AN ABSTRACT OF THE THESIS OF

Shawn M. Larson for the degree of Doctor of Philosophy in Computer Science presented on May 4, 1995.

Title: A Linear Equation Model for a Family of Interconnection Networks

Redacted for Privacy

Abstract approved: _____

Dr. Paul Cull

The most important part of parallel computation is communication. Except in the most embarassingly parallel examples, processors cannot work cooperatively to solve a problem unless they can communicate. One way to solve the problem of communication is to use an *interconnection network*. Processors are located at nodes of the network, which are joined by communication channels. Desirable aspects of an interconnection network include low maximum and average routing distances (as measured in the number of communication channels crossed), a large number of processors, and low number of communication channels per processor.

A number of published networks are created from the hypercube by rearranging the hypercube's communication links in a systematic way [23] [28] [30] [33] [50]. These networks maintain the same number of processors, communication links, and links per processor as the hypercube, but have dramatically smaller maximum and average routing distances.

This thesis derives one formal mathematical description for this family of networks. This formal description is used to derive graph-theoretic properties of existing networks, and to design new networks. The description is also used to

design generalized routing and other communications algorithms for these networks, and to show that these networks can embed and simulate other standard networks, for instance, ring and mesh networks.

A network simulator is used to model the dynamic behavior of this family of networks under both store-and-forward and wormhole routing strategies for message-passing. The simulation results are used to study and compare the networks' behavior under various message-passing loads, and to determine what properties are desirable in a network that exists in this model.

A Linear Equation Model for a Family of Interconnection Networks

by

Shawn M. Larson

A Dissertation

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Completed May 4, 1995
Commencement June 1995

Doctor of Philosophy thesis of Shawn M. Larson presented on May 4, 1995

APPROVED:

Redacted for Privacy

Major Professor, representing Computer Science

Redacted for Privacy

Head of Department of Department of Computer Science

Redacted for Privacy

Dean of Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for Privacy

Shawn M. Larson, Author

ACKNOWLEDGEMENT

I want to thank everyone who helped me with my research, especially my wife Yu-Hsi, who continually encouraged me to finish, and my major professor, Dr. Pall Cuul, who was patient enough to put up with my procrastination and many misspellings.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

A LINEAR EQUATION MODEL FOR A FAMILY OF INTERCONNECTION NETWORKS

1. INTRODUCTION AND MOTIVATION

We introduce the Twisted 3-Cube, and explain the motivation behind creating hypercube variant networks. We describe several known generalizations of the Twisted 3-Cube. We explain the need for creating a unified description of these networks, and outline our research.

1.1. INTRODUCTION

An interconnection network unites many processors to form a parallel computer. Each processor can communicate directly only to a small number of neighbors. Each processor can communicate with all other processors by forwarding messages through its neighbors.

The design of large-scale multicomputers has two conflicting goals: a large number of nodes and a low communication delay (or latency) between any two processing nodes. In addition, engineering constraints require uniformity, such as a fixed number of channels per node, reuse of processing elements at each node, etc.

The n-dimensional hypercube is one of the most popular interconnection networks. Its popularity is due in part to its highly symmetrical structure and relatively fast communication between processors. The n-dimensional hypercube has 2^n processors, n communication channels from each processor, and $n2^n$ channels total.

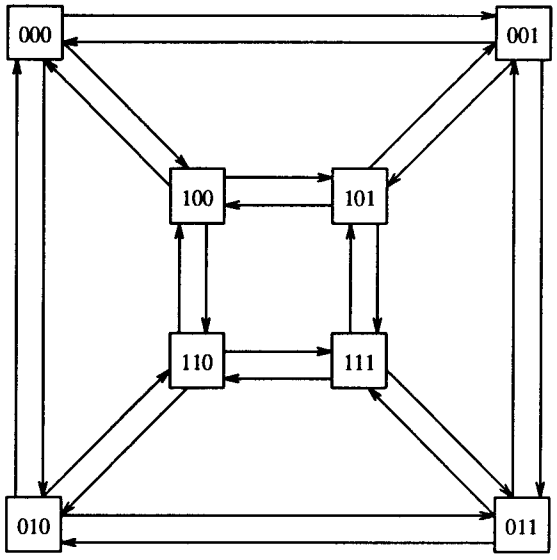

Figure 1.1. The three-dimensional hypercube Q_3.

The hypercube compares favorably with other popular interconnection networks. For example, the 2-dimensional mesh has been widely used in parallel computers because it has an extremely simple layout and only 4 communication channels per processor. However, this comes at a cost in communication delay. If N is the number of nodes in the network, then the diameter of the network (the maximum routing distance, as measured in the number of communication steps) is $\sqrt{N}$ for the mesh network, but only $\log(N)$ for the hypercube. This small diameter is a major attraction for using the hypercube network, because for many networks, a small diameter implies a small maximum communication delay.

The hypercube network is not the smallest diameter network possible for the resources it uses. By switching several of the channels, the diameter of the 3-dimensional hypercube (or Q_3) can be reduced from 3 steps to 2 steps, as in Figures

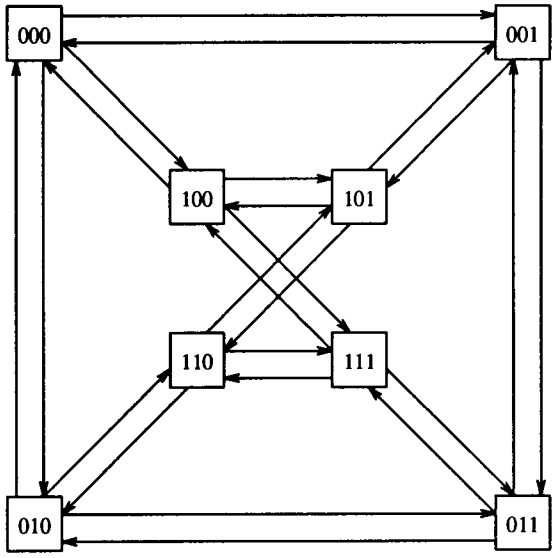

Figure 1.2. The Twisted 3-Cube TQ_3.

1.1 and 1.2. This is the smallest diameter network that can be achieved with these resources.

The Twisted 3-Cube (or TQ_3) also has an average internodal distance of 1.375 steps, compared to Q_3's average distance of 1.5 steps, as averaged over all possible source/destination pairs.

The utilization of any channel of TQ_3 is also less than or equal to the utilization of the corresponding channel on Q_3. If we use the standard hypercube routing algorithm on TQ_3, and add exceptions for any paths that are shorter, then the number of paths through any one channel on TQ_3 is always less than or equal to the number through any channel on Q_3, as shown in Figures 1.3 and 1.4.

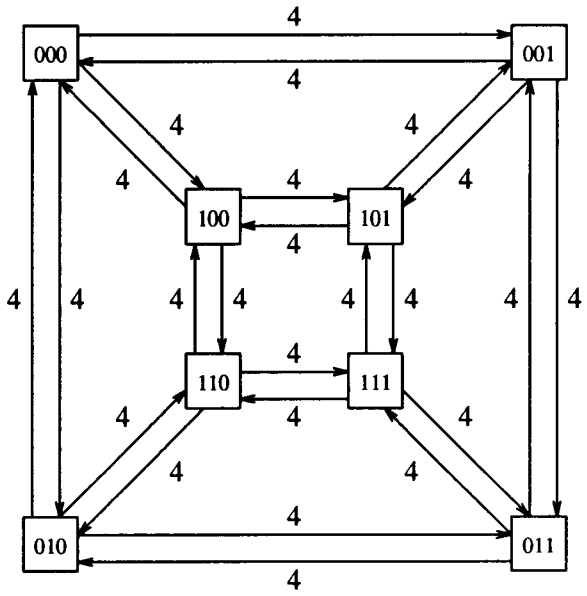

Figure 1.3. The channel utilization of the 3-Cube.

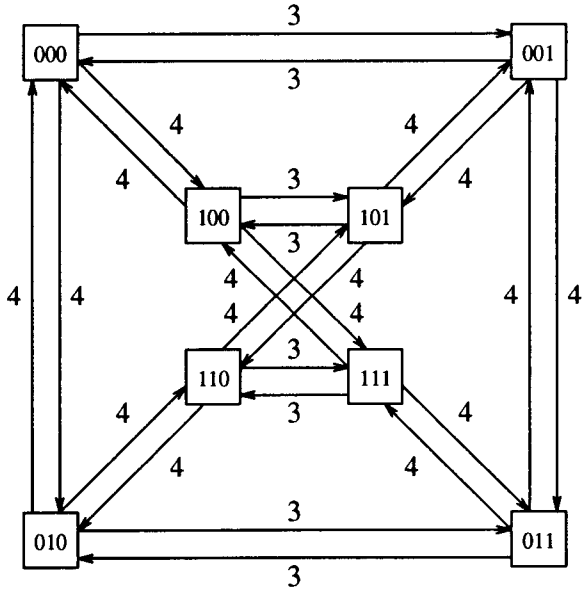

Figure 1.4. The channel utilization of the Twisted 3-Cube.

If we assume a uniform message-passing distribution, those channels will be utilized less in TQ_3 than they would be in Q_3. No channels of the TQ_3 will be utilized more.

1.2. MOTIVATION

A number of researchers have tried to improve upon the hypercube, most with the goal of reducing the communication delay between processors. These attempts have led to the creation and assessment of a number of hypercube variant networks.

Most hypercube variants attempt to improve upon the hypercube by adding processors and/or communication channels. The hypercube often appears as a squashed or relaxed embedding in these networks [45], [47] [48] [54] [52] [50]. Such "structure-preserving" variants can often use hypercube algorithms with few modifications. Their major disadvantage is that they use more hardware resources than the hypercube.

Other hypercube variant networks are "resource-preserving". They rearrange the communication paths of a hypercube to create an entirely new network [24] [29] [31] [34] [51]. These variants have the same resources as the hypercube, but a different network structure. These networks are without exception generalizations of the Twisted 3-Cube to higher dimensions. Unfortunately, resource-preserving variants often cannot use algorithms written for the hypercube, because the rearranged channels destroy the original hypercube structure.

Resource-preserving variants of the hypercube are worth study, because they seem to give "something for nothing." They show shorter communication delays than the hypercube of the same size, but without the need for additional resources. Because improvements in theoretical performance measures can translate into improvements in actual performance, study of these variants can lead to actual parallel

computers with more efficient communications. Also, existing hypercube computers may be reconfigured as a resource-preserving variant with little modification of its hardware. For some systems this reconfiguration may be as simple as re-wiring the communications channels and rewriting the communications software. The new network can give better performance with no additional purchase in hardware.

1.3. PREVIOUS WORK ON RESOURCE PRESERVING VARIANTS

The n-dimensional hypercube network Q_n is defined by assigning to each node a unique address from the vector space $\mathcal{Z}_2^n$, and allowing a channel between only nodes $\vec{X}$ and $\vec{Y}$ iff the Hamming weight of $\vec{X}+\vec{Y}$ is one, that is, the addresses $\vec{X}$ and $\vec{Y}$ differ in only one component.

About a decade ago, Hillis [35] constructed the Twisted 3-Cube in Figure 1.2. Hillis went no further with this observation. But it was obvious that his result could be generalized in two ways.

The Twisted N-Cube of Estafahanian *et al.* [31], has only one crossed pair of communication channels, so that one pair of channels connect addresses that differ by a Hamming weight of two. This "twist" gives a diameter of $n-1$. The Twisted N-Cube's routing algorithm is based on the hypercube's left-right bit correction algorithm, and has the same $O(n)$ run time, where n is the dimension of the cube. Curiously, this network contains not only a Hamiltonian circuit, but also a complete binary tree of 2^n-1 nodes.

The other obvious generalization is to consider an n-dimensional hypercube as a 3-dimensional hypercube in which all eight nodes are $(n-3)$-dimensional hypercubes. Then the twist of two channels becomes a twist of the whole set of channels joining two of the $(n-3)$-dimensional sub-networks. We can recursively apply

twists to the network so that the nodes in Figure 1.2 are now $(n-3)$-dimensional "Generalized Twisted Cubes."

This is exactly the construction given by Chedid and Chedid [12] for their Generalized Twisted cubes. They explicitly define the Twisted 3-Cube, then define their network GQ_n in higher dimensions by using graph composition. That is:

$$GQ_0 = Q_0,\ GQ_1 = Q_1,\ GQ_2 = Q_2,$$

$$GQ_{n+3} = TQ_3 \times GQ_n$$

The GQ_n routing algorithm breaks the addresses of the source and destination address into groups of three components, and does Twisted 3-Cube routing on each group of three components. The algorithm routes a message to any destination in $\lceil 2n/3 \rceil$ steps. Its average distance, not computed by the authors, can be calculated by using a sum of the average distances of the graphs used in the composition: $\frac{11}{8}\lceil \frac{n}{3} \rceil + \frac{1}{2}(n \bmod 3)$.

Chedid and Chedid show that the Generalized Twisted cubes are Hamiltonian, and "show" that they contain a complete binary tree of $2^n - 1$ nodes. (The proof is incorrect.)

Cheng and Chuang [13] have designed essentially the same network with the Varietal Hypercubes. In addition to showing many of the the same graph-theoretic properties as Chedid and Chedid did for the Generalized Twisted Cubes, Cheng and Chuang managed to show the n-dimensional Varietal Hypercubes can embed an arbitrary $2^p \times 2^q$ mesh, where $p + q \leq n$.

There are other generalizations of the Twisted 3-Cube. A summary of the ones we have discovered are listed in Table 1.1, along with their diameter, expected distance, and number of twisted channels.

