac{1}{\alpha} \end{aligned}$$

for a bound on the expected number of probes in a successful search. ■

If the hash table is half full, the expected number of probes is less than 3.387. If the hash table is 90 percent full, the expected number of probes is less than 3.670.

Exercises

12.4-1

Consider inserting the keys 10, 22, 31, 4, 15, 28, 17, 88, 59 into a hash table of length $m = 11$ using open addressing with the primary hash function $h'(k) = k \bmod m$. Illustrate the result of inserting these keys using linear probing, using quadratic probing with $c_1 = 1$ and $c_2 = 3$, and using double hashing with $h_2(k) = 1 + (k \bmod (m-1))$.

12.4-2

Write pseudocode for HASH-DELETE as outlined in the text, and modify HASH-INSERT and HASH-SEARCH to incorporate the special value DELETED.

12.4-3 ★

Suppose that we use double hashing to resolve collisions; that is, we use the hash function $h(k,i) = (h_1(k) + ih_2(k)) \bmod m$. Show that the probe sequence $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ is a permutation of the slot sequence $\langle 0, 1, \ldots, m-1\rangle$ if and only if $h_2(k)$ is relatively prime to m. (*Hint:* See Chapter 33.)

12.4-4

Consider an open-address hash table with uniform hashing and a load factor $\alpha = 1/2$. What is the expected number of probes in an unsuccessful search? What is the expected number of probes in a successful search? Repeat these calculations for the load factors 3/4 and 7/8.

12.4-5 ★

Suppose that we insert n keys into a hash table of size m using open addressing and uniform hashing. Let $p(n,m)$ be the probability that no collisions occur. Show that $p(n,m) \le e^{-n(n-1)/2m}$. (*Hint:* See equation (2.7).) Argue that when n exceeds $\sqrt{m}$, the probability of avoiding collisions goes rapidly to zero.

12.4-6 ★

The bound on the harmonic series can be improved to

$$H_n = \ln n + \gamma + \frac{\epsilon}{2n}, \qquad (12.8)$$

where $\gamma = 0.5772156649\ldots$ is known as ***Euler's constant*** and ϵ satisfies $0 < \epsilon < 1$. (See Knuth [121] for a derivation.) How does this improved approximation for the harmonic series affect the statement and proof of Theorem 12.7?

12.4-7 ★

Consider an open-address hash table with a load factor α. Find the nonzero value α for which the expected number of probes in an unsuccessful search equals twice the expected number of probes in a successful search. Use the estimate $(1/\alpha)\ln(1/(1-\alpha))$ for the number of probes required for a successful search.

Problems

12-1 Longest-probe bound for hashing

A hash table of size m is used to store n items, with $n \le m/2$. Open addressing is used for collision resolution.

a. Assuming uniform hashing, show that for $i = 1, 2, \ldots, n$, the probability that the ith insertion requires strictly more than k probes is at most 2^{-k}.

b. Show that for $i = 1, 2, \ldots, n$, the probability that the ith insertion requires more than $2\lg n$ probes is at most $1/n^2$.

Let the random variable X_i denote the number of probes required by the ith insertion. You have shown in part (b) that $\Pr\{X_i > 2\lg n\} \le 1/n^2$. Let the random variable $X = \max_{1 \le i \le n} X_i$ denote the maximum number of probes required by any of the n insertions.

c. Show that $\Pr\{X > 2\lg n\} \le 1/n$.

d. Show that the expected length of the longest probe sequence is $\mathrm{E}[X] = O(\lg n)$.

12-2 Searching a static set

You are asked to implement a dynamic set of n elements in which the keys are numbers. The set is static (no INSERT or DELETE operations), and the only operation required is SEARCH. You are given an arbitrary amount of time to preprocess the n elements so that SEARCH operations run quickly.

a. Show that SEARCH can be implemented in $O(\lg n)$ worst-case time using no extra storage beyond what is needed to store the elements of the set themselves.

b. Consider implementing the set by open-address hashing on m slots, and assume uniform hashing. What is the minimum amount of extra storage $m - n$ required to make the average performance of an unsuccessful SEARCH operation be at least as good as the bound in part (a)? Your answer should be an asymptotic bound on $m - n$ in terms of n.

12-3 Slot-size bound for chaining

Suppose that we have a hash table with n slots, with collisions resolved by chaining, and suppose that n keys are inserted into the table. Each key is equally likely to be hashed to each slot. Let M be the maximum number of keys in any slot after all the keys have been inserted. Your mission is to prove an $O(\lg n/\lg\lg n)$ upper bound on $\mathrm{E}[M]$, the expected value of M.

a. Argue that the probability Q_k that k keys hash to a particular slot is given by

$$Q_k = \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \binom{n}{k} .$$

b. Let P_k be the probability that $M = k$, that is, the probability that the slot containing the most keys contains k keys. Show that $P_k \le nQ_k$.

c. Use Stirling's approximation, equation (2.11), to show that $Q_k < e^k/k^k$.

d. Show that there exists a constant $c > 1$ such that $Q_{k_0} < 1/n^3$ for $k_0 = c \lg n / \lg\lg n$. Conclude that $P_{k_0} < 1/n^2$ for $k_0 = c \lg n/\lg\lg n$.

e. Argue that

$$\mathrm{E}[M] \le \Pr\left\{M > \frac{c \lg n}{\lg\lg n}\right\} \cdot n + \Pr\left\{M \le \frac{c\lg n}{\lg\lg n}\right\} \cdot \frac{c \lg n}{\lg\lg n} .$$

Conclude that $\mathrm{E}[M] = O(\lg n/\lg\lg n)$.

12-4 Quadratic probing

Suppose that we are given a key k to search for in a hash table with positions $0, 1, \ldots, m-1$, and suppose that we have a hash function h mapping the key space into the set $\{0, 1, \ldots, m-1\}$. The search scheme is as follows.

