NOTICE

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

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

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

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

AVIS

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

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

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

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

ISBN 0-315-59186-2

DISTRIBUTED FAULT TOLERANT ROUTING ALGORITHM FOR AN INTERCONNECTION NETWORK BASED ON BALANCED INCOMPLETE BLOCK DESIGN

Suseela T. Sarasamma

A Thesis

in

The Department

of

Electrical & Computer Engineering

Presented in Partial Fulfillment of the Requirements for the Degree of Master of Engineering at

Concordia University

Montréal, Québec, Canada

May 1990

ABSTRACT

Distributed Fault Tolerant Routing algorithm for an Interconnection Network based on Balanced Incomplete Block Design

Suseela T. Sarasamma

A new class of network model based on the concept of balanced incomplete block design is studied. The model $G(K_{2,2})^i$ possess better modularity as well as order-to-degree ratio among the above family of graphs. Hence we selected this model for further studies. A suitable labelling scheme is defined for the above network. Four types of basic transformations are then presented. Based on these, two different distributed routing algorithms are given for the fault-free environment. The upper bound on the length of the route thus computed is proved to be $2m + 1$ for the network $G(K_{2,2})^{m-1}$. Comparison of the route-length thus computed, with respect to the diameter estimate of $G(K_{2,2})^m$ as well as the shortest path length between a given (source, destination) pair are made. Two types of fault-distributions which allow up to a maximum of one-third of the total number of nodes to be in fault and yet preserving the connectivity of the network are given. A distributed fault-tolerant routing algorithm FT is then presented. It has been proved that the above algorithm will successfully track down an existing route between any pair of healthy nodes if the fault-distribution corresponds to $\mathcal{D}$. We also claim that the algorithm will be successful in finding an existing route in the case of fault-distribution $\mathcal{D}'$. The algorithms were validated by computer simulation.

ACKNOWLEDGEMENT

I am deeply indebted to my advisor Dr. Jaroslav Opatrny for introducing me to the concept of block design and suggesting this particular problem. I am grateful to him for his excellent guidance, encouragement, and the time he spent in carefully reading the manuscripts as well as the generous financial assistance given to me. His in-depth knowledge of the fundumental isues and clear vision of the underlying nature of research has not only helped me in the preparation of this thesis but has also helped me acquire the proper approach for problem solving and has rekindled my interest in scientific research.

I wish to express my sincere grattitude to Dr. P. D. Ziogas, Dr. R. Patel and Dr K. Venkatesh. Special thanks are to my husband Premchand for his constant encouragement and support. Thanks are also to the computer centre staff and to the University for the excellent facilities and services which were of immense help to me. I am grateful to the GOVT. of Canada and Quebec for granting me the visa etc to come, stay, and study in this great country. Finally I wish to thank my parents and friends.

CONTENTS

CHAPTER 1

INTRODUCTION

Today all branches of engineering and a multitude of other disciplines rely firmly on computational support. However, each discipline has important problems that, to be solved, need computers with orders of magnitude greater performance than currently available. For instance, the finite element analysis in structural design, multi dimensional modelling of earth's atmosphere for weather prediction and fluid flow studies in computational fluid dynamics need tremendous computational power. Then there are real time problems such as speech recognition, image processing, computer vision etc. Consequently, the need for very high performance computing is larger than ever and growing.

To date, high-performance computers have owed their speed primarily to advances in circuit and packaging technology. These technologies are subject to physical limits constraining the ultimate speed of a conventional uniprocessor computer. Parallel processing computers executing problem solutions expressed as parallel algorithms translated into parallel machine programs can exceed the single processor speed limit. Problems such as matrix operations. weather prediction and air traffic control have been identified as having great potential for parallel execution.

High-performance computers are broadly classified into two categories: multiprocessors and multicomputers. This classification is based on the degree of coupling between processors. The degree of coupling can be categorized as:

1. Loose coupling.
2. Moderate coupling.
3. Tight coupling.

Loosely coupled systems are characterized by serial lines, relatively low transmission speeds, relatively smaller amounts of interprocessor activity, and a high degree of error checking. Geographically, the processors can be closely located or seperated

by large distance. Moderately coupled systems are characterized by higher levels of intercomputer activity, using either high speed serial lines, parallel data buses or shared disks. The processors are closely located. In tightly coupled systems processors are closely located, share memory for data transfer and program storage, invoke minimal communication protocols and error checking. There is high degree of interprocessor communications.

Multiprocessors are tightly coupled systems and permit all processors to directly share the main memory. Multicomputers on the other hand are either moderately coupled or loosely coupled. Here each processor has its own locally addressable memory, a communication controller capable of routing messages without delaying the processor, and a small number of connections to other nodes. Multi computer architecture has been suggested as ideal for the solution of problems such as finite element analysis, partial differential equations, linear algebra, game tree searches, functional programming etc. The cooperating tasks of a parallel algorithm for solving one of these problems will execute asynchronously on different nodes and communicate via message passing.

Whether tightly coupled or loosely coupled, the peformance of a distributed (parallel) processing system is determined to a great extent by the underlying communication facility. Hence an important component of a distributed system is the system topology which defines the interprocessor communication architecture. The topology of the interprocessor communication architecture, or interconnection network as it is generally known, is the pattern of connections in its structure. The pattern is modeled by a graph in which the nodes or vertices correspond to the switching nodes in the network and the edges correspond to the communication links. Different interconnection networks(IN) are compared graphically because comparison by topology is independent of the hardware. Nodes in the graph of an IN can be numbered and then an IN can be described in terms of the algebraic relations among the nodes. The algebraic model is useful in discussing control and communication routing strategy.

An IN can be classified based on the following three operational characteristics:

1. Timing.
2. Switching.
3. Overall control.

The timing control of an IN can be either synchronous or asynchronous. Synchronous systems are characterized by a central global clock that broadcasts the clock signal to all devices on the IN so that they operate in a lockstep fashion. Asynchronous systems on the other hand support independent operation of the devices without a global clock. An IN transfers data using either circuit switching or packet switching. In circuit switching, once a device is granted a path in the IN it will occupy that path for the duration of the data transfer. In packet switching, the information is broken into small packets that individually compete for a path in the IN. Based on the overall control of the network, an IN may be classified as centralized or decentralized. In centralized control, a global controller receives all requests and transmits the message in the IN. In a decentralized system, requests are handled independently by different devices in the IN. These three operational characteristics with the topology define an IN. For example, the Butterfly parallel processor uses an asynchronous, packet switched, decentralized IN.

The nature of service demanded from the interconnection network by the multiprocessor system is quite different from that of the multicomputer. Hence two family of interconnection networks have evolved, namely, the multiprocessor IN and the multicomputer IN.

Multiprocessor INs

In this system, all data shared by the processors are stored in the shared memory. All interprocessor communications are through the shared memory which is organized as multiple memory modules. There are four basic types of processor to memory module interconnection:

1. Common bus shared memory.
2. Crossbar-switch shared memory.
3. Multibus/multiported shared memory.
4. Multistage Interconnection Networks.

The common bus interconnection is the simplest scheme with a common communication path connecting all of the functional units. Though this is less complex and less expensive, the overall system performance is severely cut down by the limitation on the overall transfer rate in the system due to the limited speed and bandwidth of the single path. The crossbar switch on the other hand allows simultaneous connection between all nonoverlapping processor-memory pairs. The conflicting requests for access to a particular memory module is resolved by special hardware which adds to the complexity and hence, cost of the system. The multibus/multiported memory interconnection uses a dedicated bus(link) to connect each processor memory pair. In other words, the control, switching and priority arbitration logic that is distributed throughout the crossbar switch matrix is concentrated at the interface to the memory units. Although there is potential for very high overall transfer rate in the system, the complex memory units makes this expensive. Multistage interconnection networks(MIN) are built using large number of simple switches arranged in several stages. Different patterns of interconnection between the inputs and outputs are realized through various combinations of the switch settings.

Multicomputer INs

Interunit communications in this system take one of two forms: redundant bus networks in which each computer can communicate with other computers over a shared bus, or redundant point to point communication. Because the nodes do not share any memory, the network in this case must efficiently support message passing. The ring, chordal ring, hypercube, cube connected cycles and the X-tree are some of the existing interconnection structures. The hypercube, also known as boolean n-cube, has received much attention. It is an n-dimensional network. Each of the 2^n vertices of the hypercube is directly connected to its n neighbors. Many of the common interconnection topologies such as the ring, the tree, the mesh etc can be embedded in a hypercube. Although the hypercube possesses some attractive features such as low internode distance characterized by a diameter of n and existence of n node disjoint paths between node pairs resulting in efficiency as well as redundancy, it also has the disadvantage of logarithmically increasing node

connectivity.

Fault-tolerance and reliability issues

Computer systems have found their way into every day life activities such as banking, vehicle traffic control, communication systems, modern day aeroplanes with auto pilot facility etc. As one can imagine, the breakdown of such a system can be costly and dangerous. Thus came the impetus for reliability and fault tolerance in present day computing systems. The most stringent fault tolerance requirements arise in real time control systems in which faulty computations can jeopardize human life or expensive equipment. The delay associated with fault recovery is also very critical in such applications. According to Avizienis [AVIZ76], a fault tolerant system is a system which has the built-in capability (without external assistance) to preserve the continued correct execution of its programs and input output systems in the presence of a certain set of operational faults. An operational fault is an unspecified change in the value of one or more logic variables in the hardware of the system which is an immediate consequence of a physical failure event. The event may be a permanent component failure, a temporary or intermittent component malfunction or externally originating interference with the operation of the system. Correct execution means that the programs, the data and the results do not contain errors and the execution time does not exceed a specified limit.

A partially fault tolerant (gracefully degrading)system is one which has the built-in capability to reduce its specified computing capacity and shrink to a smaller system by discarding some previously used programs or by slowing down below the specified rate of execution [AVIZ76]. The above reduction is due to the decrease in the hardware configuration brought about by operational faults. In general, computer systems with high reliability is achieved by redundancy and/or maintenance techniques. Redundancy can assume three different forms such as hardware, software, or time. Redundant (hardware) computer systems with several processors (or units) has been classified into the following four systems by Beaudry [BEAUD78]:

1. Massive redundant system
2. Standby redundant system

3. Hybrid redundant system
4. Gracefully degrading system

Massive redundant systems use techniques such as triple-modular redundancy, N-modular redundancy and self-purging redundancy. They execute the same task on each equivalent unit and vote on the outputs for improving the output information. Standby redundant systems execute tasks on their active units. Once failure of an active unit is detected, these systems attempt to replace the faulty unit with a spare unit. Hybrid redundant systems are composed of a massive redundant core with spares to replace failed units [LOSQ76]. Gracefully degrading systems may use all failure-free units to execute tasks. When a unit failure is detected, these systems attempt to reconfigure to a system with one fewer units.

There are four essential elements in any fault tolerant system design. They are fault detection, fault containment, fault diagnosis and fault recovery. Hardware as well as software mechanisms are used to determine whether a fault exists in the system. Fault containment refers to the techniques that are used to prevent fault-damaged information from propagating through the system during the time period between the fault occurence and its detection. Hardware and software techniques used in locating and identifying the faults form the fault diagnosis system. Fault recovery is the mechanism which is instrumental in correcting the fault by voting out the incorrect results or replacing the faulty components by spares. These four elements are present both in centralized systems as well as distributed systems. But there are some inherent fault tolerance capabilities present in some of the distributed systems available today.

Reliability and fault tolerance of the interconnection network is viewed as its capability to perform successful message transfer between healthy nodes in the presence of faults in the network. Faults can be due to the processor or switching node failure, or due to the failure of the communication links. The former is denoted as node faults and the later as edge faults. Node faults are more severe than edge faults since there are many edges incident on a node and hence possibly many communication paths will be disrupted. Most fault tolerant systems are designed

to cope with a certain set of faults, which is referred to as the fault model. In a fault model where node faults are considered, provided all nodes are of equal priority, the network is said to be *k-fault tolerant* if the network stays connected even after the failure of any k nodes. If the system stays connected for some instances of k node faults, then it is said to be *robust* rather than *k-fault tolerant* . Considering node faults, the ring and tree structure have poor fault tolerance. The hypercube which has n node disjoint paths is n-fault tolerant.

Routing Algorithms

An efficient means of controlling the message transfer within the network is crucial to any communication network. A routing ρ is a network control function which assigns a path between any pair of specified nodes in the network. The complexity of a routing scheme is decided to a great extent by the structure of the network. Regular structures make the routing process easier compared to highly irregular network topologies. Routing algorithms are broadly classified into two categories: static routing algorithms and dynamic routing algorithms. A static routing assigns to any pair of nodes in the network, a fixed path, and all communications between the two nodes travel along this path. Static routing in a network of size N is achieved by providing a fixed routing table of $O(N)$ at every node in the network. This is a very simple scheme but has the following major drawbacks. The fixed nature of the routing directory doesn't take into consideration, the network conditions such as congestion and possible failures of network components. The congestion of messages at one node has a cumulating effect and may result in the delay of the entire communication network. Further, since there is a fixed route between a source and a destination, even if there exists edge disjoint paths between a source and a destination, there is no efficient mechanism to exploit the inherent fault tolerance capability of the network. In contrast, the dynamic routing scheme allows the nodes to change the routing table.

Dynamic routing schemes fall in to two catagories namely, centralized routing scheme and decentralized routing scheme. In centralized routing scheme, the routing decisions are made by a central node. Although there is great control over the overall

communication within the network, the inability of individual nodes to take their own routing decisions often leads to congestion and delay. The central controller at times, becomes the bottleneck in the efficient operation of the entire network. This is especially the case when there are faults in the system. The reconfiguration process tends to be too slow and inefficient. Furthermore, reliance on a single node for making decisions, makes the network more vulnerable. So the current trend in fault tolerant routing is towards decentralizing or distributing the network control. Further distributed routing schemes do not need elaborate routing tables at individual stations. Hence our interest is mainly in dynamic distributed routing.

Focus of the research

The focus of this research is on network models. From the previous discussions it is quite clear that there is growing demand for highly fault-tolerant interconnection networks which possess multiple node-disjoint paths. Further, as we will see in chapter 2, the *diameter stability*, *i.e.*, the network's ability to maintain the diameter of the surviving network within close range of the original diameter is a very significant factor in the selection of interconnection network.

In chapter 2 we first give some basic definitions and concepts which are relevent in the context of this thesis. We then review the (Δ, D) graph problem and some of the significant research done in solving this extremal problem. The fault-tolerant routing algorithms propsed for the DeBruijn network is briefly scanned. Finally we examine the construction of the recently proposed BIBD networks.

In chapter 3, at first the construction of the networks $(G, n(k), n(k), k, k)$ and $(K_{k,k}, n(k), n(k), k, k)^1$ proposed by Opatrny *et al* is presented. Then we study some of the significant properties such as diameter, order, number of vertex-disjoint paths etc. of the $G(K_{k,k})^1$ graph model. We then show that the $G(K_{2,2})^1$ model is the most interesting of the above family of graphs in terms of modular structure, large order-to-degree ratio etc. A suitable labelling scheme is given for the special class of $G(K_{2,2})^1$ network. We then define four transformation schemes. Based on these transformation schemes we present two efficient dynamic distributed routing algorithms for the above communication network in a fault-free environment. An

estimate of the length of the route thus computed is presented. A comparison of the route-length computed by the above algorithm with respect to the diameter estimate of $G(K_{2,2})^i$ as well as the shortest path length between a given (source, destination) is then made.

In chapter 4, we first give a set of conditions under which the network $G(K_{2,2})^i$ can allow as many as one third the total number of nodes to be faulty and still remain connected. Then we present a distributed fault tolerant routing algorithm for the network $G(K_{2,2})^i$. It is then showed that under the above conditions the algorithm will be successful in finding an existing route between any pair of healthy nodes. Another kind of fault distribution which will also permit as many as one-third the number of vertices to be faulty with out impairing the network's connectivity is then given. We also claim that for this distribution of faults, the algorithm FT will be successful in tracking an existing route between any pair of healthy nodes.

Research contribution of this thesis is organized as a sequence of lemmas and theorems in chapters 3 and 4.

CHAPTER 2

BASIC CONCEPTS AND DEFINITIONS

In this chapter, we give the basic definitions and notations to be used in the rest of the thesis and and some well known and significant concepts which forms the building block for the forthcoming chapters. Section 2.1 gives a brief description of graph theoretic concepts to be used and some basic definitions. Section 2.2 covers the (Δ, D) graph problem and some of the significant research done in solving this extremal problem. In section 2.3, we introduce the concept of balanced incomplete block design and briefly look at the routing scheme proposed for the Shift and Replace Graph. A brief discussion of the network construction based on Balanced Incomplete Block Design is also given.

2.1 GRAPH THEORETIC CONCEPTS

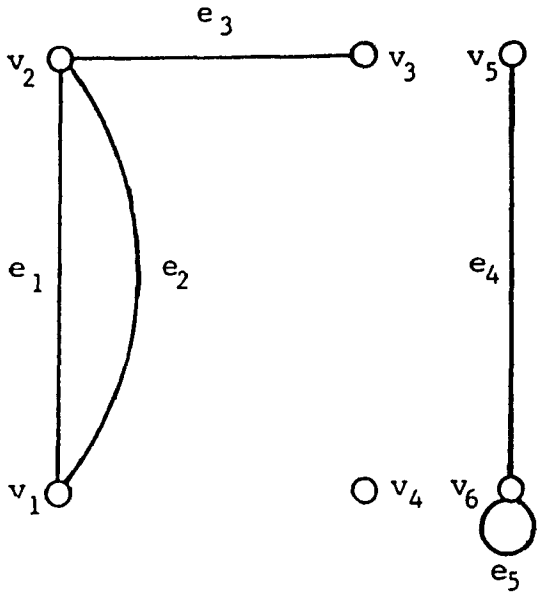
Ever since Euler solved the Königsberg bridge problem, graphs have served as models for problem solving. They form a very useful tool in the design and analysis of communication facilities. In a link-based system of interconnection network, the individual processors (units) are represented by the vertices of a graph and the communication links between processors are represented by the edges of the graph. Similarly graphs can be used to model distributed computations, with nodes representing tasks and edges representing required paths for interprocess communication. We will follow the notations given in [ORE62].

A graph G is denoted as the pair $(V(G), E(G))$, where $V(G)$ denotes the set of nodes in the graph and $E(G)$ denotes the set of edges in the graph G. An edge is a pair of vertices (a, b) between which there exists a logical connection. The nodes a and b are called the end points of the edge (a, b). An edge whose initial vertex and final vertex are the same is called a self loop. If there are several edges joining a pair of vertices, then these edges are said to be parallel edges. A graph

without self loops and parallel edges is termed a simple graph. A vertex is said to be an isolated vertex if there is no edge incident to it. A graph consisting only of isolated vertices is called a null graph. When two sets V_1 and V_2 are given, one can form the set of all pairs $\{(v_1, v_2)|v_1 \in V_1, v_2 \in V_2\}$, known as the product space $V_1 \times V_2$. Thus the graph G with given edges $E(G)$ can be viewed as a subset of the product space $V \times V$.

A graph G is said to be a directed graph if all its edges are directed. An edge (a, b) is said to be a directed edge if (a, b) is an ordered pair of $V \times V$. In this case, vertex a is known as the initial vertex and vertex b is known as the terminal vertex. If ordering is immaterial then edge (a, b) is said to be an undirected edge. Whether directed or undirected, an edge e is said to be incident to its end vertices and the end vertices are said to be incident to the edge. Similarly a graph whose edges are undirected is referred to as an undirected graph. Communication networks with unidirectional links are modelled as directed graphs whereas undirected graphs are used to model communication systems with bidirectional links. An undirected graph in which the edge set $E(G)$ is the set of all possible pairs (a, b) where $a \neq b$ and $(a, b) \in V(G)$ is called a *complete graph*. A complete graph with n nodes is denoted as K_n.

For an undirected graph G, the number of edges incident to a vertex a is termed as its degree, denoted by $\rho(a)$. A self loop incident to a vertex contributes a value of two to its degree. For directed graphs, the number of incoming edges on a node is termed its indegree $\rho^*(a)$ and the number of outgoing edges is termed as its outdegree denoted as $\rho(a)$. In the case of an undirected graph G, the minimal value of $\rho(a)$ for $a \in V(G)$ is called the mindegree $\delta(G)$ and the maximal value of $\rho(a)$ is known as the maxdegree $\Delta(G)$. An undirected graph in which every node is of the same degree is called a *regular graph*. Thus a k-regular graph will have $\rho(a) = k$ for all $a \in V$.



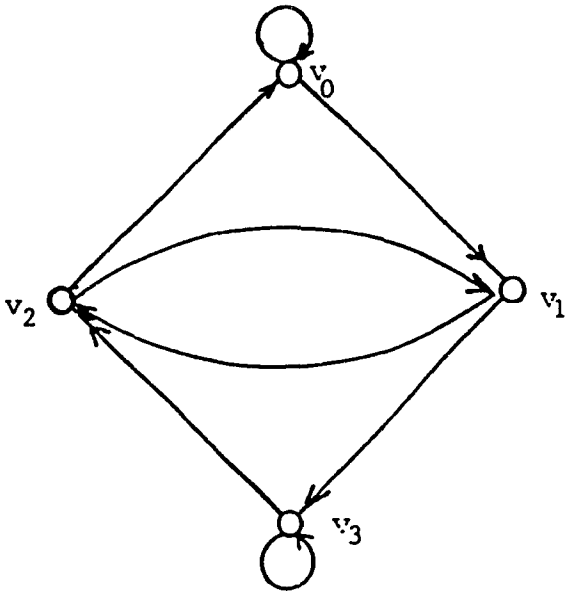


fig 2.1.(a) fig 2.1.(b)

In fig 2.1.(a), $V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$ and $E = \{e_1, e_2, e_3, e_4, e_5\}$. Note that e_1 and e_2 are parallel edges. The edge e_5 is a selfloop. Vertex v_4 is an isolated vertex. v_3 and v_5 are pendant vertices. (i.e., a vertex having only one edge incident to it) The mindegree δ of G in fig 2.1.(a) is 0 and the maxdegree Δ is 3. Fig 2.1.(b) is an example of a directed graph whose minimal in-degree as well as minimal out-degree are 2. Incidentally, the maximal in-degree Δ^* and maximal out-degree Δ are also 2.

A graph H is called a subgraph of the graph G (denoted as $G \supset H$) if the vertex set $V(H)$ of H is contained in the vertex set $V(G)$ of G and all the edges of H are edges in G. A null graph is considered to be a subgraph of every graph. To every subgraph H of G, there exists a unique complementary subgraph $\overline{H}$ consisting of all edges which do not belong to H which is indicated as $\overline{H} = G - H$. A subgraph

H is said to cover G when H has at least one edge incident at every vertex of G. Let $H_1 = (V(H_1), E(H_1))$ and $H_2 = (V(H_2), E(H_2))$ be two subgraphs of G. Then their sum graph $H = (V(H), E(H))$ where $V(H) = V(H_1) \cup V(H_2)$ and $E(H) = E(H_1) \cup E(H_2)$. Analogously, the intersection graph D of H_1 and H_2 is $(V(D), E(D))$ where $V(D) = V(H_1) \cap V(H_2)$ and $E(D) = E(H_1) \cap E(H_2)$. Two subgraphs H_1 and H_2 are said to be vertex disjoint if they have no vertices and hence no edges in common. Similarly two subgraphs are said to be edge disjoint if they have no edges in common. The ring sum of subgraphs H_1 and H_2, denoted as $H_1 \oplus H_2$ is the induced graph H on the edge set $E(H_1) \oplus E(H_2)$.

In an undirected graph G, any series of edges $S = (\ldots, e_0, e_1, \ldots, e_n, \ldots)$ is called a *sequence* if every consecutive edges e_{i-1} and e_i have a vertex in common. That is, $S = (\cdots, e_0 = (a_0, a_1), e_1 = (a_1, a_2), \cdots, e_n = (a_n, a_{n+1}), \cdots)$. If there are no edges preceeding e_0 then e_0 will be called the initial vertex of S. Similarly, if there are no edges after e_n then a_{n+1} will be called the terminal vertex of S. Any vertex a_i which belongs to two consecutive edges e_{i-1} and e_i is an *inner vertex* or *intermediate vertex*. Since vertices and edges may appear repeatedly in a sequence, an inner vertex may also be an initial vertex or terminal vertex or both. A finite sequence with both an initial vertex a_0 and a final vertex a_n can be denoted as $S = (a_0, a_n)$ where a_0 and a_n are called the end points of S. If $a_0 = a_n$, the sequence is *cyclic*. When a_i and a_j are two vertices in the sequence, the subsequence $S(a_i, a_j) = (e_i, e_{i+1}, \ldots, e_{j-1})$ is called a *section* of S.

A sequence in which no edges appear more than once is called a path. Note that a vertex in a path could possibly be traversed several times. A noncyclic path is called a *simple path* or an arc if none of its vertices is traversed more than once. A cyclic path with end points a_0 is called a *circuit* if a_0 appears only as the end points and no other vertex appears more than once. In the case of directed graphs, a directed sequence is a sequence of edges in which all edges are traversed in their prescribed directions. A cyclic directed sequence which traverses every edge of G is termed an Euler path. A graph which possesses an *Euler path* is called an *Euler graph*. For a simple path, the number of edges in its sequence

is termed the *length* of the simple path. The length of a shortest simple path between vertices a and b is called the *distance* $d(a, b)$ between a and b. For a finite graph or graph with bounded distance, *diameter* $D(G)$ is defined as the maximal distance between two of its vertices. That is, $D(G) = \max\ d(a, b)$ for $a, b \in V(G)$. The corresponding shortest arcs connecting two vertices with maximal distance are called diametral arcs.

2.1.1 Connected graphs and connected components

For an undirected graph G, two vertices a and b are said to be connected if there exists an edge sequence with a and b as the two end points. It is then evident that two nodes connected by an edge sequence are also connected by an arc. A graph is said to be connected if every pair of vertices are connected. The vertex set of a graph can be partitioned as $V = \sum_i V_i$ into disjoint subsets such that in each V_i, all vertices are connected while no vertices belonging to two different blocks are connected. That is, corresponding to each set V_i there is a connected subgraph $G(V_i)$. These subgraphs are known as the *connected components* of G.

Let A and B be two disjoint subsets of the vertex set $V(G)$ of an undirected graph G. An arc joining $a \in A$ and $b \in B$ is termed a connecting arc denoted as $P(a, b)$. A set of vertices in V is called a separating set **S** if every connecting path in G must pass through at least one of the vertices in **S**. The subsets A and B are said to be σ-vertex separated when there exists a finite separating set with at least σ number of vertices. By Menger's theorem, there exists σ disjoint simple paths between the σ-vertex separated subsets A and B of G. In other words, if a and b are two nonadjacent nodes in a connected undirected graph G with the smallest number of separating vertices equal to σ then there are σ paths between a and b having only the vertices a and b in common. Graph G is then said to be of node connectivity σ.