Network Name	Diameter	Expected Dist. (Leading Term)	Number of Twists	Routing Algorithm
Hypercube	n	$\frac{n}{2}$	0	$O(n)$
Twisted Cube [34]	$\lceil \frac{n+1}{2} \rceil$	$\approx \frac{3n}{8}$	$(n-1) \times 2^{n-4}$	$O(n)$
Twisted N-Cube [31]	$n-1$	$\approx \frac{n}{2}$	2	$O(n)$
Multiply Twisted Cube [29] Crossed Cube [30]	$\lceil \frac{n+1}{2} \rceil$	$\approx \frac{651n}{1840}$, for $n = 2k$ $\approx \frac{21n}{64}$, for $n = 2k+1$	$(k-2)2^{2k} + 2^{k+1}$ for$n = 2k$ $k2^{2k+1} - 3(2^{2k} - 2^k)$ for$n = 2k+1$	$O(n^2)$
Flip MCube [51]	$\lceil \frac{n+1}{2} \rceil$	$\approx \frac{n}{3}$	$(n-2) \times 2^{n-1}$	$O(n^2)$
0-Möbius Cube [23]	$\lceil \frac{n+2}{2} \rceil$	$\approx \frac{n}{3}$	$(n-2) \times 2^{n-1}$	$O(n)$
1-Möbius Cube [23]	$\lceil \frac{n+1}{2} \rceil$	$\approx \frac{n}{3}$	$n2^{n-1}$	$O(n)$
Generalized Twisted Cube [12]	$\lceil \frac{2n}{3} \rceil$	$\approx \frac{11n}{24}$	$2^{n-3+\lceil n/3 \rceil}$	$O(n)$
Twisted Hypercube [28]	$n-1$	$\approx n/2 - 1/8$	2^{n-1}	$O(n)$

Table 1.1. Diameter, expected distance, number of twists and routing algorithm run time of hypercube variants.

As early as 1987, Hilbers *et al.* [34] published the Twisted Cube, which was designed for odd n and had diameter $\lceil (n+1)/2 \rceil$. Their cube of dimension n is constructed from 4 sub-networks of dimension $n-2$. Their solution was to let half of a sub-network's channels be normal hypercube connections, and half of the channels be "twisted" connections. As in the hypercube, each node has a unique address in $\mathcal{Z}_2^n$. A normal connection between nodes is made by connecting a node with address $\vec{X}$ to a node with address $\vec{X} + e_i$, where e_i is the vector with a single 1 in the i-th component. Similarly a twisted connection is made by connecting $\vec{X}$ to a node with address $\vec{X} + e_i + e_{i+1}$.

For $0 \leq j \leq n/2$, the $(2j-1)$-th connections are all hypercube connections. The type of the $(2j)$-th connection is based on the parity of the node's address in components $2j$ through n. A hypercube connection is made if the parity is odd, and a cross connection is made if the parity of the remaining bits is even.

The diameter of the Twisted Cube is at least $(n+1)/2$ since at most 2 bits are corrected at each step. Hilbers shows that the diameter is exactly $(n+1)/2$ by giving an exact routing algorithm that finds a path between any pair of nodes.

Abraham and Padmanabhan ([2] and [1]) compute the expected distance of the Twisted Cube, and compare the dynamic performance of the Twisted Cube and the hypercube. A stochastic simulation of the Twisted Cube in [1] shows that it has a performance comparable to the hypercube, but not quite what would be expected from the $\lceil (n+1)/2 \rceil$ diameter.

The Multiply Twisted Cube of Kemal Efe [29] (later the Crossed Cube [30]) is quite similar to the Twisted Cube, in that it joins 4 $(n-2)$-cubes together to construct a n-cube. Its structure differs from the Twisted cube in that its connection rules cause communication channels to be twisted across several dimensions simultaneously, that is, the Hamming distance of two nodes joined by a channel can

be anywhere from 1 to n. Its diameter, like the Twisted Cube's, is $\lceil (n+1)/2 \rceil$. Efe gives a broadcasting algorithm and an optimal routing algorithm with an $O(n^2)$ run time. Efe also demonstrates an efficient broadcasting algorithm, and demonstrates SIMD algorithms for semigroup computations, matrix multiplication and sorting. On interesting feature of the Crossed Cube is that a reconfigurable network can be constructed with switchable channels, so it can behave like either a hypercube or a Crossed Cube.

Our own Möbius cubes [18] [19] [24] [42] are also generalizations of the Twisted 3-Cube. They actually are two closely related sets of twisted cubes, called the 0-Möbius cubes and the 1-Möbius cubes. Like the Twisted Cube of Hilbers *et al.*, the Möbius cubes have a simple construction rule and an $O(n)$ routing algorithm. A 0-Möbius cube of dimension n is formed by taking a 0-Möbius cube and a 1-Möbius cube of dimension $n-1$ and connecting nodes of the same address. A 1-Möbius cube of dimension n is formed by taking a 0-Möbius cube and a 1-Möbius cube of dimension $n-1$ and joining nodes whose addresses are binary complements of each other. The diameter of the 0-Möbius cube is $\lceil (n+2)/2 \rceil$ and the diameter of the 1-Möbius cube is $\lceil (n+1)/2 \rceil$, for $n \geq 4$. We have also given a routing algorithm with $O(n)$ run time for our Möbius cubes. In [21], we show an optimal routing algorithm that ultimately takes a $O(n)$ distributed run time, and we compute the diameter and expected distance. We show that the networks have a Hamiltonian ring, and a binomial tree rooted at any arbitrary node. A stochastic simulation based on Abraham and Padmanabhan's simulation showed that the Möbius cubes have a much smaller average message latency than the Twisted Cube, and somewhat smaller message latencies than the hypercube.

An entire family of networks is described by Singhvi and Ghose [51]. Their MCube (the "M" is for "Möbius" – they wanted to call their networks the Möbius

cubes, but we used the name first.) is constructed by decomposing two MCubes of dimension $n-1$ into 4 sub-networks each, and then joining each pair of cubes by a twisted connection to produce an MCube of dimension n. This method produces a family of networks because the decomposition of the two MCubes and the orientation of their sub-networks is arbitrary. A general MCube routing algorithm is presented that has a $O(n^2)$ run time. Singhvi and Ghose claim that efficient communication can be done because the MCubes have a uniform distance distribution and a uniform rate of traffic flow across all channels. One particular MCube network, the "Flip MCube," has the orientation of its sub-networks specified. The Flip MCube has diameter $\lceil (n+1)/2 \rceil$, and is shown in a dynamic simulation to have generally superior dynamic performance to the hypercube.

Finally, the Twisted Hypercube of Das *et. al.* [28] has $n/2$ of the connections in the $(n-1)$-th dimension cross both dimensions $n-1$ and n. The diameter of the network is only $n-1$, like the Twisted n-Cube. However, Das *et. al.* show that two Twisted Hypercubes of dimension $n-1$ have disjoint embeddings into the Folded Hypercube of dimension n, which improves the Folded Hypercube's fault-tolerant behavior, by allowing it to emulate a Twisted Hypercube when a regular hypercube connection fails.

These networks do not exhaust the list of possible hypercube variant networks. In this dissertation, we will introduce two new networks, the YAT (Yet Another Twisted) Cube and the Bent Cube. These two networks are patterned after the networks listed above, and compare favorably with published networks.

(On the humorous side, we have found that other researchers working on twisted hypercube variants have been inexcusably unoriginal in christening their creations. There are now five networks with the word "twisted" in their names, and three that are based on the name "Möbius". Future cube-variant designers should

consider names from the following list: Rotated, coiled, torqued, braided, wriggly, spun, skewed, warped, Kleinian, Gödelain, Escherian, bizarre, grotesque, eccentric, erratic, and oddball. To the best of our knowledge, not one of these has yet been claimed.)

Finally, it should be noted that resource-preserving variants of the hypercube can often be used to replace the hypercube structure in structure-preserving hypercube variants. Kumar and Patnaik [41] produce variant hypercube networks by taking the Enhanced Hypercube of Tzeng and Wei [53], which contains 3-Cubes, and replacing these 3-Cubes with Twisted 3-Cubes. They show that this substitution reduces the diameter of the network. Their technique can also be applied to networks like the Cube-Connected Cycles by replacing the hypercube connections with twisted cube connections.

1.4. SIGNIFICANCE OF RESEARCH

Every one of the networks discussed above has been constructed using a different method. This makes the comparison and evaluation of these networks a difficult and tedious task. A systematic approach to describing, creating, and comparing resource preserving variants is needed.

This dissertation will explore the family of resource-preserving hypercube variants. In this dissertation, we will generalize this "twisted" network into a family of networks of higher dimensions. We will do this by producing a single mathematical model that can describe most or all of these networks. This model should be simple, yet powerful enough to describe a large number of possible networks. We will use this model to derive and prove communication algorithms. We will also use this model to show bounds on some performance measures for the networks in this model, and

we will use a computer model of the networks to empirically model these networks and compute their dynamic performance.

1.4.1. The Need for a Formal Description

Fitting the hypercube variants into one mathematical model will allow a method to compare and contrast them. A single mathematical or formal description of variant networks can have a number of advantages in developing new networks. These advantages include:

Unification Presenting a model of the networks in the resource-preserving hypercube variant family is a stronger result than creating another resource-preserving variant. It gives a single method of describing what initially appear to be very different networks.

Generalization Formalizing the family of resource-preserving variants will prevent duplication of research, in the sense that results for known members of the family can extend to new members. New networks can be systematically constructed using the formal description of resource-preserving variants. Further, if limits exist for any performance measures of networks in the family, then proving membership for a new network shows that the same limits exist for that network.

A formal description may also prevent duplication of work by presenting algorithms or an algorithm schema that can (for every member of the resource-preserving variant family) compute a point-to-point route or a broadcasting tree. The routing and broadcast algorithms for these new networks may well have a systematic design.

Specialization By specifying a formal model, it may be possible to design networks that meet a specific performance measure within the limits of the family. Networks might be tailored to meet a specific processing need.

A model to describe resource-preserving variants does not need to be all-inclusive. Its main attraction should be that it can describe member networks simply, and that it is broad enough to describe a variety of networks.

1.4.2. The Need for Comparative Evaluation

All published papers on new interconnection networks include at least a comparison of the network's diameter with that of the hypercube. However, only a few authors have done an extensive evaluation of other performance measures for their networks. Only four authors have examined the mean internodal distance of any resource preserving variants [2] [31] [42] [51]. Only three authors have run stochastic simulations to compare their network's dynamic behavior, [1], [42] and [51], but they have apparently used different models of communication which makes a direct comparison of the networks difficult. Only the dynamic behavior of the hypercube and the Twisted cube of Hilbers *et al.* [34] have been analyzed in any depth [3] [2] [1].

Since no one has done an extensive evaluation of the resource preserving variants, a direct comparison and evaluation of the currently published resource-preserving variant networks is needed, for two reasons. First, differences in the descriptions of variant networks give rise to differences in their performance, as shown in a comparison of the Twisted cubes and Möbius cubes. An empirical simulation and comparison of the variants using several performance measures will tell exactly what networks have the most desirable performance measures. By examining networks with particularly good performance, we are able to specify which if any

network properties will produce the improved performance. Second, the choice of routing algorithm can affect a network's dynamic performance. In some instances, even non-optimal routing algorithms can give similar or even better performance than optimal algorithms. Simulations are a good empirical method for measuring the performance of any architecture/algorithm combination.

1.5. OVERVIEW

The rest of the dissertation is divided into the following chapters:

Chapter 2 defines most of the terms for the rest of the paper. It defines a formal model of hypercube variant networks. This model is based on linear spaces defined by vectors and matrices over $\mathcal{Z}_2$. It defines both the double matrix networks (*DM* networks) and the linear equation networks (*LE* networks). It also shows how many of the currently published hypercube variant networks can be expressed using *LE* networks.

Chapters 3 and 4 deal with the over-generality of the *LE* networks, in that the model allows disconnected networks and allows multiple descriptions for the same network. Chapter 3 shows necessary and sufficient conditions for several graph-theoretic properties of the linear equation networks, e.g., redundant channels and connectedness. Chapter 4 deals with several problems in network isomorphism.

Chapters 5 and 6 discuss general routing algorithms for *LE* networks. Chapter 5 defines the concept of expansions, and uses them to deal with minimal routing on a *DM* network. This is preparation for Chapter 6, which deals with routing on a *LE* network. Chapter 6 also shows that routing on a *LE* network is NP-complete, and discusses what conditions will permit efficient routing algorithms for the published networks.

Chapter 7 deals with several variations on routing, including non-minimal routing and wormhole routing. It discusses non-minimal routing algorithms. It also examines how the routing algorithms can be adapted to wormhole or circuit-switched routing strategies.

Chapter 8 discusses direct one-to-one embeddings of such networks as Hamiltonian cycles and binomial trees, and discusses squashed and stretched embeddings of such networks as binary trees and meshes. It also shows that the hypercube can be efficiently emulated on many *LE* networks and vice versa.

Chapter 9 discusses other communication algorithms, including broadcasting. It discusses several algorithms that can be implemented more efficiently on *LE* networks than on hypercube.

Chapter 10 gives a description of the static properties of the networks in the twisted cube family, and derives bounds on static performance measures of networks in the formal model, i.e., network diameter, mean internodal distance, etc.

Chapter 11 defines a simulation environment for testing the dynamic properties of the *LE* networks and routing algorithms. It uses this simulation to show some of the dynamic properties of networks and routing algorithms. It also examines what properties lead to better performance in *LE* networks.

Chapter 12 summarizes our conclusions and recommendations for designing and using *LE* networks, and lists some directions of future research.

2. DEFINITION OF LINEAR EQUATION NETWORKS

In this chapter, we define the class of linear equation networks and show that many new and existing Twisted Cube networks are in the class. We also show that some networks cannot be described by linear equation networks.

2.1. NETWORK DEFINITIONS

A *network* is constructed from two basic elements: channels and nodes. A *channel* is a fixed unidirectional link between two nodes which can transfer information across a serial or parallel line. A *node* has two parts: a processing element and a router. The *processing element* performs general computations. The *router* is a switching element that connects the processing element to a number of incoming and outgoing channels. The router is capable of connecting any permutation of the incoming channels to the outgoing channels, and is usually implemented as a crossbar switch.

The graph-theoretic structure of the network is its *topology.* The topology of a network can be described as a directed graph $G = (V, E)$, where V is a set of nodes (vertices) and E is a set of channels (directed edges). The topology of the network has a one-to-one correspondence with the physical structure of the network. Nodes correspond to vertices of the graph and channels correspond to edges of the graph. When convenient, we will use network terminology and graph-theoretic terminology interchangeably.

The most common way to describe the hypercube's topology is to assign each node a unique address from the vector space $\mathcal{Z}_2^n$, and allow a channel between only

nodes $\vec{X}$ and $\vec{Y}$ iff the Hamming weight of $\vec{X}+\vec{Y}$ is 1, that is, the addresses $\vec{X}$ and $\vec{Y}$ differ by only one component.

The hypercube's topology can also be described as a Cayley graph, that is, as the graph associated with the group of permutations generated by a set of permutation operators acting on a finite set. The Cayley graph representation of the hypercube uses a set of $2n$ elements and n permutation operators. For instance, the 3-dimensional hypercube can be represented using the set of permutations $\{(213456), (124356), (123465)\}$ acting on the set (123456).