1. Compute the value $i \leftarrow h(k)$, and set $j \leftarrow 0$.
2. Probe in position i for the desired key k. If you find it, or if this position is empty, terminate the search.
3. Set $j \leftarrow (j+1) \bmod m$ and $i \leftarrow (i+j) \bmod m$, and return to step 2.

Assume that m is a power of 2.

a. Show that this scheme is an instance of the general "quadratic probing" scheme by exhibiting the appropriate constants c_1 and c_2 for equation (12.5).

b. Prove that this algorithm examines every table position in the worst case.

12-5 k-universal hashing

Let $\mathcal{H} = \{h\}$ be a class of hash functions in which each h maps the universe U of keys to $\{0, 1, \ldots, m-1\}$. We say that $\mathcal{H}$ is ***k-universal*** if, for every fixed sequence of k distinct keys $\langle x_1, x_2, \ldots, x_k\rangle$ and for any h chosen at random from $\mathcal{H}$, the sequence $\langle h(x_1), h(x_2), \ldots, h(x_k)\rangle$ is equally likely to be any of the m^k sequences of length k with elements drawn from $\{0, 1, \ldots, m-1\}$.

a. Show that if $\mathcal{H}$ is 2-universal, then it is universal.

b. Show that the class $\mathcal{H}$ defined in Section 12.3.3 is not 2-universal.

c. Show that if we modify the definition of $\mathcal{H}$ in Section 12.3.3 so that each function also contains a constant term b, that is, if we replace $h(x)$ with

$$h_{a,b}(x) = a \cdot x + b ,$$

then $\mathcal{H}$ is 2-universal.

Chapter notes

Knuth [123] and Gonnet [90] are excellent references for the analysis of hashing algorithms. Knuth credits H. P. Luhn (1953) for inventing hash tables, along with the chaining method for resolving collisions. At about the same time, G. M. Amdahl originated the idea of open addressing.

13 Binary Search Trees

Search trees are data structures that support many dynamic-set operations, including SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and DELETE. Thus, a search tree can be used both as a dictionary and as a priority queue.

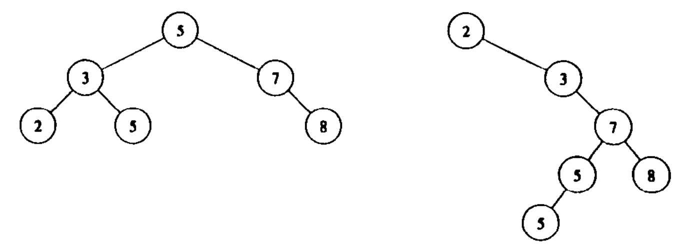
Basic operations on a binary search tree take time proportional to the height of the tree. For a complete binary tree with n nodes, such operations run in $\Theta(\lg n)$ worst-case time. If the tree is a linear chain of n nodes, however, the same operations take $\Theta(n)$ worst-case time. We shall see in Section 13.4 that the height of a randomly built binary search tree is $O(\lg n)$, so that basic dynamic-set operations take $\Theta(\lg n)$ time.

In practice, we can't always guarantee that binary search trees are built randomly, but there are variations of binary search trees whose worst-case performance on basic operations can be guaranteed to be good. Chapter 14 presents one such variation, red-black trees, which have height $O(\lg n)$. Chapter 19 introduces B-trees, which are particularly good for maintaining data bases on random-access, secondary (disk) storage.

After presenting the basic properties of binary search trees, the following sections show how to walk a binary search tree to print its values in sorted order, how to search for a value in a binary search tree, how to find the minimum or maximum element, how to find the predecessor or successor of an element, and how to insert into or delete from a binary search tree. The basic mathematical properties of trees were introduced in Chapter 5.

13.1 What is a binary search tree?

A binary search tree is organized, as the name suggests, in a binary tree, as shown in Figure 13.1. Such a tree can be represented by a linked data structure in which each node is an object. In addition to a *key* field, each node contains fields *left*, *right*, and *p* that point to the nodes corresponding to its left child, its right child, and its parent, respectively. If a child or the parent is missing, the appropriate field contains the value NIL. The root node is the only node in the tree whose parent field is NIL.



(a) (b)

Figure 13.1 Binary search trees. For any node x, the keys in the left subtree of x are at most $key[x]$, and the keys in the right subtree of x are at least $key[x]$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional to the height of the tree. **(a)** A binary search tree on 6 nodes with height 2. **(b)** A less efficient binary search tree with height 4 that contains the same keys.

The keys in a binary search tree are always stored in such a way as to satisfy the ***binary-search-tree property***:

> Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $key[y] \le key[x]$. If y is a node in the right subtree of x, then $key[x] \le key[y]$.

Thus, in Figure 13.1(a), the key of the root is 5, the keys 2, 3, and 5 in its left subtree are no larger than 5, and the keys 7 and 8 in its right subtree are no smaller than 5. The same property holds for every node in the tree. For example, the key 3 in Figure 13.1(a) is no smaller than the key 2 in its left subtree and no larger than the key 5 in its right subtree.

The binary-search-tree property allows us to print out all the keys in a binary search tree in sorted order by a simple recursive algorithm, called an ***inorder tree walk***. This algorithm derives its name from the fact that the key of the root of a subtree is printed between the values in its left subtree and those in its right subtree. (Similarly, a ***preorder tree walk*** prints the root before the values in either subtree, and a ***postorder tree walk*** prints the root after the values in its subtrees.) To use the following procedure to print all the elements in a binary search tree T, we call INORDER-TREE-WALK(*root*[T]).

```
INORDER-TREE-WALK(x)
if x ≠ NIL
   then INORDER-TREE-WALK(left[x])
        print key[x]
        INORDER-TREE-WALK(right[x])
```