With A and B as defined above, a family of edges, $T = \{e_i\}$ form a *separating edge set* for A and B when every connecting path between these sets must pass through at least one edge in T. Clearly an edge $e_i \in T$ do not belong to either the subgraph G_A or G_B. Any subset **S** of vertices containing at least one end point

of $e_i \in T$ is a vertex separating set and any family of edges containing all edges from the vertices in a vertex separation set is an edge separation set. The minimal number of nodes or edges that must be removed from a graph in order to remove all existing paths between any remaining pair of nodes is termed the *connectivity* of the graph [WILK70].

Some special graphs which have been considered in communication networks are the tree, the cycle, completely connected undirected graph K_n, de Bruijn graphs, bipartite graphs, the hypercube etc. A connected undirected graph is called a tree when it has no circuits. In a tree, any two vertices are connected by a unique path. The de Bruijn graph is a directed Euler graph, denoted by $G_{s,n}$. Here s denotes the number of elements in an alphabet **A** and n is a positive integer. The construction of $G_{s,n}$ is as follows:

1. V encompasses all the s^{n-1} words of length $n-1$ over the alphabet **A**.
2. E is the set of all the s^n words of length n over the alphabet **A**.
3. The edge $b_1, b_2, \ldots, b_n$ has $b_1, b_2, \ldots, b_{n-1}$ as its initial vertex and $b_2, b_3, \ldots, b_n$ as its final vertex.
4. At each vertex, there are s incoming edges and s outgoing edges.

Given in figure 2.2 is the de Bruijn graph $G_{2,4}$. The sequence of edges 0000, 0001, 0011, 0111, 1111, 1110, 1101, 1011, 0110, 1100, 1001, 0010, 0101, 1010, 0100, 1000 is a directed Euler trail. The first alphabet of these words, when concatenated yields the de Bruijn sequence 0000111101100101. It is known that this type of graph gives reasonably small value of diameter for a given order $N(\Delta, D)$.

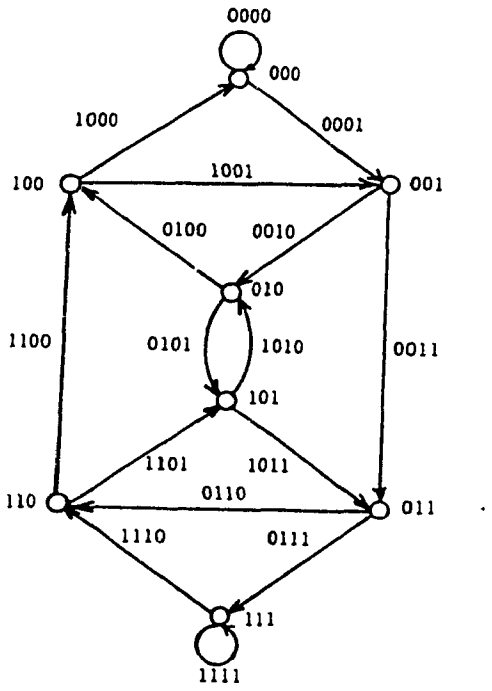


fig. 2.2 De Bruijn graph.

The Shift and Replace graph (SRG) is essentially the de Bruijn graph with the number of nodes, $n = r^m$ and degree $\Delta = r$. The difference lies in the fact that SRG is undirected, has no self loops or parallel edges. This topology was studied by Pradhan and Reddy in [PRAD82]. The diameter of such an SRG is m. With $m = 1$, the resulting graph is a completely connected graph with r nodes and when $r = 1$, it is a single node. Any node i in SRG could be represented by the radix-r representation

$$r(i) = i_{m-1}, i_{m-2}, \ldots, i_1, i_0 \qquad \text{where } 0 \leq i \leq r^m - 1.$$

With such a representation, it can be easily stated that node i is connected to node j if $r_j = (i_{m-2}, i_{m-3}, \ldots, i_1, i_0, y)$ or $r_j = (y, i_{m-1}, i_{m-2}, \ldots, i_1)$ for $y = 0, 1, 2, \ldots, r-1$. The graph thus emulates the state diagram of a shift register with alphabet size r. The next states of the register containing $(i_{m-1}, i_{m-2}, \ldots, i_0)$ can be determined by left or right shifting of the register wherein the least(most)

significant digit is replaced by x for $x = 0, 1, \ldots, r-1$. Thus the graph is promptly named the Shift and Replace Graph. It has been shown in [PRAD82] that the SRG has $n - r^2$ nodes of degree $2r$, r nodes of degree $2r-2$ and $r^2 - r$ nodes of degree $2r-1$. Esfahanian and Hakimi have shown that the connectivity of an SRG with $n = r^m$ nodes is $2r-3$ [ESFA85].

Ease of routing and existence of multiple paths are the two important characteristics of the SRG graphs. With the nodes labelled by the corresponding radix-r representation, the routing algorithm reduces to the task of properly shifting the labels to obtain the internal nodes in the route. Pradhan and Reddy have presented two routing algorithms for the SRG for the following network conditions: 1. No node failures. 2. Number of failed nodes $\leq r$ -- 1. Esfahanian and Hakimi have proved that the system can tolerate up to $2r-3$ processor failures without being disconnected. Thus this topology has the two important and desirable features of an interconnection network such as reliability (due to multiple paths) and simplicity of routing scheme.

A *bipartite* graph $G = (V, V')$ is a graph in which the vertex set $V(G)$ can be partitioned into two disjoint sets V and V' such that each edge $e = (V, V')$ connects a vertex $v \in V$ to another vertex $v' \in V$. A subgraph of a bipartite graph is bipartite. If G is connected then each vertex $v \in V$ has an even distance from vertices in V and an odd distance from vertices in V'. A graph G can be represented as a bipartite graph if and only if all circuits in G has even lengths. If in a simple bipartite graph G with bipartition (V, V'), there is an edge (v_i, v_j) for every vertex $v_i \in V$ and every vertex $v_j \in V'$, then G is called a complete bipartite graph. We will denote a complete bipartite graph as $K_{r,k}$, where $r = |V|$ and $k = |V'|$. Shown in figure 2.3 is a $K_{3,3}$ graph.

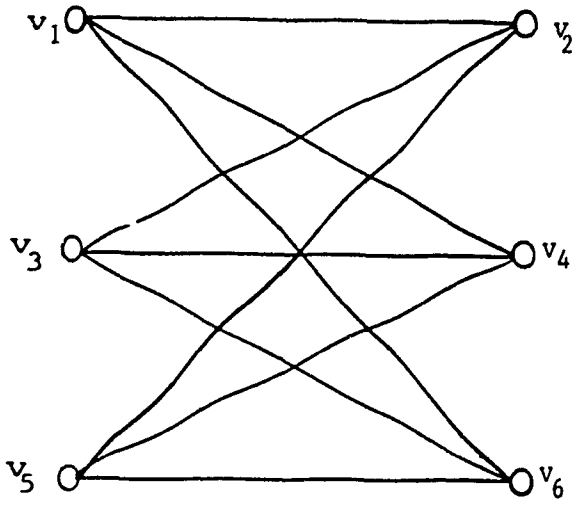


fig. 2.3 The complete bipartite graph $K_{3,3}$

2.2. The (Δ, D) graph problem

It is well known that one of the cost factors of interconnection among processors(computers) is the number of physical lines between them. For a communication network model to be of practical significance, it is essential that the transmission delay be kept a minimum. This delay is closely related to the diameter of the graph representing the network. Furthermore, in a large network, the shortest route between many pairs of nodes will have to pass through several intermediate nodes. However, there is a certain amount of distortion, switching delay and vulnerability associated with each node. In this perspective, it can be said that minimizing the distance between every pair of nodes is of great importance in the design of interconnection networks. Hence, given a number of nodes, interconnecting them with minimal number of links and minimal diameter is a major issue in the design of INs. The converse of this problem, *i.e.*, given a maximum degree Δ and a maximum diameter D, the problem of finding a graph with the maximum number of vertices is known as the (Δ, D) graph problem. The (Δ, D) graph problem was first set by Elspas and it addresses the issue of maximizing the order $N(\Delta, D)$ of a

graph, given Δ and D. Optimizing the order $N(\Delta, D)$ of a graph, given its degree and diameter is considered to be a difficult theoretical problem. A theoretical upper bound on $N(\Delta, D)$ is given by Moore as:

$$N(2, D) \leq 2D + 1 \qquad \text{for } \Delta = 2 \qquad \text{and}$$

$$N(\Delta, D) \leq \frac{\Delta(\Delta - 1)^D - 2}{\Delta - 2} \qquad \text{for } \Delta > 2$$

It has been proved that the Moore graphs can exist only if:

1. $\Delta = 2$, the corresponding graph being the $2D + 1$-cycle **or**
2. $D = 1$, the graph being the $(\Delta + 1)$-cliques **or**
3. if $D = 2$ and $\Delta = 3, 7, 57$.

For $\Delta = 3$ there exists a unique Moore graph, the Petersen's graph on 10 vertices. The Hoffman-Singleton's graph on 50 vertices is a unique Moore graph for $\Delta = 7$. The existence of Moore graphs is not known for $\Delta = 57$. Sachs proved that these graphs exist only if $\Delta - 1$ is a power of a prime number [SACHS64]. R. S. Wilkov gave a construction for $\Delta \in \{2, 3, 4, 5, 6\}$. Many constructions which give a lower bound for some values of (Δ, D) graph problem were also given. Akers [AKER65] constructed $(\Delta, \Delta - 1)$ graphs based on coding theory which has

$$N_A(\Delta, \Delta - 1) = \binom{2\Delta - 1}{\Delta} \qquad \text{nodes.}$$

Arden and Lee proposed the multitree structured (Δ, D) graphs for $\Delta \leq 3$ [ARDEN78]. They claim to have given the best known solution for the (Δ, D) graph problem for $\Delta = 3$. Imase and Itoh gave a construction [IMASE81] for directed graphs which gives

$$n_I(\Delta, D) = (\Delta/2)^D \qquad \text{for } \Delta \text{ even.}$$

The nodes are numbered from 0 to $(\Delta/2)^D - 1$. Nodes i and j are connected if and only if $j = id + \alpha \bmod (n)$ or $i = jd + \alpha \bmod (n)$ where $\alpha = 0, 1, 2, \ldots, d - 1$. However reliability and extensibility are also very important for an interconnection network.

2.2.1 Reliability and Extensibility aspects

The simplest criterion for reliability or survivability is considered to be the connectivity of the graph model which represents the interconnection network. It corresponds to the minimum number of communication links or switching stations that must fail in order to destroy the communication between any pair of functional switching nodes. In this respect, it can be said that when all nodes are of equal importance, the design of a maximally reliable communication network is closely related to the construction of a graph having maximum connectivity for a given number of nodes and edges. Another important criterion for reliability is the network's ability to sustain its performance within reasonable bounds in the event of a certain number of node failures. In other words, the length of the communication path should not increase substantially in the presence of node failures.

The extensibility problem can be stated as : Given a graph G, find $G_1 \subset G_2 \subset G_3 \subset \ldots \subset G_n = G$ such that G_i keeps properties such as optimal connectivity, diameter etc. close to the final graph G. Memmi and Raillard proposed two constructions to improve the average diameter and maximal connectivity. Later Bermond *et al* generalized these constructions and came up with larger (Δ, D) graphs. Recently Opatrny *et al* have come up with another construction [OPAT86] based on Balanced Incomplete Block Design(BIBD).

2.2.2 Surviving graphs and surviving route graphs

A routing ρ in a network is a function which assigns a path between any specified pair of nodes in the network. A static routing scheme assigns to any pair of nodes in the network, a fixed path, and all communications between the two nodes travel along this path. A minimal routing is one that always gives a path of minimal length for two vertices a and b in G [BROD84]. The fault tolerance properties of networks are compared either on the basis of the surviving graph or on the basis of the surviving route graph. Given a graph G and a set of edge as well as node faults F, the surviving graph $G_s = (V_s, E_s)$ where $V_s = \{v_i | v_i \in V(G) \quad \text{and } v_i \notin F\}$ and $E_s = \{e_i \in E(G) | e_i \notin F \text{ and } e_i \text{ is not incident to any vertex } a \in F\}$. Comparison

based on surviving route graph is done in networks supporting static routing.

Given a graph G, a routing ρ and a set of faults F, the surviving route graph is made up of the vertex set $V_R = V(G) - \{v_i \in F\}$ and the edge set $E_R = \{e_i\}$ where $e_i = (a, b)$ for all $a, b \in V_R$ such that there is a route between a and b which avoids F. In the event of faults in the network, although some of the existing routes in the network are unusable due to the faults, communication between surviving nodes will still be possible by routing the messages through a sequence of surviving routes. In this context, the diameter of the surviving route graph is a measure of the worst case performance degradation caused by the faults. The diameter of the surviving route graph $D(R(G,\rho)/F)$ gives the maximum number of routes that must be used by any two processors for sending a message [DOLEV84].

2.3 A Look at some recently proposed fault tolerant networks

2.3.1 Routing in SRG model

A short description of the routing strategies mentioned in [ESFA85] is given below. Consider the nodes i and j, whose radix-r representations are respectively $r(i) = (i_{m-1}, i_{m-2}, \ldots, i_1, i_0)$ and $r(j) = (j_{m-1}, j_{m-2}, \ldots, j_1, j_0)$. Node j will be called a right-neighbor of node i if $r(j) = (x, i_{m-1}, i_{m-2}, \ldots, i_1)$ where $0 \leq x \leq \Delta - 1$. Similarly, node j will be a left-neighbor of node i if $r(j) = (i_{m-2}, i_{m-3}, \ldots, i_0, x)$. The set of right-neighbors of node i can be termed $R(i)$ and the set of left-neighbors, $L(i)$.

Given an arbitrary pair of nodes i and j, a path, $i = v_0\text{-}v_1\text{-}v_2\text{-}\cdots\text{-}v_k = j$ is said to be right consistent if for all p, such that $1 \leq p \leq k$, $v_p \in R(v_{p-1})$. Similarly the path, $i = v_0\text{-}v_1\text{-}\cdots\text{-}v_k = j$ is said to be left consistent if v_p is a left-neighbor of v_{p-1} for all p such that $1 \leq p \leq k$. When there is zero node failure, the routing scheme of Pradhan and Reddy is based on finding the shortest consistent path (left or right) between a pair of nodes. In brief, the algorithm can be stated as:

step 1 : Find the largest x such that the last x characters of i are equal to the first x characters of j.

step 2 : Find the largest y such that the last y characters of j are equal to the first

y characters of i.

step 3 : If $x > y$ then left shift $r(i)$ by the last $m-x$ characters of j else shift right $r(i)$ by the last $m-y$ characters of j.

Let F denote a set of faulty nodes, where $|F| \leq r-1$. Given a source node i and destination j, the alternate routing algorithm proposed by Pradhan and Reddy consists of establishing a route

$$(i_{m-2}, i_{m-3}, \ldots, i_0, e),$$
$$(i_{m-3}, i_{m-4}, \ldots, i_0, e, e),$$
$$\ldots$$
$$(e, e, \ldots, e),$$
$$(j_0, e, e, \ldots, e),$$
$$\ldots$$
$$(j_{m-1}, j_{m-2}, \ldots, j_0) \qquad \textbf{or}$$

$$(e, i_{m-1}, \ldots, i_1),$$
$$(e, e, i_{m-1}, \ldots, i_2),$$
$$\ldots$$
$$(e, e, \ldots, e),$$
$$(e, e, \ldots, e, j_{m-1}),$$
$$\ldots$$
$$(j_{m-1}, j_{m-2}, \ldots, j_1, j_0)$$

such that e is not equal to the most(least) significant digit of the radix-r representation of any node in F. The algorithm doesn't guarrantee a route when $|F| > r-1$.

For $r \leq |F| \leq 2r-3$, Esfahanian and Hakimi have given routing algorithms which are based on finding a node $b = (b_{x-1}, b_{x-2}, \ldots, b_1, b_0)$ such that there exists a path from node i to node b and from node b to node j. If $|F \cap (L(i) \cup R(j))| \leq |F \cap (R(i) \cup L(j))|$ then the route would consist of the left consistent path from i to b denoted as PL(i-b) followed by the left consistent path from b to j. Otherwise, the route would be a right consistent path from i to b followed by a right consistent path from b to j. Note that in this case the fault tolerant route is either left consistent

or right consistent unlike in the previous case, where a left(right) consistent path was followed by a right(left) consistent path.

The Shift and Replace Graph, as we have seen, is one of the best known constructions proposed for large interconnection networks. In comparison with the existing hypercube network, the SRG gives more optimal diameter and therefore lesser communication delay for larger values of $N(\Delta, D)$. For example, consider $N(\Delta, D) = 65536$. A hypercube with these many nodes would have a diameter D as well as degree Δ of 16. But an SRG with the same order can be constructed with diameter $D = 8$ and degree $\Delta = 8$ (since SRG is an offspring of the DeBruijn graph which has order $N(\Delta, D) = (\Delta/2)^D$ for Δ even. One deterrent for the hypercube is its logarithmic increase in degree(node-connectivity) as the number of nodes increases. But the hypercube possesses a very desirable property as far as fault tolerance is concerned. That is, it's highly regular structure, which makes reconfiguration much easier in the event of node failure. The SRG as already seen, is not a regular graph. Further, regular graphs make the routing process less tedius. Thus, the not totally independent nature, of the desirable properties of an IN such as higher order, lower degree and diameter, regularity, and the need for existence of multiple node-disjoint paths between stations present a very difficult design problem.

2.3.2 Network construction based on BIBD

The following notion of balanced incomplete block design will be needed in the construction of the special class of interconnection network which we are studying. A Balanced Incomplete Block Design (BIBD) essentially deals with the arrangement of a set of objects into a specified number of subsets or blocks so that the resulting arrangement has the following properties:

1. Each block(subset) has the same number of elements as every other block.
2. The number of different blocks in which an element appears is the same for all elements.
3. Each collection of elements say a pair, triple, or quadruple appears in exactly the same number of blocks.

To be more precise, we use the following definition [HALL67]. A balanced incomplete block design is an arrangement of n distinct objects into b blocks such that each block contains exactly k distinct objects, each object occurs in exactly r different blocks, and every pair of distinct objects a_i, a_j occurs together in exactly λ blocks.

Thus there is a certain relation of incidence depicting which objects belong to which blocks. The fact that not all nCk combinations form blocks makes this arrangement an incomplete block design. But the property that each pair of elements a_i, a_j occurs the same number of times makes it balanced. The two elementary relations among the five parameters, b, n, r, k, λ, of a block design are:

$$bk = nr$$

$$r(k-1) = \lambda(n-1)$$

A block design is said to be symmetric if $b = n$. Obviously,in a symmetric BIBD, $r = k$. Furthermore, in a symmetric BIBD, any two different blocks have exactly λ elements in common. Symmetric block designs are referred to as (n, k, λ) designs. Let $n = n(k) = k^2 - k + 1$. With reference to [HALL67] we give without proof, the following theorem as a sufficient condition for the existence of a symmetric balanced incomplete block design.

Theorem 2.1. *Given $n(k) = k^2 - k + 1$, a symmetric balanced incomplete block design $(n(k), k, \lambda)$ exists if $k - 1$ is a power of a prime.*

Now let's look at some examples of BIBD.

example 2.1

Let $n = b = 7$, $r = k = 3$, and $\lambda = 1$. The corresponding symmetric BIBD is as follows:

$$B_1 : [1,2,4]$$
$$B_2 : [2,3,5]$$
$$B_3 : [3,4,6]$$
$$B_4 : [4,5,7]$$
$$B_5 : [5,6,1]$$
$$B_6 : [6,7,2]$$
$$B_7 : [7,1,3]$$

Here we can see that there are 7 objects arranged in 7 blocks. Each object appears in exactly 3 different blocks and each pair appears together in one and only one block. This is an example of a symmetric block design.

example 2.2

Let $n = 9$, $b = 12$, $r = 4$, $k = 3$, and $\lambda = 1$. The corresponding blocks are:

$B_1 : [1,2,3]$	$B_4 : [1,4,7]$	$B_7 : [1,5,9]$	$B_{10} : [1,6,8]$
$B_2 : [4,5,6]$	$B_5 : [2,5,8]$	$B_8 : [2,6,7]$	$B_{11} : [2,4,9]$
$B_3 : [7,8,9]$	$B_6 : [3,6,9]$	$B_9 : [3,4,8]$	$B_{12} : [3,5,7]$

In this case there are 9 objects which are arranged in 12 different blocks. By the definition of symmetry, this is not a symmetric BIBD. Note that $n \neq b$ and $r \neq k$. Each pair of objects appear together in exactly one block and each pair of blocks has exactly one element in common. A table listing many block designs can be found in [HALL67].

Here we give a brief description of the interconnection scheme proposed by Opatrny *et al*[OPAT89]. Recall from the earlier discussion of BIBD that the different blocks form a well-overlapping system of sets. This fact should give some intuition that

given a graph G and a certain block design (b, n, r, k, λ), one could associate some sort of mapping from the elements of the blocks to the nodes which would give us a new structure with possibly many paths between pairs of nodes. And this indeed is the case. We will look at the constructions given in [OPAT89] for $(b, n, r, k, 1)$ design with graphs K_1, K_k and $K_{i,k}$ yielding respectively, the network models (K_1, b, n, r, k), (K_k, b, n, r, k) and $(K_{i,k}, b, n, r, k)^{t-i}$.

2.3.2.1 Construction of (K_1, b, n, r, k)

For any integer k, $n(k)$ is defined as $n(k) = k^2 - k + 1$. It may be clear from the discussion on complete graph K_n that K_1 is comprised of a single isolated vertex. Let $B = \{B_1, B_2, \ldots, B_b\}$ be the b blocks of a $(b, n, r, k, 1)$ design on the set $S = \{v_1, v_2, \ldots, v_n\}$.

1. The vertex set $V = \{v_1, v_2, \ldots, v_n\} \cup \{u_1, u_2, \ldots, u_b\}$ where u_i, for $1 \leq i \leq b$ represents the block B_i.
2. The edge set $E = \{(v_i, u_j) | v_i \in B_j\}$.

The resulting graph is a bipartite graph with degree sequence (r, k).

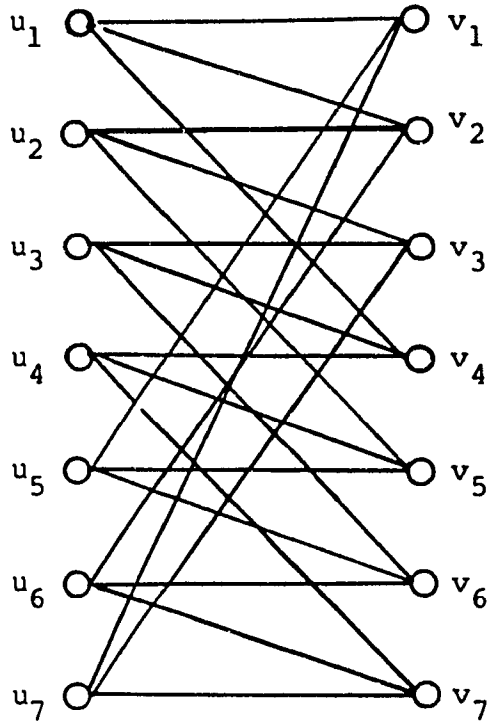


fig. 2.4 $(K_1, 7, 7, 3, 3)$ model.

The following results for $G = (K_1, b, n, r, k)$ have been proved in [OPAT89].

1.
$$D(G) = \begin{cases} 3 & \text{if } B_i \cap B_j \neq \emptyset \\ 4 & \text{otherwise cr} \end{cases}.$$

2. There are k node disjoint paths of length ≤ 4 between every pair of nodes.
3. For every integer k such that $k-1$ is a prime power, $(K_1, n(k), n(k), k, k, 1)$ is a k-regular communication network model having the properties:
   a. G is k-connected.
   b. $D(G) = 3$.
   c. $|V(G)| = 2(k^2 - k + 1)$
   d. For any set F of faults, $F \subseteq V(G)$ and $|F| \leq k-1$

$$D(G/F) \leq 4 \qquad \text{and}$$

$$D(R(G,\rho)/F) = 2 \qquad \text{for any minimal routing } \rho.$$

2.3.2.2 Construction of (K_k, b, n, r, k)

Let $B_1, B_2, \ldots, B_b$ be the blocks of the $(b, n, r, k\ 1)$ design on the set $S = \{v_1, v_2, \ldots, v_n\}$. Let $B_i = \{x_{i_1}, x_{i_2}, \ldots, x_{i_k}\}$ for $1 \leq i \leq b$ and $1 \leq j \leq k$ and $x_{i_j} \in S$.

Make b copies of the complete graph K_k. Label the nodes as $u_{i_1,i}, u_{i_2,i}, \ldots, u_{i_k,i}$ for the graph representing the block B_i. Introduce n additional nodes $v_1, v_2, \ldots, v_n$ corresponding to the elements of S. Join node v_j to node $u_{j,i}$ for $1 \leq i \leq b$ and $1 \leq j \leq n$. The resulting graph $G = (K_k, b, n, r. k)$ has degree sequence (r, k) and $n(r+1)$ nodes.