In fact, Cayley graphs are a useful formal description of a large number of networks that are vertex symmetric [5] [7], [8]. However, a number of the Twisted Cube networks in the current literature are (in general) neither vertex symmetric nor edge symmetric [21] and are not members of the Cayley graph family.

2.1.1. Double Matrix Networks

The hypercube graph can also be described using a vector basis B over $\mathcal{Z}_2^n$, where n is the dimension of the hypercube. Let the set of basis vectors $B_i \in B$ be $B_i = e_i$ for $1 \leq i \leq n$, where e_i is the vector with a 1 in position i and zeroes elsewhere. Then the nodes V and the channels E of the network can be defined for $1 \leq i \leq n$ by:

$$V = \mathcal{Z}_2^n$$

$$E = \{(\vec{X} \in V, \vec{Y} \in V) : \vec{X} + \vec{Y} = B_i\}$$

The basis B can be represented as an $n \times n$ matrix, where B_i is the i-th column of the matrix B.

This model as presented can describe only one network – the hypercube. Any other basis B over $\mathcal{Z}_2^n$ will describe a network isomorphic to the hypercube.

We can extend the description by using two matrices instead of one. This addition defines a family of digraphs of 2^n nodes that are regular and have out-degree $2n$.

Definition 2.1.1 *The **double matrix network** (**DM network**) is defined as $G = (B^0, B^1)$ of dimension n, where B^0 and B^1 are each n-element sets of n-element vectors over $\mathcal{Z}_2^n$, and nodes and channels are defined for $\phi \in \{0,1\}$ and $1 \leq i \leq n$ by:*

$$V = \mathcal{Z}_2^n$$

$$E = \{(\vec{X} \in V, \vec{Y} \in V) : \vec{X} + \vec{Y} = B_i^{\phi}\}$$

Both B^0 and B^1 can be described as $n \times n$ matrices, by placing the i-th element of B^0 or B^1 on the i-th column of the matrix B^0 or B^1, respectively.

The hypercube Q_n is trivially described by a *DM* network, by setting $B^0 = \mathbf{I}$ and $B^1 = \mathbf{0}$, though there are two channels for every one channel in the original hypercube. The Folded Hypercube of [44] and [6] and the Enhanced Hypercube of [54] [36] are also described by *DM* networks. The n-dimensional Folded Hypercube is described by the set of vectors:

$$B^0 = \{e_1, e_2, \ldots, e_n\}$$

$$B^1 = \{e_1 + e_2, e_2 + e_3, \ldots, e_{n-1} + e_n, e_n\}$$

The n-dimensional Enhanced Hypercube is described by the set of vectors:

$$B^0 = \{e_1, e_2, \ldots, e_n\}$$

$$B^1 = \{e_1 + e_2 + \ldots + e_n, e_2, \ldots, e_{n-1}, e_n\}$$

If we don't want the extra channels, we can replace columns 2 through n of B^1 with $\vec{0}$.

We also describe a special class of the *DM* networks, in which B^0 and B^1 are restricted:

Definition 2.1.2 *The **lower triangular double matrix network (**LTDM **network)** is a* DM *network with the following properties:* B^0 *and* B^1 *are lower triangular matrices, with* $B^0_{i,i} = B^1_{i,i} = 1$ *for* $1 \leq i \leq n$.

Notice that the n-dimensional hypercube and both n-dimensional Möbius cubes are subgraphs of the Folded Hypercube. This suggests that we may be able to use one common description to describe the Twisted Cube networks.

2.1.2. Linear Equation Networks

The *DM* network model is too broad for our needs. It defines up to $2n$ channels per node, when only n are desired. Clearly, it does not preserve the hypercube's resources.

We can describe a class of networks that incorporates most of the Twisted Cube networks by introducing a selector function $SEL : \mathcal{Z}_2^n \rightarrow \mathcal{Z}_2$ that will choose which channels from B^0 and B^1 are actually used. A channel $(\vec{X}, \vec{Y})$ is then defined iff:

$$\vec{X} + \vec{Y} = B_i^{SEL(\vec{X})_i}$$

The selector function, $SEL(\vec{X})$, is usually a linear function, that is,

$$SEL(\vec{X}) = A\vec{X}$$

where A is an $n \times n$ matrix.

The choice of B_i^0 or B_i^1 is always forced by the selector function, that is, exactly one of B_i^0 and B_i^1 specifies a channel leading from $\vec{X}$.

Definition 2.1.3 *The **linear equation network*** (LE *network) is defined by* $G = (B^0, B^1, A)$ *of dimension* n*, where* B^0 *and* B^1 *are* n*-element sets of* n*-element vectors over* $\mathcal{Z}_2^n$*, and* A *is an* $n \times n$ *matrix over* $\mathcal{Z}_2^n$*, has nodes and* ***directed*** *channels defined for* $1 \leq i \leq n$ *by:*

$$V = \mathcal{Z}_2^n$$

$$E = \{(\vec{X} \in V, \vec{Y} \in V) : \vec{X} + \vec{Y} = B_i^{(A\vec{X})_i}\}$$

The *LE* networks are clearly related to the *DM* networks in that a *DM* network $G_1 = (B^0, B^1)$ always contains the *LE* network $G_2 = (B^0, B^1, A)$ as a subgraph. However, while the former is a digraph, the latter may not be, because the set E contains only directed channels.

As before, both B^0 and B^1 can be described by $n \times n$ matrices, by placing the i-th vector in the i-th column of the matrix. These two matrices, plus the matrix A, can be used to completely describe a resource preserving hypercube variant.

Example: The Twisted 3-Cube of Figure 1.2 can be defined by the 3 matrices:

$$B^0 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, B^1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}, A = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

∎

For much of this dissertation, we will be considering special cases of *LE* networks, where B^0, B^1, and A are restricted matrices. These special cases include:

Definition 2.1.4 *A **lower triangular linear equation network*** (**LTLE *network***) $G = (B^0, B^1, A)$ *is a* LE *network with the following properties:* B^0 *and* B^1 *are lower triangular matrices, with* $B^0_{i,i} = B^1_{i,i} = 1$ *for* $1 \leq i \leq n$*.* A *is a strictly lower triangular matrix, with* $A_{i,i} = 0$ *for* $1 \leq i \leq n$*.*

Definition 2.1.5 *A* ***banded lower triangular linear equation network*** ***(banded* LTLE *network*** $G = (B^0, B^1, A)$ *is a* LE *network with the following properties:* B^0 *and* B^1 *are banded lower triangular matrices, with* $B^0_{i,i} = B^1_{i,i} = 1$ *for* $1 \leq i \leq n$, *and* $B^0_{i,j} = B^1_{i,j} = 0$ *for* $i - j > k$ *and* k *constant.* A *is also strictly lower triangular, with* $A_{i,i} = 0, 1 \leq i \leq n$.

2.1.3. Neighbor Description of *LE* Networks

The above definitions of *DM* networks and *LE* networks define channels between nodes by explicitly defining a set of channels for the entire network. A second way to define the network is with a set of "neighbor" functions that define the channels from a given node.

For a *LE* network, we can define the n neighbors of a processor with address $\vec{X}$ by an $n \times n$ matrix $N(\vec{X})$, where each column i is the address of the i-th neighbor of $\vec{X}$:

$$N(\vec{X}) = \vec{X}\mathbf{1}_{1\times n} + B^0 + \left(B^0 + B^1\right) DIAG(A\vec{X})$$

where 1 is the matrix containing only 1's in each position, and $DIAG(\vec{X})$ returns an $n \times n$ matrix with X_i in element (i, i) and zeroes elsewhere.

The linear function uses $\vec{X}$ to pick the columns of $N(\vec{X})$ so that B^0_i is a column i of $N(\vec{X})$ iff $(A\vec{X})_i = 0$ and B^1_i is a column i of $N(\vec{X})$ iff $(A\vec{X})_i = 1$. The choice is always forced, that is, exactly one of $\vec{X} + B^0_i$ or $\vec{X} + B^1_i$ is in $N(\vec{X})$.

$N(\vec{X})$ can be computed in $O(n^2)$ bit operations. Although there is one matrix multiplication, this operation can be simplified to just comparisons and value copying. The rest of the operations run in only $O(n^2)$ bit operations using classical matrix algorithms.

We can use a simpler computation to compute the i-th neighbor of $\vec{X}$:

$$N_i(\vec{X}) = \vec{X} + B_i^0 + (B_i^0 + B_i^1)[A\vec{X}]_i = \vec{X} + B_i^{(A\vec{X})_i}$$

$N_i(\vec{X})$ can be computed in $O(n)$ bit operations. The operation takes a $(1 \times n)$ by $(n \times 1)$ matrix multiply to compute $(A\vec{X})_i$ (by multiplying the i-th row of A with $\vec{X}$) and a n-element vector addition (of $\vec{X}$ and a column of B^0 or B^1), which totals to $3n = O(n)$ bit operations.

We note what conditions will make N_i one-to-one:

Lemma 2.1.1 *The neighbor function $N_i(\vec{X}) = \vec{X} + B_i^{(A\vec{X})_i}$ is one-to-one and onto iff any one of the following conditions is true:*

- $row_i(A) = \mathbf{0}$
- $B_i^0 = B_i^1$
- $(AB_i^0)_i = (AB_i^1)_i$

Proof: if N_i is 1-1, it is also clearly onto, because its domain and range are the same size.

Consider if $row_i(A) = \mathbf{0} \vee B_i^0 = B_i^1$. If $\text{row}_i(A) = \mathbf{0}$, then only B_i^0 would legally define a channel. if $B_i^0 = B_i^1$, the two vectors are the same, so either one will define the same channel. Since $\vec{X_1} \neq \vec{X_2} \Rightarrow \vec{X_1} + B_i^0 \neq \vec{X_2} + B_i^0$, the neighbor function N_i is 1-1 in this case.

Now consider if $(AB_i^0)_i = (AB_i^1)_i$. As before, we know that:

$$\vec{X_1} \neq \vec{X_2} \Rightarrow \vec{X_1} + B_i^0 \neq \vec{X_2} + B_i^0$$

$$\vec{X_1} \neq \vec{X_2} \Rightarrow \vec{X_1} + B_i^1 \neq \vec{X_2} + B_i^1$$

So any two vectors $\vec{X_1}$ and $\vec{X_2}$ with $(A\vec{X_1})_i = (A\vec{X_2})_i$ will not map to the same image under N_i. But now consider $(A\vec{X_1})_i \neq (A\vec{X_2})_i$. Let (wlog) $(A\vec{X_1})_i = 0$ and

$(A\vec{X}_2)_i = 1$. If $(AB_i^0)_i = (AB_i^1)_i = 0$, then $(A\vec{X}_1)_i + (AB_i^0)_i = (A[\vec{X}_1 + B_i^0])_i = 0$ and $(A\vec{X}_2)_i + (AB_i^1)_i = (A[\vec{X}_2 + B_i^1])_i = 1$. Since $\vec{X}_1 + B_i^0$ is the image of $\vec{X}_1$ under N_i and $\vec{X}_2 + B_i^1$ is the image of $\vec{X}_2$ under N_i, the two vectors cannot map to the same image under N_i. If we assume $(AB_i^0)_i = (AB_i^1)_i = 1$, then using the same argument, $\vec{X}_1$ and $\vec{X}_2$ do not map to the same image under N_i.

If we assume that $\text{row}_i(A) \neq \mathbf{0} \wedge B_i^0 \neq B_i^1 \wedge (AB_i^0)_i \neq (AB_i^1)_i$. Then all we have to do is choose $\vec{X}_1 + B_i^0 = \vec{X}_2 + B_i^1$, and these will map to the same image under N_i. ∎

From this proof, we can conclude that all lower triangular networks have N_i be 1-1 for $1 \leq i \leq n$.

2.1.4. Definitions

There will be a number of definitions related to *LE* networks that will be used throughout this dissertation. These definitions refer mostly to properties of and relationships between channels in the network. These terms are defined here for convenience.

First, we assign a name to an element of either set B^0 or B^1:

Definition 2.1.6 *Given a network $G = (B^0, B^1, A)$, a **term** is an element of either set B^0 or B^1. A term is also a column of either matrix B^0 or B^1.*

We also define a label for the relationship between terms and channels in both *DM* networks and *LE* networks.

Definition 2.1.7 *A term B_i^ϕ **defines** a directed channel $(\vec{X}, \vec{Y})$ iff $\vec{X} + \vec{Y} = B_i^{(A\vec{X})_i}$, and **does not define** channel $(\vec{X}, \vec{Y})$ otherwise.*

Clearly every term in a column of B^0 or B^1 "defines" a channel from any given node in a *DM* network. But in *LE* networks, a term may not define a channel from a particular node, because a particular channel's existence in a *LE* network depends on the source address and the selector function.

The defined channels of a network have some properties, based on the weight of the term that defines them.

Definition 2.1.8 *A channel* $(\vec{X}, \vec{X} + B_i^{(A\vec{X})_i})$ ***spans*** *dimension* j *iff* $B_{j,i}^{(A\vec{X})_i} = 1$.

Definition 2.1.9 *A channel is a* ***twisted*** *channel iff it spans more than one dimension.*

A term may define a channel from a given node and not define a channel from a node neighboring the first, or vice versa. It will be useful for routing algorithms to know whether a channel exists at a neighboring node, especially if does not exist at the current node.

Assume that the channels of a *LE* network $G = (B^0, B^1, A)$ are contained in the set E. Assume also that we are at a node $\vec{X}$ and that the term $B_j^{(A\vec{X})_j}$ defines the channel $(\vec{X}, \vec{X} + B_j^{(A\vec{X})_j})$. Let $\vec{Y} = \vec{X} + B_i^{(A\vec{X})_i}$ be a node adjacent to $\vec{X}$. We note that if the term $B_j^{(A\vec{X})_j}$ does not define the channel $(\vec{Y}, \vec{Y} + B_j^{(A\vec{Y})_j})$, then:

$$(\vec{X}, \vec{X} + B_j^{(A\vec{X})_j}) \in E \Leftrightarrow (\vec{X}, \vec{X} + B_j^{(A\vec{X})_i + (AB_i^{(A\vec{X})_i})_j}) \notin E$$

Which implies:

$$(A\vec{X})_j \neq (A\vec{X})_j + (AB_i^{(A\vec{X})_i})_j$$

So in binary $(AB_i^{(A\vec{X})_i})_j$ is forced to be 1.