LEAVING A BROADCAST CHANNEL

CROSS-REFERENCE TO RELATED APPLICATIONS

This application is related to U.S. Patent Application No. ______________, entitled "BROADCASTING NETWORK," filed on July 31, 2000 (Attorney Docket No. 030048001 US); U.S. Patent Application No.______________, entitled "JOINING A BROADCAST CHANNEL," filed on July 31, 2000 (Attorney Docket No. 030048002 US); U.S. Patent Application No.______________, "LEAVING A BROADCAST CHANNEL," filed on July 31, 2000 (Attorney Docket No. 030048003 US); U.S. Patent Application No.______________, entitled "BROADCASTING ON A BROADCAST CHANNEL," filed on July 31, 2000 (Attorney Docket No. 030048004 US); U.S. Patent Application No.______________, entitled "CONTACTING A BROADCAST CHANNEL," filed on July 31, 2000 (Attorney Docket No. 030048005 US); U.S. Patent Application No.______________, entitled "DISTRIBUTED AUCTION SYSTEM," filed on July 31, 2000 (Attorney Docket No. 030048006 US); U.S. Patent Application No.______________, entitled "AN INFORMATION DELIVERY SERVICE," filed on July 31, 2000 (Attorney Docket No. 030048007 US); U.S. Patent Application No.______________, entitled "DISTRIBUTED CONFERENCING SYSTEM," filed on July 31, 2000 (Attorney Docket No. 030048008 US); and U.S. Patent Application No.______________, entitled "DISTRIBUTED GAME ENVIRONMENT," filed on July 31, 2000 (Attorney Docket No. 030048009 US), the disclosures of which are incorporated herein by reference.

TECHNICAL FIELD

The described technology relates generally to a computer network and more particularly, to a broadcast channel for a subset of a computers of an underlying network.

BACKGROUND

There are a wide variety of computer network communications techniques such as point-to-point network protocols, client/server middleware, multicasting network

protocols, and peer-to-peer middleware. Each of these communications techniques have their advantages and disadvantages, but none is particularly well suited to the simultaneous sharing of information among computers that are widely distributed. For example, collaborative processing applications, such as a network meeting programs, have a need to distribute information in a timely manner to all participants who may be geographically distributed.

The point-to-point network protocols, such as UNIX pipes, TCP/IP, and UDP, allow processes on different computers to communicate via point-to-point connections. The interconnection of all participants using point-to-point connections, while theoretically possible, does not scale well as a number of participants grows. For example, each participating process would need to manage its direct connections to all other participating processes. Programmers, however, find it very difficult to manage single connections, and management of multiple connections is much more complex. In addition, participating processes may be limited to the number of direct connections that they can support. This limits the number of possible participants in the sharing of information.

The client/server middleware systems provide a server that coordinates the communications between the various clients who are sharing the information. The server functions as a central authority for controlling access to shared resources. Examples of client/server middleware systems include remote procedure calls ("RPC"), database servers, and the common object request broker architecture ("CORBA"). Client/server middleware systems are not particularly well suited to sharing of information among many participants. In particular, when a client stores information to be shared at the server, each other client would need to poll the server to determine that new information is being shared. Such polling places a very high overhead on the communications network. Alternatively, each client may register a callback with the server, which the server then invokes when new information is available to be shared. Such a callback technique presents a performance bottleneck because a single server needs to call back to each client whenever new information is to be shared. In addition, the reliability of the entire sharing of information depends upon the reliability of the single server. Thus, a failure at a single computer (*i.e.*, the server) would prevent communications between any of the clients.

The multicasting network protocols allow the sending of broadcast messages to multiple recipients of a network. The current implementations of such multicasting network

protocols tend to place an unacceptable overhead on the underlying network. For example, UDP multicasting would swamp the Internet when trying to locate all possible participants. IP multicasting has other problems that include needing special-purpose infrastructure (*e.g.*, routers) to support the sharing of information efficiently.

The peer-to-peer middleware communications systems rely on a multicasting network protocol or a graph of point-to-point network protocols. Such peer-to-peer middleware is provided by the T.120 Internet standard, which is used in such products as Data Connection's D.C.-share and Microsoft's NetMeeting. These peer-to-peer middleware systems rely upon a user to assemble a point-to-point graph of the connections used for sharing the information. Thus, it is neither suitable nor desirable to use peer-to-peer middleware systems when more than a small number of participants is desired. In addition, the underlying architecture of the T.120 Internet standard is a tree structure, which relies on the root node of the tree for reliability of the entire network. That is, each message must pass through the root node in order to be received by all participants.

It would be desirable to have a reliable communications network that is suitable for the simultaneous sharing of information among a large number of the processes that are widely distributed.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 illustrates a graph that is 4-regular and 4-connected which represents a broadcast channel.

Figure 2 illustrates a graph representing 20 computers connected to a broadcast channel.

Figures 3A and 3B illustrate the process of connecting a new computer Z to the broadcast channel.

Figure 4A illustrates the broadcast channel of Figure 1 with an added computer.

Figure 4B illustrates the broadcast channel of Figure 4A with an added computer.

Figure 4C also illustrates the broadcast channel of Figure 4A with an added computer.

Figure 5A illustrates the disconnecting of a computer from the broadcast channel in a planned manner.

Figure 5B illustrates the disconnecting of a computer from the broadcast channel in an unplanned manner.

Figure 5C illustrates the neighbors with empty ports condition.

Figure 5D illustrates two computers that are not neighbors who now have empty ports.

Figure 5E illustrates the neighbors with empty ports condition in the small regime.

Figure 5F illustrates the situation of Figure 5E when in the large regime.

Figure 6 is a block diagram illustrating components of a computer that is connected to a broadcast channel.

Figure 7 is a block diagram illustrating the sub-components of the broadcaster component in one embodiment.

Figure 8 is a flow diagram illustrating the processing of the connect routine in one embodiment.

Figure 9 is a flow diagram illustrating the processing of the seek portal computer routine in one embodiment.

Figure 10 is a flow diagram illustrating the processing of the contact process routine in one embodiment.

Figure 11 is a flow diagram illustrating the processing of the connect request routine in one embodiment.

Figure 12 is a flow diagram of the processing of the check for external call routine in one embodiment.