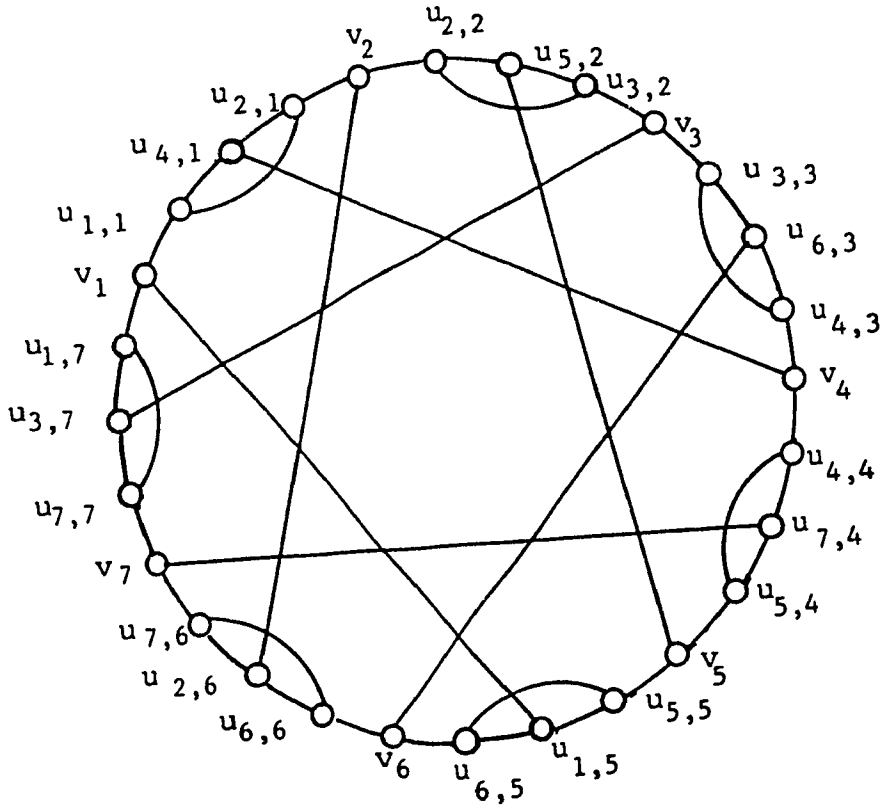


Fig. 2.5 $G = (K_3, 7, 7, 3, 3)$

It can be seen that there is a loop structure embedded in this graph as in the case of the SRG. But this graph is more regular. The graph, $G = (K_k, n(k), n(k), k, k, 1)$ is indeed regular. The following results are true for $G = (K_k, b, n, r, k)$:

1. There exists exactly one shortest path between every pair of nodes.

2.

$$D(G) = \begin{cases} 4 & \text{if } B_i \cap B_j \neq \emptyset \\ 5 & \text{otherwise} \end{cases}.$$

3. The graph (K_k, b, n, r, k) is k-connected with every pair of nodes having at least k node-disjoint paths of length ≤ 7.
4. For the network structure, $G = (K_k, b, n, r, k)$ with minimal routing ρ and any set F of faulty nodes such that $|F| \leq (k-1)$, $d(R(G,\rho)/F) \leq 2$.
5. For every integer k such that $(k-1)$ is a power of a prime number, the graph $G = (K_k, n(k), n(k), k, k, 1)$ is a k-regular communication network model having the properties:
   a. G is k-connected.
   b. $D(G) = 4$.
   c. $|V(G)| = (k^2 - k + 1)(k+1)$.
   d. For any set F of faulty nodes, $|F| \leq (k-1)$, $D(G^*/F) \leq 7$ and

$$D(R(G,\rho)/F) \leq 2 \text{ for the minimal routing } \rho.$$

It has also been shown that (K_1, b, n, r, k) and (K_k, b, n, r, k) models give large networks with smaller diameter. Further, they give better surviving route graphs. The $(K_{i,k}, b, n, r, k)^{k-i}$ model generates graphs with higher diameter than that of (K_1, b, n, r, k) and (K_k, b, n, r, k) models.

2.3.2.3 Construction of graph $(K_{i,k}, b, n, r, k)^1$

Construct a complete bipartite graph $K_{i,k}$ for each block B_j, where $1 \leq j \leq b$. The nodes of $K_{i,k}$ with degree k are labelled as $\{z_{1,i,j}, z_{2,1,j}, \ldots, z_{i,1,j}\}$ and the nodes with degree i are labelled as $\{u_{j_1,1,j}, u_{j_2,1,j}, \ldots, u_{j_k,1,j}\}$. Corresponding to the n elements of S, nodes $v_{1,1}, v_{1,2}, \ldots, v_{1,n}$ are added to the vertex set V and the edges $(v_{1,t}, u_{t,1,x})$ for $1 \leq t \leq n$ and $1 \leq x \leq b$ to the edge set E. The resulting graph has $b(i+k)+n$ nodes and is of degree sequence $(r, k\ i+1)$. For any integer $s \geq 2$, the graph $(K_{i,k}, b, n, r, k)^s$ is derived from $(K_{i,k}, b, n, r, k)^{s-1}$ as follows:
Make b copies of the graph $(K_{i,k}, b, n, r, k)^{s-1}$. Relabel the nodes of the b^{s-1} copies of $K_{i,k}$ such that node u_{α_l} now corresponds to $u_{\alpha_l,\beta_l,m,j}$ and node z_{β_l} corresponds

to $z_{\beta_l,m,j}$ for the m^{th} copy of $K_{i,k}$ for $1 \leq l \leq k$ and $1 \leq m \leq b^{s-1}$. Further, relabel the nodes $v_{\gamma_1}, v_{\gamma_2}, \ldots$ of the j^{th} copy of $(K_{i,k}, b, n, r, k)^{s-1}$ as $v_{\gamma_1,j}, v_{\gamma_2,j}, \ldots$. Let $w = b^{s-1}$. Corresponding to the n elements of S, b^{s-1} new nodes labelled $v_{s,1,1}, v_{s,1,2}, \ldots, v_{s,1,w}, v_{s,2,1}, \ldots, v_{s,2,w}, \ldots, v_{s,n,1}, \ldots, v_{s,n,w}$ are introduced. Node $v_{s,t,x}$ is joined to node $u_{\alpha,t,x,y}$ for $1 \leq t \leq n$, $1 \leq x \leq w$ and $1 \leq y \leq b$. The resulting graph $(K_{i,k}, b, n, r, k)^s$ has degree sequence $(r, k.i + s)$ and order $b^{s-1}(b(i + k) + sn)$. It has been proved in [OPAT89] that diameter of $(K_{i,k}, b, n, r, k)^s \leq 4s + 2$. Furthermore, there exists $s + i$ node disjoint paths of length $\leq 4s + 6$ between every pair of nodes of graph $(K_{i,k}, b, n, r, k)^s$. For $s = k - i$, the resulting structure is regular. With $(k - 1)$ as a prime power, $(K_{i,k}, n(k), n(k), k, k)^{k-i}$ is a k-regular communication network model. Figure 2.6 gives the 3 regular graph corresponding to $(K_{2,3}, 7, 7, 3, 3)$. These networks not only possess good fault tolerant properties but also have a hierarchical structure. Furthermore, some of these models are the largest known (Δ, D) graphs which are regular.

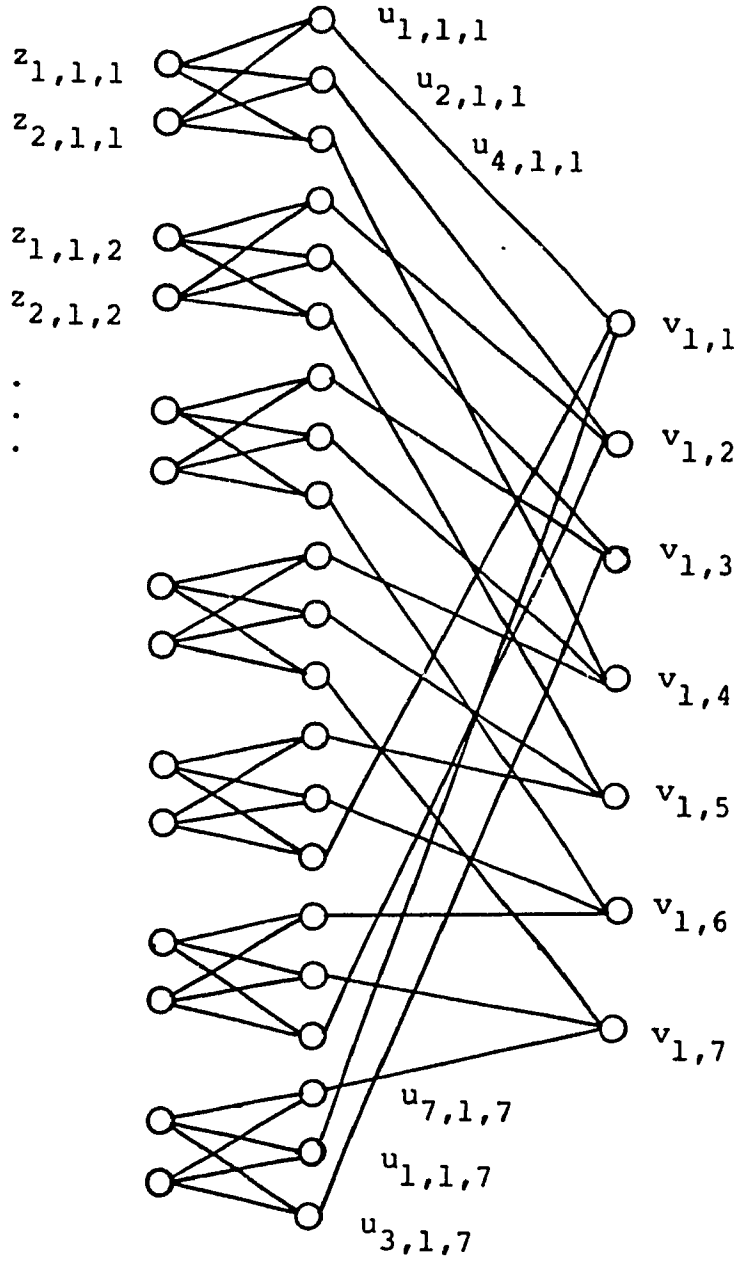


fig. 2.6 $(K_{2,3}, 7, 7, 3, 3)$

CHAPTER 3

DISTRIBUTED ROUTING ALGORITHM FOR THE NETWORK MODEL $G(K_{2,2})^i$

In this chapter, first we give the construction of network models $(G, n(k), n(k), k, k)$ and $(K_{k,k}, n(k), n(k), k, k)^i$. We then study their basic properties such as the diameter, the order, and the number of edge disjoint paths. A suitable labelling scheme is introduced. Using this scheme, we present a class of algorithms called *Order Preserving algorithms (OP)* for efficient routing of messages. Then we present the *Opportunistic algorithm (O)* and prove its optimality with respect to route length. It is also proved that the route length given by this algorithm may exceed a shortest path length by at most $\lfloor 2/3D_R \rfloor + \lfloor 2/3D_V \rfloor$.

3.1 Construction of network model $(G, n(k), n(k), k, k)$

In this section, we describe the construction of $(G, n(k), n(k), k, k)$ by the application of $(n(k), n(k), k, k, 1)$ design on graph G whenever order of $G \geq k$. Select k subsets $S_1, S_2, \ldots, S_k$ of $V(G)$ such that $S_i \cap S_j = \emptyset$, for $i \neq j$ and $|S_i| = m \geq 1$. Make $n(k)$ copies of G. Note that corresponding to each block p in the block design, we have a copy of the original graph G say G_p. Observe that there are k elements in each block p. Furthermore, G_p contains exactly k subsets. Therefore we establish a one to one correspondence between each element in the block p and the set of subsets in G_p. For example, if p_i denotes the i^{th} element of block p, then the i^{th} subset of G_p will be assigned to p_i. Thus a subset can be uniquely identified by a pair of indices of the form p, p_i. Within each subset, the m individual elements can be uniquely identified as $v_{p,p_i,i_1}, v_{p,p_i,i_2}, \ldots, v_{p,p_i,i_m}$. Node v_{p,p_i,i_j} is joined to node v_{q,q_l,l_j} whenever $p_i = q_l$, for $1 \leq p, q \leq n(k)$, $1 \leq j \leq m$. The resulting graph is $(G, n(k), n(k), k, k)$. Given the graph G, its subsets $S_1, S_2, \ldots, S_k$ and the block design $(n(k), n(k), k, k, 1)$, the construction is unique up to isomorphism. Hence in our future discussions on the

construction of network model $(G, n(k), n(k), k, k)$, it is enough to specify the graph G, the subsets $S_1, S_2, \ldots, S_k$ and the block design $(n(k), n(k), k, k, 1)$.

Example 1

Let $G = \langle V, E \rangle$, where $V = \{a, b, c, d, e, f\}$ and $E = \{(a, b), (b, c), (c, d), (e, f), (f, a)\}$ be the graph on which the symmetric block design $(3, 3, 2, 2, 1)$ is to be applied. We first select the subsets S_1 and S_2 of V such that $S_1 \cap S_2 = \emptyset$. Let $S_1 = \{a, b\}$ and $S_2 = \{f, c\}$ (refer to fig. 3.1 (a)). The selection is arbitrary or in other words, a different selection of nodes could be made. First we will make $n(k) = 3$ copies of G and call them G_1, G_2 and G_3. Further, call the i^{th} subset of G_p as S_{p,p_i}. Refer to fig. 3.1.(b). According to the labelling scheme described in the construction, a in $S11$ will be known as $v_{11,1_1}$, b in S_{11} as $v_{11,1_2}$ etc. Next we introduce the edges $(v_{p,p_i,i_j}, v_{q,q_l,l_j})$, where $p_i = q_l$, for $1 \leq p, q \leq 3$, $1 \leq j \leq 2$. The resulting graph is shown in fig. 3.2.

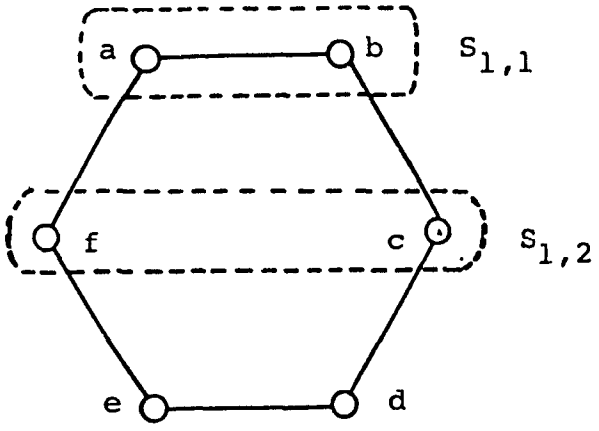
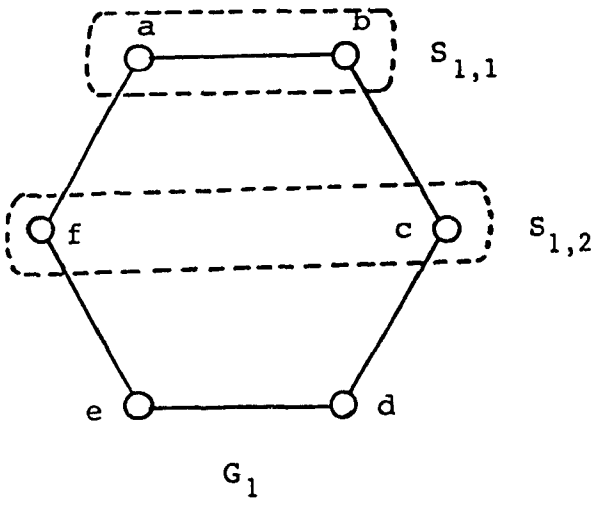


fig. 3.1.(a) Basic graph G



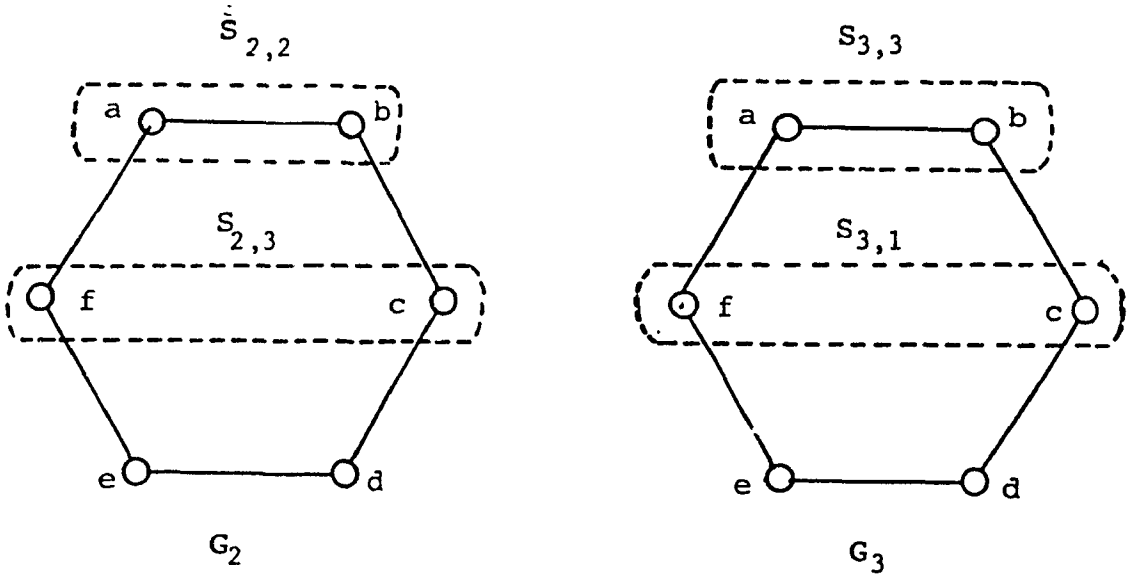


fig. 3.1.(b) $n(k)$ copies of basic graph G

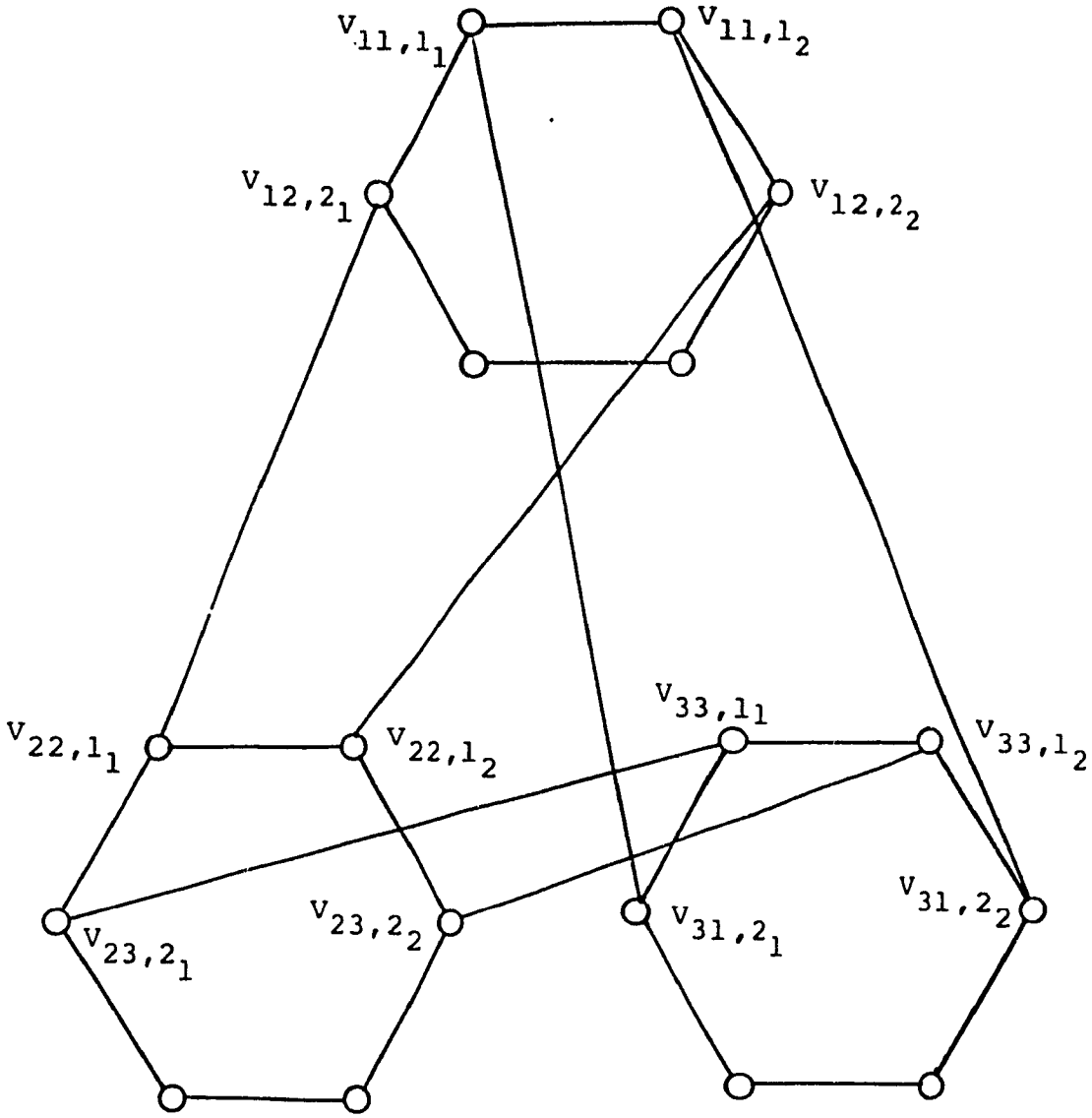


fig. 3.2 The graph $(G, n'k), n(k), k, k)$

3.1.1 Construction of network model $(K_{k,k}, n(k), n(k), k, k)^1$

Consider the bipartite graph $K_{k,k}$ with bipartition (α, β). Partition the set of nodes α into k singleton subsets $S_1, S_2, \ldots, S_k$. Now apply the design $(n(k), n(k), k, k, 1)$ on $K_{k,k}$. The resulting graph is called $(K_{k,k}, n(k), n(k), k, k)^1$. Further, if $v_{p,i,j}$, where $i = 1, 2, \ldots, k$ are the labels of α nodes in the p^{th} copy, then β-nodes appearing in the same copy will be denoted by $u_{p,i,j}$, where $i = 1, 2, \ldots, k$.

Example 2. $(K_{2,2}, 3, 3, 2, 2)^1$ is given below:

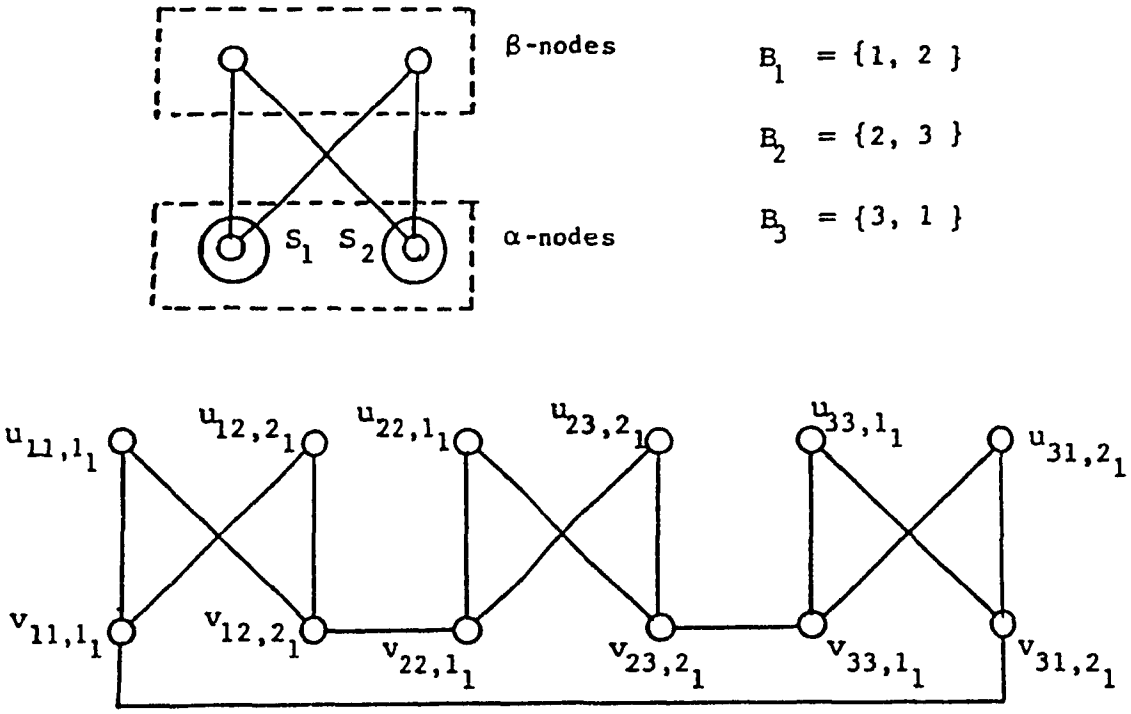


fig. 3.3 $(K_{2,2}, 3, 3, 2, 2)^1$

We now give some of the important results concerning $(K_{k,k}, n(k), n(k), k, k)^1$. Note that in the above construction, $|S_i| = 1$. Hence to simplify the notation, we will denote a node $v_{i,j,k}$ simply as $v_{i,j}$. Further, if there is no need to distinguish between different β nodes appearing in the i^{th} copy, we shall simply denote it as u_i.

Theorem 3.1. *The network model $(K_{k,k}, n(k), n(k), k, k)$ is of diameter 4 having $2kn(k)$ nodes and degree sequence $(k, 2k-1)$. Furthermore, the minimal routing in $(K_{k,k}, n(k), n(k), k, k)$ is unique up to β-nodes.*

proof:

Since there are $n(k)$ copies of $K_{k,k}$, the total number of nodes is $2kn(k)$. Note that β-nodes are not involved in this construction. Hence their degree remains at k. As per the block design property 2, every element appears in k distinct blocks. Thus in the construction, each node in α is connected to $k-1$ new nodes. Hence the degree of a node in α is $k+(k-1)=2k-1$. We now proceed to prove the rest of the results stated in the theorem. Since the subsets involved in the construction are singleton, we will discard the last index of the node labels for the rest of the proof.

case 1. Consider the pair $v_{i,p}, v_{j,s} \in \alpha$.

If $p = s$ or $i = j$ then $d(v_{i,p}, v_{j,s}) \leq 2$ and the shortest path is unique up to the β-nodes. Suppose $p \neq s$. Then according to the block design, there exists a block B_t containing the pair (p, s). Thus there exists a path $v_{i,p} \rightarrow v_{t,p} \rightarrow u_t \rightarrow v_{t,s} \rightarrow v_{j,s}$. Here u_t is a β-node in the block B_t. The above path is a shortest path between $v_{i,p}$ and $v_{j,s}$. Uniqueness up to the selection of u_t holds since the pair (p, s) appears in exactly one block.

case 2. Consider the pair $(v_{i,p}, u_j)$, where $u_j \in B_j \cap \beta$.

If $i = j$, then these two nodes are adjacent and the result follows. Suppose $i \neq j$. Let s be the element common to blocks B_i and B_j. If $s \neq p$ then $v_{i,p} \rightarrow u_i \rightarrow v_{i,s} \rightarrow v_{j,s} \rightarrow u_j$ is the unique shortest path between $v_{i,p}$ and u_j (up to selection of u_i). Otherwise, if $s = p$, $v_{i,p} \rightarrow v_{j,p} \rightarrow u_j$ is the unique shortest path.

case 3. Consider the pair (u_i, u_j) where $i \neq j$ and $u_i \in \beta \cap B_i$.

Let s be the common element in blocks B_i and B_j. Then the path $u_i \rightarrow v_{i,s} \rightarrow v_{j,s} \rightarrow u_j$ is the unique shortest path between u_i and u_j.

Thus the diameter of the graph is 4.

Corollary 3.1. *There is a path of length 3 between any pair of β-nodes in graph $(K_{k,k}, n(k), n(k), k, k)^1$.*

proof: Refer case 3 above.

Theorem 3.2. *Graph $(K_{k,k}, n(k), n(k), k, k)^1$ is k-connected with every pair of nodes having k node disjoint paths of length ≤ 8.*

proof: The proof is by considering every possible pair of nodes which fall into three cases. For every l, $1 \leq l \leq n(k)$ let C_l be the set of indices of the blocks of $(n(k), n(k), k, k, 1)$ which contains l. Let $C_l = \{l_1, l_2, \ldots, l_k\}$, where $1 \leq l \leq n(k)$.

case 1. $(v_{i,p}, v_{j,s})$. Suppose $p \neq s$.