Simply put, a term B_i^{ϕ} that defines a channel leading from $\vec{X}$ will not define a channel leading from $\vec{Y} = \vec{X} + B_j^{\psi}$ and vice versa iff $(AB_j^{\psi}) = 1$. We describe this relationship by the definition below:

Definition 2.1.10 *The terms B_i^0 and B_i^1* ***depend on*** *another term B_j^ψ iff $B_i^0 \neq B_i^1$ and $(AB_j^\psi)_i = 1$.*

B_i^0 and B_i^1 are not dependent on B_j^ψ if $B_i^0 = B_i^1$, because:

$$\begin{aligned}(\vec{X}, \vec{X} + B_i^1) \in E &\Rightarrow (\vec{X}, \vec{X} + B_i^0) \in E \\ &\Rightarrow (\vec{Y}, \vec{Y} + B_i^1) \in E \\ &\Rightarrow (\vec{Y}, \vec{Y} + B_i^0) \in E\end{aligned}$$

Example: Consider the term B_1^0 in the network defined below:

$$B^0 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, B^1 = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}, A = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

For B_1^1, the product is:

$$(AB_1^1) = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$$

and so both B_2^0 and B_2^1 depend on B_1^1, but B_1^0, B_1^1, B_3^0 and B_3^1 do not depend on B_1^1. ∎

Pre-computing $(AB_i^\phi)_j$ for all $i, j \in \{1, \ldots, n\}$ requires $O(n^3)$ bit operations (using classical multiplication) and fills a table of $2n^2$ bits. Once we have the table, we can look up whether a term depends on another or not in at most $O(n)$ bit operations.

2.2. NETWORKS IN THE *LE* NETWORK MODEL

Most of the Twisted Cube networks we described in Section 1.3 can be represented by the *LE* model given in Definition 2.1.3. We describe how to describe

each of the published Twisted Cube networks using the model, and explain why two cannot be described using the model. We will also introduce two new networks and give their descriptions.

2.2.1. Existing *LE* Networks

The hypercube is trivially included in the *LE* networks:

Theorem 2.2.1 *The hypercube can be described as an* LE *network.*

Proof: For the hypercube, set $B^0 = I$, $B^1 = I$, and $A = 0$. ∎

Our own Möbius cubes can also be expressed as *LE* networks:

Theorem 2.2.2 *The Möbius Cubes can be described as* LE *networks.*

Proof: This is the most direct representation of the twisted cube networks. For the 0-Möbius Cube, set B_i^0, B_i^1, and $A_{i,j}$, $1 \leq i \leq n$ and $1 \leq j \leq n$, to:

$$B_i^0 = e_i,$$

$$B_i^1 = \sum_{k=i}^{n} e_k,$$

$$\begin{aligned} A_{i,j} &= 1, && i = j-1 \\ &= 0, && i \neq j-1 \end{aligned}$$

For the 1-Möbius Cube, set B_i^0, B_i^1, and $A_{i,j}$, $1 \leq i \leq n$ and $1 \leq j \leq n$, to:

$$
\begin{aligned}
B_i^0 &= \textstyle\sum_{k=1}^{n} e_k, \quad i = 1 \\
B_i^1 &= e_i, \qquad\qquad i > 1
\end{aligned}
$$

$$
\begin{aligned}
B_i^0 &= e_1, \qquad\qquad i = 1 \\
B_i^1 &= \textstyle\sum_{k=i}^{n} e_k, \quad i > 1
\end{aligned}
$$

$$
\begin{aligned}
A_{i,j} &= 1, \qquad\qquad i = j \\
&= 0, \qquad\qquad i \neq j - 1
\end{aligned}
$$

∎

The only difference between the 0-Möbius Cube and the 1-Möbius Cube is that the 1-Möbius Cube has the two columns B_1^0 and B_1^1 exchanged. For instance, the six-dimensional 0-Möbius Cube can be described by the matrices:

$$
B^0 = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}
\quad
B^1 = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 & 1
\end{bmatrix}
\quad
A = \begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0
\end{bmatrix}
$$

The six-dimensional 1-Möbius Cube differs from the 0-Möbius Cube only in columns B_1^0 and B_1^1 :

$$B^0 = \begin{bmatrix} 1&0&0&0&0&0 \\ 1&1&0&0&0&0 \\ 1&0&1&0&0&0 \\ 1&0&0&1&0&0 \\ 1&0&0&0&1&0 \\ 1&0&0&0&0&1 \end{bmatrix} \quad B^1 = \begin{bmatrix} 1&0&0&0&0&0 \\ 0&1&0&0&0&0 \\ 0&1&1&0&0&0 \\ 0&1&1&1&0&0 \\ 0&1&1&1&1&0 \\ 0&1&1&1&1&1 \end{bmatrix} \quad A = \begin{bmatrix} 0&0&0&0&0&0 \\ 1&0&0&0&0&0 \\ 0&1&0&0&0&0 \\ 0&0&1&0&0&0 \\ 0&0&0&1&0&0 \\ 0&0&0&0&1&0 \end{bmatrix}$$

The Generalized Twisted Cube (GQ_n) of Chedid and Chedid can also be expressed in our general model. In Chedid and Chedid's paper, GQ_3 is defined differently from the Twisted 3-Cube representation in Section 2.1:

Theorem 2.2.3 *The Generalized Twisted Cubes can be described as a* LE *network.*

Proof: Set B_i^0, B_i^1, and $A_{i,j}$, $1 \leq i \leq n$ and $1 \leq j \leq n$, to:

$$B_i^0 = e_i,$$

$$\begin{aligned} B_i^1 &= e_i + e_{i+1}, \quad i = 3k+2,\ 0 \leq k \leq n/3 \\ &= e_i \qquad\qquad \text{otherwise} \end{aligned}$$

$$\begin{aligned} A_{i,j} &= 1, \qquad i = j+1 \text{ and } i = 3k+1,\ 0 \leq k \leq n/3 \\ &= 0, \qquad \text{otherwise} \end{aligned}$$

∎

The 6-dimensional Generalized Twisted Cube's matrix description is:

$$B^0 = \begin{bmatrix} 1&0&0&0&0&0 \\ 0&1&0&0&0&0 \\ 0&0&1&0&0&0 \\ 0&0&0&1&0&0 \\ 0&0&0&0&1&0 \\ 0&0&0&0&0&1 \end{bmatrix} \quad B_1 = \begin{bmatrix} 1&0&0&0&0&0 \\ 0&1&0&0&0&0 \\ 0&1&1&0&0&0 \\ 0&0&0&1&0&0 \\ 0&0&0&0&1&0 \\ 0&0&0&0&1&1 \end{bmatrix} \quad A = \begin{bmatrix} 0&0&0&0&0&0 \\ 1&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&1&0&0 \\ 0&0&0&0&0&0 \end{bmatrix}$$

This version of the Generalized Twisted Cube is different from the one given by Chedid and Chedid. They used a different node labeling of the Twisted 3-Cube that has a different matrix description. This network is trivially isomorphic to the Generalized Twisted Cube, because the Twisted 3-cubes used in the graph composition construction are isomorphic. Its only advantage is that it is a *LTLE* network.

Before the Generalized Twisted Cube was published, an associate had devised a very similar network generalization we called the "JimTwist Cube" [37] which has the matrix description above.

Not surprisingly, these two descriptions are not the only formulations of this network. The Varietal Hypercube of Cheng and Chuang [13] has exactly the same description as the Generalized Twisted Cube.

The Folded Hypercube of Kim and Shin [40] has exactly half of the edges in one dimension twisted. This makes the description simple:

Theorem 2.2.4 *The Folded Hypercube can be described as a* LE *network.*

Proof: Set B_i^0, B_i^1, and $A_{i,j}$, $1 \leq i \leq n$ and $1 \leq j \leq n$, to:

$$B_i^0 = e_i,$$

$$\begin{aligned} B_i^1 &= e_i + \ldots + e_n, \quad i = 2 \\ &= e_i \qquad\qquad \text{otherwise} \end{aligned}$$

$$\begin{aligned} A_{i,j} &= 1, \qquad\qquad i = 2, j = 1 \\ &= 0, \qquad\qquad \text{otherwise} \end{aligned}$$

∎

The 6-dimensional Folded Hypercube's matrix description is:

$$B^0 = \begin{bmatrix} 1&0&0&0&0&0 \\ 0&1&0&0&0&0 \\ 0&0&1&0&0&0 \\ 0&0&0&1&0&0 \\ 0&0&0&0&1&0 \\ 0&0&0&0&0&1 \end{bmatrix} B_1 = \begin{bmatrix} 1&0&0&0&0&0 \\ 0&1&0&0&0&0 \\ 0&1&1&0&0&0 \\ 0&1&0&1&0&0 \\ 0&1&0&0&1&0 \\ 0&1&0&0&0&1 \end{bmatrix} A = \begin{bmatrix} 0&0&0&0&0&0 \\ 1&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \end{bmatrix}$$

The Twisted Cube of Hilbers is one of the earliest Twisted Cube networks.

Theorem 2.2.5 *The Twisted Cube can be described as a* LE *network.*

Proof: Set B_i^0, B_i^1, and $A_{i,j}$, $1 \leq i \leq n$ and $1 \leq j \leq n$, to:

$$
\begin{aligned}
B_i^0 &= e_i, && i = 2k+1,\ 0 \le k \le n/2 \\
&= e_i + e_{i-1}, && i = 2k,\ 0 \le k < n/2 \\
&= e_i, && i = n
\end{aligned}
$$

$$
B_i^1 = e_i
$$

$$
\begin{aligned}
A_{i,j} &= 1, && i = 2k \text{ and } i > j \\
&= 0, && \text{otherwise}
\end{aligned}
$$

∎

This is the description of the Twisted Cube given by Seth Abraham [2], and is slightly different from the definition of the Twisted Cube edges from [34]. The six-dimensional Twisted cube the matrix description:

$$
B^0 = \begin{bmatrix}
1 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}
B^1 = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}
A = \begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
$$

The MCube described by Singhvi and Ghose is an alternate description of generalizing the Twisted 3-Cube. One specific network they define is the Flip MCube.

Theorem 2.2.6 *The Flip MCube can be described as a* LE *network.*

Proof: Set B_i^0, B_i^1, and $A_{i,j}$, $1 \le i \le n$ and $1 \le j \le n$, to:

$$
\begin{aligned}
B_i^0 &= e_i, && 1 \le i \le n \\
\\
B_i^1 &= e_i + e_{i+1}, && 1 \le i \le n-2 \\
&= e_i, && n-1 \le i \le n \\
\\
A_{i,j} &= 1, && i > j \\
&= 1, && j = n \\
&= 0, && \text{otherwise}
\end{aligned}
$$

. ∎

For example, the six-dimensional Flip MCube can be represented using the matrices:

$$
B^0 = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}
B^1 = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}
A = \begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 1 \\
1 & 1 & 0 & 0 & 0 & 1 \\
1 & 1 & 1 & 0 & 0 & 1 \\
1 & 1 & 1 & 1 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 & 1
\end{bmatrix}
$$

(The last two rows of the A matrix can actually be anything we wish, because the last two columns of B^0 and B^1 are equal – it doesn't matter which we choose.)

2.2.2. New *LE* Networks

There are two new networks that we introduce in this dissertation. These are the Bent Cube [27], and the YAT Cube - an acronym for "Yet Another Twisted."

In searching for networks that offer a compromise between lower diameter and a simple routing algorithm, we discovered the Bent Cube. This network offers

a diameter that is the same as the Generalized Twisted Cube, but offers a smaller expected distance.

The Bent Cube is constructed by setting B_i^0, B_i^1, and $A_{i,j}$, $1 \leq i \leq n$ and $1 \leq j \leq n$, to:

$$B_i^0 = e_i,$$

$$\begin{aligned} B_i^1 &= e_i + e_{i+1}, \quad 1 \leq i < n \\ &= e_n, \qquad\qquad i = n \end{aligned}$$

$$\begin{aligned} A_{i,j} &= 1, \qquad\qquad i = j + 1 \\ &= 0, \qquad\qquad i \neq j + 1 \end{aligned}$$

For instance, the Bent Cube of six dimensions can be represented using the matrices:

$$B^0 = \begin{bmatrix} 1&0&0&0&0&0 \\ 0&1&0&0&0&0 \\ 0&0&1&0&0&0 \\ 0&0&0&1&0&0 \\ 0&0&0&0&1&0 \\ 0&0&0&0&0&1 \end{bmatrix} \quad B^1 = \begin{bmatrix} 1&0&0&0&0&0 \\ 1&1&0&0&0&0 \\ 0&1&1&0&0&0 \\ 0&0&1&1&0&0 \\ 0&0&0&1&1&0 \\ 0&0&0&0&1&1 \end{bmatrix} \quad A = \begin{bmatrix} 0&0&0&0&0&0 \\ 1&0&0&0&0&0 \\ 0&1&0&0&0&0 \\ 0&0&1&0&0&0 \\ 0&0&0&1&0&0 \\ 0&0&0&0&1&0 \end{bmatrix}$$

The diameter of the network is not difficult to derive [26]. The address pairs:

$$\vec{X} = (000000\ldots000)$$

$$\vec{Y} = (110110\ldots110)$$

require exactly $2n/3$ steps to route between. For any n, the diameter is $\lceil 2n/3 \rceil + n \bmod 3$ – larger than the diameters of most published networks, with the exception of the Generalized Twisted Cubes.

The YAT cube is related to the Flip MCube. It is identical to the Flip MCube, except that the two columns B_n^0 and B_n^1 are replaced by $e_{n-1} + e_n$. The six dimensional YAT cube can be represented by:

$$B^0 = \begin{bmatrix} 1&0&0&0&0&0 \\ 0&1&0&0&0&0 \\ 0&0&1&0&0&0 \\ 0&0&0&1&0&0 \\ 0&0&0&0&1&1 \\ 0&0&0&0&0&1 \end{bmatrix} \quad B^1 = \begin{bmatrix} 1&0&0&0&0&0 \\ 1&1&0&0&0&0 \\ 0&1&1&0&0&0 \\ 0&0&1&1&0&0 \\ 0&0&0&1&1&1 \\ 0&0&0&0&0&1 \end{bmatrix} \quad A = \begin{bmatrix} 0&0&0&0&0&1 \\ 1&0&0&0&0&1 \\ 1&1&0&0&0&1 \\ 1&1&1&0&0&1 \\ 1&1&1&1&0&1 \\ 1&1&1&1&1&1 \end{bmatrix}$$

2.3. NETWORKS THAT ARE NOT *LE* NETWORKS

There are a number of published networks that cannot be represented as a *LE* network. These include the Twisted N-Cube and the Multiply Twisted Cube. Interestingly, the Twisted N-Cube cannot be described using a *LE* network because it has too few twisted channels, and the Multiply Twisted Cube cannot be described because it has "too many" twisted channels.