Figure 13 is a flow diagram of the processing of the achieve connection routine in one embodiment.

Figure 14 is a flow diagram illustrating the processing of the external dispatcher routine in one embodiment.

Figure 15 is a flow diagram illustrating the processing of the handle seeking connection call routine in one embodiment.

Figure 16 is a flow diagram illustrating processing of the handle connection request call routine in one embodiment.

Figure 17 is a flow diagram illustrating the processing of the add neighbor routine in one embodiment.

Figure 18 is a flow diagram illustrating the processing of the forward connection edge search routine in one embodiment.

Figure 19 is a flow diagram illustrating the processing of the handle edge proposal call routine.

Figure 20 is a flow diagram illustrating the processing of the handle port connection call routine in one embodiment.

Figure 21 is a flow diagram illustrating the processing of the fill hole routine in one embodiment.

Figure 22 is a flow diagram illustrating the processing of the internal dispatcher routine in one embodiment.

Figure 23 is a flow diagram illustrating the processing of the handle broadcast message routine in one embodiment.

Figure 24 is a flow diagram illustrating the processing of the distribute broadcast message routine in one embodiment.

Figure 26 is a flow diagram illustrating the processing of the handle connection port search statement routine in one embodiment.

Figure 27 is a flow diagram illustrating the processing of the court neighbor routine in one embodiment.

Figure 28 is a flow diagram illustrating the processing of the handle connection edge search call routine in one embodiment.

Figure 29 is a flow diagram illustrating the processing of the handle connection edge search response routine in one embodiment.

Figure 30 is a flow diagram illustrating the processing of the broadcast routine in one embodiment.

Figure 31 is a flow diagram illustrating the processing of the acquire message routine in one embodiment.

Figure 32 is a flow diagram illustrating processing of the handle condition check message in one embodiment.

Figure 33 is a flow diagram illustrating processing of the handle condition repair statement routine in one embodiment.

Figure 34 is a flow diagram illustrating the processing of the handle condition double check routine.

DETAILED DESCRIPTION

A broadcast technique in which a broadcast channel overlays a point-to-point communications network is provided. The broadcasting of a message over the broadcast channel is effectively a multicast to those computers of the network that are currently connected to the broadcast channel. In one embodiment, the broadcast technique provides a logical broadcast channel to which host computers through their executing processes can be connected. Each computer that is connected to the broadcast channel can broadcast messages onto and receive messages off of the broadcast channel. Each computer that is connected to the broadcast channel receives all messages that are broadcast while it is connected. The logical broadcast channel is implemented using an underlying network system (*e.g.*, the Internet) that allows each computer connected to the underlying network system to send messages to each other connected computer using each computer's address. Thus, the broadcast technique effectively provides a broadcast channel using an underlying network system that sends messages on a point-to-point basis.

The broadcast technique overlays the underlying network system with a graph of point-to-point connections (*i.e.*, edges) between host computers (*i.e.*, nodes) through which the broadcast channel is implemented. In one embodiment, each computer is connected to four other computers, referred to as neighbors. (Actually, a process executing on a computer is connected to four other processes executing on this or four other computers.) To broadcast a message, the originating computer sends the message to each of its neighbors using its point-to-point connections. Each computer that receives the message then sends the message to its three other neighbors using the point-to-point connections. In this way, the message is propagated to each computer using the underlying network to effect the broadcasting of the message to each computer over a logical broadcast channel. A graph in which each node is connected to four other nodes is referred to as a 4-regular graph. The use of a 4-regular graph means that a computer would become disconnected from the broadcast channel only if all four of the connections to its neighbors fail. The graph used by the broadcast technique also has the property that it would take a failure of four computers to

divide the graph into disjoint sub-graphs, that is two separate broadcast channels. This property is referred to as being 4-connected. Thus, the graph is both 4-regular and 4-connected.

Figure 1 illustrates a graph that is 4-regular and 4-connected which represents the broadcast channel. Each of the nine nodes A-I represents a computer that is connected to the broadcast channel, and each of the edges represents an "edge" connection between two computers of the broadcast channel. The time it takes to broadcast a message to each computer on the broadcast channel depends on the speed of the connections between the computers and the number of connections between the originating computer and each other computer on the broadcast channel. The minimum number of connections that a message would need to traverse between each pair of computers is the "distance" between the computers (*i.e.*, the shortest path between the two nodes of the graph). For example, the distance between computers A and F is one because computer A is directly connected to computer F. The distance between computers A and B is two because there is no direct connection between computers A and B, but computer F is directly connected to computer B. Thus, a message originating at computer A would be sent directly to computer F, and then sent from computer F to computer B. The maximum of the distances between the computers is the "diameter" of broadcast channel. The diameter of the broadcast channel represented by Figure 1 is two. That is, a message sent by any computer would traverse no more than two connections to reach every other computer. Figure 2 illustrates a graph representing 20 computers connected to a broadcast channel. The diameter of this broadcast channel is 4. In particular, the shortest path between computers 1 and 3 contains four connections (1-12, 12-15, 15-18, and 18-3).

The broadcast technique includes (1) the connecting of computers to the broadcast channel (*i.e.*, composing the graph), (2) the broadcasting of messages over the broadcast channel (*i.e.*, broadcasting through the graph), and (3) the disconnecting of computers from the broadcast channel (*i.e.*, decomposing the graph) composing the graph.