The paths $v_{i,p} \rightarrow v_{p_t,p} \rightarrow u_{p_t} \rightarrow v_{p_t,r_t} \rightarrow v_{s_t,r_t} - u_{s_t} \rightarrow v_{s_t,s} \rightarrow v_{j,s}$, $t = 1, 2, \ldots, k$ where r_t is the common element of blocks B_{p_t} and B_{s_t} are node disjoint paths of length ≤ 7 between $v_{i,p}$ and $v_{j,s}$. Assume that $p = s$. First of all note that $v_{i,p}$ and $v_{j,p}$ are adjacent nodes and thus there is a path of length 1. Since p appears in k different blocks, including blocks i and j, there exists $k-2$ blocks (other than i and j) in which p appears. Let B_t be such a block. Then $v_{i,p} \rightarrow v_{t,p} \rightarrow v_{j,p}$ is a path of length 2. Thus there are $k-1$ paths of length ≤ 2 between $v_{i,p}$ and $v_{j,p}$. Further the path $v_{i,p} \rightarrow u_i \rightarrow v_{i,s} \rightarrow v_{s,s} \rightarrow u_s - v_{s,j} \rightarrow v_{j,j} \rightarrow u_j \rightarrow v_{j,p}$ is a path of length 8 where $s \in B_i - \{p\}$.

case 2. The nodes $v_{i,p}, u_j$.

The paths $v_{i,p} \rightarrow v_{p_t,p} \rightarrow u_{p_t} \rightarrow v_{p_t,r_t} \rightarrow v_{j,r_t} \rightarrow u_j$, $t = 1, 2, \ldots, k$ where r_t is common to blocks B_{p_t} and B_j are node disjoint paths of length ≤ 5 between $v_{i,p}$ and u_j.

case 3. The nodes u_i and u_j.

The paths $u_i \rightarrow v_{i,p_t} \rightarrow v_{m,p_t} \rightarrow u_m \rightarrow v_{m,r} \rightarrow u_{j,r} \rightarrow u_j$, $1 \leq t \leq k$, where pair (i, m) is in block B_{p_t} and (m, j) in B_t are k node disjoint paths of length 6 between u_i and u_j.

3.1.2 Construction of $(K_{k,k}, n(k), n(k), k, k)^i$, $i \geq 2$

Inductively assume that $(K_{k,k}, n(k), n(k), k, k)^{i-1}$ has been constructed. without loss of generality assume that α-nodes were involved in the construction of $(K_{k,k}, n(k), n(k), k, k)^{i-1}$. Hence in this step, β-nodes alone will be used. Assume that $S'_{p,1}, S'_{p,2}, \ldots, S'_{p,k}$ were the subsets involved n the p^{th} copy of the previous construction. Since β-nodes are involved in this level, let $S''_{p,j} = \{u_{s,t,r} | v_{s,t,r} \in S'_{p,j}\}$. In this level of construction, the j^{th} subset is constructed as $S_j = \cup_{p=1}^{n(k)} S''_{p,j}$. Further,

if $S''_{p,j}$ is the ordered set $\{x_{p_1}, x_{p_2}, \ldots, x_{p_q}\}$ then the order in S_j is given by

$$x_{1_1}, x_{1_2}, \ldots, x_{1_q}, x_{2_1}, x_{2_2}, \ldots, x_{2_q}, \ldots, x_{n(k)_1}, x_{n(k)_2}, \ldots, x_{n(k)_q}.$$

Now the network model $(K_{k,k}, n(k), n(k), k, k)^i$ is obtained by the application of block design $(K_{k,k}, n(k), n(k), k, k, 1)$ on the graph $(K_{k,k}, n(k), n(k), k, k)^{i-1}$ with $S_1, S_2, \ldots, S_k$ as subsets. Thus

$$(K_{k,k}, n(k), n(k), k, k)^i = ((K_{k,k}, n(k), n(k), k, k)^{i-1}, n(k), n(k), k, k).$$

Example 4. Construction of $(K_{2,2}, 3, 3, 2, 2)^2$.

Recall that (example 2) $(K_{2,2}, 3, 3, 2, 2)^1$ is given by fig. 3.4.(a). Now the sets S_1 and S_2 are $S_1 = \{u_{1,1,1_1}, u_{2,2,1_1}, u_{3,3,1_1}\}$ $S_2 = \{u_{1,2,2_1}, u_{2,3,2_1}, u_{3,1,2_1}\}$. According to the construction described, the graph $(K_{k,k}, n(k), n(k), k, k)^2$ is given by fig. 3.4.(b).

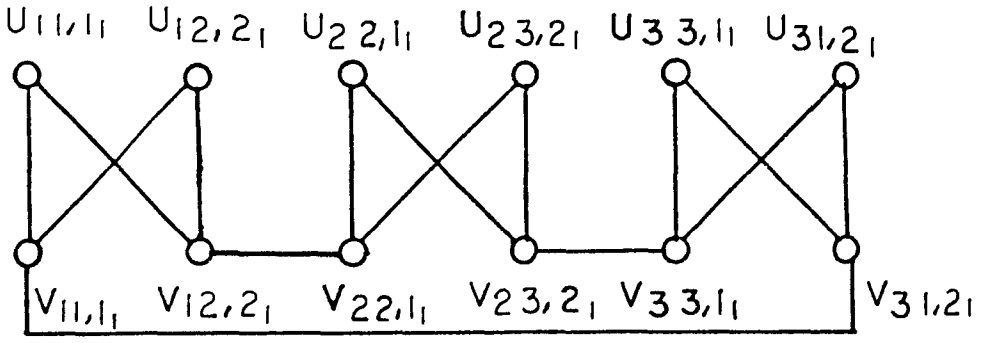


fig. 3.4.(a) $(K_{2,2}, 3, 3, 2, 2)^1$

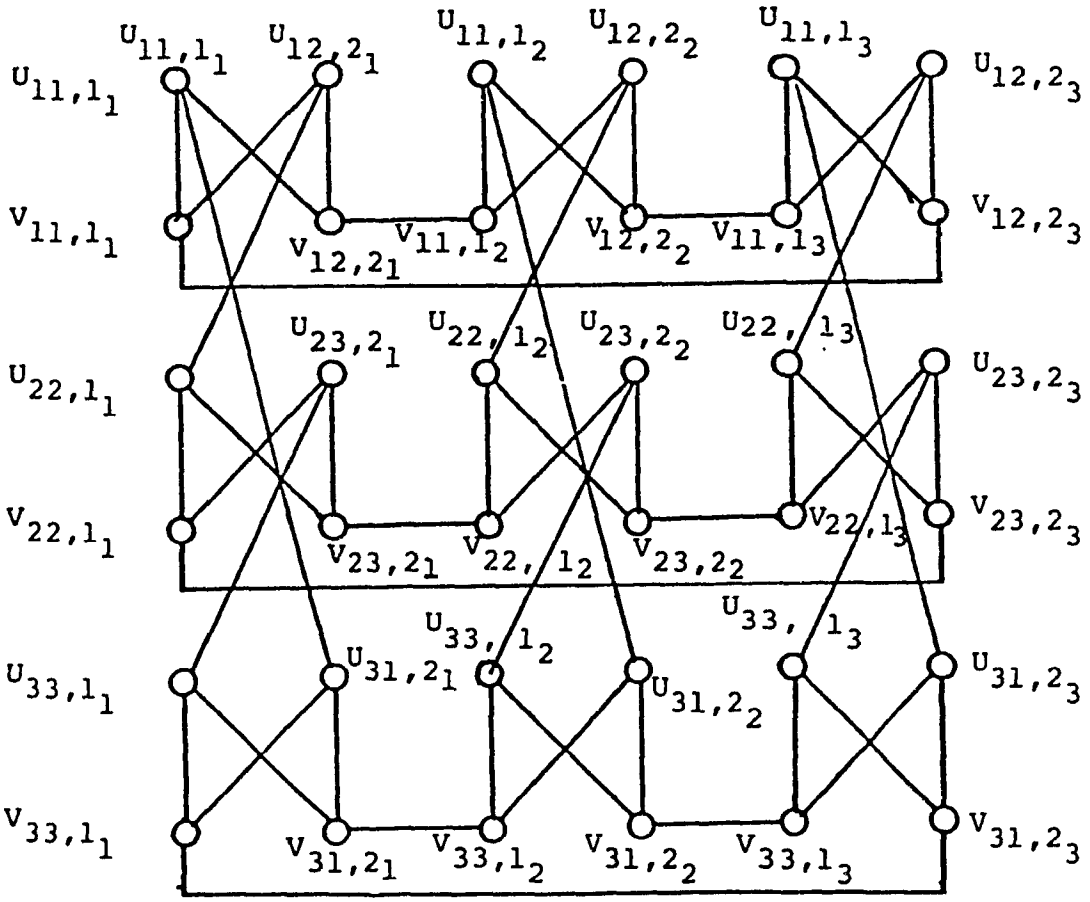


fig. 3.4.(b) $(K_{2,2}, 3, 3, 2, 2)^2$

Theorem 3.3. *$(K_{k,k}, n(k), n(k), k, k)^2$ is a network model of diameter 5 having $2k(n(k))^2$ nodes and degree $2k-1$.*

proof:

Since $n(k)$ copies of $(K_{k,k}, n(k), n(k), k, k)^1$ are used in the construction of $(K_{k,k}, n(k), n(k), k, k)^2$, there are $n(k)(2kn(k)) = 2k(n(k))^2$ nodes. In this construction, β-nodes only are involved and hence the degree of α-nodes remains as $2k-1$. Since each β-node is connected to $k-1$ additional nodes, degree of a β-node is $k+(k-1) = 2k-1$. Now we proceed to prove that the network has diameter 5. We consider the following three cases.

case 1. Two β-type nodes.

Let $(u_{i,j,l}, u_{i',j',l'})$ be a pair of β-nodes. If $i = i'$ or $j = j'$ or $l = l'$ the proof is simpler. Hence let $i \neq i'$, $j \neq j'$, and $l \neq l'$. Let B_p be the block in which j and

j' occur together. Then $u_{i,j,l} \rightarrow u_{p,j,l} \xrightarrow{3} u_{p,j',l'} \rightarrow u_{i',j,l'}$ is a path of length 5 (by corollary 3.1).

case 2. Two α-type nodes $(v_{i,j,l}, v_{i',j',l'})$.

Let j_1 and j_2 denote the middle indices of the nodes $v_{i,j,l}$ and $v_{i',j',l'}$ in the previous level of the construction. Let B_p denote the block in which the pair (j_1, j_2) appear. Let $s = B_i \cap B_{i'}$. Now it may be varified that $v_{i,j,l} \rightarrow v_{i,x,p} \rightarrow u_{i,s,p} \rightarrow u_{i',s,p} \rightarrow v_{i',y,p} \rightarrow v_{i',j',l'}$ is a path of length 5 for some values of x and y. Thus there is a path of length 5 between two α-type nodes.

case 3. A β-type and an α-type node $(u_{i,j,l}, v_{i',j',l'})$. From corollary 3.1, we know that there is a path of length 3 between $u_{i,j,l}$ and $u_{i,j',l'}$. Hence $u_{i,j,l} \xrightarrow{3} u_{i,j',l'} \rightarrow u_{i',j',l'} \rightarrow v_{i',j',l'}$ is a path of length 5.

This completes the proof.

Theorem 3.4. *There are $2k-1$ node disjoint paths of length ≤ 8 between any pair of nodes in $(K_{k,k}, n(k), n(k), k, k)^2$.*

proof: We consider the following three cases which will encompass all the node pairs.

case 1. Two β-type nodes, $u_{i,j,l}$ and $u_{i',j',l'}$.

Let $B_i = \{j_1, j_2, \ldots, j_k\}$. Similarly let $B_{i'} = \{j'_1, j'_2, \ldots, j'_k\}$. Let $p \in B_i \cap B_{i'}$. Then $u_{i,j,l} \rightarrow v_{i,j_t,l} \rightarrow v_{i,j_t,l_t} \rightarrow u_{i,p,l_t} \rightarrow u_{i',p,l_t} \rightarrow v_{i',j'_t l_t} \rightarrow v_{i',j_{t'},l'} \rightarrow u_{i',j',l'}$ is a path of length 7 for $1 \leq t \leq k$. Now there exists values $i_1, i_2, \ldots, i_{k-1}$ $(i_t \neq i)$ such that $u_{i,j,l}$ is adjacent to the nodes labelled $u_{i_1,j,l}, u_{i_2,j,l}, \ldots, u_{i_{k-1},j,l}$. Choose $\hat{i}_t$ and s_t such that $B_{\hat{i}_t} \cap B_{i'} = \{j'\}$ and $B_{i_t} \cap B_{\hat{i}_t} = s_t$. Now $u_{i,j,l} \rightarrow u_{i_t,j,l} \xrightarrow{3} u_{i_t,s_t,l'} \rightarrow u_{\hat{i}_t,s_t,l'} \rightarrow v_{\hat{i}_t,s_t,l'} \rightarrow u_{\hat{i}_t,j',l'} \rightarrow u_{i',j',l'}$ is a path of length ≤ 8 for $t = 1, 2, \ldots, k-1$. Thus there are $2k-1$ node disjoint paths of length ≤ 8.

case 2. Two α nodes $(v_{i,j,l}, v_{i',j',l'})$.

Let $p = B_i \cap B_{i'}$. Let $j_1, j_2, \ldots, j_{k-1}$ be the $k-1$ elements other than p in B_i. Now $v_{i,j,l} \rightarrow v_{i,s_t,l_t} \rightarrow u_{i,p,l_t} \rightarrow u_{i',p,l_t} \rightarrow v_{i',r_t,l_t} \rightarrow v_{i',j',l'}$ is a path of length ≤ 5 for $t = 1, 2, \ldots, k-1$. Let $j'_1, j'_2, \ldots, j'_{k-1}$ be the other $k-1$ elements in B'_i. Let B_{i_t} be the unique block containing the pair j_t and j'_t. Now $v_{i,j,l} \rightarrow u_{i,j_t,l} \rightarrow u_{i_t,j_t,l} \xrightarrow{3} u_{i_t,j_t,l'} \rightarrow u_{i',j'_t,l'} \rightarrow v_{i',j',l'}$ is a path of length ≤ 7 for $t = 1, 2, \ldots, k-1$.

Thus there are $2k-1$ node disjoint paths.

case 3. One α-node and one β-node $(v_{i,j,l}, u_{i',j',l'})$

Let $p = B_i \cap B_{i'}$. Let $j_1, j_2, \ldots, j_{k-1}$ be the $k-1$ elements other than p in B_i. Now $v_{i,j,l} \rightarrow v_{i,s_t,l_t} \rightarrow u_{i,p,l_t} \rightarrow u_{i',p,l_t} \xrightarrow{3} u_{i',j',l'}$ is a path of length ≤ 6 for $t = 1, 2, \ldots, k$. Let B_{i_t} be the block containing the pair (j_t, j'). Clearly $v_{i,j,l} \rightarrow u_{i,j_t,l} \rightarrow u_{i_t,j_t,l} \xrightarrow{3} u_{i_t,j',l'} \rightarrow u_{i',j',l'}$ is a path of length ≤ 6, for $t = 1, 2, \ldots, k-1$. Thus there are $2k-1$ node disjoint paths of length ≤ 8 between any pair of nodes. For notational convenience we will denote graph $(K_{k,k}, n(k), n(k), k, k)^i$ also by $G(K_{k,k})^i$.

Theorem 3.5. *Diameter of* $(K_{k,k}, n(k), n(k), k, k)^i$ *is* $2 + 3i/2$ *if* i *is even and* $4 + 3(i-1)/2$ *if* i *is odd.*

proof: We will treat the even and odd cases separately. Let (x, y) be a pair of nodes.

case 1. $i = 2l$, where l is any integer ≥ 1.

Proof is by induction on l.

Basis: $l = 1$

Between any pair of β-type nodes, there exists a path of length 4 (by corollary 1 and the fact that there is an edge $(u_{i,j,l}, u_{i',j,l})$. Now, considering any two α-type nodes, there is a path $\alpha \rightarrow \alpha \rightarrow \beta \rightarrow \beta \rightarrow \alpha \rightarrow \alpha$ of length 5.

Between an α-type node and a β-type node, there exists a path $\alpha \rightarrow \beta \rightarrow \beta \rightarrow \alpha \rightarrow \alpha \rightarrow \beta$ or $\beta \rightarrow \alpha \rightarrow \alpha \rightarrow \beta \rightarrow \beta \rightarrow \alpha$ in the worst case. Therefore diameter of $G(K_{2,2})^2 = 5 = 2 + 3(2/2)$. Hence the result is true for $l = 1$.

Induction hypothesis: Assume the result is true for all $l \leq n$.

$$\text{i.e diameter of } G(K_{2,2})^{2n} = 2 + 3(\tfrac{2n}{2}).$$

Induction step: Let $l = n + 1$.

Let A denote the subgraph $G(K_{k,k})^{2n}$ of $G(K_{k,k})^{2n+2}$ containing the node x and A' that of y. The worst case distance between a and b is when there is no direct edge from A to A'. Let A'' be another subgraph $G(K_{k,k})^{2n}$ of $G(K_{k,k})^{2n+2}$ which is adjacent to A as well as A'. By virtue of construction there exists such a subgraph. Now there exists a certain node z in A'' such that there is an arc of length 3 from either x or y to z. Further if z is at distance 3 from x then there is a path of length at most $D(G(K_{k,k})^{2n})$ from z to y. Similarly if z is at

distance 3 from y then there is a path of length $D(G(K_{k,k})^{2n})$ from z to x. This is due to the fact that there is direct link from each $K_{k,k}$ subgraph H_p in $A(A')$ to the subgraph H_p'' in A'' for $1 \leq p \leq n(k)^{2n}$. That is, there exists direct links from the subgraph say, G_j'' containing node z in A'' to the subgraph say, G_i in $A(A')$ containing node $x(y)$. However by construction there are edges from the $K_{k,k}$ subgraphs in the subgraph G_i containing node $x(y)$ to another subgraph G_j in $A(A')$. Hence the maximum distance from vertex $x(y)$ to vertex z is the diameter of graph $G(K_{k,k})^{2n}$. Hence the maximum distance (diameter),

$$\begin{aligned} D(x,y) &= \text{diameter of } G(K_{k,k})^{2n} + 3 \\ &= \{2 + 3(\tfrac{2n}{2})\} + 3 \\ &= 2 + 3(\tfrac{2n+2}{2}) \\ &= 2 + 3(\tfrac{2(n+1)}{2}) \end{aligned}$$

Hence the result is true for $l = n + 1$.

case 2: $i = 2l - 1$

For $l = 1$, the result holds, since by theorem 3.1, diameter of

$G(K_{k,k})^1 = 4 = 4 + 3(\frac{2\times1-1-1}{2})$.

Induction hypothesis: Assume the result holds for $l = n$. That is, diameter of $G(K_{k,k})^{2n-1} = 4 + 3(\frac{2n-1-1}{2})$.

Induction step: Let $l = n + 1$.

Let A denote the subgraph $G(K_{k,k})^{2n-1}$ which contains the node x and A', that which contains node y. Consider the worst case situation where there exists no direct link from A to A'. Let A'' be another subgraph $G(K_{k,k})^{2n-1}$ which has direct links to both A and A'. By the same reasons as mentioned in the induction step for the even case there exists a node z in A' which is at distance 3 from either x or y and at distance $D(G(K_{k,k})^{2n-1})$ from y or x. This being the worst case, we have

$$\begin{aligned} \text{diameter of } G(K_{k,k})^{2(n+1)-1} &= \text{diameter of } G(K_{k,k})^{2n-1} + 3 \\ &= 4 + 3(\tfrac{2n-2}{2}) + 3 \\ &= 4 + 3(\tfrac{2(n+1)-1-1}{2}). \end{aligned}$$

Thus the result is true for $l = n + 1$. Hence the result.

Let us now compare the $G(K_{k,k})^i$ graphs for different values of k and i with respect to the (Δ, D) graph problem. For $k = 2$, we have $n(k) = 3$. With $i = 2$, the value of $\Delta = 2k-1 = 3$, $D = 5$ and $N(\Delta, D) = 36$. For the same value of i but $k = 3$ we have $\Delta = 5$, $D = 5$ and $N(\Delta, D) = 2kn(k)^2 = 294$. However we could construct a graph with 324 nodes, degree 4, and diameter 8 using $k = 2$ but $i = 4$. This graph has more modularity as well as lesser degree compared to the $N(5,5)$ graph constructed by using $k = 3$. At this point note that a torus with 324 nodes also has a degree of 4. However, it is of diameter 18. As the value of k and i increases this difference in structural modularity as well as degree becomes more and more pronounced. Keeping the degree of individual nodes in a communication network to a minimum is of great importance. At the same time the network should be able to connect quite a large number of nodes to meet the present day need for computing power. In this context, the model $G(K_{2,2})^i$ seems to be the best choice among the above family of graphs. Hence we choose the special case of $(K_{2,2}, 3, 3, 2, 2)^i$ for our further studies.

Theorem 3.6. *The graph $G(K_{2,2})^i$ has the properties:*

1. *There are $4(3)^i$ vertices.*
2. *The mindegree δ is $2 + \lfloor i/2 \rfloor$ and the maxdegree Δ is $2 + \lfloor i + 1/2 \rfloor$.*
3. *For even values of i, the graph $G(K_{2,2})^i$ is regular.*

proof: Proof of (1) is by induction.

basis: $i = 1$.

The result holds in this case since by theorem 3 1, there are $2kn(k) = 4(3)$ nodes in $G(K_{2,2})^1$.

Induction hypothesis: Assume the result is true for all $i \leq n$. *i.e.*, Number of vertices in $G(K_{2,2})^n = 4(3)^n$.

Induction step: Let $i = n + 1$.

By construction of $G(K_{2,2})^i$, $G(K_{2,2})^{n+1}$ is made up of three copies of $G(K_{2,2})n$. Therefore the number of vertices in $G(K_{2,2})^{n+1}$ = Number of vertices in $G(K_{2,2})^n * 3 = 4(3)^{n+1}$. Hence the result.

Proof of (2) is again by induction.

basis: $i = 1$.

The result holds for $i = 1$ since the mindegree of $\delta = k = 2 = 2 + \lfloor 1/2 \rfloor$ (by theorem 3.1) and the maxdegree $\Delta = 2k - 1 = 3 = 2 + \lfloor \frac{1+1}{2} \rfloor$.

Induction hypothesis: Assume the result is true for all $i \leq n$. *i.e.*, $\delta = 2 + \lfloor n/2 \rfloor$ and $\Delta = 2 + \lfloor \frac{n+1}{2} \rfloor$ for $G(K_{2,2})^n$.

Induction step: Let $i = n + 1$.

Note that, by construction, if α-nodes were involved in the construction of $G(K_{2,2})^n$ then β-nodes are involved in the construction of $G(K_{2,2})^{n+1}$. That is, if α-nodes were of degree $2 + \lfloor n/2 \rfloor$ in $G(K_{2,2})^n$, then each of the α-nodes will have $k - 1$ new edges incident to it in $G(K_{2,2})^{n+1}$ whereas the degree of β-nodes remain at $2 + \lfloor \frac{n+1}{2} \rfloor$.

$$\begin{aligned} \delta &= 2 + \lfloor \tfrac{n+1}{2} \rfloor \qquad \text{and} \\ \Delta &= 2 + \lfloor \tfrac{n}{2} \rfloor + k - 1 = 2 + \lfloor \tfrac{n}{2} \rfloor + 1 \\ &= 2 + \lfloor \tfrac{n+2}{2} \rfloor. \end{aligned}$$

Hence the result holds for $i = n + 1$ and hence the result (2). Since $\delta = \Delta$ for all even values of i, part 3 follows.

Network	Order	Degree	Diameter
Hypercube	4	2	2
	8	3	3
	16	4	4
	32	5	5
	1024	10	10
	2048	11	11
	4096	12	12
	8192	13	13
	65536	16	16
SRG	4	4	2
	16	4,8	4,2
	32	4	5
	1024	4,8,16	10,5,2
	2048	4	11
	4096	4,8,16,32,128	12,6,4,3,2
	65536	4,8,32,512	16,8,4,2
$G(K_{2,2})^i$	4	2	2
	12	3	4
	36	3	5
	108	4	7
	324	4	8
	972	5	10
	2916	5	11
	8748	6	13
	26244	6	14
	78732	7	16

Table of comparison of hypercube, SRG, and $G(K_{2,2})^i$.

3.2. Routing in network $G(K_{2,2})^s$

It is the function of any message delivery system to find a route along which to send each message from the originator, the source to its destination. If the same message is not passed more than once to any intermediate station en route then the route is a simple path. If the route is known beforehand then it can be appended along with the message, thus permitting intermediate stations to send the message on by looking at the routing information available to it. The selection of the route for a specific node pair is done by means of a routing function ρ. In most cases the routing function ρ is selected in such a way that the resulting route is a minimal length path between the source and destination But in cases such as that of the spokes graph where there are certain nodes which are of much larger degree than the rest of the nodes, the selection of routing function ρ based on the minimal route length alone seems to be insufficient.

The scheme in which the choice of route is based on apriory knowledge of routing information for all possible (source,destination) pairs makes use of a routing table. The routing table is maintained at each individual station and this adds to the space complexity of the whole system. Since computation of the entire route for all node pairs is quite time consuming, this kind of routing is generally followed in networks which have a static topology or networks where reconfigurations are very rare. Any rerouting in the event of failures or chronic congestion in certain sections of the network needs elaborate recomputing of the new routes. This problem along with the inefficient use of valuable memory space has lead to decentralized or distributed routing schemes. However, a table based routing scheme would be preferable for very high speed high bandwidth communication network.

In a distributed routing scheme, elaborate routing tables are not maintained. The absence of a well defined route directory calls for a well coordinated functioning among the individual stations so as to ensure that the messages originated at a particular node eventually reaches its destination without looping en route. This in turn calls for a suitable labelling scheme for the individual nodes. The labels should be able to convey as clear a picture of the network topology as possible.

Furthermore the individual labels should bear a certain relation among themselves such as to provide information regarding the pattern of connection among the nodes. With such a labelling scheme and some minimum routing informations provided along with the message to be routed, the routing task is much simplified. A route between any specified pair of nodes could be established by suitably transforming the labels of the source node such that each step of transformation yields an internal node and eventually the destination is reached.