Also, though the Flip MCube is a *LE* network, many networks that are MCubes are not *LE* networks.

We begin by counting the number of twisted channels in a *LE* network.

Lemma 2.3.1 *Let G be a* LE *network. Exactly $k2^{n-1}$ channels of G are "twisted", for some $0 \leq k \leq 2n$.*

Proof: Let T_i denote the number of twisted channels in dimension i, and let $W_H(\vec{X})$ be the Hamming weight of $\vec{X}$. For a single dimension i, the selector function $(A\vec{X})_i$ is a parity function over $\mathcal{Z}_2^n$.

If row i of A is $\vec{0}$, then $(A\vec{X})_i = 0$ for all $\vec{X}$. The number of twisted channels in dimension i are:

1. If $W_H(B_i^0) < 2$, then $T_i = 0$.

2. If $W_H(B_i^0) \geq 2$, then $T_i = 2^n = 2(2^{n-1})$

If row i of A is nonzero, then $(A\vec{X})_i = 0$ for half of $\vec{X}$ in $\mathcal{Z}_2^n$, and $(A\vec{X})_i = 1$ for the rest. The number of twisted channels in dimension i are:

1. If $W_H(B_i^0) < 2$ and $W_H(B_i^1) < 2$, then $T_i = 0$.

2. If $W_H(B_i^0) \geq 2$ or $W_H(B_i^1) \geq 2$, then $T_i = 2^{n-1}$.

3. If $W_H(B_i^0) \geq 2$ and $W_H(B_i^1) \geq 2$, then $T_i = 2(2^{n-1})$.

The total number of twisted channels in the entire network is a multiple of 2^{n-1}. The smallest number of twisted channels is 0 and the largest (of course) is $n2^n = (2n)(2^{n-1})$. ∎

This lemma allows us to show that the Twisted N-Cube is not an LE network.

Theorem 2.3.1 *The Twisted N-Cube cannot be described using an* LE *network for dimension $n > 3$.*

Proof: The Twisted N-Cube has exactly four processors with twisted channels in only one dimension of the entire network. A LE network allows only $k2^{n-1}$ twisted channels in the network and so can have exactly four twisted channels for only $n = 2$ or $n = 3$. ∎

If we relax the definition of the network so that the selector function $SEL_i(\vec{X})$ is not a linear function, we can then describe the Twisted N-Cube using B^0 and B^1 matrices and a selector function that returns 1 iff $\vec{X}$ is one of the four nodes with a twisted channel.

The Crossed Cube cannot be represented even using a non-linear selector function:

Theorem 2.3.2 *The Crossed Cube cannot be described using a* LE *network for dimension* $n > 3$.

Proof: Consider an n-dimensional Crossed Cube CQ_n. The neighbor function N_1 for a *LE* network can allow only two possible neighbors for $\vec{X}$: $\vec{X} + B_1^0$ and $\vec{X} + B_1^1$.

The Crossed Cube architecture has its neighbors defined by a "pair-wise" relation. Two binary strings $x = x_1x_2$ and $y = y_1y_2$ are *pair-related*, denoted $x\tilde{y}$ iff $(x, y) \in \{(00, 00), (10, 10), (01, 11), (11, 01)\}$.

The channel $(u_1 \ldots u_n, v_1 \ldots v_n)$ is in CQ_n iff for some l, we have all of the following:

- $u_1 \ldots u_{l-1} = v_1, v_{l-1}$
- $u_l \neq v_l$
- $u_{l+1} = v_{l+1}$ if $n - l$ is odd
- for $1 \leq i \leq \lfloor (n-l)/2 \rfloor$, $u_{n-2i}u_{n-2i+1}\tilde{v}_{n-2l}v_{n-2i+1}$

(The indexing here is slightly changed from the definition in [30]).

The Crossed Cubes CQ_1, CQ_2, CQ_3 and CQ_4 can be described using *LE* networks. However, CQ_5 cannot be described using a *LE* network. In a *LE* network, an edge $(\vec{X}, \vec{Y})$ exists only if $\vec{X} + \vec{Y} = B_i^\phi$, where $1 \leq i \leq n$ and $\phi \in \{0, 1\}$. For channels in dimension i, our model allows only B_i^0 and B_i^1 to be the mod 2 sum between connected nodes. But CQ_n allows at least four different vector sums in the cube's $(n-4)$-th dimension:

$(0\ldots0000000, 0\ldots0100000) \in E,$

implies $0\ldots0000000 + 0\ldots0100000 = 0\ldots0100000$

$(0\ldots0000001, 0\ldots0100011) \in E,$

implies $0\ldots0000001 + 0\ldots0100011 = 0\ldots0100010$

$(0\ldots0000100, 0\ldots0101100) \in E,$

implies $0\ldots0000100 + 0\ldots0101100 = 0\ldots0101000$

$(0\ldots0000001, 0\ldots0100011) \in E,$

implies $0\ldots0000101 + 0\ldots0101111 = 0\ldots0101010$

So CQ_n is impossible to represent for $n \geq 5$. ∎

While the Flip MCube is a *LE* network, Singhvi and Ghose describe a whole set of networks called MCubes. This is because an MCube of dimension n is recursively defined from two MCubes of dimension $n-1$. More than one MCube can be designed under this specification, because the authors allow any arbitrary orientation of the two sub-MCubes.

The set of networks described by the MCube construction and the set of the *LE* networks intersect with each other, but neither set is a subset of the other. This is trivially shown. First, the set of *LE* networks contains disconnected networks, while the MCube model does not. Second, the (2,2)-MCube network shown in Figure 2.1 cannot be represented using *LE* networks - the number of twisted edges in each dimension of the MCube is incorrect, by Lemma 2.3.1.

However, these networks can be described if we again lift the restriction that the selector function must be linear. Since each channel in dimension i of an MCube

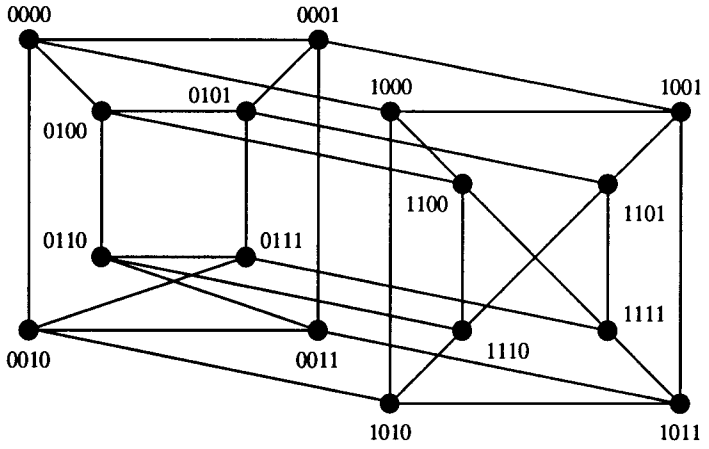

Figure 2.1. A (2,2)-MCube that is not a *LE* network.

network can either be a hypercube channel or twisted channel across dimensions i and $i+1$, $2n$ vectors are enough to describe all the possible channels.

2.4. SUMMARY

We have described a linear equation model for representing interconnection networks, and have shown that a number of published Twisted Cube networks are included in the family of networks described by this model. We have also developed several new networks using this model. Though not all Twisted Cube networks can be described using *LE* networks, we have shown that some extensions to the *LE* network model can include those networks as well.

3. NETWORK PROPERTIES

In this chapter, we examine some basic properties of general *LE* networks, including conditions for network connectedness. We show that a number of these properties are *NP*-complete or *NP*-hard. We then show that all *LTLE* networks share many properties, including connectedness.

3.1. CHANNEL PROPERTIES OF *LE* NETWORKS

The *LE* network uses unidirectional channels in its definition. This allows it to be general enough to describe most published Twisted Cube networks. Unfortunately, this allows it to describe not only networks in which all channels are bidirectional (using two unidirectional channels), but it can also describe networks that have strictly unidirectional channels. It can also describe networks with self-looping or redundant channels.

We define three properties of directed channels in a general network, then show what conditions must hold for *DM* networks and *LE* networks to have channels with these properties. We then give algorithms to compute these conditions and bound their run time complexity.

Definition 3.1.1 *Let $G = (V, E)$ be any network. Then for any $\vec{X}, \vec{Y} \in V$:*

Reflexive *If there is a channel $(\vec{X}, \vec{X}) \in E$, it is a* reflexive *channel.*

Redundant *If there is more than one channel $(\vec{X}, \vec{Y}) \in E$, then all such channels are* redundant *channels.*

Reciprocal *If there is a channel* $(\vec{X}, \vec{Y}) \in E$ *and another channel* $(\vec{Y}, \vec{X}) \in E$, *then they are both* reciprocal *channels. (They may also be called a single* bidirectional *channel for convenience.)*

A desirable property of a network is that it has only reciprocal channels. The purpose of a channel is to communicate information, so it is not useful to have a reflexive channel. For instance, the two self-loops in a deBruijn network are removed when the network is implemented in hardware [45]. It also is not useful to have redundant channels in a network, unless there is a need to increase the traffic throughput between two adjacent nodes in the network. On the other hand, it is useful to have all channels in the network be reciprocal. A network with only reciprocal channels can always respond to a message using the same path, if needed.

What are the necessary and sufficient conditions for the two network classes above to have each kind of these channels? The two theorems below describe what conditions are needed to remove the undesirable types of channels from the network and ensure that only desirable channels are in the network. First, we consider channel properties on *DM* networks:

Theorem 3.1.1 *Let* $G = (B^0, B^1)$ *be an n-dimensional* DM *network. Then:*

- *G has* no *reflexive channels iff* $0 \notin \{B^0, B^1\}$.
- *G has* no *redundant channels iff* $\forall i, j \in \{1, \ldots, n\} : \forall \phi, \psi \in \{0, 1\} : B_i^\phi \neq B_j^\psi$.
- *Every channel in G is reciprocal.*

Proof: These follow directly from Definition 2.1.1 for *DM* networks and from addition of binary vectors. ∎

Next, we consider channel properties in *LE* networks:

Theorem 3.1.2 *Let* $G = (B^0, B^1, A)$. *Then:*

1. G *has* no *reflexive channels iff* $\forall \vec{X} \in \mathcal{Z}_2^n : \forall i, 1 \leq i \leq n : B_i^{(A\vec{X})_i} \neq 0$.

2. G *has* no *redundant channels iff* $\forall \vec{X} \in \mathcal{Z}_2^n : \forall i, \forall j, 1 \leq i \neq j \leq n : B_i^{(A\vec{X})_i} \neq B_j^{(A\vec{X})_j}$.

3. G *has* only *reciprocal channels iff* $\forall \vec{X} \in \mathcal{Z}_2^n : \forall i, 1 \leq i \leq n : \exists j, 1 \leq j \leq n : B_i^{(A\vec{X})_i} = B_j^{(A\vec{X})_j + (AB_i^{(A\vec{X})_i})_j}$.

Proof: Each of these follow from the definition of LE networks.

1. To have reflexive channels, we must have:

$$B_i^{(A\vec{X})_i} = 0 \Leftrightarrow \vec{X} + \vec{X} + B_i^{(AX_i)}$$
$$\Leftrightarrow (\vec{X}, \vec{X}) \in E$$

2. To have redundant channels, we must have:

$$B_i^{(A\vec{X})_i} = B_j^{(A\vec{X})_j} \Leftrightarrow \vec{X} + \vec{Y} = B_i^{(A\vec{X})_i} \wedge \vec{X} + \vec{Y} = B_j^{(A\vec{X})_j}$$
$$\Leftrightarrow (\vec{X}, \vec{Y}) \in E$$

3. To have only reciprocal channels, we must have:

$$(\vec{X}, \vec{Y}) \in E \iff \vec{X} + \vec{Y} = B_i^{(A\vec{X})_i}$$

And:

$$(\vec{Y}, \vec{X}) \in E \iff \vec{Y} + \vec{X} = B_j^{(A\vec{Y})_j} = B_j^{(A\vec{X})_j + (AB_i^{(A\vec{X})_i})_j}$$

∎

There are efficient methods to compute whether or not a *DM* network has reflexive, redundant, or non-reciprocal channels.

To compute if a *DM* network has reflexive channels, we need to find if $\vec{0} \in \{B^0, B^1\}$. This check would require at most $O(n^2)$ bit operations to compute. To find if a *DM* network has redundant channels, we need to find if $\exists \vec{X}, \vec{Y} \in \{B^0, B^1\} : \vec{X} = \vec{Y}$. This check would require at most $O(n^3)$ bit operations to compute. However, a *DM* network always has only reciprocal channels, by Theorem 3.1.1.

Computing if a *LE* network has channels with these properties will be somewhat more involved. In Figure 3.1, Algorithm ReflexiveChannel computes if any reflexive channels exist in a given network.

Theorem 3.1.3 *Let* $G = (B^0, B^1, A)$ *be an n-dimensional* LE *network. Algorithm ReflexiveChannel correctly tests to see any channels in G are reflexive in* $O(n^3)$ *bit operations.*

Proof: If some $B_i^\phi = 0$, then we must check if ϕ can be chosen by $(A\vec{X})_i$. For $\phi = 0$, $\vec{X} = 0$ will always make $(A\vec{X})_i = 0$; for $\phi = 1$, $A_{i,j} = 1$ and $\vec{X} = e_i$ will make $(A\vec{X})_i = 1$, but if $row_i(A) = 0$, then $(A\vec{X})_i$ can never be equal to one. This exhausts all possibilities for a reflexive channel at any node $\vec{X}$, so the algorithm is correct.

Finding all $B_i^\phi = \vec{0}$ takes $O(n^2)$ bit operations and determining if a given B_i^ϕ causes any reflexive channels take $O(n)$ bit operations, so at worst $0(n^3)$ bit operations are needed. ∎

In Figure 3.2, Algorithm RedundantChannel computes if any redundant channels exist in a given network.

Input: A n-dimensional LE network $G = (B^0, B^1, A)$.

Output: "Yes" iff G has any reflexive channels, "No" otherwise.

```
Procedure ReflexiveChannel( G = (B^0, B^1, A) )
    for each B_i^φ ∈ {B^0, B^1} do
        if B_i^φ = 0 then
            if φ = 0 then
                output "Yes : X⃗ = ", 0
                Stop
            else if φ = 1 and ∃j : A_{i,j} = 1 and row_i(A) ≠ 0 then
                output "Yes : X⃗ = ", e_j
                Stop
            end if
        end if
    end for
    output "No"
    Stop
end procedure
```

Figure 3.1. Algorithm ReflexiveChannel.