Composing the Graph

To connect to the broadcast channel, the computer seeking the connection first locates a computer that is currently fully connected to the broadcast channel and then

establishes a connection with four of the computers that are already connected to the broadcast channel. (This assumes that there are at least four computers already connected to the broadcast channel. When there are fewer than five computers connected, the broadcast channel cannot be a 4-regular graph. In such a case, the broadcast channel is considered to be in a "small regime." The broadcast technique for the small regime is described below in detail. When five or more computers are connected, the broadcast channel is considered to be in the "large regime." This description assumes that the broadcast channel is in the large regime, unless specified otherwise.) Thus, the process of connecting to the broadcast channel includes locating the broadcast channel, identifying the neighbors for the connecting computer, and then connecting to each identified neighbor. Each computer is aware of one or more "portal computers" through which that computer may locate the broadcast channel. A seeking computer locates the broadcast channel by contacting the portal computers until it finds one that is currently fully connected to the broadcast channel. The found portal computer then directs the identifying of four computers (*i.e.*, to be the seeking computer's neighbors) to which the seeking computer is to connect. Each of these four computers then cooperates with the seeking computer to effect the connecting of the seeking computer to the broadcast channel. A computer that has started the process of locating a portal computer, but does not yet have a neighbor, is in the "seeking connection state." A computer that is connected to at least one neighbor, but not yet four neighbors, is in the "partially connected state." A computer that is currently, or has been, previously connected to four neighbors is in the "fully connected state."

Since the broadcast channel is a 4-regular graph, each of the identified computers is already connected to four computers. Thus, some connections between computers need to be broken so that the seeking computer can connect to four computers. In one embodiment, the broadcast technique identifies two pairs of computers that are currently connected to each other. Each of these pairs of computers breaks the connection between them, and then each of the four computers (two from each pair) connects to the seeking computer. Figures 3A and 3B illustrate the process of a new computer Z connecting to the broadcast channel. Figure 3A illustrates the broadcast channel before computer Z is connected. The pairs of computers B and E and computers C and D are the two pairs that are identified as the neighbors for the new computer Z. The connections between each of these pairs is broken, and a connection between computer Z and each of computers B, C, D, and E

is established as indicated by Figure 3B. The process of breaking the connection between two neighbors and reconnecting each of the former neighbors to another computer is referred to as "edge pinning" as the edge between two nodes may be considered to be stretched and pinned to a new node.

Each computer connected to the broadcast channel allocates five communications ports for communicating with other computers. Four of the ports are referred to as "internal" ports because they are the ports through which the messages of the broadcast channels are sent. The connections between internal ports of neighbors are referred to as "internal" connections. Thus, the internal connections of the broadcast channel form the 4-regular and 4-connected graph. The fifth port is referred to as an "external" port because it is used for sending non-broadcast messages between two computers. Neighbors can send non-broadcast messages either through their internal ports of their connection or through their external ports. A seeking computer uses external ports when locating a portal computer.

In one embodiment, the broadcast technique establishes the computer connections using the TCP/IP communications protocol, which is a point-to-point protocol, as the underlying network. The TCP/IP protocol provides for reliable and ordered delivery of messages between computers. The TCP/IP protocol provides each computer with a "port space" that is shared among all the processes that may execute on that computer. The ports are identified by numbers from 0 to 65,535. The first 2056 ports are reserved for specific applications (*e.g.*, port 80 for HTTP messages). The remainder of the ports are user ports that are available to any process. In one embodiment, a set of port numbers can be reserved for use by the computer connected to the broadcast channel. In an alternative embodiment, the port numbers used are dynamically identified by each computer. Each computer dynamically identifies an available port to be used as its call-in port. This call-in port is used to establish connections with the external port and the internal ports. Each computer that is connected to the broadcast channel can receive non-broadcast messages through its external port. A seeking computer tries "dialing" the port numbers of the portal computers until a portal computer "answers," a call on its call-in port. A portal computer answers when it is connected to or attempting to connect to the broadcast channel and its call-in port is dialed. (In this description, a telephone metaphor is used to describe the connections.) When a computer receives a call on its call-in port, it transfers the call to another port. Thus, the

seeking computer actually communicates through that transfer-to port, which is the external port. The call is transferred so that other computers can place calls to that computer via the call-in port. The seeking computer then communicates via that external port to request the portal computer to assist in connecting the seeking computer to the broadcast channel. The seeking computer could identify the call-in port number of a portal computer by successively dialing each port in port number order. As discussed below in detail, the broadcast technique uses a hashing algorithm to select the port number order, which may result in improved performance.

A seeking computer could connect to the broadcast channel by connecting to computers either directly connected to the found portal computer or directly connected to one of its neighbors. A possible problem with such a scheme for identifying the neighbors for the seeking computer is that the diameter of the broadcast channel may increase when each seeking computer uses the same found portal computer and establishes a connection to the broadcast channel directly through that found portal computer. Conceptually, the graph becomes elongated in the direction of where the new nodes are added. Figures 4A-4C illustrate that possible problem. Figure 4A illustrates the broadcast channel of Figure 1 with an added computer. Computer J was connected to the broadcast channel by edge pinning edges C-D and E-H to computer J. The diameter of this broadcast channel is still two. Figure 4B illustrates the broadcast channel of Figure 4A with an added computer. Computer K was connected to the broadcast channel by edge pinning edges E-J and B-C to computer K. The diameter of this broadcast channel is three, because the shortest path from computer G to computer K is through edges G-A, A-E, and E-K. Figure 4C also illustrates the broadcast channel of Figure 4A with an added computer. Computer K was connected to the broadcast channel by edge pinning edges D-G and E-J to computer K. The diameter of this broadcast channel is, however, still two. Thus, the selection of neighbors impacts the diameter of the broadcast channel. To help minimize the diameter, the broadcast technique uses a random selection technique to identify the four neighbors of a computer in the seeking connection state. The random selection technique tends to distribute the connections to new seeking computers throughout the computers of the broadcast channel which may result in smaller overall diameters.

Broadcasting Through the Graph

As described above, each computer that is connected to the broadcast channel can broadcast messages onto the broadcast channel and does receive all messages that are broadcast on the broadcast channel. The computer that originates a message to be broadcast sends that message to each of its four neighbors using the internal connections. When a computer receives a broadcast message from a neighbor, it sends the message to its three other neighbors. Each computer on the broadcast channel, except the originating computer, will thus receive a copy of each broadcast message from each of its four neighbors. Each computer, however, only sends the first copy of the message that it receives to its neighbors and disregards subsequently received copies. Thus, the total number of copies of a message that is sent between the computers is 3N+1, where N is the number of computers connected to the broadcast channel. Each computer sends three copies of the message, except for the originating computer, which sends four copies of the message.