We are interested in a dynamic distributed routing scheme. Further, we would prefer providing minimal global information to individual nodes. Each station would be provided with general information such as the network size and local information such as the labels of immediate neighbors. Informations such as the source and destination address, the route list which include the list of nodes so far visited etc will be appended along with the message. Further, individual nodes will have a limited local information regarding the status of the neighbor nodes. The routing strategy could be expressed informally as follows:

1. Node x generates a message to be sent to node y and appends the message with the address of the source and the destination; $x' \leftarrow x$.
2. Forward the message to node x'' such that edge (x', x'') is a section of the path $G_{(x,y)}$. Include x'' in the list of nodes visited by the message.
3. At node x'', check whether the destination address is x''. If not then $x' \leftarrow x''$; Repeat steps 2 and 3 until $x'' = y$.
4. Otherwise remove the message and acknowledge receipt.

From the above discussion, it can be seen that the selection of the intermediate nodes plays a significant role in deciding the route and hence the route length.

3.2.1 Labelling scheme for $G(K_{2,2})^1$

In the special case of network model $(K_{2,2}, 3, 3, 2, 2)^1$, we will use the characters R, S, U, V to represent the four nodes involved in the basic graph where U and V are α-type nodes and R and S are β-type nodes (see fig. 3.5). Each level of construction associates a certain element of the block design to these nodes. Further, the value assigned in the i^{th} level need not necesarily be the same as that of the $i-1^{st}$ level. Hence it would be desirable to include these information in the label. Thus each node is assigned the label $i_1, i_2, \ldots, i_j$ where $i_1 \in \{R, S, U, V\}$, and i_l identifies the unique copy of $(K_{2,2}, 3, 3, 2, 2)^{l-1}$ in which node $i_1, i_2, \ldots, i_j$ belonged to at the l^{th} level of construction. For example, the label R_{31} gives us the information that the corresponding node belonged to the 3^{rd} copy in level one of the construction and to the first copy in level two construction. With the convention that β-nodes (R and S) are involved in even levels of construction, it becomes clear that the above node represented one of the elements of the block $B_1 = \{1, 2\}$ in level two construction. We will further adopt the convention that U and R nodes always represent the elements in the particular block which is synonymous with the block index. Then R_{31} will represent the element 1 in the 1^{st} copy for level two construction. Similarly U_{22} will represent the element 2 which belongs to block B_2 in level one construction. Fig. 3.5.(c) shows the complete labelling scheme of graph $G(K_{2,2})^2$.

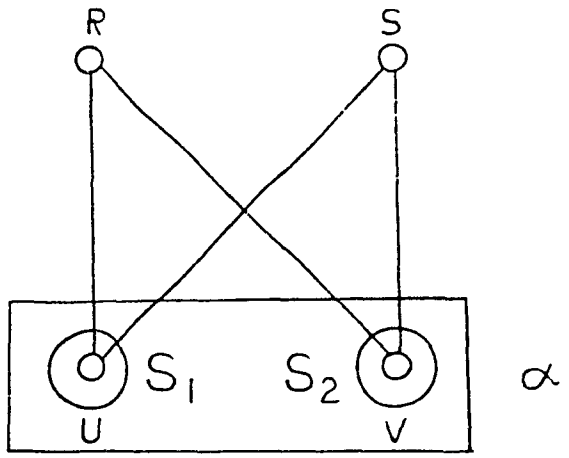


fig. 3.5.(a) Basic graph $K_{2,2}$

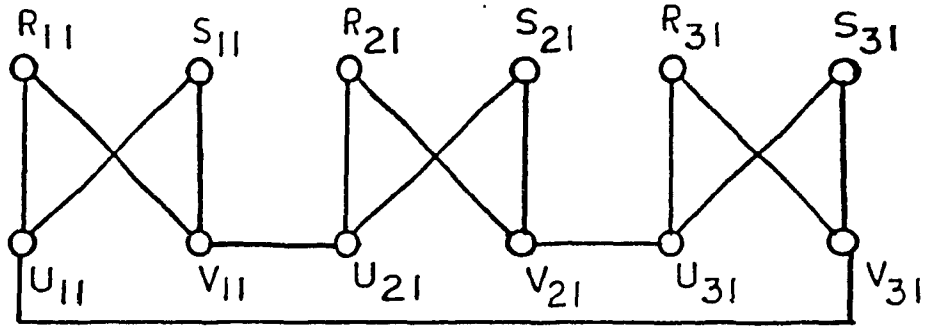


fig. 3.5.(*b*) $(K_{2,2}, 3, 3, 2, 2)^1$

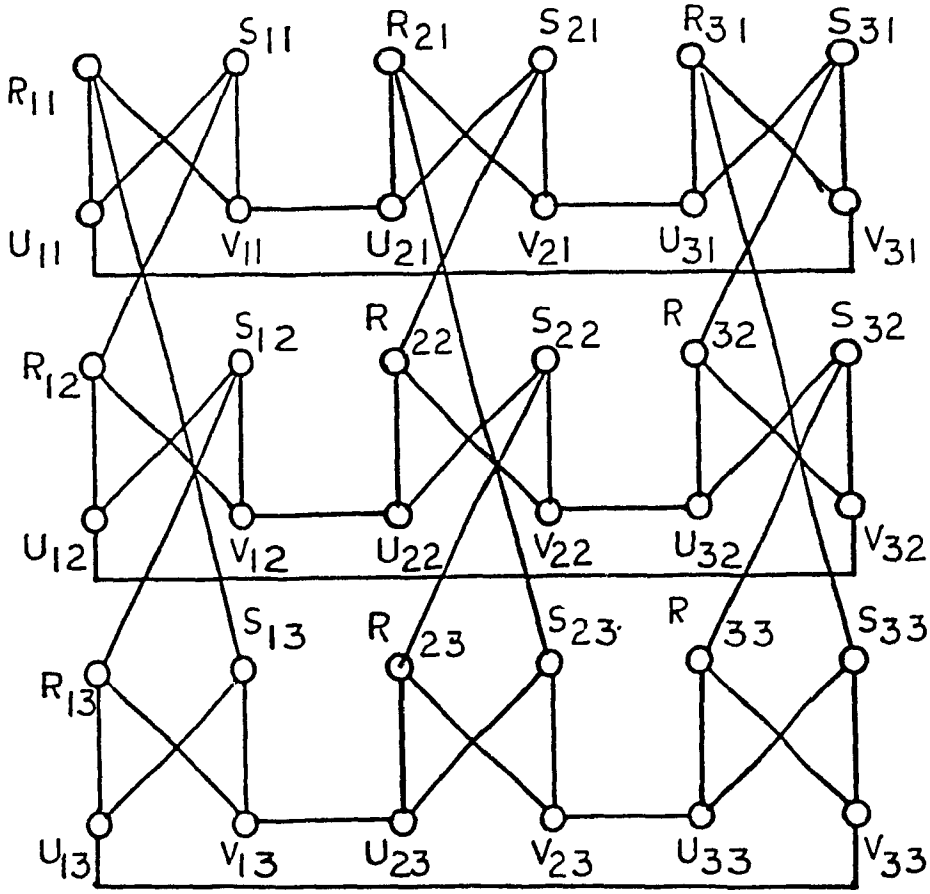


fig. 3.5.(*c*) $(K_{2,2}, 3, 3, 2, 2)^2$

Let $i = (i_1, i_2, \ldots, i_m)$ and $j = (j_1, j_2, \ldots, j_m)$ be a pair of nodes in $G(K_{2,2})^{m-1}$. Now we have the following results:

1. If $i_1 \in \alpha, \beta$ and $j_1 \in \beta, \alpha$ then there exists an edge (i, j) only if $i_2 = j_2$ and $i_l \neq j_l$, for $2 < l \leq m$.
2. If $i_1 = j_1$, there does not exist an edge (i, j).
3. If $i_1 \in \alpha, \beta$ and $j_1 \in \alpha, \beta$, then the edge (i, j) exists only if for some l, where $2 \leq l \leq m, i_l \neq j_l$ and for p such that $2 \leq p \leq m$ $i_p = j_p$ for $p \neq l$. Further, if $i_1 = R(U)$ and $j_1 = S(V)$ then edge (i, j) exists if $i_l = j_l + 1$ or $i_l = j_l - 2$. Similarly if $i_1 = S(V)$ and $j_1 = R(U)$ then (i, j) exists if $i_l = j_l - 1$ or $i_l = j_l + 2$.

Having labelled the nodes as described above, we will now present a simple and efficient distributed routing strategy. Our method is based on four basic transformation schemes. Given any two nodes as source and destination, the required basic transformations can be easily determined. Further, the path length of the route established by this method is optimal in the sense that the length of the route thus obtained exceeds the distance between the nodes by a small fraction.

3.2.2 Basic Transformations

Let $P = (p_1, p_2, \ldots, p_m)$ be the source and $Q = (q_1, q_2, \ldots, q_m)$ be the destination. Recall that $p_1, q_1 \in \{R, S, U, V\}$ and $p_l, q_l \in \{1, 2, 3\}$ for $1 < l \leq m$. Define the difference vector Δ_{PQ} as $\Delta_{PQ} = (\delta_1, \delta_2, \ldots, \delta_m)$ where

$$\delta_1 = \begin{cases} 0, & \text{if } p_1 = q_1; \\ 1, & \text{if } p_1 \text{ and } q_1 \text{ are adjacent in } G(K_{2,2}); \\ 2, & \text{otherwise.} \end{cases}$$

and $\delta_l = p_l - q_l$ for $2 \leq l \leq m$. It is easy to see that $\delta_l \in \{-2, -1, 0, 1, 2\}$. Since Δ_{PQ} is a null vector only when $P = Q$, the routing process can be viewed as a sequence of steps wherein, at each step called a segment, a particular coordinate of Δ_{PQ} is made equal to zero. Now each step can be realized by means of one of the following four basic transformations.

case 1: $\delta_l \in \{+1, -2\}$ and l is even.

In this case, we first route the message to node P' in the same subgraph $K_{2,2}$ of $G(K_{2,2})^{m-1}$, such that $p'_1 = U$. Since the source and P' are in the same $K_{2,2}$, the only coordinate changed, is the first coordinate. From the construction and the labelling scheme described earlier, we know that there exists a node P'' adjacent to P' whose first coordinate $p''_1 = V$ and $p''_l - q_l = 0$, $p''_i - q_i = p'_i - q_i = p_i - q_i$ for $i = 1, 2, \ldots, l-1, l+1, \ldots, m$. We call such a step, a $U \rightarrow V$ transformation. Note that $U \rightarrow V$ transformation selects a new source P'' such that Δ_{PQ} and $\Delta_{P''Q}$ are same except for the first and the l^{th} coordinate values. Further $\Delta_{P''q}$ has its l^{th} coordinate value zero. Once P'' is selected, message is forwarded to P'' and P'' becomes the new source.

case 2: $\delta_l \in \{-1, +2\}$ and l is even; $l \neq 1$.

In this case, the message is first routed to a node P' in the same subgraph $K_{2,2}$ such that $p'_1 = V$. As in case 1, only the first coordinate is changed. From the construction, there exists a node P'' such that $p''_1 = U$ and $p''_l - q_l = 0$; $p''_i - q_i = p'_i - q_i = p_i - q_i$ for $i = 1, 2, \ldots, l-1, l+1, \ldots, m$. This step is called a $V \rightarrow U$ transformation. Once P'' is selected, message is forwarded to P'' and P'' becomes the new source.

case 3: $\delta_l \in \{+1, -2\}$, l is odd and $l \neq 1$.

We first route the message to the node P' in the same $K_{2,2}$ whose first coordinate value is R. Note that P' has an adjacent node P'' whose first coordinate value is S and $p''_l - q_l = 0$; $p''_i - q_i = p'_i - q_i = p_i - q_i$ for $i = 1, 2, \ldots, l-1, l+1, \ldots, m$. This step is called an $R \rightarrow S$ transformation. As in cases 1 and 2, message is routed to P'' and P'' becomes the new source.

case 4: $\delta \in \{-1, +2\}$, l is odd and $l \neq 1$.

This is the dual of case 3 and hence we omit detailed discussion. This procedure is called an $S \rightarrow R$ transformation.

The above four transformations enable us to change the source node successively so that when $\delta_l = 0$, for $2 \leq l \leq m$, the source and destination belong to the same $K_{2,2}$. At this stage, since $K_{2,2}$ is connected, there exists a path from the current source to the destination of length at most 2. Note that all nonzero coordinates

of Δ_{PQ} have to be made zero through basic transformations. The lack of a central controller in a distributed routing scheme puts forth the need for a certain prespecified global rule for selecting the order in which the coordinates of Δ_{PQ} are made zero. Clearly, one of the simplest of all possible orderings is the ascending order of transformation. In this case, δ_l is made zero before δ_{l+1} for $l = 2, 3, \ldots, m-1$. The descending order of transformation is defined similarly. The set of all routing algorithms which possesses a predefined order is called an order preserving algorithm. In particular, we refer to the ascending order of transformation as the order preserving algorithm for the rest of our discussion. We will consider a routing algorithm to be *based on systematic transformations* if the transformations selected at each segment are in conformance with the value of δ_k as specified in the four basic transformations. For instance, consider the case where $\delta_2 = +2$. The transformation which is in conformance with this value is $V \rightarrow U$. Before formally presenting the algorithm, we shall illustrate this routing process by an example.

Example 1. Let the source and destination be R_{1312} and S_{2321} respectively.

Let $P = [R, 1, 3, 1, 2]$ and $Q = [S, 2, 3, 2, 1]$. We shall treat P and Q as vectors and apply the order preserving algorithm to obtain the route. In this strategy, the first step is to make the second coordinates identical. Now the difference vector $\Delta_{PQ} = [2, -1, 0, -1, 1]$ implies that we need to apply a $V \rightarrow U$ transformation for the second coordinate. Clearly, there is a path of length one from $P = [R, 1, 3, 1, 2]$ to $i_2' = [V, 1, 3, 1, 2]$. Further, there is an edge between i_2' and $i_2'' = [U, 2, 3, 1, 2]$. Hence the message is routed to i_2'' and i_2'' is treated as the new source node. The difference vector is now given by $\Delta_{i_2''Q} = [1, 0, 0, -1, 1]$. Next we pick the fourth coordinate since the 3^{rd} coordinate is already zero. $\delta_4 = -1$ implies that we have to apply a $V \rightarrow U$ transformation. Since there is a path from $[U, 2, 3, 1, 2]$ to $[V, 2, 3, 1, 2]$ first, the message is send to $[V, 2, 3, 1, 2]$. From $[V, 2, 3, 1, 2]$ we route the message to $[U, 2, 3, 2, 2]$. Now the difference vector is given by $[1, 0, 0, 0, 1]$. Pick the last coordinate. Observe that by applying an $R \rightarrow S$ transformation, we can make the last coordinate zero. The resulting route is

$$R_{1312} \rightarrow V_{1312} \rightarrow U_{2312} \rightarrow R_{2312} \rightarrow V_{2312} \rightarrow U_{2322} \rightarrow R_{2322} \rightarrow S_{2321}$$

We now formally present the order preserving algorithm.

Algorithm OP {Order preserving routing algorithm }

Input: $P = [p_1, p_2, \ldots, p_m]$ { the source } and $Q = [q_1, q_2, \ldots, q_m]$ { the destination }

Output: Routing of a message from P to Q.

step 1. If $P \neq Q$ do the following steps.

step 2. { Initialization } $i_2 \leftarrow P; T_1 \leftarrow p_1$; $k \leftarrow 2$

step 3. { looping }

while $(k \leq m)$ **do**

3.1 { Compute the difference vector }

$\Delta = i_k - Q$

while $(\delta_k = 0)$ **and** $(k < m)$ **do**

$k \leftarrow k + 1$; $i_k \leftarrow i_{k-1}$

3.2 { Determine the basic transformations. }

if k is even then

if $\delta_k = -1$ or 2 **then** $H_k \leftarrow V$; $T_k \leftarrow U$

else $H_k \leftarrow U$; $T_k \leftarrow V$

else

if $\delta_k = -1$ or 2 **then** $H_k \leftarrow S$; $T_k \leftarrow R$

else $H_k \leftarrow R$; $T_k \leftarrow S$

3.3 Send the message from the current source

$i_k = [T_{k-1}, q_2, \ldots, q_{k-1}, p_k, \ldots, p_m]$ to $i'_k = [H_k, q_2, \ldots, q_{k-1}, p_k, \ldots, p_m]$.

{ Note that this involves a routing inside a $K_{2,2}$ only. }

3.4 Send the message from i'_k to the node $i''_k = [T_k, q_2, \ldots, q_k, p_{k+1}, \ldots, p_m]$

3.5 **if** $(k < m)$ **then** $i_{k+1} \leftarrow i''_k$

else send the message from $i''_m = [T_m\ q_2, \ldots, q_m]$

to the destination $[q_1, q_2, \ldots, q_m]$.

{ Note that this involves a routing inside a $K_{2,2}$ only. }

step 4. **end**

Lemma 3.1. *Algorithm OP is correct and the path length between any pair of source and destination is less than or equal to* $(2m+1)$, *where* m *is the number of coordinates of the difference vector.*

proof: In each iteration, one coordinate of the source and destination is matched. In fact, to be more specific, at the beginning of the k^{th} iteration { in step 3 } the source node is of the form $[T_{k-1}, q_2, \ldots, q_{k-1}, p_k, \ldots, p_m]$. By steps 3.3 and 3.4, we make sure that at the beginning of the $(k+1)^{st}$ iteration the message has reached the node $[T_k, q_2, \ldots, q_k, p_{k+1}, \ldots, p_m]$ for $k = 2, 3, \ldots, m$. Thus after $m-1$ steps, the source node is of the form $[T_m, q_2, \ldots, q_m]$. Now by step 3.5, the correctness follows.

It may be observed that the transformation corresponding to the second coordinate may result in a partial route of length 3. For $i = 3, 4, \ldots, m$ the partial route created has length less than or equal to 2 if $\delta_{i-1} \neq 0$ and is at most 3 otherwise. Therefore the sum of the partial route lengths generated for $i = 2, 3, \ldots, m$ { in steps 3.3 and 3.4 } is bounded by $3 + 2(m-2)$. Furthermore, step 3.5 may result in an additional partial path of length at most 2. Hence the total path length is bounded by $3 + 2(m-2) + 2 = 2m + 1$.

Example 2. Let the source be V_{1212} and the destination R_{3123}. Let $P = [V, 1, 2, 1, 2]$ and $Q = [R, 3, 1, 2, 3]$. The difference vector $\Delta_{PQ} = [1, -2, 1, -1, -1]$. The transformations for the various coordinates are given in the following table.

coordinate	δ_k	H_k	T_k
2	-2	U	V
3	$+1$	R	S
4	-1	V	U
5	-1	S	R

Table 3.1

The route based on the OP algorithm for the above transformations is $V_{1212} \rightarrow R_{1212} \rightarrow U_{1212} \rightarrow V_{3212} \rightarrow R_{3212} \rightarrow S_{3112} \rightarrow V_{3112} \rightarrow U_{3122} \rightarrow S_{3122} \rightarrow R_{3123}$. Note that the route length is 9.

Now let us consider an alternate ordering scheme for the above example. To start with, the current source is of the type V. Now the value of H_4 is also V. So we choose the fourth coordinate in the first iteration which will result in a partial route of length 1. Then the current source for iteration 2 is $[U,1,2,2,2]$. Since H_2 is U, we will pick the second coordinate for the next iteration. This yields a partial route of length 1 leaving us $[V,3,2,2,2]$ as the current source for iteration 3. Since there are no more $U \rightarrow V$ transformations needed, we have to select between an $R \rightarrow S$ and an $S \rightarrow R$ transformation. Since the destination is an R-node, we select the $R \rightarrow S$ transformation next. Thus we have the following route:
$V_{1212} \rightarrow U_{1222} \rightarrow V_{3222} \rightarrow R_{3222} \rightarrow S_{3122} \rightarrow R_{3123}$ In comparison to the route established by the order preserving algorithm, this route is much shorter with length 5. In example 2, we have seen that it is possible to reduce the route length by properly choosing the coordinates without necessarily following a predefined order. An algorithm based on these ideas is presented next

Algorithm O { opportunistic routing algorithm }

Input : $P = [p_1, p_2, \ldots, p_m]$ { the source } and $Q = [q_1, q_2, \ldots, q_m]$ { the destination }

Output : Routing of a message from P to Q.

step 1. If $P \neq Q$ do the following steps.

step 2. { Initialization } $currentsource \leftarrow P; \Delta \leftarrow P - Q$; $K = \{2,3,\ldots,m\}$

step 3. { Determine the basic transformations. }

For $k = 2$ to m do the following:

if $\delta_k = 0$ **then**

$K = K - \{k\}$

else

if k is even **then**

if $\delta_k = -1$ or 2 **then** $H_k \leftarrow V$; $T_k \leftarrow U$

else $H_k \leftarrow U$; $T_k \leftarrow V$

else

if $\delta_k = -1$ or 2 **then** $H_k \leftarrow S$; $T_k \leftarrow R$

else $H_k \leftarrow R$; $T_k \leftarrow S$

step 4.**while** K not empty **do**

4.1 { Pick the closest H_k. }
Let X be the first coordinate of the current source. Choose k such that $X = H_k$. If no such k exists then choose a k such that both X and H_k do not belong to either α or β. Break any tie by selecting the H_k which is the same as q_1. If this is not possible, choose any k; $K \leftarrow K - \{k\}$.

4.2 Route the message to the node $[H_k, \ldots, p_k, \ldots]$ in the $K_{2,2}$ containing the current source.

4.3 Route the message to the node $[T_k, \ldots, q_k, \ldots]$.

step 5. Route the message from $[T_m, q_2, \ldots, q_m]$ to $[q_1, q_2, \ldots, q_m]$.

step 6. **end**

The proof of correctness of algorithm O is similar to that of algorithm OP and hence omitted.

Lemma 3.2. *Let $\mathcal{A}$ be the class of routing algorithms based on the systematic transformation of coordinates of the difference vector Δ_{PQ}. Let $l(\gamma, P, Q)$ denote the path length of the route from P to Q as determined by the algorithm $\gamma \in \mathcal{A}$. Then we have $l(O, P, Q) \leq l(\gamma, P, Q)$ for all γ, P, Q.*

proof: Each route is computed in $(m-1)$ iterations, where m is the length of the label. In each iteration, a partial route is generated. A partial route has two components, an edge e_i as determined by the basic transformation selected and a path t_i leading from the current source to the head node of e_i. Any algorithm in $\mathcal{A}$ has to perform the same number of iterations for the pair (P, Q). Further, in each iteration, an edge e_i is always generated. Therefore the only factor that decides the partial path length during the i^{th} iteration is the length of t_i. Since the algorithm O picks the smallest t_i at each iteration, the partial path generated is always kept minimum. Therefore the length of the path calcu ated by the algorithm O is always less than or equal to the length of the path determined by any other algorithm $\gamma \in \mathcal{A}$.

3.2.3 Comparison of route length generated by opportunistic algorithm to the diameter estimate

Let $P = [p_1, p_2, \ldots, p_{m+1}]$ and $Q = [q_1, q_2, \ldots, q_{m+1}]$ be the source and destination nodes respectively and $\Delta_{PQ} = [\delta_1, \delta_2, \ldots, \delta_{m+1}]$ the corresponding difference vector. Let (H_k, T_k) represent the basic transformation corresponding to the k^{th} coordinate. Define $|H_X|$ as the number of occurence of X as H_k for $2 \leq k \leq m+1$ and $X \in \{R, S, U, V\}$. T_X is similarly defined. Let CN denote all the node pairs (P, Q) in $V(G(K_{2,2})^m)$ such that $|T_V| = 0$ or $|T_U| = 0$ and $|T_R| = 0$ or $|T_S| = 0$, and $\delta_k \neq 0$ for $2 \leq k \leq m+1$.

Lemma 3.3. *For any pair of nodes (P, Q) in CN, the length of the route computed by the opportunistic algorithm is at most $2m+2$.*

proof: We will assume the case where $|T_V| = 0$ and $|T_R| = 0$. The proof of the other case is exactly similar. If $p_1 \in \{V, R\}$ it is quite clear that the first transformation selected by the opportunistic algorithm will result in a partial route length of 1. If $q_1 \in \{U, S\}$ then the last section of the route will also have a length of one, with no need for the additional routing with in a $K_{2,2}$. We will consider the case where $p_1 \notin \{V, R\}$ and $q_1 \notin \{U, S\}$ such that no such saving in route length is possible. Whether $p_1 = U$ or $p_1 = S$, there exists more than one selection of transformation resulting in the very first section of the route being of length 2. Since P and Q are in CN, the remaining sections will each have a partial route length of 2 (*i.e.*, of the form $T_k \rightarrow X = H_l \rightarrow T_l$ where $H_l \in \{V, R\}$ and H_l is adjacent to $[T_k, p_2, \ldots, q_k, p_{k+1}, \ldots, p_{m+1}]$). After $m-1$ such transformations, we reach some node $[Y, q_2, \ldots, q_{m+1}]$ which could be at a distance of at most two from Q.

Hence the total route length $= 2 + 2(m-1) + 2 = 2m + 2$.

From theorem 3.5, the diameter of $G(K_{2,2})^m$ is at most $2+3(m/2)$ if m is even and $4 + 3(\frac{m-1}{2})$ if m is odd. Hence for any pair of nodes in CN, the route length computed by the opportunistic algorithm would exceed the diameter by at most $(2m+2)-(2+3(m/2)) = m/2$ if m is even and $(2m+2)-(4+3(m-1)/2) = (m-1)/2$ if m is odd.

Let us now consider the case where $||H_R| - |T_R|| = l > 0$ and $||H_V| - |T_V|| = j > 0$. In this case it can be seen that there are $(|H_R| + |T_R| - l) + (|H_V| + |T_V| - j) = m - (l + j)$ sections where the resulting partial routes are of length one. That means, the route length in this case would be $(2m+2) - (m - l - j) = m + l + j + 2$. Then the route length established by the opportunistic algorithm will exceed the diameter of $G(K_{2,2})^m$ by $(m + l + j + 2 - (2 + 3m/2) = l + j - m/2$ if m is even and $(m + l + j + 2 - (4 + 3(m - 1)/2) = l + j - (m + 1)/2$ if m is odd.

3.2.4 Comparison with respect to the shortest path length

Let P, Q be any pair of nodes. $|H_V|$ and $|H_R|$ are defined as in the previous section. Let *H_set* denote an array containing the various values of H_k for $2 \leq k \leq m + 1$. *T_set* is similarly defined.

Lemma 3.4. *For any source node P and destination node Q, if*

1. $p_1 \in H\_set$ *and* $q_1 \in T\_set$;
2. $|H_V| = |T_V|$ *and* $|H_R| = |T_R|$ *then the length of the route computed by the opportunistic algorithm matches the shortest path length from P to Q.*

proof:

First, note that a shortest path from P to Q also involves transformations but not always systematic. Hence it is sufficient to show that, under the conditions of the lemma, a route based on systematic transformations is of lesser length than a route obtained by any combination of non-systematic transformations. Now, for each non-systematic transformation used in a route, a non-systematic transformation of the same type will have to be used to get the same effect as that of a systematic transformation.