Input: A n-dimensional *LE* network $G = (B^0, B^1, A)$.

Output: "Yes" iff G has redundant channels, "No" otherwise.

Procedure RedundantChannel($G = (B^0, B^1, A)$)
    for each $B_{i_1}^{\phi_1}, B_{i_2}^{\phi_2} \in \{B^0, B^1\}$ do
    if $B_{i_1}^{\phi_1} = B_{i_2}^{\phi_2}$ then
        if $\phi_1 = 0$ and $\phi_2 = 0$ then
            Output "Yes" : $\vec{X}$ = ", $\mathbf{0}$
            Stop
        else if $\phi_1 = 0$ and $\phi_2 = 1$ then
            if $\exists j : A_{i_1,j} = 0 \wedge A_{i_2,j} = 1$ then
                Output "Yes" : $\vec{X}$ = ", e_j
                Stop
            end if
        else if $\phi_1 = 1$ and $\phi_2 = 0$ then
            if $\exists j : A_{i_1,j} = 1 \wedge A_{i_2,j} = 0$ then
                Output "Yes" : $\vec{X}$ = ", e_j
                Stop
            end if
        else if $\phi_1 = 1$ and $\phi_2 = 1$ then
            if $\exists j : A_{i_1,j} = A_{i_2,j} = 1$ then
                Output "Yes" : $\vec{X}$ = ", e_j
                Stop
            else if $\exists j_1, j_2 : A_{i_1,j_1} = A_{i_2,j_2} = 1 \wedge A_{i_1,j_2} = A_{i_2,j_1} = 0$ then
                Output "Yes" : $\vec{X}$ = ", $e_{j_1} + e_{j_2}$
                Stop
            end if
        end if
    end if
    end for
    Output "No"
    Stop
end procedure

Figure 3.2. Algorithm RedundantChannel.

Theorem 3.1.4 *Let $G = (B^0, B^1, A)$ be an n-dimensional* LE *network. Algorithm RedundantChannel tests to see any channels in G are redundant in $O(n^4)$ bit operations.*

Proof: The algorithm finds a pair of terms $B_{i_1}^{\phi_1} = B_{i_2}^{\phi_2}$, then based on the possible values of ϕ_1 and ϕ_2, it generates a node $\vec{X}$ that sets $(A\vec{X})_{i_1} = \phi_1$ and $(A\vec{X})_{i_2} = \phi_2$.

For $\phi_1 = \phi_2 = 0$, $\vec{X} = \mathbf{0}$ always suffices.

For $\phi_i \neq \phi_2$, if there is a j so that $A_{i_1,j} = \phi_1$ and $A_{i_2,j} = \phi_2$, then $\vec{X} = e_j$ suffices, otherwise $B_{i_1}^{\phi_1}$ and $B_{i_2}^{\phi_2}$ do not define redundant channels.

Finally, for $\phi_1 = \phi_2 = 1$, if there is a j so that $A_{i_1,j} = A_{i_2,j} = 1$, then $\vec{X} = e_j$ suffices. Alternately, if there are j_1 and j_2 so that $A_{i_1,j_1} = A_{i_2,j_2} = 1$ $A_{i_1,j_2} = A_{i_2,j_1} = 0$, then $\vec{X} = e_{j_1} + e_{j_2}$ suffices. Otherwise $B_{i_1}^{\phi_1}$ and $B_{i_2}^{\phi_2}$ do not define redundant channels.

This exhausts all possibilities for a redundant channel from any node $\vec{X}$, so the algorithm is correct. An example node for which a redundant channel exists is also output. Finding all pairs $B_{i_1}^{\phi_1} = B_{i_2}^{\phi_2}$ takes $O(n^3)$ bit operations, and then each check will take $O(n)$ bit operations, so at worst $O(n^4)$ bit operations are needed. ∎

In Figure 3.3, Algorithm ReciprocalChannel computes if a given network contains only channels that are reciprocal.

Theorem 3.1.5 *Let $G = (B^0, B^1, A)$ be an n-dimensional* LE *network. Algorithm ReciprocalChannel tests to see all channels in G are reciprocal in $O(n^4)$ bit operations.*

Proof: For each B_i^{ϕ}, the algorithm looks for other terms equal to B_i^{ϕ}, then does a series of tests to see if the other terms define channels reciprocal to B_i^{ϕ}. The first test determines if some B_j^{ψ} defines a reciprocal channel for all $\vec{X}$.

Input: A n-dimensional LE network $G = (B^0, B^1, A)$.

Output: "Yes" iff G has only reciprocal channels, "No" otherwise.

```
Procedure ReciprocalChannel( G = (B^0, B^1, A) )
    for each B_i^φ ∈ {B^0, B^1} : ∃X⃗ : (AX⃗)_i = φ do
        ReciprocalChannel ← false
        if ∃j : row_j(A) = 0 ∨ B_j^0 = B_i^φ then
            ReciprocalChannel ← true
        else if ∃j_1, j_2 : row_{j_1}(A) = row_{j_2}(A) ∨ B_{j_1}^0 = B_{j_2}^1 = B_i^φ then
            ReciprocalChannel ← true
        else if ∃j : row_j(A) = row_i(A) ∨ B_j^φ = B_i^φ ∨ (AB_i^φ)_j = 0 then
            ReciprocalChannel ← true
        end if
        if not ReciprocalChannel then
            Output "No : B_i^φ = ", B_i^φ
            Stop
        end if
    end for
    Output "Yes"
    Stop
end procedure
```

Figure 3.3. Algorithm ReciprocalChannel.

The second test determines if two different terms $B_{j_1}^{\psi_1}$ and $B_{j_2}^{\psi_2}$ might be used together to define reciprocal channels for all $\vec{X}$. The only way this can happen is if one term defines a reciprocal channel only when the other does not. Thus we must have $row_{j_1}(A) = row_{j_2}(A)$ and $\psi_1 \neq \psi_2$.

The final test determines if B_j^{ψ} defines a reciprocal channel only when B_i^{ϕ} defines a channel. This could only happen if $row_i(A) = row_j(A)$. Further, since the channel must be reciprocal, at node $\vec{X} + B_i^{\phi}$ we must have $A(\vec{X} + B_i^{\phi})_j = A(\vec{X})_j + (AB_i^{\phi})_j = (A\vec{X})_i$. Since $(A\vec{X})_j = (A\vec{X})_i$, we must have $(AB_i^{\phi})_j = 0$.

This exhausts all of the possible ways that a reciprocal channel can exist. All other conditions for an channel defined by B_i^{ϕ} from some node $\vec{X}$ allow a possible reciprocal channel to be undefined for at least one node $\vec{X}$.

It takes $O(n)$ operations to find out if a given B_i^1 defines a channel (that is, to find if $row_i(A) \neq \vec{0}$). Since B_i^0 is always used as an channel, we don't need to check if it does. Once we verify that B_i^{ϕ} is used as an channel, each of the tests can take up to $O(n^3)$ bit operations to verify (especially finding a possible j_1 and j_2 in the second test). Since there are $2n$ of the B_i^{ϕ}, we must use in the worst case $O(n^4)$ bit operations to compute this algorithm. ∎

The asymptotic run time complexity of Algorithm ReflexiveChannel is not prohibitively high. But if we wish to simplify the algorithm's complexity, we can restrict the definition of reciprocal channels to allow each term B_i^{ϕ} to define not only a channel, but also define to its won reciprocal channel:

Definition 3.1.2 *An n-dimensional* LE *network* $G = (B^0, B^1, A)$ *has* ***restricted reciprocal channels*** *iff* $\forall \vec{X} : \forall i \; B_i^{(A\vec{X})_i} = B_i^{(A\vec{X})_i + (AB_i^{(A\vec{X})_i})_i}$.

This restriction leads to a simpler test for computing whether only restricted reciprocal channels exist:

Theorem 3.1.6 *Let $G = (B^0, B^1, A)$ be an n-dimensional network. All channels in G are restricted reciprocal iff $\forall i \in \{1, \ldots, n\} : [AB^0]_{i,i} = [AB^1]_{i,i} = 0$, and this can be tested in $O(n^3)$ bit operations.*

Proof: The i-th neighbor of any node with address $\vec{X}$ can be computed by:

$$N_i(\vec{X}) = \vec{X} + B_i^{(A\vec{X})_i}$$

To guarantee that each channel of the network under our model is reciprocal, we need:

$$\begin{aligned}
\vec{X} &= N_i(N_i(\vec{X})) \\
&= N_i(\vec{X}) + B_i^{(AN_i(\vec{X}))_i} \\
&= \vec{X} + B_i^{(A\vec{X})_i} + B_i^{(A[\vec{X}+B_i^{(A\vec{X})_i}])_i} \\
&= \vec{X} + B_i^{(A\vec{X})_i} + B_i^{(A\vec{X}+AB_i^{(A\vec{X})_i})_i}
\end{aligned}$$

Assume that $[A\vec{X}]_i = 0$. Then:

$$\begin{aligned}
\vec{X} &= \vec{X} + B_i^0 + B_i^{0+(AB_i^0)_i} \\
B_i^0 &= B_i^{(0+AB_i^0)_i}
\end{aligned}$$

This forces $(AB_i^0)_i = (AB^0)_{i,i} = 0$. Assuming that $(A\vec{X})_i = 1$ also forces $(AB_i^1)_i = (AB^1)_{i,i} = 0$. Simply put, the network has only reciprocal channels iff the products AB^0 and AB^1 have zeroed diagonals.

We compute the products of AB^0 and AB^1 in $O(n^3)$ operations using the classical matrix multiply, then we check if the diagonals of AB^0 and AB^1 are zero in $O(n)$ bit operations. So this test requires $O(n^3)$ bit operations. ∎

This limited definition of reciprocal channels may not seem as robust as the previous definition, because it does not include some networks that have exclusively

reciprocal channels. However, all published *LE* networks have restricted reciprocal channels. The limited definition also has an advantage over the first in that it is conceptually simpler.

3.2. CONNECTEDNESS OF *LE* NETWORKS

Connectedness is (without doubt!) a major issue in interconnection networks. If we want to route a message through a network, then we must guarantee that the network is strongly connected.

The *DM* and *LE* networks are flexible and powerful, because they allow a large number of hypercube-variant networks to be described compactly. But in a practical sense, they are also too general because they can describe networks that are disconnected.

Consider the *LE* network described by the matrices:

$$B^0 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad B^1 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad A = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

This network is illustrated in Figure 3.4. Clearly it is disconnected.

It is also possible to describe networks that are only weakly connected. Consider the network described by the matrices:

$$B^0 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad B^1 = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \quad A = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

This network is illustrated in Figure 3.5. The nodes with addresses (000) and (111) each have in-degree 0 and cannot receive messages from the rest of the network.

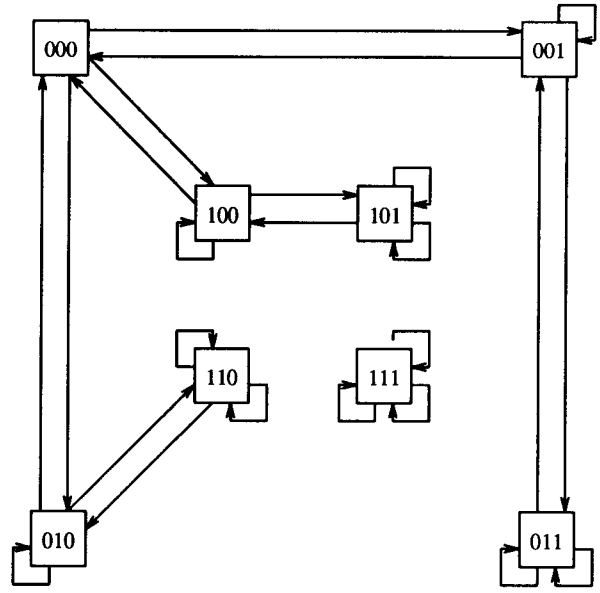

Figure 3.4. A disconnected 3-cube.

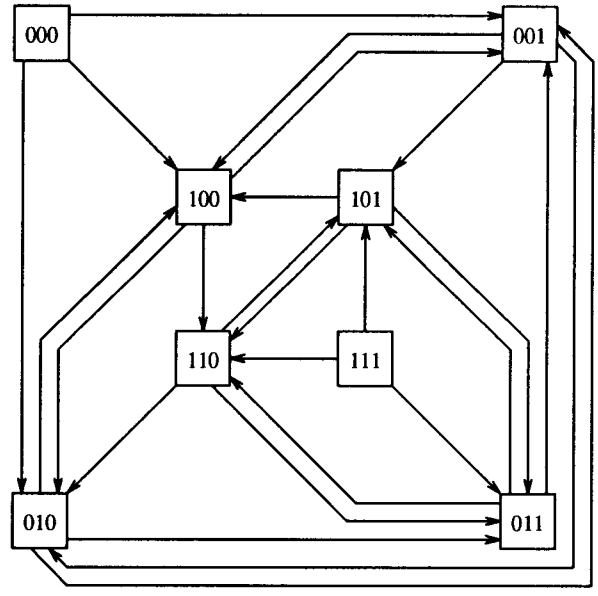

Figure 3.5. A weakly connected 3-cube.

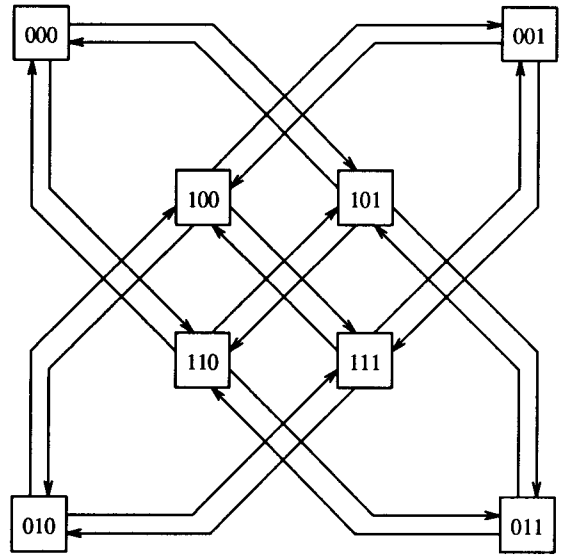

Figure 3.6. A disconnected 3-cube using restricted reciprocal channels. For instance there is no path from 000 to 111.