The redundancy of the message sending helps to ensure the overall reliability of the broadcast channel. Since each computer has four connections to the broadcast channel, if one computer fails during the broadcast of a message, its neighbors have three other connections through which they will receive copies of the broadcast message. Also, if the internal connection between two computers is slow, each computer has three other connections through which it may receive a copy of each message sooner.

Each computer that originates a message numbers its own messages sequentially. Because of the dynamic nature of the broadcast channel and because there are many possible connection paths between computers, the messages may be received out of order. For example, the distance between an originating computer and a certain receiving computer may be four. After sending the first message, the originating computer and receiving computer may become neighbors and thus the distance between them changes to one. The first message may have to travel a distance of four to reach the receiving computer. The second message only has to travel a distance of one. Thus, it is possible for the second message to reach the receiving computer before the first message.

When the broadcast channel is in a steady state (*i.e.*, no computers connecting or disconnecting from the broadcast channel), out-of-order messages are not a problem because each computer will eventually receive both messages and can queue messages until all earlier ordered messages are received. If, however, the broadcast channel is not in a

steady state, then problems can occur. In particular, a computer may connect to the broadcast channel after the second message has already been received and forwarded on by its new neighbors. When a new neighbor eventually receives the first message, it sends the message to the newly connected computer. Thus, the newly connected computer will receive the first message, but will not receive the second message. If the newly connected computer needs to process the messages in order, it would wait indefinitely for the second message.

One solution to this problem is to have each computer queue all the messages that it receives until it can send them in their proper order to its neighbors. This solution, however, may tend to slow down the propagation of messages through the computers of the broadcast channel. Another solution that may have less impact on the propagation speed is to queue messages only at computers who are neighbors of the newly connected computers. Each already connected neighbor would forward messages as it receives them to its other neighbors who are not newly connected, but not to the newly connected neighbor. The already connected neighbor would only forward messages from each originating computer to the newly connected computer when it can ensure that no gaps in the messages from that originating computer will occur. In one embodiment, the already connected neighbor may track the highest sequence number of the messages already received and forwarded on from each originating computer. The already connected computer will send only higher numbered messages from the originating computers to the newly connected computer. Once all lower numbered messages have been received from all originating computers, then the already connected computer can treat the newly connected computer as its other neighbors and simply forward each message as it is received. In another embodiment, each computer may queue messages and only forwards to the newly connected computer those messages as the gaps are filled in. For example, a computer might receive messages 4 and 5 and then receive message 3. In such a case, the already connected computer would forward queue messages 4 and 5. When message 3 is finally received, the already connected computer will send messages 3, 4, and 5 to the newly connected computer. If messages 4 and 5 were sent to the newly connected computer before message 3, then the newly connected computer would process messages 4 and 5 and disregard message 3. Because the already connected computer queues messages 4 and 5, the newly connected computer will be able to process message 3. It is possible that a newly connected computer will receive a set of messages from an originating computer through one neighbor and then receive another set of message from the

same originating computer through another neighbor. If the second set of messages contains a message that is ordered earlier than the messages of the first set received, then the newly connected computer may ignore that earlier ordered message if the computer already processed those later ordered messages.

Decomposing the Graph

A connected computer disconnects from the broadcast channel either in a planned or unplanned manner. When a computer disconnects in a planned manner, it sends a disconnect message to each of its four neighbors. The disconnect message includes a list that identifies the four neighbors of the disconnecting computer. When a neighbor receives the disconnect message, it tries to connect to one of the computers on the list. In one embodiment, the first computer in the list will try to connect to the second computer in the list, and the third computer in the list will try to connect to the fourth computer in the list. If a computer cannot connect (*e.g.*, the first and second computers are already connected), then the computers may try connecting in various other combinations. If connections cannot be established, each computer broadcasts a message that it needs to establish a connection with another computer. When a computer with an available internal port receives the message, it can then establish a connection with the computer that broadcast the message. Figures 5A-5D illustrate the disconnecting of a computer from the broadcast channel. Figure 5A illustrates the disconnecting of a computer from the broadcast channel in a planned manner. When computer H decides to disconnect, it sends its list of neighbors to each of its neighbors (computers A, E, F and I) and then disconnects from each of its neighbors. When computers A and I receive the message they establish a connection between them as indicated by the dashed line, and similarly for computers E and F.

When a computer disconnects in an unplanned manner, such as resulting from a power failure, the neighbors connected to the disconnected computer recognize the disconnection when each attempts to send its next message to the now disconnected computer. Each former neighbor of the disconnected computer recognizes that it is short one connection (*i.e.*, it has a hole or empty port). When a connected computer detects that one of its neighbors is now disconnected, it broadcasts a port connection request on the broadcast channel, which indicates that it has one internal port that needs a connection. The port connection request identifies the call-in port of the requesting computer. When a connected

computer that is also short a connection receives the connection request, it communicates with the requesting computer through its external port to establish a connection between the two computers. Figure 5B illustrates the disconnecting of a computer from the broadcast channel in an unplanned manner. In this illustration, computer H has disconnected in an unplanned manner. When each of its neighbors, computers A, E, F, and I, recognizes the disconnection, each neighbor broadcasts a port connection request indicating that it needs to fill an empty port. As shown by the dashed lines, computers F and I and computers A and E respond to each other's requests and establish a connection.