Since $p_1 \in H\_set$, the very first section of the route yields the shortest possible partial route length of one for that section. Since $|H_V| = |T_V|$ and $|H_R| = |T_R|$, either $|H_R|$ or $|H_V|$ number of transformations are possible successively with each yielding least partial route length of one. Following that, with an additional edge of the form $(V \text{ or } U, R \text{ or } S)$, the next successive steps of $|H_V|$ or $|H_R|$ transformations are possible with least partial route length of one. The total number of transforma-

tions required $= 2 \cdot |H_V| + 2 \cdot |H_R|$. With no loss of generality, we will assume that $p_1 \in \{U, V\}$. If q_1 is also in $\{U, V\}$ and provided $|H_R| > 0$ we will need another edge to reach Q. The total route length in this case is $2|H_R| + 1 + 2|H_V| + 1$. If $|H_R| = 0$ we will have a route of length at most $2|H_V| + 2$. Consider any other route with at least one non-systematic transformation. It will need at least one extra edge for the route. Hence the route established by the opportunistic algorithm gives a shortest path from P to Q under the conditions of the lemma.

Example 3. Let $P = [V, 1, 1, 1, 1]$; $Q = [U, 2, 2, 3, 3]$

$\Delta_{PQ} = [2, -1, -1, -2, -2]$

$H_2 = V$; $T_2 = U$

$H_3 = S$; $T_3 = R$

$H_4 = U$; $T_4 = V$

$H_5 = R$; $T_5 = S$

$|H_V| = 1$; $|T_V| = 1$; $|H_R| = 1$; $|T_R| = 1$

Route generated by opportunistic algorithm is as follows:

$V_{1111} \xrightarrow{2} U_{2111} \xrightarrow{4} V_{2131} \rightarrow S_{2131} \xrightarrow{3} R_{2231} \xrightarrow{5} S_{2233} \rightarrow U_{2233}$ This has a length of 6. The value written over $\rightarrow$ indicates the particular coordinate which had been transformed. Now consider performing a non-systematic transformation for the third coordinate of Δ_{PQ} (*i.e.*, performing $R \rightarrow S$ instead of $S \rightarrow R$). We will have a route: $V_{1111} \rightarrow U_{2111} \rightarrow V_{2131} \rightarrow R_{2131} \rightarrow S_{2331} \rightarrow U_{2331} \rightarrow R_{2331} \rightarrow S_{2333} \rightarrow U_{2333} \rightarrow R_{2333} \rightarrow S_{2233} \rightarrow U_{2233}$. Note that this route has a length of 11 which is greater than that of the opportunistic algorithm.

We will now consider the case where $|H_R| \neq |T_R|$ and $|H_V \neq |T_V|$. Let $D_R = ||H_R| - |T_R||$ and $D_V = ||H_V| - |T_V||$.

Lemma 3.5. *The length of a route established between a pair of nodes by the opportunistic algorithm will exceed a shortest path length by at most* $\lfloor 2/3 D_R \rfloor + \lfloor 2/3 D_V \rfloor$.

proof:

case 1. $D_R + D_V = m$. That is, either $|H_R| = 0$ or $|T_R| = 0$ and either $|H_V| = 0$ or $|T_V| = 0$. With no loss of generality, we will assume that $|T_R| = 0$ and $|T_V| = 0$.

Now all but the first transformation done by the opportunistic algorithm will have a partial route length of two for each section. If $p_1 \in H_{set}$ and $q_1 \in T_{set}$ then the route length will be $2|H_R| + 2|H_V| + 2 = 2m + 2$.

Let $p_1 = U$ (i.e., $p_1 \notin H_{set}$). Then the following scheme of transformation could yield a shortest path.

$$U \overset{k/2}{\rightarrow} V \overset{2}{\rightarrow} U \overset{k/2}{\rightarrow} V \overset{4}{\rightarrow} U \overset{l/2}{\rightarrow} V \overset{m}{\rightarrow} U \overset{l/2}{\rightarrow} V \cdots$$

That is, $\lceil |H_V|/3 \rceil$ transformations are non-systematic. They need a total of $2\lceil |H_V|/3 \rceil$ edges. This combined with the $(|H_V| - \lceil |H_V|/3 \rceil)$ systematic transformations has a partial route length of $2\lceil |H_V|/3 \rceil + (|H_V| - \lceil |H_V|/3 \rceil) = |H_V| + \lceil |H_V|/3 \rceil$. Similarly the other set of remaining transformations have a route length of $|H_R| + \lceil |H_R|/3 \rceil$. In addition to this, there will be an edge linking these two partial routes and in the worst case, another edge leading to the destination. Thus a shortest path length for the above case is

$$|H_V| + \lceil |H_V|/3 \rceil + |H_R| + \lceil |H_R|/3 \rceil + 2. \tag{1}$$

Now, the equation for the opportunistic algorithm can be rewritten as

$$|H_V| + |H_R| + \lceil |H_V|/3 \rceil + (|H_V| - \lceil |H_V|/3 \rceil) + \lceil |H_R|/3 \rceil + (|H_R| - \lceil |H_R|/3 \rceil) + 2. \tag{2}$$

Subtracting (1) from (2), we have

$$\begin{aligned}(|H_V| - \lceil |H_V|/3 \rceil) + (|H_R| - \lceil |H_R|/3 \rceil) &= \lfloor 2/3|H_V| \rfloor + \lfloor 2/3|H_R| \rfloor \\ &= \lfloor 2/3 D_R \rfloor + \lfloor 2/3 D_V \rfloor.\end{aligned}$$

This completes case 1.

case 2. $||H_V| - |T_V|| = D_V > 0$ and $||H_R| - |T_R| = D_R > 0$.

In this case, the very first section of the route will have a partial route length of one. With no loss of generality we will assume that $|H_V| > |T_V|$ and $|H_R| > |T_R|$. Here the length of a route given by the opportunistic algorithm will be at most

$$2|T_V| + 2|T_R| + 2D_V + 2D_R + 2. \tag{3}$$

A shortest path will have a route length of

$2\lceil D_V/3\rceil + 2\lceil D_R/3\rceil + 2|T_V| + 2|T_R| + (D_V - \lceil D_V/3\rceil) + (D_R - \lceil D_R/3\rceil) + 2.$

That is,

$$2|T_V| + 2|T_R| + D_V + \lceil D_V/3\rceil + D_R + \lceil D_R/3\rceil + 2. \tag{4}$$

Subtracting (4) from (3), we have

$$(D_V - \lceil D_V/3\rceil) + (D_R - \lceil D_R/3\rceil) = \lfloor 2/3 D_R\rfloor + \lfloor 2/3 D_V\rfloor.$$

case 3. $D_V = 0$; $D_R = 0$.

This is similar to case 2. This completes the proof.

CHAPTER 4

NETWORK FAULT TOLERANCE AND DISTRIBUTED FAULT TOLERANT ROUTING ALGORITHM

In this chapter, we first look briefly at the ways and means of achieving fault tolerance in a multicomputer system and the importance of network fault tolerance in the system level fault tolerance. Conventional criterion for network fault tolerance is considered to be the network connectivity. Network connectivity however cannot exceed the minimum degree δ of the network model. So robustness of the network is equally important. Hence we give a set of conditions under which the network could allow faults which exceed by far the degree of the network. In fact this in a way, takes into account the total number of nodes in the network and hence reflects the network's resilience. In section 4.2 we give a distributed routing algorithm for the case when some faulty nodes occur in G^m.

4.1 Resilience of the network

The trend towards constructing computing systems incorporating many processing elements has resulted in a complex phenomenon involving reliability and fault tolerance considerations. The multiplicity of processing elements can be used to enhance the system reliability. With a good number of processing resources in the system, the failure of one or several of them could be gracefully tolerated in a way that does not seriously affect the overall system performance. But that is only one side of the issue. With several active elements in a computing system, the probability of a failure at some point in the system at any time is also non zero. As the number of processing elements increases, the probability of component failure tends to increase with it. With the growing emphasis on decentralized(distributed) control in such systems, the ways and means to deal with such failures also have become increasingly complex.

According to Kuhl and Reddy [Kuhl86], hardware fault tolerance in a computer system is achieved in one of two ways:

1. By masking the effects of faults
2. By identifying the sources of failure and then following the appropriate actions to compensate for the effects of identified failures.

In the case of masking, some kind of hardware redundancy in the form of replication and masking is employed to mask the failures. This is considered to be a reliable but expensive option. The other option works out to be cheaper but has its own nontrivial subtasks such as fault detection, fault diagnosis, reconfiguration and system recovery. The above two options are system-level fault tolerance methodologies based on the exchange of information between processing nodes of a multicomputer. The importance of reliability and fault tolerance of the underlying communication facility, the interconnection network in successful execution of message transfers cannot be overlooked.

The communication facilities generally fall into one of the following three catagories:

1. link-based
2. bus-oriented
3. connection network based

Nodes connected by discrete communication links make a link-based system. There will be several links incident on a node, linking it to neighboring nodes in the system. Communication with nodes having no direct links is achieved by routing messages to the non neighbor node over several links through several intermediate nodes. Bus oriented architecture places several processing nodes on a common shared bus. Use of connection network as inter-node communication facility allows a large number of node pairs to establish direct communication paths simultaneously by use of a switching circuit such as crossbar, multistage interconnection network etc. Robustness and reconfigurability are the two essential features the communication facility should possess to ensure system reliability and fault tolerance.

Robustness refers to the inherent redundancy of communication paths pro-

vided by the interconnection structure. This in turn is a measure of the ability of the system to maintain reliable communication between processors in the presence of failures of nodes or links. Informally speaking, reconfigurability is a measure of the degree to which the inherent redundancy can be practically exploited. The flow of information between nodes in large systems is controlled by complex routing algorithms. It is essential that the overheads associated with these routing algorithms be kept as minimum as possible to enable fast and efficient flow of information in the system. Reconfigurability is also closely linked to the level to which the simplicity, efficiency and well structured nature of a good routing algorithm is preserved in the presence of failures. Robustness on the other hand is directly related to the connectivity of the network topology. Recall that a network is modelled as a graph with the nodes being the processors and the edges, the communication links between nodes. Connectivity is also defined as the minimum number of edges or nodes that have to be removed to break a graph into two or more components. Thus it is a measure of the communication facility's ability to function in the presence of node or edge faults.

It can be seen that the network model $(K_{k,k}, n(k), n(k), k, k)'$ fits in well with the description of a link-based structure. Since the model $(K_{2,2}, 3, 3, 2, 2)'$ is the most interesting in the above family of graphs, in terms of larger order to degree ratio and fine structure, we will concentrate on this type of network topology. In chapter 3 we have seen that the minimum degree δ of $G(K_{2,2})'$ is $2 + \lfloor i/2 \rfloor$ and the maximum degree Δ is $2 + \lfloor (i+1)/2 \rfloor$. Furthermore, the network $G(K_{2,2})'$ is i-regular for even values of i. Therefore the minimum number of nodes to be removed to disconnect this network is δ. Thus presence of any $\delta - 1$ faults pause no threat of the graph getting disconnected. We will view the network fault tolerance as a measure of the number of faults the network can cope with before being disconnected. Since node failures are much more serious and common than link failures, we will consider only node failures. With δ faults in the network, there is nonzero probability of some node getting disconnected from the network, since it could be that all the δ faults fall in the neighborhood of a single healthy node say x. But so far as message routing is

concerned, this situation doesn't affect the successful message transfer between any pair of nodes in $V(G) - x$. Here $V(G)$ denotes the vertex set of $G(K_{2,2})^1$. Failure of $\delta + 1$ nodes doesn't pose any additional threat so long as $\delta > 3$.

As few as δ nodes could result in the network disconnection. But as many as $2/3^{rd}$ of the total number of nodes could fail and still the surviving network could support communication between any surviving pair of nodes. For a network with large number of nodes it is quite difficult to track down all the different ways in which an arbitrary large number of nodes failures could occur and still leave the network intact. However we could pin down quite many of these different fault distributions by a set of general conditions. But first we will give few more definitions.

Any basic graph $K_{2,2}$ in $G(K_{2,2})$ has two types of connections, one involving the α-type nodes and the other involving the β-type nodes. Then a network $G(K_{2,2})^1$ could as well be identified in terms of rows in the α-direction and β-direction respectively. Here an α-row consists of a basic graph $K_{2,2}$ and all the $K_{2,2}$ graphs to which it has α-type connections. Similarly a β-row will include any $K_{2,2}$ graph and all the $K_{2,2}$ graphs with which it has β-type connections. To make it simple we will use the term *row* for α-row and *column* for β-row with no loss of generality. In the network $G(K_{2,2})^m$, there are $n(k)^{\lfloor m/k \rfloor}$ rows and $n(k)^{\lceil m/k \rceil}$ columns. We will call a row in $G(K_{2,2})^2$, a *row_element* and a column in $G(K_{2,2})^2$ graph a *column_element* .

Consider any node x. A node y will be called its *zero_neighbor* if there is an edge (x, y) in the basic graph $K_{2,2}$ containing node x. A node y' will be called a *two_neighbor* of node x if there exists an edge (x, y') in the $G(K_{2,2})^2$ graph containing node x. Note that x and y' would either belong to the same row_element or to the same column_element. Let $\mathcal{D}$ be a class of fault distributions which satisfy the following conditions :

1. Every node in the surviving graph $G(K_{2,2})^m/F$ has at least one zero_neighbor.
2. Every node in $G(K_{2,2})^m/F$ has at least one two_neighbor which is not a zero_neighbor.
3. At most one third of the number of nodes in any $G(K_{2,2})^2$ could be faulty.

Lemma 4.1. *If $G(K_{2,2})^m$ has a fault distribution $d \in \mathcal{D}$ then there will be at most one $G(K_{2,2})^0$ in any $G(K_{2,2})^2$ with all four nodes faulty.*

proof: Proof is by contradiction.
For simplicity we will use the notations G^0 and G^2 respectively for $G(K_{2,2})^0$ and $G(K_{2,2})^2$. Let us assume there are two G^0 subgraphs in a G^2 with all four nodes faulty. Then eight nodes in these subgraphs alone will be faulty. Now each node in a G^0 is linked to some other node in a different G^0 belonging to either the same row_element or the same column_element. Considering that the two G^0 graphs under consideration belong to the same row(column)_element there are still seven more nodes belonging to seven different G^0 subgraphs which are adjacent to the above two subgraphs. That means altogether there are fifteen nodes in fault in the G^2 under consideration. This obviously violates the third condition for d to be in $\mathcal{D}$. Hence the G^2 in G^m/F cannot have two G^0 subgraphs with all four nodes faulty and we have a contradiction. Hence the lemma.

Lemma 4.2. *For a G^m with fault distribution $d \in \mathcal{D}$, every G^1 in G^m is connected*

proof: Proof is obvious since by condition 1, each G^0 in G^1 is connected and by condition 2 each G^0 in G^1 is linked to at least one other G^0 in the same G^1.

Lemma 4.3. *If G^m has a fault distribution $d \in \mathcal{D}$ then every column_element in G^m/F is connected.*

proof: Proof is similar to that of lemma 4.2 and hence omited.

Lemma 4.4. *Every G^2 in G^m/F has at least one G^1 from which there exists one or more links to each of the other G^1 in the same G^2 provided, the corresponding fault distribution belongs to $\mathcal{D}$.*

proof: Here again proof is by contradiction.
We will assume that there exists no G^2 in G^m/F which has links to each of the other G^1(row_element)s.
First we will consider the case where there exists a row_element m which has a G^0 which is fully down (all four nodes in fault). Then by condition (2) the four

neighbors of this G^0 has to be faulty, which brings the number of faults to eight. By conditions (1) and (2) at most two more nodes could be faulty in the m^{th} row. If so, the neighbors of those two nodes in the other two rows say, l^{th} and p^{th} rows will have to be faulty too taking the total number of faults to twelve. If the surviving links of row m are one each to the l^{th} and p^{th} rows, we are done (m^{th} row has links to each of the other rows and hence contradiction). If not, let us assume that the two surviving links are to row p alone. Then if p doesn't have any links to row l then there is no row_element which has links to each of the other two row_elements. However, in that case, all the three nodes in row p responsible for $p \rightarrow l$ links and their neighbors in row l will have to be faulty (by condition (2)). This would bring the total number of faults to eighteen which exceeds the limit set in condition (3). Therefore row p has links to row m as well as row l. Hence we have a contradiction and so the lemma holds.

Lemma 4.5. *Not one row_element or column_element in G^m/F could be fully down provided the fault distribution belongs to $\mathcal{D}$.*

proof: Proof follows directly from lemmas 4.1, 4.2 and 4.3.

Lemma 4.6. *For a network G^m with fault distribution as in $\mathcal{D}$, every G^2 in G^m/F has at most one connected component.*

proof: Proof is evident from lemma 4.1 and lemma 4.2. That is, each row_element in itself has at most one connected component. Further, there is a connected row_element in each G^2 with a minimum of one link to each of the other connected row_elements. Therefore G^2/F has at most one connected component.

Theorem 4.1. *If network G^m has a fault distribution which belongs to the class of fault distributions $\mathcal{D}$ then the surviving graph G^m/F is connected and hence reconfigurable.*

proof: By lemma 4.6 the smallest subgraph which could be a connected component of G^m/F is a G^2. Since every G^2 in G^m/F is connected in itself it is sufficient to show that any G^2 possesses at least one link to each of the other G^2 neighbor in the

α direction as well as β directon. Assume there is at least one G^2/F say G_A, which has no links to any other G^2. Now the smallest subgraph of G^m which contains G_A is a G^3. The above assumption would then mean that none of the row_elements in G_A will have a surviving edge which links that row_element to the corresponding row_elements of the G^2 neighbors in $G^3 \supset G_A - G_A$. Similarly, none of the column_elements of G_A will have surviving edges linking it to the corresponding column_elements of the G^2 neighbors in the β-direction.

We will now consider the case of G_A not having any surviving edge to each of the other G^2 neighbors in the β-direction of $G^3 \supset G_A$. We will assume there is a column_element, say the i^{th}, in G_A which has no surviving edge linking it to each of the other column_elements in the i^{th} column. By lemma 4.1 and condition (1), this could not be due to the failure of all β nodes in the i^{th} column_element of G_A. At most four β nodes in a column_element could be faulty. But this can happen only if a $K_{2,2}$ subgraph in the i^{th} column_element of G_A is fully down. Let us now consider the remaining column_elements, say the l^{th} and the p^{th}, of G_A. By lemma 4.1 there cannot be any $K_{2,2}$ subgraph which is fully down in either of these column_elements. For the l^{th} column_element to not have any edge to each of the other column_elements in column l of G^m/F, each of these column_elements should have at least four β nodes in fault which is possible only if each of them (this excludes the l^{th} column_element in G_A) has a $K_{2,2}$ subgraph which is fully down. Now two of the three column_elements of G_A do not have any links to each of the remaining column_elements in their respective columns. We are now left with the p^{th} column_element of G_A. We have seen that the respective G^2 subgraphs containing the l^{th} column cannot have any more column_element with a $K_{2,2}$ subgraph which is fully down. By conditions (1) and (2) each of the column_elements in the p^{th} column has four β nodes which are surviving, two R nodes and two S nodes each. So is the case with the p^{th} column_element of G_A. Hence there should be a minimum of one surviving edge from the p^{th} column_element of G_A to each of the other column_elements in the p^{th} column.

Assume there is a row_element, say the i^{th}, in G_A which has no links to

any other row_element in row i of $G^3 \supset G_A$. This could happen only if all three edges from the $K_{2,2}$ subgraphs in the i^{th} row_element of G_A to each of the i^{th} row_elements in $G^3 \supset G_A$ are down. By condition (2), this is possible only if at least four α-type nodes in each of the above row_elements are in fault. But by condition (1) this would not happen unless there is a $K_{2,2}$ subgraph in each of the above row_elements which is fully down. By lemma 4.1, there cannot be any more $K_{2,2}$ subgraphs in G_A or its G^2 neighbors in $G^3 \supset G_A$ which is fully down. That means, from each of the remaining row_elements in G_A, there exists a minimum of one link to each of the corresponding row_elements in the G^2 neighbors in $G^3 \supset G_A$. Hence there exists a minimum of two edges linking G_A to its G^2 neighbors in the α-direction of $G^3 \supset G_A$. Since every G^2 has this property, any G^2 has a minimum of two links to each of its G^2 neighbors in the α-direction.

Thus it is quite clear that G_A has a minimum of one surviving edge to each of its G^2 neighbor in the β-direction and a minimum of two surviving edges to each of its G^2 neighbor in the α-direction. Since every G^2 in G^m/F possesses the above property G^m/F has at most one connected component and hence is connected and thus reconfigurable.

4.2. Fault tolerant distributed routing in G^m

In a communication network with a large number of processing stations, there is always the possibility of some stations being down. This would mean that in the link-based structure we are considering, some links could be down due to either one of its end nodes or both being faulty. Hence a routing algorithm designed for the ideal faultless situation would not be sufficient to ensure reliable communication between the nodes. It is evident that there should be a mechanism to diagnose and detect the faults in the system as and when it happens. Furthermore, the information regarding the faults should be conveyed to the relevent nodes. Now, the state of a node as being faulty or fault free could be made known to all the nodes in the network. This is particularly suited for routing algorithms based on elaborate routing tables. This approach can result in undue tying up of the

communication links. Also, it is quite impossible to have the table of healthy nodes made up to date at all times. In a totally distributed routing environment, it is not necessary to broadcast the fault information to all the nodes in the network. It is just sufficient to let the state(faulty, fault-free) of a node be known to its immediate neighbors. To show the state of its neighbors, each node can at regular intervals, send a test message to each of the neighbors. If no acknowledgement is recieved from a node within a specified time, such a node is considered faulty. Now it is up to the routing algorithm to avoid routing the message through the links and nodes which are down, and at the same time, find as optimal a path as possible between the source and destination.

There are two possible strategies for a distributed fault tolerant routing. The first strategy is to use a backtracking algorithm. With this scheme, the local as well as global information available to individual nodes could be kept very minimum. The dissemination of information regarding a node's state can be limited to its adjacent nodes. One could proceed as far as possible till a node is reached, from where, there is no further links to nodes other than what has already been visited. At this juncture, the algorithm could retrace the path to some convenient node from where there exists an unexplored path. This could go on till finally the destination is reached or till there is no further path to explore. This is quite time consuming Further, extensive book-keeping must be done to avoid the possibility of cycles in the route.

The other approach is to use a non backtracking algorithm. In this type of algorithm, restricting the dissemination of fault information to the adjacent nodes alone could cut down the chances of successful message routing even when there are paths still existing in the network. This situation could arise when the fault distribution is such as to leave leaf nodes in the network. Barring this, the non backtracking algorithm is less complex.

We propose a non backtracking algorithm for the routing of message in network $G(K_{2,2})'$. It is assumed that the nodes are able to detect the faults in the adjacent nodes by regular probes. The algorithm essentially selects the internal

nodes and edges as suggested by the transformations described in chapter 3. But before selecting a certain transformation it does make sure that the corresponding nodes and hence the edges are healthy. In a faultless situation, it creates a route very similar to that of the algorithm O. But first let us look at some examples. Consider the network $G(K_{2,2})^2$ shown in figure 4.1. Here, $\otimes$ represents a faulty node and all the edges incident on such nodes are removed.

Example 1

Consider the source node as $S_{2,1}$ and the destination node as $U_{1,2}$. We have the difference vector, $\Delta = [1, 1, -1]$. this calls for an $S \rightarrow R$ transformation for the third coordinate and a $U \rightarrow V$ transformation for the second coordinate. If one were to construct a route by the algorithm O, we will have the route $S_{21} \rightarrow R_{22} \rightarrow U_{22} \rightarrow V_{12} \rightarrow R_{12} \rightarrow U_{12}$. Now the edges (S_{21}, R_{22}) and (R_{22}, U_{22}) are down. Since the node S_{21} could ascertain that the only healthy neighbor available is U_{21}, it should pick up the edge (S_{21}, U_{21}) instead of (S_{21}, R_{22}). Then we have the route $S_{21} \rightarrow U_{21} \rightarrow V_{11} \rightarrow S_{11} \rightarrow R_{12} \rightarrow U_{12}$.

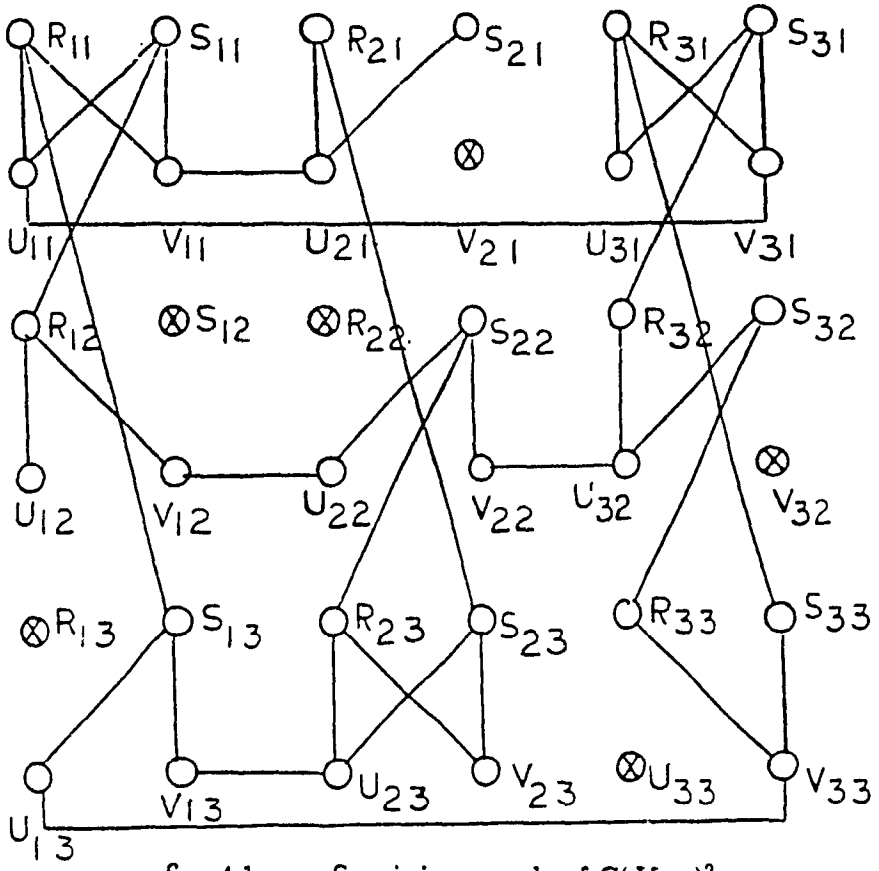


fig. 4.1 Surviving graph of $G(K_{2,2})^2$

Example 2.