The first example allowed reflexive channels. The second example allowed non-reciprocal channels. It could be conjectured that using these types of channels permits cubes that are disconnected. However, if we disallow reflexive and non-reciprocal channels, the network can still be disconnected, as in the network described by the matrices:

$$B^0 = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \quad B^1 = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \quad A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

This network is illustrated in Figure 3.6. Though all the channels are reciprocal, the network still has two disconnected components.

For *DM* networks, it is simple to tell if a given matrix description defines a connected network. If there in a complete basis over $\mathcal{Z}_2^n$ in the columns of the B^0

and B^1 matrices, then the network is connected. This is because there is a linear combination of terms in B^0 and B^1 for every vector $\vec{X} \in \mathcal{Z}_2^n$, and so there is a path from $\vec{0}$ to $\vec{X}$.

Unfortunately, for *LE* networks it is usually not easy to tell from a given matrix description whether the network it describes is connected or not. One possible approach is to ignore the matrix description and represent the entire network as a graph. We can then use a standard graph connectedness algorithm which has a run time that is linear in the number of vertices. But because there are 2^n vertices, the run time will be exponential in n, the dimension of the network.

We can also ignore the graph description and try to compute connectedness from the matrix description alone. But the problem of network connectedness is more difficult than it might first appear. Even the apparently simpler problem of finding whether a single path exists in a given network is in general *NP*-hard, as we shall see.

3.2.1. Path Existence

We will examine the problem of whether a path exists between two given nodes in a network. This problem is defined as:

Definition 3.2.1 ***Network Path Existence (NPath):***

Instance: *An n-dimensional* LE *network* $G = (B^0, B^1, A)$, *and two nodes* $\vec{X}, \vec{Y} \in \mathcal{Z}_2^n$.

Question: *Is there a path from* $\vec{X}$ *to* $\vec{Y}$ *in* G*?*

If a path from $\vec{X}$ to $\vec{Y}$ exists, then each edge $(\vec{U}, \vec{V})$ in the routing path from $\vec{X}$ to $\vec{Y}$ can be represented by the term $B_i^{\phi} = \vec{U} + \vec{V}$. We can then represent the entire routing path as a sequence of T terms:

$$B_{i_1}^{\phi_1}, B_{i_1}^{\phi_1}, \ldots, B_{i_T}^{\phi_T} \tag{3.1}$$

This path must meet some requirements. The first is that these steps must lead from $\vec{X}$ to $\vec{Y}$:

$$\vec{X} + \sum_{U=1}^{T} B_{i_U}^{\phi_U} = \vec{Y} \tag{3.2}$$

The second requirement is that every term must correspond to an network channel when it is used. That is, for $1 \leq V \leq T$:

$$(A\vec{X})_{i_V} + \sum_{U=1}^{V} (AB_{i_U}^{\phi_U})_{i_V} = \phi_V$$

We can use these two requirements to verify that a path exists using at most $O(Tn^2)$ bit operations.

Unfortunately, there is a problem in showing a bound on the length T of a minimal path between $\vec{X}$ and $\vec{Y}$. The longest minimal path is the diameter of the network. Trivially, a loose upper bound of $2^n - 1$ channels can be placed on the diameter of any *LE* network, because that is the longest path that can be constructed using 2^n nodes. But we have not discovered a polynomial upper bound on the diameter of every *LE* network.

If we limit the steps of the path to be unique terms, then we can artificially put a polynomial bound on the length of the path:

Definition 3.2.2 ***Limited Network Path Existence (LNPath):***

Instance: *An n-dimensional* LE *network* $G = (B^0, B^1, A)$, *and two nodes* $\vec{X}, \vec{Y} \in \mathcal{Z}_2^n$.

Question: *Is there a path from* $\vec{X}$ *to* $\vec{Y}$ *in G, where each term in* B^0 *or* B^1 *defines at most one channel in the path?*

A limited path in a connected network can have a maximum $2n$ terms in its expansion, because there are at most $2n$ unique terms in B^0 and B^1.

We consider *LNPath* to be a valid limitation of *NPath*. First, we want paths in a network to be quite short – ideally, they should be linearly bounded. After all, we are searching for networks that are improvements upon the hypercube design. If the hypercube has a maximum path length of n steps, and a variant cube has a maximum path length of $O(n^2)$, then clearly we have failed in our search. Second, paths that include several channels defined by one term B_i^ϕ are very likely to cause network bottlenecks, because the channels defined by B_i^ϕ are being over-utilized. Using *LNPath* helps us avoid these two problems.

The polynomial bound on the maximum path length allows us to show that *LNPath* is *NP*-complete:

Theorem 3.2.1 LNPath *is* NP*-complete.*

Proof: The proof is given by reduction from the problem *3SAT*:

LNPath $\in$ NP: Guess a path from $\vec{X}$ to $\vec{Y}$. Using Equations 3.1 and 3.2, verify that the path uses no term B_i^ϕ more than once, that the path correctly leads from $\vec{X}$ to $\vec{Y}$ and that each channel on the path exists. This verification takes $O(n^3)$ steps.

3SAT $\leq$ LNPath: Start with an instance of *3SAT* over a set X of variables $x_1, x_2, \ldots, x_{|X|}$ and a set C of clauses, where the occurrence of each variable t in each clause j is identified by $c(t,j) \in C$, where $t \in \{x_1, \ldots, x_{|X|}, \overline{x_1}, \ldots, \overline{x_{|X|}}\}$ and $1 \leq j \leq |C|$.

We will construct a $(2|X| + 3|C|)$-dimensional network. We assign labels to each of the rows and columns. For the first $2|X|$ rows and columns, assign the labels of x_i and $\overline{x_i}$, $1 \leq i \leq |X|$. For the rest of the rows and columns of the 3 matrices

B^0, B^1 and A, assign the labels of $c(t,j)$, for each occurrence of variable t in clause j for $1 \leq j \leq |C|$.

Now, we assign the elements of B^0. First, for all $x_i \in X$, set $B^0_{x_i,x_i} = 1$ and $B^0_{\overline{x_i},x_i} = 1$. Second, for all $c(t,j) \in C$, set $B^0_{c(t,j),t} = 1$. All other elements of B^0 are zero.

Next, we assign the elements of B^1. For all $c(t,j) \in C$, set $B^1_{c(t,j),c(t,j)} = 1$. All other elements of B^1 are zero.

Finally, we assign values to A. For $t \in \{x_1, \ldots, x_{|X|}, \overline{x_1}, \ldots \overline{x_{|X|}}\}$, set $A_{t,t} = 1$. For each $c(t_1,j) \in C$, $c(t_2,j) \in C$, and $c(t_3,j) \in C$, set $A_{c(t_1,j),c(t_2,j)} = 1$, set $A_{c(t_2,j),c(t_3,j)} = 1$ and set $A_{c(t_3,j),c(t_1,j)} = 1$. All other elements of A are zero.

Finally, set $\vec{X} = (000\ldots0)$ and $\vec{Y} = (111\ldots1)$.

The problem is now transformed. The construction clearly takes $O(n^2)$ assignments and at most $O(n)$ time to scan through the clauses, so the transformation is polynomial time.

There are two key points to note. First, for every x_i:

$$(AB^0_{x_i})_{x_i} = (AB^0_{x_i})_{\overline{x_i}} = (AB^0_{\overline{x_i}})_{x_i} = (AB^0_{\overline{x_i}})_{\overline{x_i}} = 1$$

This means that once $B^0_{x_i}$ defines a routing step in the path, then $B^0_{\overline{x_i}}$ cannot define a routing step at all. This is also true for $B^0_{\overline{x_i}}$.

Also note that for all B^ϕ_j with $j \neq i$, $(AB^\phi_j)_{x_i} = (AB^\phi_j)_{\overline{x_i}} = 0$. This means that because we are starting from node $(000\ldots0)$, either $B^0_{x_i}$ or $B^0_{\overline{x_i}}$ can freely be used exactly once to define a routing step at any step in the path.

Second, for any j, the three columns that correspond to the terms $c(t_1,j)$, $c(t_2,j)$, $c(t_3,j)$ in the j-th clause have:

$$(AB^1_{c(t_1,j)})_{c(t_2,j)} = 1$$

$$(AB^1_{c(t_2,j)})_{c(t_3,j)} = 1$$

$$(AB^1_{c(t_3,j)})_{c(t_1,j)} = 1$$

Further, because of the way we set each $B^0_{t_i}$:

$$(AB^0_{t_1})_{c(t_2,j)} = 1$$

$$(AB^0_{t_2})_{c(t_3,j)} = 1$$

$$(AB^0_{t_3})_{c(t_1,j)} = 1$$

This means that none of $B^1_{c(t_1,j)}$, $B^1_{c(t_2,j)}$, $B^1_{c(t_3,j)}$ can define a routing step until at least one of $B^0_{t_1}$, $B^0_{t_2}$, or $B^0_{t_3}$ defines a routing step. If we allow $B^0_{t_i}$ to define a routing step, then $B^1_{c(t_{(i+1) \bmod 3},j)}$ can define routing step later in the routing path, and then $B^1_{c(t_{(i+2) \bmod 3},j)}$ can also define a routing step still later. Only in this way can all components $c(t_1, j)$, $c(t_2, j)$ and $c(t_3, j)$ can be set to 1 if starting from $(000\ldots0)$.

Assume that the *3SAT* expression has an assignment of values for x_1, x_2, ..., $x_{|X|}$ that sets the expression true. Then at least one value of every clause is true, satisfying every clause. Now for each variable assignment x_i with $1 \leq i \leq |X|$, include the term $B^0_{x_i}$ if x_i is set to true, and include the term $B^0_{\overline{x_i}}$ if x_i is set to false. Now for every j with $1 \leq j \leq |C|$, at least one of the indices $c(t_1, j)$, $c(t_2, j)$ and $c(t_3, j)$ is set to 1, and the rest can now be set to 1 by adding the routing steps $B^1_{c(t_1,j)}$, $B^1_{c(t_2,j)}$, and/or $B^1_{c(t_3,j)}$ to the end of the path as described in the preceding paragraph.

Note that every $B^0_{t_i}$ and $B^1_{c(t_i,j)}$ is used either once or not at all. The path meets the requirements of the limited network path.

Now assume that the *3SAT* expression has no assignment of values x_1, x_2, ..., $x_{|X|}$ that can set the expression true. Then for every assignment of x_1, x_2, ..., $x_{|X|}$, at least one clause j has no variables set to true. This means that the indices $c(t_1, j)$, $c(t_2, j)$ and $c(t_3, j)$ are all zero, and none of $B^1_{c(t_1,j)}$, $B^1_{c(t_2,j)}$, $B^1_{c(t_3,j)}$ can be

used in the routing path to set these indices. Thus $(111\ldots1)$ is unreachable from $(000\ldots0)$. ∎

Example: Let $\vec{X} = \{x_1, x_2, x_3\}$ and let $C = (x_1 \vee \overline{x_2} \vee \overline{x_3}), (\overline{x_1} \vee \overline{x_2} \vee x_3)$. Then the constructed matrices are (with labeled rows and columns) shown in Figure 3.7.

Because the expression is satisfiable, there is a limited minimal routing path from (000000000000) to (111111111111). That path is B_1^0, B_4^0, B_5^0, B_9^1, B_{12}^1, B_{10}^1. ∎

The proof that *3SAT* $\leq$ *LNPath* is also trivially a proof that *3SAT* $\leq$ *NPath*, because *LNPath* is a special case of *NPath*. If we can show a bound on the maximum path length in any network, we could show that *NPath* is also *NP*-complete. However, for now we have:

Corollary 3.2.1 NPath *is* NP*-hard.*

Proof: This follows directly from the reduction of *3SAT* to *NPath*. ∎

There are two possibilities for *NPath*. First, the maximum path length could be bounded by a polynomial in n, in which case *NPath* is *NP*-complete. Second, some networks could have diameters that are exponential to n, which implies that *NPath* has an exponential run time. As it currently stands, however, the upper bound on diameter is still an open question.

Our original plan was to use *NPath* or *LNPath* to construct an proof that would show that network connectedness was *NP*-complete. We thought to show network connectedness by examining only a representative subset of node pairs for path existence. However, assuming that all paths are polynomial in length, we would need an exponential number of such paths to make sure we cover every node, unless we could show that each path examined guaranteed an exponential number of nodes

$B^0 =$

	x_1	$\overline{x_1}$	x_2	$\overline{x_2}$	x_3	$\overline{x_3}$	$c(x_1,1)$	$c(\overline{x_2},1)$	$c(\overline{x_3},1)$	$c(\overline{x_1},2)$	$c(\overline{x_2},2)$	$c(x_3,2)$
x_1	1	0	0	0	0	0	0	0	0	0	0	0
$\overline{x_1}$	0	1	0	0	0	0	0	0	0	0	0	0
x_2	0	0	1	0	0	0	0	0	0	0	0	0
$\overline{x_2}$	0	0	0	1	0	0	0	0	0	0	0	0
x_3	0	0	0	0	1	0	0	0	0	0	0	0
$\overline{x_3}$	0	0	0	0	0	1	0	0	0	0	0	0
$c(x_1,1)$	1	0	0	0	0	0	0	0	0	0	0	0
$c(\overline{x_2},1)$	0	0	0	1	0	0	0	0	0	0	0	0
$c(\overline{x_3},1)$	0	0	0	0	0	1	0	0	0	0	0	0
$c(\overline{x_1},2)$	0	1	0	0	0	0	0	0	0	0	0	0
$c(\overline{x_2},2)$	0	0	0	1	0	0	0	0	0	0	0	0
$c(x_3,2)$	0	0	0	0	1	0	0	0	0	0	0	0

$B^1 =$

	x_1	$\overline{x_1}$	x_2	$\overline{x_2}$	x_3	$\overline{x_3}$	$c(x_1,1)$	$c(\overline{x_2},1)$	$c(\overline{x_3},1)$	$c(\overline{x_1},2)$	$c(\overline{x_2},2)$	$c(x_3,2)$
x_1	0	0	0	0	0	0	0	0	0	0	0	0
$\overline{x_1}$	0	0	0	0	0	0	0	0	0	0	0	0
x_2	0	0	0	0	0	0	0	0	0	0	0	0
$\overline{x_2}$	0	0	0	0	0	0	0	0	0	0	0	0
x_3	0	0	0	0	0	0	0	0	0	0	0	0
$\overline{x_3}$	0	0	0	0	0	0	0	0	0	0	0	0
$c(x_1,1)$	0	0	0	0	0	0	1	0	0	0	0	0
$c(\overline{x_2},1)$	0	0	0	0	0	0	0	1	0	0	0	0
$c(\overline{x_3},1)$	0	0	0	0	0	0	0	0	1	0	0	0
$c(\overline{x_1},2)$	0	0	0	0	0	0	0	0	0	1	0	0
$c(\overline{x_2},2)$	0	0	0	0	0	0	0	0	0	0	1	0
$c(x_3,2)$	0	0	0	0	0	0	0	0	0	0	0	1

$A =$

	x_1	$\overline{x_1}$	x_2	$\overline{x_2}$	x_3	$\overline{x_3}$	$c(x_1,1)$	$c(\overline{x_2},1)$	$c(\overline{x_3},1)$	$c(\overline{x_1},2)$	$c(\overline{x_2},2)$	$c(x_3,2)$
x_1	1	0	0	0	0	0	0	0	0	0	0	0
$\overline{x_1}$	0	1	0	0	0	0	0	0	0	0	0	0
x_2	0	0	1	0	0	0	0	0	0	0	0	0
$\overline{x_2}$	0	0	0	1	0	0	0	0	0	0	0	0
x_3	0	0	0	0	1	0	0	0	0	0	0	0
$\overline{x_3}$	0	0	0	0	0	1	0	0	0	0	0	0
$c(x_1,1)$	0	0	0	0	0	0	0	1	0	0	0	0
$c(\overline{x_2},1)$	0	0	0	0	0	0	0	0	1	0	0	0
$c(\overline{x_3},1)$	0	0	0	0	0	0	1	0	0	0	0	0
$c(\overline{x_1},2)$	0	0	0	0	0	0	0	0	0	0	1	0
$c(\overline{x_2},2)$	0	0	0	0	0	0	0	0	0	0	0	1
$c(x_3,2)$	0	0	0	0	0	0	0	0	0	1	0	0

Figure 3.7. The construction of a *LE* network from an instance of *3SAT*.

were connected together. This seems unlikely, because each path is polynomial in length.