It is possible that a planned or unplanned disconnection may result in two neighbors each having an empty internal port. In such a case, since they are neighbors, they are already connected and cannot fill their empty ports by connecting to each other. Such a condition is referred to as the "neighbors with empty ports" condition. Each neighbor broadcasts a port connection request when it detects that it has an empty port as described above. When a neighbor receives the port connection request from the other neighbor, it will recognize the condition that its neighbor also has an empty port. Such a condition may also occur when the broadcast channel is in the small regime. The condition can only be corrected when in the large regime. When in the small regime, each computer will have less than four neighbors. To detect this condition in the large regime, which would be a problem if not repaired, the first neighbor to receive the port connection request recognizes the condition and sends a condition check message to the other neighbor. The condition check message includes a list of the neighbors of the sending computer. When the receiving computer receives the list, it compares the list to its own list of neighbors. If the lists are different, then this condition has occurred in the large regime and repair is needed. To repair this condition, the receiving computer will send a condition repair request to one of the neighbors of the sending computer which is not already a neighbor of the receiving computer. When the computer receives the condition repair request, it disconnects from one of its neighbors (other than the neighbor that is involved with the condition) and connects to the computer that sent the condition repair request. Thus, one of the original neighbors involved in the condition will have had a port filled. However, two computers are still in need of a connection, the other original neighbor and the computer that is now disconnected from the computer that received the condition repair request. Those two computers send out port connection requests. If those two computers are not neighbors, then they will connect to

each other when they receive the requests. If, however, the two computers are neighbors, then they repeat the condition repair process until two non-neighbors are in need of connections.

It is possible that the two original neighbors with the condition may have the same set of neighbors. When the neighbor that receives the condition check message determines that the sets of neighbors are the same, it sends a condition double check message to one of its neighbors other than the neighbor who also has the condition. When the computer receives the condition double check message, it determines whether it has the same set of neighbors as the sending computer. If so, the broadcast channel is in the small regime and the condition is not a problem. If the set of neighbors are different, then the computer that received the condition double check message sends a condition check message to the original neighbors with the condition. The computer that receives that condition check message directs one of it neighbors to connect to one of the original neighbors with the condition by sending a condition repair message. Thus, one of the original neighbors with the condition will have its port filled.

Figure 5C illustrates the neighbors with empty ports condition. In this illustration, computer H disconnected in an unplanned manner, but computers F and I responded to the port connection request of the other and are now connected together. The other former neighbors of computer H, computers A and E, are already neighbors, which gives rise to the neighbors with empty ports condition. In this example, computer E received the port connection request from computer A, recognized the possible condition, and sent (since they are neighbors via the internal connection) a condition check message with a list of its neighbors to computer A. When computer A received the list, it recognized that computer E has a different set of neighbor (*i.e.*, the broadcast channel is in the large regime). Computer A selected computer D, which is a neighbor of computer E and sent it a condition repair request. When computer D received the condition repair request, it disconnected from one of its neighbors (other than computer E), which is computer G in this example. Computer D then connected to computer A. Figure 5D illustrates two computers that are not neighbors who now have empty ports. Computers E and G now have empty ports and are not currently neighbors. Therefore, computers E and G can connect to each other.

Figures 5E and 5F further illustrate the neighbors with empty ports condition. Figure 5E illustrates the neighbors with empty ports condition in the small regime. In this

example, if computer E disconnected in an unplanned manner, then each computer broadcasts a port connection request when it detects the disconnect. When computer A receives the port connection request form computer B, it detects the neighbors with empty ports condition and sends a condition check message to computer B. Computer B recognizes that it has the same set of neighbors (computer C and D) as computer A and then sends a condition double check message to computer C. Computer C recognizes that the broadcast channel is in the small regime because is also has the same set of neighbors as computers A and B, computer C may then broadcast a message indicating that the broadcast channel is in the small regime.

Figure 5F illustrates the situation of Figure 5E when in the large regime. As discussed above, computer C receives the condition double check message from computer B. In this case, computer C recognizes that the broadcast channel is in the large regime because it has a set of neighbors that is different from computer B. The edges extending up from computer C and D indicate connections to other computers. Computer C then sends a condition check message to computer B. When computer B receives the condition check message, it sends a condition repair message to one of the neighbors of computer C. The computer that receives the condition repair message disconnects from one of its neighbors, other than computer C, and tries to connect to computer B and the neighbor from which it disconnected tries to connect to computer A.

Port Selection

As described above, the TCP/IP protocol designates ports above number 2056 as user ports. The broadcast technique uses five user port numbers on each computer: one external port and four internal ports. Generally, user ports cannot be statically allocated to an application program because other applications programs executing on the same computer may use conflicting port numbers. As a result, in one embodiment, the computers connected to the broadcast channel dynamically allocate their port numbers. Each computer could simply try to locate the lowest number unused port on that computer and use that port as the call-in port. A seeking computer, however, does not know in advance the call-in port number of the portal computers when the port numbers are dynamically allocated. Thus, a seeking computer needs to dial ports of a portal computer starting with the lowest port number when locating the call-in port of a portal computer. If the portal computer is

connected to (or attempting to connect to) the broadcast channel, then the seeking computer would eventually find the call-in port. If the portal computer is not connected, then the seeking computer would eventually dial every user port. In addition, if each application program on a computer tried to allocate low-ordered port numbers, then a portal computer may end up with a high-numbered port for its call-in port because many of the low-ordered port numbers would be used by other application programs. Since the dialing of a port is a relatively slow process, it would take the seeking computer a long time to locate the call-in port of a portal computer. To minimize this time, the broadcast technique uses a port ordering algorithm to identify the port number order that a portal computer should use when finding an available port for its call-in port. In one embodiment, the broadcast technique uses a hashing algorithm to identify the port order. The algorithm preferably distributes the ordering of the port numbers randomly through out the user port number space and only selects each port number once. In addition, every time the algorithm is executed on any computer for a given channel type and channel instance, it generates the same port ordering. As described below, it is possible for a computer to be connected to multiple broadcast channels that are uniquely identified by channel type and channel instance. The algorithm may be "seeded" with channel type and channel instance in order to generate a unique ordering of port numbers for each broadcast channel. Thus, a seeking computer will dial the ports of a portal computer in the same order as the portal computer used when allocating its call-in port.