Let the source node $P = [R, 2, 1]$ and the destination $Q = [U, 3, 2]$. The difference vector is $\Delta_{PQ} = [1, -1, -1]$. The transformations for the second and third coordinates are $V \rightarrow U$ and $S \rightarrow R$ respectively. From the node R_{21}, the state of V_{21} is detected as faulty. So the link (V_{21}, U_{31}) is known to be unusable and hence of course the $V \rightarrow U$ transformation is not possible at this point. But the effect of a $V \rightarrow U$ transformation is equivalent to that of two $U \rightarrow V$ transformations. Which means we could use the links (U_{21}, V_{11}) and (U_{11}, V_{31}) instead of (V_{21}, U_{31}). From R_{21}, U_{21} is closer compared to S_{21}. That is, we could pick $P' = U_{21}$ and $P'' = V_{11}$. With P'' as the new source, the difference vector $\Delta_{P''Q} = [2, -2, -1]$. The required transformations are a $U \rightarrow V$ for coordinate 2 and $S \rightarrow R$ for coordinate 3. S_{11} being closest to V_{11}, we pick $P' = S_{11}$ and $P'' = R_{12}$. From R_{12} we have two possible options to pick, either U_{12} or V_{12}. But since the difference vector $\Delta_{PQ} = [1, -2, 0]$ which means a $U \rightarrow V$ transformation is the best choice, we will pick the node

U_{12}. But once node U_{12} is reached, we find that both V_{32} and S_{12} are faulty. With no provision for backtracking, this would mean the message would not be routed further. The partial route thus far being $R_{21} \rightarrow U_{21} \rightarrow V_{11} \rightarrow S_{11} \rightarrow R_{12} \rightarrow U_{12}$. That is, with still more than one path existing in the network, the algorithm could fail to find a path. This could be avoided to a great extent by allowing the node being probed to pass on the information regarding its immediate neighbor. That is, while at node R_{12}, had we known that the link (U_{12}, V_{32}) required for the $U \rightarrow V$ transformation is down by letting the node U_{12} to inform R_{12} of the state of V_{32}, we could have picked the alternate path.

It is desirable that the routing algorithm possess the following features:

1. It is optimum or at least near optimum.
2. The message is not caught in a cycle that could prevent it from reaching the ultimate destination.
3. The algorithm doesn't give up the search for a route too quickly.

The first of the above features could be met by picking the transformations according to algorithm O. The second feature could be achieved by appending information about the nodes visited, to the message and thus enabling the choice of a node as close as possible to the current source and at the same time not already traversed. The third feature could be best accomplished by a backtracking algorithm, which has other disadvantages as well. In a nonbacktracking algorithm, the third feature could be achieved by avoiding leaf-nodes enroute. Features 2 and 3 are contradictory and hence a suitable balance must be chosen. We will try to avoid leaf-nodes by the following approaches:

1. Substitution
2. Look-ahead

Substitution: Consider the nodes S_{2312} and V_{3123}. If we were to set up a path from S_{2312} to V_{3123} a near optimum solution is to follow the transformations indicated by the coordinates of the difference vector $[1, -1, 2, -1, -1]$ given in the table below

coordinate	transfo mation
2	$V \rightarrow U$
3	$S \rightarrow R$
4	$V \rightarrow U$
5	$S \rightarrow R$

which gives a possible route $S_{2312} \rightarrow R_{2313} \rightarrow V_{2313} \rightarrow U_{2323} \rightarrow S_{2323} \rightarrow R_{2123} \rightarrow V_{2123} \rightarrow U_{3123} \rightarrow R_{3123} \rightarrow V_{3123}$. But due to the construction of the network we can break up one transformation of the form $X \rightarrow Y$ into two half-transformations of the form $Y \rightarrow X$, with the two half operations being not necessarily contiguous. This would have the same effect in nullifying a coordinate of the difference vector as the single operation, with obviously different partial route lengths. We call this a substitution. This could result in a shorter route length as in the case of the above pair of nodes. For instance, let us substitute for the fourth coordinate's transformation. We then have the following route:
$S_{2312} \xrightarrow{T_5} R_{2313} \rightarrow U_{2313} \xrightarrow{T_4/2} V_{2333} \xrightarrow{T_2} U_{3333} \xrightarrow{T_4/2} V_{3323} \rightarrow S_{3323} \xrightarrow{T_3} R_{3123} \rightarrow V_{3123}$. In a fault prone network, such a substitution can lead us to an existing path since the likelihood of both the nodes of the α or β group getting disconnected is much lesser compared to that of a single node getting disconnected.

Look-ahead is aimed at making sure that a selected transformation is indeed possible. Let $N(X)$ denote the list of neighbors of node X. Let the current source be $P'' = [p''_1, p''_2, \ldots, p''_{m+1}]$. Let head[$k$] denote $[H_k, p''_2, \ldots, p''_{m+1}]$ and tail[k] denote $[T_k, p''_2, \ldots, q_k, p''_{k+1}, \ldots, p''_{m+1}]$. If $head[k] \in N(P'')$ then the link (head[k],tail[k]) will be selected as a possible transformation only if head[k] as well as tail[k] are not faulty. With both substitution and look-ahead permitted, we will have the following route established for the pair of nodes in example 2 which was otherwise not possible: $R_{21} \rightarrow U_{21} \rightarrow V_{11} \rightarrow S_{11} \rightarrow R_{12} \rightarrow V_{12} \rightarrow U_{22} \rightarrow S_{22} \rightarrow V_{22} \rightarrow U_{32}$. The route length is 9 which is one short of double the nominal route length, which in this case is the diameter of the network $G(K_{2,2})^2$.

Before presenting the algorithm, we will introduce a few terms and notations

that are used in the algorithm.

$N(P)$:	The set of immediate neighbors of some node P.
P'' :	The current source, $[p_1'', p_2'', \ldots, p_{m+1}'']$.
Zero-neighbor of P :	The set of neighbors of node P which belong to the same $K_{2,2}$ as P.
current_loc :	The node at which the message is currently resident.
basic_transform :	The procedure that determines the basic transformations.
(H_k, T_k) :	The transformation called for by the k^{th} coordinate of the difference vector. It also represents the edge (head[k],tail[k]).
head[k] :	A node in the same $K_{2,2}$ as the current source with first coordinate H_k, i.e $[H_k, p_2'', \ldots, p_{m+1}'']$.
tail[k] .	The node $[T_k, p_2'', \ldots, q_k, p_{k+1}'', \ldots, p_{m+1}'']$.
bad_links :	The set of all transformations (H_k, T_k) for which head[k] is a neighbor of P'' and for which head[k] or tail[k] or both are faulty or marked visited.
substitute bad_links :	A procedure which performs substitutions for the non-usable bad_links and then assigns the weight maxint to those transformations which are still impossible.
weight[k] :	The distance from P'' to head[k] which is zero if $P'' = head[k]$, one if P'' and head[k] are zero-neighbors, two if P'' and head[k] belong to the same $K_{2,2}$, and maxint if head[k] or tail[k] is faulty.
Noroute :	A boolean variable which will be true if further routing is impossible.
k_{min} :	The coordinate whose weight is minimal among all $k \in K$.

At each intermediate node, the node is marked visited and a copy of the message is forwarded to the next node. A route_list is maintained which keeps track of the nodes so far visited.

Algorithm FT { Fault-tolerant routing algorithm }

Input : $P = [p_1, p_2, \ldots, p_{m+1}]$, the source and $Q = [q_1, q_2, \ldots, q_{m+1}]$, the destination.

Output :

A route from node P to node Q if possible or else a partial route and a message of "Further routing impossible."

step 1. If $P \neq Q$ do the following.

step 2. { initialization }

current_loc $\leftarrow P$; bad_links $\leftarrow \emptyset$;

$K \leftarrow \{2, 3, \ldots, m+1\}$; noroute $\leftarrow$ false;

step 3. { looping }

while K not empty **and** noroute = false **do**

3.1 { Compute the difference vector.}

$P'' \leftarrow$ current_loc;

$\Delta = P'' - Q$;

failure $\leftarrow$ false; $K \leftarrow K$ - $\{k | \delta_k = 0\}$;

3.2 { Determine the basic transformations. }

basic_transform(Δ);

for $k \in K$ **do**

head[k] $\leftarrow [H_k, p''_2, \ldots, p''_{m+1}]$;

tail[k] $\leftarrow [T_k, p''_2, \ldots, q_k, p''_{k+1}, \cdots, p''_{m+1}]$;

3.3 { locate the bad_links. }

For head[k] $\in N(P'')$ and $k \in K$ **do**

if head[k] or tail[k] faulty or n arked visited **then**

bad_links $\leftarrow$ bad_links $\cup(head[k], tail[k])$

3.4 substitute bad_links;

3.5 { Assign weights to the possible transformations. }

for $k \in K$ **do**

if head[k] faulty or marked visited **then**

weight[k] $\leftarrow$ maxint

else if weight[k] $\neq$ maxint **then**

if $head[k] = P''$ **then** $weight[k] \leftarrow 0$

else if (head[k] of type α and P'' of type β)

or (head[k] of type β and P'' of type α) **then**

weight[k] $\leftarrow 1$

else weight[k] $\leftarrow 2$

3.6 choose k_{min}

3.7 **if** weight[k_{min}] ≤ 1 **then**

{P' is either the current source or its zero_neighbor.}

forward the message to tail[k] via head[k];

mark P'' and head[k] as visited;

current_loc $\leftarrow$ tail[k];

$K \leftarrow K - \{k\}$

else if $weight[k_{min}] = 2$ **then**

{P' is a two_neighbor but not a zero_neighbor of current_loc. }

if there exists a zero-neighbor $\hat{P}$ of P''

which is not faulty and not visited **then**

mark P'' as visited;

forward the message to head[k] via $\hat{P}$;

current_loc $\leftarrow head[k]$;

if tail[k] not faulty and not visited **then**

forward the message to tail[k];

mark head[k] as visited;

current_loc $\leftarrow tail[k]$; $K \leftarrow K - \{k\}$

else failure $\leftarrow true$;

if failure= true or $weight[k_{min}] = maxint$ **then**

{P' is any other healthy non-visited neighbor of current_loc. }

if there exists $\check{P}$, a non-faulty non-visited neighbor of current_loc **then**

mark current_loc as visited;

forward the message to $\check{P}$;

current_loc $\leftarrow \check{P}$

else noroute $\leftarrow$ *true*;

step 4. **if** current_loc $\neq Q$ and current_loc $\notin N(Q)$ **then**

{ current_loc is in the destination basic block. }

if there exists $\hat{Q}$, a non-faulty and non-visited neighbor of current_loc **then**

forward the message to Q via $\hat{Q}$;

mark $\hat{Q}$ and Q as visited

else noroute $\leftarrow$ true;

step 5. **if** noroute = true **then** further routing impossible

else successful completion of task.

end

procedure basic_transform(Δ)

input : The difference vector Δ

output: The values of H_k and T_k for $k \in K$

begin

for $k \in K$ **do**

if k is even **then**

if $\delta_k = -1$ or 2 **then** $H_k \leftarrow V$; $T_k \leftarrow U$

else $H_k \leftarrow U$; $T_k \leftarrow V$

else if $\delta_k = -1$ or 2 then $H_k \leftarrow S$;$T_k \leftarrow R$

else $H_k \leftarrow R$; $T_k \leftarrow S$

end

```
procedure substitute (bad_links)
input : The set of bad_links
output: The set of links which substitute the bad_links.
begin
        for e in bad_links do
        begin
            head[k] ← [T_k, p''_2, ..., p''_{m+1}];
            tail[k] ← [H_k, p''_2, ρ_k, p''_{k+1}, ..., p''_{m+1}]
            { Here ρ_k stands for the k^th index of the label of the
            neighbor of node [T_k, p''_2, ..., p''_{m+1}]
            if head[k]or tail[k] faulty or visited then
                    weight[k] ← maxint;
        end
end
```

4.2.1 On the performance of algorithm FT

In the event of no faults, the algorithm FT generates a route similar to that of the opportunistic algorithm. We will view the performance of the algorithm in the light of its success in tracking down an existing route. First we will examine the situation when the fault distribution is of type $\mathcal{D}$ mentioned in section 4.1. In this case, it can be seen that pendant vertices are absent in the surviving graph G^m. Further, due to conditions (1) and (2) mentioned in section 4.1, there exists at least two vertex-disjoint paths between every pair of vertices in G^m/F for $m > 1$.

Lemma 4.7. *The routing algorithm FT will succeed in finding an existing simple path between any pair of healthy nodes in the surviving graph G^m/F if the fault distribution belongs to $\mathcal{D}$ (from section 4.1).*

proof: First of all, the surviving graph G^m/F is a connected graph (by Theorem 4.1). The algorithm can fail in tracking down an existing path in one of two ways:

1. It reaches a pendant vertex from where there is no way out.

2. It reaches a certain vertex whose surviving neighbors have all been previously visited.

Scenario (1) cannot happen in this case since there are no pendant vertices in G^m/F. Scenario (2) could happen in the following way. In a certain $K_{2,2}$ say G_A, a transformation H_k of weight 2 is performed. In the course of further routing, due to faults enroute, we are forced to come back to a node say x in G_A by some substitution. At this point x has no more non-visited neighbors ir G_A. If there are no other surviving neighbors for x then further routing is impossible. However, the algorithm FT always selects the transformation with the least possible weight. Every surviving node has a surviving zero_neighbor as well as a surviving two_neighbor which is not a zero_neighbor. As a direct consequence of the above, selection of a transformation with weight < 1 will not take place, unless the source and destination belong to the same row or column. In such a case, by lemma 4.2 and 4.3 as well as due to the existence of two or more vertex-disjoint paths between every pair of nodes, scenario (2) cannot happen. Thus none of the situations leading to the failure of the algorithm in finding an existing route can occur if the fault distribution belongs to $\mathcal{D}$. Note also that the algorithm always avoids visiting a previously visited node. Hence the lemma.

We will now consider another type of fault distribution. Let $\mathcal{D}'$ be any class of fault distribution such that

(a). Every column_element in G^m/F has at least one $K_{2,2}$ subgraph with surviving links to each of the other $K_{2,2}$ subgraphs in the same column_element.

(b). Every row_element in G^m/F has at least one $K_{2,2}$ subgraph with surviving links to each of the other $K_{2,2}$ subgraphs in the same row_element.

Lemma 4.8. *If G^m has a fault distribution $d' \in \mathcal{D}'$, then*

1. *Every $K_{2,2}$ subgraph will be connected in itself.*
2. *There will be no $K_{2,2}$ subgraph with both α or both β type nodes in fault.*
3. Graph G^m/F is connected.

proof: Proof of (1) and (2) is obvious from (a) and (b). Proof of (3) is similar to that of Theorem 4.1 and is hence omitted.

Claim 4.1. *For a fault distribution $d' \in \mathcal{D}'$, algorithm FT will succeed in finding a route between any pair of healthy nodes in G''' / F.*

This claim is based on the following observations:

1. Algorithm FT always selects the transformation with the least weight.
2. Even if a transformation is not possible due to faulty nodes either
   a) It's substitution will be possible **or**
   (b) Another transformation with similar or lesser weight will be possible **or**
   (c) A non-visited non-faulty neighbor $\check{P}$ will be available.

Observation (2) stems from the fact that between each column_element in a column there exists a minimum of one edge and between each row_element in a row there exists a minimum of one edge.

CHAPTER 5

CONCLUSION

We have discussed in the previous chapters, different aspects of interconnection network $G(K_{2,2})^t$. Interconnection network plays a very significant role in any large distributed system. Quite many improvements are possible over the existing networks such as cube, cube-connected cycles etc. Maintaining the degree of a node to a minimum is of great practical interest in network design. At the same time, minimizing the delay and distortion is extremely important in the efficient operation of the network. With more and more communicating processors in present day systems, there is increasing stress on the design of gracefully degradable systems. We have seen some of the network designs aimed at achieving the above requirements in chapter 2. It was also seen that the network models based on balanced incomplete block design, in addition to meeting the above requirements gives surviving route graphs of diameter 2 for a minimal routing ρ.

Chapter 3 and chapter 4 contains the major contribution of this thesis. In chapter 3, we studied some of the significant properties of network model $G(K_{k,k})^t$. We then showed that the network $G(K_{2,2})^t$ is the most interesting of the above family of graphs in terms of better *order-to-degree* ratio and modular structure. A labelling scheme for $G(K_{2,2})^t$ was proposed. Four basic transformations such as $U \rightarrow V$, $V \rightarrow U$, $S \rightarrow R$ and $R \rightarrow S$ were then defined. Based on these transformations, a class of distributed routing algorithms called *order preserving* (OP) was presented. It was shown that the length of the route computed by the OP algorithm for a network $G(K_{2,2})^{m-1}$ is at most $2m + 1$. An improved version of the above algorithm known as algorithm O was then presented. It was proved that the length of the route computed by algorithm O is optimal among the class of routing algorithms based on *systematic transformation*. It was also proved that the maximal length of the route computed by such an algorithm is $2m + 1$ for the

graph $G(K_{2,2})^{m-1}$. In fact, m is equal to the mindegree δ of $G(K_{2,2})^m$. We also proved that the diameter of $G(K_{2,2})^m$ is equal to $2 + 3(m/2)$ for even values of m and $4 + 3(\frac{m-1}{2})$ for odd values of m. We also proved that there are $4(3)^m$ vertices in graph $G(K_{2,2})^m$. Furthermore, it was shown that the mindegree δ of $G(K_{2,2})^m$ is $2 + \lfloor m/2 \rfloor$ and the maxdegree Δ is $2 + \lfloor \frac{m+1}{2} \rfloor$. A comparison of the route-length computed by the opportunistic algorithm with respect to the diameter of the graph $G(K_{2,2})^m$ as well as the shortest path length between any source and destination was also done.

In chapter 4, we first saw the general techniques of achieving fault tolerance in a system. In the context of network fault tolerance, the connectivity of the network is the conventional criteria for the measure of fault tolerance. However, in networks with high order to degree ratio, this doesn't take into account the significantly large number of nodes in the system.

Hence we explored the robustness of $G(K_{2,2})^t$ network under a set of conditions. These conditions accomodate quite many fault distributions with as much as one-third the number of vertices of $G(K_{2,2})^t$ in fault. We did prove that the network has at most one connected component for any fault distribution satisfying the above conditions. A distributed fault tolerant routing algorithm FT was then presented. Since the algorithm does substitutions for the basic transformations, when confronted with failed nodes, it is a non-systematic algorithm. No backtracking is involved. The algorithm assumes status fed back from the adjacent nodes as well as their immediate neighbors of a node.

REFERENCES

[AKER65] S. B. Akers, Jr, "On the construction of (d,k) graphs", *IEEE Trans. Electron comput.*, vol. EC-14, p. 488, June 1965.

[ARDEN78] B. W. Arden and H. Lee, "A multi-tree structured network", *in Proc. COMPCON 78*, pp. 201–210, Sept. 1978.

[AVIZ76] A. Avizienis, "Fault-Tolerant Systems", *IEEE Trans. Comput.*, vol. C-25, pp. 1304–1312.

[BEAUD78] M. D. Beaudry, "Performance-Related Reliability Measures for Computing Systems", *IEEE Trans. Comput.*, vol. C-27, pp. 540–547.

[BERM82] J. C. Bermond, C .Delorme, and J. J. Quisquater, "Tables of large graphs with given degree and diameter", *Inf. Proc. Lett.*, vol 15, no. 1, pp. 10–13, 1982.

[BRODS84] A. Broder, M. Fischer, D. Dolev, and B. Simons, "Efficient fault tolerant routings in networks." in *Proc. 16th Annual ACM Symp. on Theory of Computing*, pp. 536–541, 1984.

[CHEN90] M. Y. Chen, K. G. Shin, and D. D. Kandlur, "Addressing, Routing, and Broadcasting in Hexagonal mesh Multiprocessors", *IEEE Trans Comput.*, vol. C-39, pp. 10–18, January 1990.

[DOLEV84] D. Dolev, J. Halpern, B. Simons, and R. Strong, "A New Look at Fault-Tolerant network routing", *Proc. of Sixteenth Annual ACM Symp. on Theory of Computing*, pp. 526–535, 1984.

[DUTT88] S. Dutt and J. P. Hayes, "Design and Reconfiguration Strategies for Near-optimal k-Fault-Tolerant Tree Architectures", in *The Eighteenth International Symposium on Fault-Tolerant Computing*, pp. 328–333, June 27–30, 1988.

[ESFA85] A. Esfahanian, S. L. Hakimi, " Fault-Tolerant Routing in De Bruijn Communication Networks ", *IEEE Trans. Comput.*, vol. C-34, pp. 777–788, September 1985.

[FORT85] J. Fortes and C. Raghavendra, "Gracefully degradable processor arrays", *IEEE Trans. Comput.*, vol. C-34, pp. 1033-1044, November 1985.

[HALL67] M. Hall, Jr, *Combinatorial Theory*, New York: Wiley, 1967.

[IMASE81] M. Imase and M. Itoh, "Design to minimize diameter on building-block network", *IEEE Trans. Comput.*, vol. C-30, pp. 439- 442, June 1981.

[KUHL86] J. G. Kuhl and S. M. Reddy, " Fault-Tolerance Considerations in large Multiple-Processor Systems", *IEEE COMPUTER*, pp. 56-67, March 1986.

[LOSQ76] J. Losq, "A Highly Efficient Redundancy Scheme : Self-Purging Redundancy", *IEEE Trans. Comput.*, vol. C-25, pp. 569-578, 1976.

[MEMMI82] G. Memmi and Y. Raillard, "Some new results about the (d, k) graph problem", *IEEE Trans. Comput.*, vol. C-31, pp. 784-791, 1982.

[NAJJ90] W. Najjar and J. L. Gaudiot, "Network Resilience: A measure of Network Fault Tolerance", *IEEE Trans. Comput.*, vol. C-39, pp. 174- 181, Feb 1990.

[OPAT86] J. Opatrny, N. Srinivasan, and V. S. Alagar, " Construction of Large Fault-Tolerant Communication Network models ", *Proc. Int. Symp. on Fault-Tolerant Computing*, Vienna, Austria, pp 110-116, July 1986.

[OPAT89] J. Opatrny, N. Srinivasan, and V. S. Alagar, "Highly Fault-Tolerant Communication Network Models", *IEEE Trans. on Circuits and Systems*, vol. 36, pp. 23-30, January 1989.

[ORE62] O. Ore, "Theory of graphs", *Amer. Math. Soc.*, Providence, RI, 1962.

[PRAD82] D. K. Pradhan and S. M. Reddy, "A Fault-Tolerant Communication Architecture for distributed Systems", *IEEE Trans. Comput.*, vol. C-31, pp. 863-870, September 1982.

[PRAD85] D. Pradhan, "Fault-tolerant multiprocessor link and bus network architectures", *IEEE Trans. Comput.*, vol. C-34, pp. 33-45, January 1985.

[WILK70] R. S. Wilkov, "Construction of maximally reliable communication networks with minimum transmission delay", *Proc. IEEE Int. Conf. Commun.*, vol. 6,

pp. 42-10–42-15, June 1970.

[RENN86] D. A.Rennels, "On implementing fault-tolerance in binary hypercubes", in *Proc. IEEE Symp. Fault-Tolerant Comput.*, pp. 344–349, 1986.

[RENN84] D. A.Rennels, "Fault-Tolerant Computing- Concepts and Examples", *IEEE Trans. Comput.*, vol. C-33, pp. 1116–1129, December 1984.