If many computers are at the same time seeking connection to a broadcast channel through a single portal computer, then the ports of the portal computer may be busy when called by seeking computers. The seeking computers would typically need to keep on redialing a busy port. The process of locating a call-in port may be significantly slowed by such redialing. In one embodiment, each seeking computer may each reorder the first few port numbers generated by the hashing algorithm. For example, each seeking computer could randomly reorder the first eight port numbers generated by the hashing algorithm. The random ordering could also be weighted where the first port number generated by the hashing algorithm would have a 50% chance of being first in the reordering, the second port number would have a 25% chance of being first in the reordering, and so on. Because the seeking computers would use different orderings, the likelihood of finding a busy port is reduced. For example, if the first eight port numbers are randomly selected, then it is

possible that eight seeking computers could be simultaneously dialing ports in different sequences which would reduce the chances of dialing a busy port.

Locating a Portal Computer

Each computer that can connect to the broadcast channel has a list of one or more portal computers through which it can connect to the broadcast channel. In one embodiment, each computer has the same set of portal computers. A seeking computer locates a portal computer that is connected to the broadcast channel by successively dialing the ports of each portal computer in the order specified by an algorithm. A seeking computer could select the first portal computer and then dial all its ports until a call-in port of a computer that is fully connected to the broadcast channel is found. If no call-in port is found, then the seeking computer would select the next portal computer and repeat the process until a portal computer with such a call-in port is found. A problem with such a seeking technique is that all user ports of each portal computer are dialed until a portal computer fully connected to the broadcast channel is found. In an alternate embodiment, the seeking computer selects a port number according to the algorithm and then dials each portal computer at that port number. If no acceptable call-in port to the broadcast channel is found, then the seeking computer selects the next port number and repeats the process. Since the call-in ports are likely allocated at lower-ordered port numbers, the seeking computer first dials the port numbers that are most likely to be call-in ports of the broadcast channel. The seeking computers may have a maximum search depth, that is the number of ports that it will dial when seeking a portal computer that is fully connected. If the seeking computer exhausts its search depth, then either the broadcast channel has not yet been established or, if the seeking computer is also a portal computer, it can then establish the broadcast channel with itself as the first fully connected computer.

When a seeking computer locates a portal computer that is itself not fully connected, the two computers do not connect when they first locate each other because the broadcast channel may already be established and accessible through a higher-ordered port number on another portal computer. If the two seeking computers were to connect to each other, then two disjoint broadcast channels would be formed. Each seeking computer can share its experience in trying to locate a portal computer with the other seeking computer. In particular, if one seeking computer has searched all the portal computers to a depth of eight,

then the one seeking computer can share that it has searched to a depth of eight with another seeking computer. If that other seeking computer has searched to a depth of, for example, only four, it can skip searching through depths five through eight and that other seeking computer can advance its searching to a depth of nine.

In one embodiment, each computer may have a different set of portal computers and a different maximum search depth. In such a situation, it may be possible that two disjoint broadcast channels are formed because a seeking computer cannot locate a fully connected port computer at a higher depth. Similarly, if the set of portal computers are disjoint, then two separate broadcast channels would be formed.

Identifying Neighbors for a Seeking Computer

As described above, the neighbors of a newly connecting computer are preferably selected randomly from the set of currently connected computers. One advantage of the broadcast channel, however, is that no computer has global knowledge of the broadcast channel. Rather, each computer has local knowledge of itself and its neighbors. This limited local knowledge has the advantage that all the connected computers are peers (as far as the broadcasting is concerned) and the failure of any one computer (actually any three computers when in the 4-regular and 4-connect form) will not cause the broadcast channel to fail. This local knowledge makes it difficult for a portal computer to randomly select four neighbors for a seeking computer.

To select the four computers, a portal computer sends an edge connection request message through one of its internal connections that is randomly selected. The receiving computer again sends the edge connection request message through one of its internal connections that is randomly selected. This sending of the message corresponds to a random walk through the graph that represents the broadcast channel. Eventually, a receiving computer will decide that the message has traveled far enough to represent a randomly selected computer. That receiving computer will offer the internal connection upon which it received the edge connection request message to the seeking computer for edge pinning. Of course, if either of the computers at the end of the offered internal connection are already neighbors of the seeking computer, then the seeking computer cannot connect through that internal connection. The computer that decided that the message has

traveled far enough will detect this condition of already being a neighbor and send the message to a randomly selected neighbor.

In one embodiment, the distance that the edge connection request message travels is established by the portal computer to be approximately twice the estimated diameter of the broadcast channel. The message includes an indication of the distance that it is to travel. Each receiving computer decrements that distance to travel before sending the message on. The computer that receives a message with a distance to travel that is zero is considered to be the randomly selected computer. If that randomly selected computer cannot connect to the seeking computer (*e.g.*, because it is already connected to it), then that randomly selected computer forwards the edge connection request to one of its neighbors with a new distance to travel. In one embodiment, the forwarding computer toggles the new distance to travel between zero and one to help prevent two computers from sending the message back and forth between each other.

Because of the local nature of the information maintained by each computer connected to the broadcast channel, the computers need not generally be aware of the diameter of the broadcast channel. In one embodiment, each message sent through the broadcast channel has a distance traveled field. Each computer that forwards a message increments the distance traveled field. Each computer also maintains an estimated diameter of the broadcast channel. When a computer receives a message that has traveled a distance that indicates that the estimated diameter is too small, it updates its estimated diameter and broadcasts an estimated diameter message. When a computer receives an estimated diameter message that indicates a diameter that is larger than its own estimated diameter, it updates its own estimated diameter. This estimated diameter is used to establish the distance that an edge connection request message should travel.

External Data Representation

The computers connected to the broadcast channel may internally store their data in different formats. For example, one computer may use 32-bit integers, and another computer may use 64-bit integers. As another example, one computer may use ASCII to represent text and another computer may use Unicode. To allow communications between heterogeneous computers, the messages sent over the broadcast channel may use the XDR ("eXternal Data Representation") format.