APPENDIX

Given below are the procedures which were used in the implementation and testing of the algorithms, order preserving, opertunistic, and FT. The main programs are also enclosed.

```
{ ------------------------------------------------------------------------}
function mth$random(var formalseed :integer) : real;
fortran;

function random : real;
begin
        random := mth$random(globalseed);
end;
{ ------------------------------------------------------------------------}
function differ(P, Q : string; len : integer): vect;
{ ------------------------------------------------------------------------}
{ Given the source P, destination Q and the length of the label used to represent
the node, this function will return the difference vector.
{ ------------------------------------------------------------------------}
var
        i : integer;
        tem : vect;
begin
        if p[1] = q[1] then tem[1] := 0
        else
        if ((p[1] in alpha) and (q[1] in beta)) or
        ((p[ 1 ] in beta) and (q[ 1 ] in alpha)) then
                tem[ 1 ] := 1
        else
                tem[ 1 ] := 2;
        for i := 2 to len do
                tem[ i ] := ord(p[ i ]) - ord(q[ i ]);
        differ := tem;
end;
 { ------------------------------------------------------------------------}
```

```
procedure make_fault ;
var
        c : real;
        numfault, i, r : integer;
begin
                writeln(' How many faults ?');
                readln(numfault);
                writeln;
                writeln(' Number of faults, not necessarily distinct = ',numfault);
                writeln;
                writeln(' The globalseed ?');
                readln(globalseed);
                writeln(' globalseed = ',globalseed);
                for i := 1 to numfault do
                begin
                        c := random;
                        r := trunc( n * random);
                        if r = 0 then r := 1;
                                writeln(' fault[', i:3,' ] = ',r);
                        vertices[ r ] ↑ . fault := true;
                end;
                writeln(' GLOBALSEED = ', GLOBALSEED);
end;
 { ------------------------------------------------------------------------------}
procedure pringraph;
var
        i, numnodes : integer;
        t : node;
begin
        writeln(' How many nodes ? ');
        read(numnodes);
        for i := 1 to numnodes do
        begin
                t := vertices[ i ];
                while t <> nil do
                begin
                        write(t↑ .index:4, '(', t↑ .name, ') ',t↑ .fault,' ', t↑ .visit);
                        t := t↑ .link;
                end;
                writeln;
        end;
end; { of procedure pringraph }
 { ------------------------------------------------------------------------------}
```

```
procedure readgr;
{ This procedure reads the input graph from the file ingraph and
stores it in the form of an adjacency list named vertices. }
var
        rear,t : node;
        i, k, s: integer;
        nodlab : string;
begin
        reset(outc);
        read(outc, n, len);
        writeln(' len = ', len);
        for i := 1 to n do
        begin
                read(outc, k, nodlab);
                new(t);
                t↑ .index := k;
                t↑ .name := nodlab;
                t↑ .link := nil;
                vertices[ k ] := t;
                rear := t;
                while not eoln(outc) do
                begin
                        read(outc, s, nodlab);
                        new(t);
                        t↑ .index := s;
                        t↑ .name := nodlab;
                        t↑ .link := nil;
                        rear↑ .link := t;
                        rear := t;
                end; {of while not eoln(outc) }
                readln(outc);
        end;
end; { of procedure readgr }
{ ---------------------------------------------------------------------- }
```

```
procedure createhead(divect : svect; n : integer; var headlist, taillist :nstring);
{ Inputs : The difference vector divect, the length of the label
Outputs: The headlist which is a one dimensional array which contains
the type of head node involved in the i position of the label,
the taillist which stores the type of tail node for the i position. }
var
     i : integer;
begin
     for i := 2 to n do
          if divect[ i ] = 0 then
          begin
               headlist[ i ] := ' ';
               taillist[ i ] := ' ';
          end
          else
          if odd(i) then
               case divect[ i ] of
                    -1, +2 : begin
                                  headlist[ i ] := 'S';
                                  taillist[ i ] := 'R';
                             end;
                    +1, -2 : begin
                                  headlist[ i ] := 'R';
                                  taillist[ i ] := 'S';
                             end;
               end
               else
               case divect[ i ] of
                    -1, 2 : begin
                                  headlist[ i ] := 'V';
                                  taillist[ i ] := 'U';
                             end;
                    +1, -2: begin
                                  headlist[ i ] := 'U';
                                  taillist[ i ] := 'V';
                             end;
               end; { of case }
               for i := 2 to n do
               begin
                    writeln;
                    writeln('headlist[', i:3, '] = ', headlist[ i ]);
                    writeln('taillist[', i:3, '] = ', taillist[ i ]);
               end;
end; { of procedure createhead }
```

```
procedure generate(divect : svect;len : integer; var h1, t1 : nany);
```

Given the difference vector, this pocedure generates two arrays H1 and T1 where H1 contains all the headnodes for the transformation of coordinates [2 .. lim] and their respective distance from the current source. Note that lim is the length of the label. Similarly T1 contains the names of the tailnodes for transformations [2 .. lim].

```
var
    i,j : integer;
begin
    for i := 2 to lim do
        if divect[ i ] <> 0 then
        begin
            h1[ i ].lab[ 1 ] := headlis[ i ];
            t1[ i].lab[1] := taillis[ i ];
        for j := 2 to lim do
        begin
            h1[ i ].lab[ j ] := first[ j ];
            t1[ i ].lab[ j ] := first[ j ];
        end;
        t1[ i ].lab[ i ] := chr(ord(t1[ i ].lab[ i ]) - divect[ i ]);
        if first[ 1 ] = h1[ i ].lab[ 1 ] then
            h1[ i ].leng := 0
        else
        if ((first[ 1 ] in alpha) and (h1[ i ].lab[ 1 ] in beta))
        or ((first[ 1 ] in beta) and (h1[ i ].lab[ 1 ] in alpha)) then
            h1[ i ].leng := 1
        else
            h1[ i ].leng := 2;
        end
        else begin
                h1[ i ].lab := blank;
                t1[ i ].lab := blank;
                h1[ i ].leng := maxint;
        end;
end; { of procedure generate }
{ ---------------------------------------------------------------------- }
```

```
procedure Attatch(P : integer; S : string);
{ Adds a node to the rear of the routelist. The integer
as well as the label for the node are stored. }
var
        leaf : node;
begin
        new(leaf);
        leaf↑ .index := P;
        leaf↑ .name := S;
        leaf↑ .link := nil;
        if routelist = nil then
                routelist := leaf;
        if rearlist <> nil then
                rearlist↑ .link := leaf;
        rearlist := leaf;
end;
{ ------------------------------------------------------------------------------ }
procedure route_order(p,q : integer);
{ Inputs : The integer values of the source P and the destination Q, the starting
and ending locations of the label.
Output : Creates the route.
This procedure generates a route between nodes P and Q according to the order
preserving algorithm. }
var
     divect          : svect;
     axle            : vect;
     t               : node;
     k,last,start    : integer;
     headnode, tailnode, temp, first, second : string;
```

```
begin
    attatch(p, vertices[p]↑.name);
    start := 2;
    first := vertices[p]↑.name;
    second := vertices[q]↑.name;
    writeln(' source = ', first);
    writeln(' Destination = ', second);
    if p <> q then
    begin
        last := len - 1;
        x := differ(first, second, len);
        write(' The difference vector = ');
        for i := 1 to len do
            write(x[i]:3);
        for k := 2 to len do
            divect[k] := x[k];
        if substr(first, 2, last) = substr(second, 2, last)
        then state := 5
        else
            state := 1;
        while (start <= len) and (state <> 6) do
        begin
        case state of
            1 :  begin
                    if divect[start] <> 0 then
                    begin
                        createhead(divect, len, headlis, taillis);
                        headnode[1] := headlis[start];
                        tailnode[1] := taillis[start];
                        for k := 2 to lim do
                        begin
                            headnode[k] := first[k];
                            tailnode[k] := first[k];
                        end;
                        tailnode[start] := chr(ord(tailnode[start]) -
                                            divect[start]);
                        writeln(' headnode = ', headnode);
                        writeln(' tailnode = ', tailnode);
                        if headnode[1] = first[1] then
                            state := 3
                        else
                            state := 2;
                    end
```

```
        else
        begin
            start := start + 1;
        end;
    end; { end of state 1 }
2 : begin
        t := vertices[p]↑.link;
        axle := differ(first, headnode, len);
        if axle[1] = 2 then
        begin
            while substr(t↑.name, 2, last) ¡¿ substr(first, 2, last)
            do t := t↑.link;
            if substr(t↑.name, 2, last) = substr(first, 2, last)
            then attatch(t↑.index, t↑.name);
            t := vertices[t↑.index]↑.link;
        end;
        while (t <> nil) do
            if t↑.name = headnode then
            begin
                attatch(t↑.index, t↑.name);
                t := nil;
            end
            else
                t := t↑.link;
            state := 3;
    end;
3 : begin
        t := vertices[rearlist↑.index]↑.link;
        while (t <> nil) do
        if t↑.name = tailnode then
        begin
            attatch(t↑.index, t↑.name);
            p := t↑.index;
            first := t↑.name;
            t := nil;
        end
        else
            t := t↑.link;
        state := 4;
    end;
```

```
                4 :  begin
                         x := differ(first, second, len);
                         write(' The difference vector = ');
                         for i := 1 to len do
                             write(x[i]:3);
                         for k := 2 to len do
                             divect[k] := x[k];
                         if substr(first, 2, last) = substr(second, 2, last)
                             then state := 5
                         else
                         begin
                             state := 1;
                             start := start + 1;
                         end;
                     end;
                5 :  begin
                         if p <> q then
                         begin
                             t := vertices[p]↑.link;
                             if x[1] = 2 then
                             begin
                                 while t <> nil do
                                 if substr(t↑.name,2,last) = substr(first,2,last)
                                 then begin
                                     attatch(t↑.index, t↑.name);
                                     t := nil;
                                 end
                                 else
                                     t := t↑.link;
                             end;
                             attatch(q, second);
                         end;
                         state := 6;
                     end;
                end; { end of case }
            end;
        end;
    end;
```

The program for the implementation of order-preserving algorithm is given below.

```
program orderroute(input, outc, output);
const
        maxnodes                =       4000;
        lim                     =       10;
type
        irange                  =       1 .. 6;
        vrange                  =       -2 .. 2;
        string                  =       packed array [1 .. lim] of char;
        vect                    =       array [1 .. 10] of vrange;
        svect                   =       array [2 .. 10] of vrange;
        nstrir.g                =       array [2 .. 10] of char;
        atype                   =       set of char;
        node                    =       ↑pointer;
        pointer                 =       record
                                          index : integer;
                                          name  :  string;
                                          visit · boolean;
                                          link    node;
                                        end;
        graphnodes              =       array [1 .. maxnodes] of node;
        list                    =       array[1 .. maxnodes] of string;
var
        state                   :       irange;
        M,routelist,rearlist    :       node:
        vertices                :       graphnodes;
        table                   :       list;
        outc                    :       text;
        p. q, level             :       integer;
        len, i, rlength         :       integer;
        x                       :       vect;
        alpha, beta             :       atype;
        divect, null            :       svect.
        headlis, taillis        :       nstring;
```

```
{ main program for OP algorithm begins here. }
begin
    alpha := ['U', 'V'];
    beta := ['R', 'S'];
    readgr;
    pringraph;
    writeln(' Source please :');
    readln(p);
    writeln(' Destination please :');
    readln(q);
    route_order(p, q);
    M := routelist;
    writeln;
    writeln(' The route is as follows : ');
    writeln;
    rlength := -1;
    while M <> nil do
    begin
        write(M↑.name,' ');
        rlength := rlength + 1;
        M := M↑.link;
    end;
    writeln;
    writeln(' The length of the route = ', rlength :3);
end.
```

```
procedure route_oper(p,q : integer);
{ Inputs : The integer values of the source P and the destination Q, the starting
and ending locations of the label.
{ Output : Creates the route. }
var
        divect                  :       svect;
        axle                    :       vect;
        t                       :       node;
        h1, t1                  :       nany;
        prime, dprime, blank    :       string;
        last, small, i, j       :       integer;
{ ------------------------------------------------------------------------------}
begin
    first := vertices[ p ]↑ .name;
    second := vertices[ q ]↑ .name;
    writeln;
    writeln(' source = ', p,'(',first, ')');
    writeln;
    writeln(' Destination = ', q, '(', second, ')');
    writeln:
    attatch(p, first);
    if p <> q then
    begin
        last := len - 1;
        if substr(first, 2, last) = substr(second, 2, last)
        then state := 5
        else
            state := 1;
        x := differ(first, second, len);
        writeln:
        write(' The difference vector = ');
        for i := 1 to len do
            write(x[ i ]: 3);
        while (state <= 5) do
            case state of
```

```
1 :  begin
        for i := 2 to len do
        divect[ i ] := x[ i ];
        createhead(divect, len, headlis, taillis);
        generate(divect, len, h1, t1);
        small := 2;
        for i := 2 to len do
            if h1[ i ].leng < h1[small].leng then
                small := i;
        prime := h1[small].lab;
        dprime := t1[small].lab;
        case h1[small].leng of
            0 : state := 4;
            1 : state := 3;
            2 : state := 2;
        end; { of case }
    end; { end of state 1 }
2 :  begin
        t := vertices[p]↑ .link;
        while substr(t↑ .name,2,last)
        <> substr(first,2,last) do
            t := t↑ .link;
        attatch(t↑ .index, t↑ .name);
        state := 3;
end;
3 :  begin
        t := vertices[rearlist↑ .index]↑ .link;
        while (t↑ .name <> prime ) do
            t := t↑ .link;
        attatch(t↑ .index, t↑ .name);
        state := 4;
    end;
```

```
        4 :  begin
                 t := vertices[rearlist↑ .index]↑ .link;
                 while (t↑ .name <> dprime) do
                     t := t↑ .link;
                 attatch(t↑ .index,t↑ .name);
                 p := t↑ .index;
                 first := t↑ .name;
                 x := differ(first, second, len);
                 writeln;
                 write(' The difference vector = ');
                 for i := 1 to len do
                     write(x[ i ]: 3);
                 if substr(first, 2, last) = substr(second, 2, last) then
                     state := 5
                 else
                     state := 1;
             end;
             5 : begin
                 if p <> q then
                 begin
                     t := vertices[ p ]↑ .link;
                     if x[ 1 ] = 2 then
                     begin
                     while t <> nil do
                     if substr(t↑ .name, 2, last) = substr(first, 2. last)
                         then begin
                             attatch(t↑ .index, t↑ .name);
                             t := nil;
                         end
                         else
                             t := t↑ .link;
                     end;
                     attatch(q, second);
                 end;
                 state := 6;
             end;
         end; { end of case }
     end;
end;
{ ——————————————————————————————————————————————————————————————————————}
```

The following program generates a route between two given nodes P and Q according to the opertunistic algorithm. The graph is stored as a one-dimensional array of linked lists where each list corresponds to the adjacency list of that node. Each node in the list is a record containing the fields index, name, visit, and link. The index is used for the easy access of the neighbor nodes of a particular node.

```
program Opertunist(input, outc, output);
const
        maxnodes        =       4000;
        lim             =       10;
type
        irange          =       1 .. 6;
        vrange          =       -2 .. 2;
        string          =       packed array [1 .. lim] of char;
        vect            =       array [1 .. 10] of vrange;
        svect           =       array [2 .. 10] of vrange;
        nstring         =       array [2 .. 10] of char;
        atype           =       set of char;
        node            =       ↑ pointer;
        pointer         =  record
                               index : integer;
                               name  : string,
                               visit : boolean,
                               link  : node;
                           end;
        next            =  record
                               lab  : string;
                               leng : integer;
                           end;
        nany            =       array [2 .. lim] of next;
        graphnodes      =       array [1 .. maxnodes] of node;
        list            =       array[1 .. maxnodes] of string;
```

```
var
        state                   :       irange;
        M, routelist, rearlist  :       node;
        vertices                :       graphnodes;
        table                   :       list;
        outc                    :       text;
        p, q, level             :       integer;
        len, i, rlength         :       integer;
        x                       :       vect;
        alpha, beta             :       atype;
        divect, null            :       svect;
        headlis, taillis        :       nstring;
        first, second           :       string;
{ main program for opertunistic algorithm begins here. }
begin
    alpha := ['U', 'V'];
    beta := ['R', 'S'];
    readgr;
    read(p);
    while p <> 0 do
    begin
        read(q);
        readln;
        route_oper(p, q);
        M := routelist;
        writeln;
        writeln(' The route is as follows : ');
        writeln;
        rlength := -1;
        while M <> nil do
        begin
            writeln(M↑ .name,' →');
            rlength := rlength + 1;
            M := M↑ .link;
        end;
        writeln;
        writeln(' The length of the route = ', rlength :3);
        routelist := nil;
        writeln;
        read(p);
    end;
end.
```

```
procedure route_fault(p,q : integer);
    { This procedure tries to establish a route between two given nodes P and
Q in a faulty environment. If successful, it will give the route and its length as the
output. Otherwise a message, 'Further routing impossible!' and the route thus far
will be given.}
var
        divect                      :       svect;
        axle                        :       vect;
        h1, t1                      :       many;
        tem                         :       char;
        prefer                      :       string;
        t, s, temper                :       node;
        done, found                 :       boolean;
        last, small, i, j, count, ch :      integer;
begin
    first := vertices[p]↑ .name;
    second := vertices[q]↑ .name;
    writeln(' source = ', p, '(', first, ')');
    writeln(' destination = ',q, '(', second, ')');
    if (vertices[p]↑ .fault or vertices[q]↑ .fault) then
        writeln(' Routing impossible between faulty nodes.')
    else
    begin
        attatch(p, first);
        if p <> q then
        begin
            last := len - 1;
            x := differ(first, second, len);
            if substr(first, 2, last) = substr(second, 2, last) then
                state := 6
            else
                state := 1;
            while (state <= 6) and not(noroute) do
```

```
case state of
    1:  begin
            writeln;
            writeln('p = ', p);
            writeln('first = ', first);
            writeln;
            for i := 2 to len do
                divect[i] := x[i];
            createhead(divect, len, headlis, taillis);
            generate(divect, len, h1, t1);
            t := vertices[p];
            while t <> nil do
            begin
                i := 2;
                found := false;
    while (i <= last) and not(found) do
        if t↑ .name = h1[i].lab then
            found := true
        else
            i := i + 1;
        if (found) then
        begin
            s := vertices[t↑ .index]↑ .link;
            while s↑ .name <> t1[i].lab do
                s := s↑ .link;
            h1[i].ind := t↑ .inde>;
            if h1[i].ind > n then
                h1[i].leng := maxint;
            if ((vertices[s↑ .index]↑ .fault)
            or vertices[s↑ .index]↑ .visit
            or vertices[t↑ .index]^ .fault)
            or vertices[t↑ .index]↑ .visit
            and (h1[i].lab[1] <> taillis[i]) then
            begin
                tem := h1[i].lab[1];
                h1[i].lab[1] := t1[i].lab[1];
                t1[i].lab[1] := tem;
                ch := ord(t1[i].lab[i]) - divect[i];
                if ch = ord('0') then
                    ch := ord('3')
```

```
                    else
                    if ch > ord('3') then
                    begin
                        ch := ch mod ord('3') + ord('0');
                    end
                    else
                    if ch < ord('0') then
                        ch := ch + 3;
                    t1[i].lab[i] := chr(ch);
                    s := vertices[p];
                    while ((s <> nil) and (s↑ .name <> h1[i].lab)) do
                        s := s↑ .link;
                    if s = nil then
                    begin
                        s :=vertices[ vertices[p]↑ .link↑ .index];
                        while (s↑ .name <> h1[i].lab) and (s <> nil) do
                            s := s↑ .link;
                    end;
                    h1[i].ind := s↑ .index;
                    if vertices[s↑ .index]↑ .fault or
                    vertices[s↑ .index]↑ .visit or (h1[i].ind > n ) then
                        h1[i].leng := maxint
                    else
                    if first[1] = h1[i] lab[1] then
                        h1[i].leng := 0
                    else
                    if ((first[1] in alpha) and (h1[i].lab[1] in beta))
                    or ((first[1] in beta) and (h1[i].lab[1] in alpha)) then
                        h1[i].leng := 1
                    else
                        h1[i].leng := 2;
                end;
            end;
            t := t↑ .link;
        end;{ end of while t <> nil }
        state := 2;
        vertices[rearlist↑ .index]↑ .visit := true;
end; { of state 1 }
```

```
2: begin
     small := 2;
     for i := 2 to last do
          if (h1[i].leng < h1[small].leng)
               then small := i;
          writeln(' h1[', small:3, '].lab = ',h1[small].lab);
          writeln(' h1[', small:3,'].ind = ', h1[small].ind);
          writeln('h1[', small:3,'].leng = ', h1[small].leng);
          case h1[small].leng of
               0 : state := 5;
               maxint : state := 4;
               2 : state := 3;
               1 :begin
                    if (h1[small].ind < 1) or (h1[small].ind > n)
                    then state := 4
                    else
                    if not(vertices[h1[small].ind]↑ .visit)then
                    begin
                         writeln(' h1.ind = ', h1[small].ind);
                         attatch(h1[small].ind, h1[small].lab);
                         vertices[h1[small].ind]↑ .visit := true;
                         state := 5;
                    end
                    else state := 4;
               end;
          end; { end of case }
     end; { of state 2 }
```

```
3: begin
     t := vertices[p]↑ .link;
     done := false;
     while t <> nil do
          if (substr(t↑ .name, 2, last) = substr(first, 2, last))
          and (not(vertices[t↑ .index]↑ .fault)
          and not(vertices[t↑ .index]↑ .visit)) then
          begin
               attatch(t↑ .index, t↑ .name);
               done := true;
               vertices[t↑ .index]↑ .visit := true;
               t := nil;
          end
          else
          begin
               t := t↑ .link;
          end;
          if (not done) then
               state := 4
          else
          begin
               t := vertices[rearlist↑ .index]↑ .link;
               while t↑ .name <> h1[small].lab do
                    t := t↑ .link;
               if vertices[t↑ .index]↑ .fault then
                    state := 4
               else
               begin
                    attatch(t↑ .index, t↑ .name);
                    vertices[t↑ .index]↑ .visit := true;
                    state := 5;
               end;
          end;
end; { of state 3 }
```

```
4: begin
     j := 0;
     prefer := blank;
     t := vertices[rearlist↑ .index]↑ .link;
     writeln(' t↑ .name = ', t↑ .name);
     while t <> nil do
     if not(vertices[t↑ .index]↑ .fault) and
     not(vertices[t↑ .index]↑ .visit) then
     begin
          j := t↑ .index;
          prefer := t↑ .name;
          t := nil;
     end
     else
          t := t↑ .link;
          writeln(' j = ', j, 'prefer = ', prefer);
          if prefer = blank then
          begin
               writeln(' Further routing impossible.');
               noroute := true;
          end
          else
          begin
               attatch(j, prefer);
               p := j;
               first := prefer;
               x := differ(first,second,len);
               state := 1;
          end;
     end; { end of state 4 }
```

```
5: begin
     t := vertices[rearlist↑ .index]↑ .link;
     done := false;
     while t <> nil do
          if (t↑ .name = t1[small].lab) and
          not(vertices[t↑ .index]↑ .fault) and
          not(vertices[t↑ .index]↑ .visit) then
          begin
               attatch(t↑ .index,t↑ .name);
               done := true;
               p := t↑ .index;
               first := t↑ .name;
               t := nil;
          end
          else
               t := t↑ .link;
               if not done then
                    state := 4
               else
               begin
                    x := differ(first, second, len);
                    if substr(first, 2, last) = substr(second, 2, last) then
                         state := 6
                    else state := 1;
               end;
     end; { of state 5 }
```

```
                6: begin
                    if p <> q then
                    begin
                        done := false;
                        t := vertices[p]↑ .link;
                        if x[1] = 2 then
                        while t <> nil do
                        if substr(t↑ .name, 2, last) = substr(first, 2, last) then
                            if not(vertices[t↑ .index]↑ .fault) then
                            begin
                                attatch(t↑ .index,t↑ .name);
                                done := true;
                                t↑ .visit := true;
                                t := nil;
                            end
                            else t := t↑ .link;
                            if (not(done)) and (x[1] = 2) then
                            begin
                                noroute := true;
                                writeln(' Further routing impossible due to
                                        disconnected block.');
                            end
                            else
                                attatch(q, second);
                    end;
                    state := 7;
                end; { end of state 6 }
            end; { of case statement }
        end;
    end;
end; { of procedure fault_route }
```

```
{ Declarations for program Fault.}
{ ------------------------------------------------------------------------}
const
        maxnodes                =       4000;
        lim                     =       10;
type
        irange                  =       1 .. 6;
        vrange                  =       -2 .. 2;
        string                  =       packed array [1 .. lim] of char;
        vect                    =       array [1 .. 10] of vrange;
        svect                   =       array [2 .. 10] of vrange;
        nstring                 =       array [2 .. 10] of char;
        atype                   =       set of char;
        node                    =       ↑ pointer;
        pointer                 =  record
                                       index : integer;
                                       name : string;
                                       visit : boolean;
                                       fault : boolean;
                                       link : node;
                                   end;
        next                    =  record
                                       lab  : string;
                                       leng : integer;
                                       ind  : integer;
                                   end;
        nany                    =       array [2 .. lim] of next;
        graphnodes              =       array [1 .. maxnodes] of node;
        list                    =       array[1 .. maxnodes] of string;
var
        blank                   :       string;
        state                   :       irange;
        M, routelist, rearlist  :       node;
        vertices                :       graphnodes;
        table                   :       list;
        outc                    :       text;
        p, q, level, n, len, i  :       integer;
        globalseed, rlength     :       integer;
        x                       :       vect;
        alpha, beta             :       atype;
        divect, null            :       svect;
        headlis, taillis        :       nstring;
        first, second           :       string;
        noroute                 :       boolean;
```

```
{ main program begins here }
begin
        alpha := ['U', 'V'];
        beta := ['R', 'S'];
        noroute := false;
        for i := 1 to lim do
        blank[i] := ' ';
        readgr;
        make_fault;
        writeln(' source ? ');
        read(p);
        writeln(' destination ?');
        read(q);
        writeln;
        writeln;
        route_fault(p, q);
        pringraph;
        M := routelist;
        writeln;
        if noroute then
        writeln(' Route thus far : ')
        else
        writeln(' The route is as follows : ');
        writeln;
        rlength := -1;
        while M <> nil do
        begin
                writeln(M↑ .name,' → ');
                rlength := rlength + 1;
                M := M↑ .link;
        end;
        writeln;
        writeln(' The length of the route = ', rlength :3);
        routelist := nil;
        writeln;
end.
```

The following program for the creation of the graph $G(K_{2,2})^i$ is an adaptation of Dr. J. Opatrny's program COPIES.

```
program copy(input,inn,inpairs,outc,output);
(* Program that generates and labels the graph G(K2,2)^i.*)

const
        maxdeg = 10;
        size = 4000;
        lim = 10;
type
        string = packed array[1 .. lim] of char;
        node = record
                        index : integer;
                        name : string;
                end;
        graph = array[1 .. size,1 .. maxdeg] of node;
var
        ind, m, i, no, sh, k, j, level, po : integer;
        deg : array[1 .. size] of integer;
        G : graph;
        inn,inpairs,outc: text;
{ ------------------------------------------------------------------------------}
```

```
procedure incop(var no, level : integer);
var
        i, j : integer;
begin
        readln(inn, no, level);
        writeln('no = ', no);
        i := 1;
        while not eof(inn) do
        begin
                j := 0;
                while not eoln(inn) do
                begin
                        j := j + 1;
                        read(inn, G[i, j].index, G[i, j].name);
                end;
                deg[i] := j;
                i := i + 1;
                readln(inn);
        end;
        (*for k := 1 to i do
        begin
                for j := 1 to deg[k] do
                begin
                        write(' G[', k:3, j:3, '].index = ', G[k, j].index:3);
                        write(' G[', k:3, j:3, '].name = ', G[k, j].name);
                end;
                writeln;
        end; *)
end; (* of incop *)
{ ---------- ----------}
```

```
procedure makepair;
var
        shift, norep, i, nopairs, per1, per2, j : integer;
        tag1, tag2 : array[1 .. 100] of string;
        p1, p2 · array[1 .. 100] of integer;
begin
    ·   writeln('how many pairs ?');
        readln(nopairs);
        writeln;
        for i := 1 to nopairs do
                readln(inpairs, p1[i], p2[i]);
        writeln('what are the periods ?');
        readln(per1, per2);
        writeln('number of repetitions ?');
        readln(norep);
        shift := 0;
        for i := 1 to norep do
        begin
                for j := 1 to nopairs do
                begin
                        deg[p1[j]] := deg[p1[j]] + 1
                        G[p1[j], deg[p1[j]]].index := p2[j];
                        G[p1[j], deg[p1[j]]].name := G[p2[j], 1].name;
                        deg[p2[j]] := deg[p2[j]] + 1;
                        G[p2[j], deg[p2[j]]].index := p1[j];
                        G[p2[j], deg[p2[j]]].name := G[p1[j], 1].name;
                        p1[j] := p1[j] + per1;
                        p2[j] := p2[j] + per2;
                end;
                shift := shift + per1;
        end;
end;
```

```
procedure copout;
var
        i, j : integer;
begin
        writeln(outc, no * k : 5, level:5);
        for i := 1 to no * k do
        begin
                for j := 1 to deg[i] do
                begin
                        G[i, j].name[level] := '1';
                        write(outc, G[i, j].index:5, G[i, j].name);
                end;
                writeln(outc);
        end;
end; (* of procedure copout *)
{ ------------------------------------------------------------------------------}
```

```
(* main program begins here *)
begin
        reset(inn);
        reset(inpairs);
        rewrite(outc);
        writeln(' how many copies ?');
        readln(k);
        incop(no, level);
        sh := no;
        po := 1;
        for j := 1 to k - 1 do
        begin
                for ind := 1 to no do
                begin
                        for m := 1 to deg[ind] do
                        begin
                                G[sh + ind, m].index := G[ind, m].index + sh;
                                G[sh + ind, m].name := G[ind, m].name;
                                G[sh + ind, m].name[level] :=
                                chr(ord(G[ind, m].name[level]) + po);
                        end;
                        deg[sh + ind] := deg[ind];
                end;
                sh := sh + no;
                po := po + 1;
                if po > 3 then po := 1;
        end;
        makepair;
        level := level + 1;
        writeln(' level = ',level);
        copout;
end.
{ ---------- }
```