3.2.2. Network Connectedness

We now examine the problem of network connectedness, or whether a path exists between *every* pair of nodes in the network.

Definition 3.2.3 ***Network Connectedness:***

Instance: *An n-dimensional* LE *network* $G = (B^0, B^1, A)$.

Question: *Is there a path from every pair of nodes* $\vec{X}$, $\vec{Y}$ *in* G*?*

Though we have been able to define some sufficient conditions and some necessary conditions for network connectedness, we have not been able to show both necessary and sufficient conditions simultaneously. At this time, it is not known if the problem of network connectedness is in P, or in NP, or is has a run time that is strictly exponential in the dimension n.

We examine some necessary conditions for network connectedness first.

Theorem 3.2.2 *Let* $G = (B^0, B^1)$ *be an n-dimensional* DM *network. If* G *is connected, then some combination of n vectors from* B^0 *and* B^1 *are linearly independent.*

Proof: G has $2n$ edges $\vec{X} + \vec{Y} = B_i^{\phi}$ defined for every $\vec{X}$. The addresses of the nodes clearly form a vector space over $\mathcal{Z}_2^n$ using mod 2 addition. To have every vector in $\mathcal{Z}_2^n$ reachable from $(000\ldots0)$, we need at least one set of k linearly independent vectors in B^0 and B^1, so that every vector in $\mathcal{Z}_2^n$ can be represented as a linear combination of those vectors. Then the path from $(000\ldots0)$ to $\vec{X}$ is composed of channels defined by terms in the linear combination of vectors.∎

This leads to the result:

Theorem 3.2.3 *Let* $G = (B^0, B^1, A)$ *be an n-dimensional* LE *network.* G *is connected only if the* DM *network* $\check{G} = (B^0, B^1)$ *is connected.*

Proof: Because G is a sub-network of $\check{G}$, its connectedness depends on whether $\check{G}$ is connected. Normally, it is possible for the subgraph of a disconnected graph to be connected, but only if the subgraph has fewer vertices, i.e., the subgraph is in one connected component of the disconnected graph. However, G has all the nodes of $\check{G}$, but not all of the channels, so if $\check{G}$ is disconnected, then so is G. ∎

A necessary condition for connectedness in an *LE* network is that at least one combination of the column vectors from B^0 and B^1 must be linearly independent. Though this is sufficient to guarantee that a *DM* network is connected, it is not sufficient to guarantee that a *LE* network is connected.

3.3. PROPERTIES OF *LTLE* NETWORKS

The *LTLE* networks are a special case of *LE* networks, and so they have a number of special properties, such as reciprocal channels, connectedness, and a recursive construction that allows the *LTLE* networks to be subdivided into smaller *LTLE* networks. In this section, we show and examine these properties.

3.3.1. Channel Properties

First, we examine the channel properties of *LTLE* networks. We note that from Theorem 3.1.6 we can show that all *LTLE* networks have common channel properties.

Theorem 3.3.1 *Let* $G = (B^0, B^1, A)$ *be an n-dimensional* LTLE *network. Then* G *has no reflexive or redundant channels, and only reciprocal channels.*

Proof: By Definition 2.1.4, B_i^0 and B_i^1, $1 \leq i \leq n$, cannot be zero, for all i. So G cannot have reflexive channels.

Now for $1 \leq i \leq n$, $1 \leq j \leq n$, we have $B_i^0 \neq B_j^0$, $i \neq j$, because if $i < j$, $B_{i,i}^0 = 1$ and $B_{i,j}^0 = 0$, and if $i > j$, $B_{j,j}^0 = 1$ and $B_{j,i}^0 = 0$. For the same reasons, $B_i^1 \neq B_j^1$ and $B_i^0 \neq B_j^1$ for $j \neq i$. Now, it is possible to have $B_i^0 = B_i^1$, but by definition of *LE* networks, only one can define a channel from a given node $\vec{X}$. So G cannot have redundant channels.

Finally, by Definition 2.1.4, $A_{i,i} = 0$ for all i, and B^0, B^1, and A are lower triangular matrices, so $(AB^0)_{i,i} = 0$ and $(AB^1)_{i,i} = 0$ for $1 \leq i \leq n$. By Theorem 3.1.6, all the channels of G are reciprocal. ∎

The theorem above shows that if we want a network with only reciprocal channels, we can simply use a *LTLE* network.

3.3.2. Connectedness

The *LTLE* networks share more than just channel properties. We can show that every *LTLE* network is constructed from two smaller *LTLE* networks and hence is connected.

In general, a *LE* network is not decomposable into smaller *LE* networks. If we partition the nodes of a connected n-dimensional *LE* network into two groups, one containing the addresses with $X_1 = 0$ and the other with $X_1 = 1$, we would be able to describe the channels between nodes in each group as two *LE* networks of dimension $n - 1$, but these two sub-networks may not themselves be connected.

This means that many *LE* networks are not expansible, in that they cannot be built up from smaller *LE* networks. Further, it means that parallel algorithms that use a "divide and conquer" strategy cannot be easily computed on a general

LE network, because a subproblem cannot be recursively handed to a connected sub-network to be solved.

However, there is a way to decompose a *LTLE* network into smaller *LTLE* networks. This method gives us a simple method to construct divide-and-conquer algorithms for *LTLE* networks.

Theorem 3.3.2 *Let* $G = (B^0, B^1, A)$ *be a* LTLE *network of dimension* n. *Then* G *is composed of two disjoint* LTLE *networks of dimension* $n-1$.

Proof: The address space can be broken into two subspaces, the first with vectors that have $\vec{X}_1 = 0$ and the second with vectors that have $\vec{X}_1 = 1$. The first column of B^0 and B^1 are the only terms that define channels joining the first subspace with the second.

Consider $G_0 = (\check{B}^0, \check{B}^2, \check{A})$, where:

$$\check{B}^0 = \begin{bmatrix} B^0_{2,2} & \dots & B^0_{2,n} \\ \vdots & \ddots & \vdots \\ B^0_{n,2} & \dots & B^0_{n,n} \end{bmatrix} \check{B}^1 = \begin{bmatrix} B^1_{2,2} & \dots & B^1_{2,n} \\ \vdots & \ddots & \vdots \\ B^1_{n,2} & \dots & B^1_{n,n} \end{bmatrix} \check{A} = \begin{bmatrix} A_{2,2} & \dots & A_{2,n} \\ \vdots & \ddots & \vdots \\ A_{n,2} & \dots & A_{n,n} \end{bmatrix}$$

The network G_0 generated by these matrices is clearly a *LTLE* network and has dimension $n-1$.

Now consider the network $G_1 = (\hat{B}^0, \hat{B}^1, \hat{A})$ where:

$$\hat{B}^0 = \begin{bmatrix} B^{A_{2,1}}_{2,2} & \dots & B^{A_{n,1}}_{2,n} \\ \vdots & \ddots & \vdots \\ B^{A_{2,1}}_{n,2} & \dots & B^{A_{n,1}}_{n,n} \end{bmatrix} \hat{B}^1 = \begin{bmatrix} B^{\overline{A_{2,1}}}_{2,2} & \dots & B^{\overline{A_{n,1}}}_{2,n} \\ \vdots & \ddots & \vdots \\ B^{\overline{A_{2,1}}}_{n,2} & \dots & B^{\overline{A_{n,1}}}_{n,n} \end{bmatrix} \hat{A} = \begin{bmatrix} A_{2,2} & \dots & A_{2,n} \\ \vdots & \ddots & \vdots \\ A_{n,2} & \dots & A_{n,n} \end{bmatrix}$$

The network G_1 generated by these matrices is a *LTLE* network and has dimension $n-1$. This network is the same as G_1 but with B^0_i and B^1_i exchanged if $A_{i,1} = 1$.

Now G_0 is a subgraph of G, because each address $\vec{X} \in \check{V}$ corresponds to an address $(0\vec{X}) \in V$, and because an edge $(\vec{X}, \vec{Y}) \in \check{E}$ corresponds to an edge $((0\vec{X}), (0\vec{Y})) \in E$:

$$\begin{aligned}(\vec{X}, \vec{Y}) \in \check{E} &\Rightarrow \vec{X} + \vec{Y} = \check{B}_i^{(\check{A}\vec{X})_i} \\ &\Rightarrow (0\vec{X}) + (0\vec{Y}) = B_{i+1}^{(A(0\vec{X}))_{i+1}} \\ &\Rightarrow ((0\vec{X}), (0\vec{Y}) \in E\end{aligned}$$

This is because $(A(0\vec{X}))_i = (\check{A}\vec{X})_i$.

Now G_1 is a subgraph of G, because each address $\vec{X} \in \check{V}$ corresponds to an address $(1\vec{X}) \in V$, and because an edge $(\vec{X}, \vec{Y}) \in \hat{E}$ corresponds to an edge $((1\vec{X}), (1\vec{Y})) \in E$:

$$\begin{aligned}(\vec{X}, \vec{Y}) \in \hat{E} &\Rightarrow \vec{X} + \vec{Y} = \hat{B}_i^{(\hat{A}\vec{X})_i} \\ &\Rightarrow (1\vec{X}) + (1\vec{Y}) = \begin{cases} B_{i+1}^{(A(1\vec{X}))_{i+1}} & \text{iff} A_{i+1,1} = 0 \\ \overline{B_{i+1}^{(A(1\vec{X}))_{i+1}}} & \text{iff} A_{i+1,1} = 1 \end{cases} \\ &\Rightarrow ((1\vec{X}), (1\vec{Y}) \in E\end{aligned}$$

This is because $(A(1\vec{X}))_{i+1} = A_{i+1,1} + (\hat{A}\vec{X})_i$.

Finally, the node addresses of G_0 are the sub-space of vectors with $\vec{X}_1 = 0$ and the node addresses of G_2 are the sub-space of vectors with $\vec{X}_1 = 1$. Thus G_1 and G_2 are disjoint and together use all vertices of G. ∎

Example: In [42], we demonstrated that a 0-Möbius Cube of dimension $n-1$ and a 1-Möbius Cube of dimension $n-1$ can be joined together to form a 0-Möbius or 1-Möbius Cube of dimension n. The definition of the 1-Möbius Cube differs only from the 0-Möbius Cube in that it has only the columns B_1^0 and B_1^1 exchanged. This is because $A_{2,1} = 1$ is the only nonzero element of the first column in A, for both network definitions. ∎

From this, we can conclude that *LTLE* networks are connected:

Corollary 3.3.1 *Let* $G = (B^0, B^1, A)$ *be an n-dimensional* LTLE *network. Then G is connected.*

Proof: The proof is inductive. For our base case, the 1-dimensional *LTLE* network $G = [B^0, B^1, A]$ is trivially connected. Now we assume that for $n' < n$, any n'-dimensional *LTLE* network is connected. An n-dimensional *LTLE* network is composed of two connected $(n-1)$-dimensional *LTLE* networks, which are joined at every node by the neighbor function N_1. ∎

We can also determine the maximum path length for *LTLE* networks.

Corollary 3.3.2 *Let* $G = (B^0, B^1, A)$ *be an n-dimensional* LTLE *network. Then the maximum path length in G is n.*

Proof: The proof is inductive. For our base case, the 1-dimensional *LTLE* network $G = [B^0, B^1, A]$ is trivially takes 1 maximum routing step to communicate a message between any two nodes. Now we assume that for $n' < n$, any n'-dimensional *LTLE* network requires at most n' routing steps. We can route to the $(n-1)$-dimensional sub-network that contains the destination by using using a channels defined by N_1, then recursively routing on the $(n-1)$-dimensional *LTLE* network.

∎

3.4. SUMMARY

The *LE* model is overly general for describing interconnection networks. It allows descriptions of networks that are disconnected, weakly connected or have non-reciprocal, redundant or reflexive channels. There is strong evidence that there

is no efficient algorithm for determining if a network is connected. Even the upper bound on the maximum path length for *LE* networks remains an open question.

However, one subclass of the *LE* model has a number of attractive features and properties. and resolves several of the problems listed above. *LTLE* networks are always strongly connected with only reciprocal, non-redundant, non-reflexive channels. The n-dimensional *LTLE* networks has a maximum path length of n channels, and a simple decomposition into 2 disjoint $(n-1)$-dimensional *LTLE* networks. These properties make *LTLE* networks preferable to a general *LE* network.

All of these features make the *LTLE* networks very attractive as potential interconnection network topologies. For the rest of this dissertation, many of the results we show will be specifically for *LTLE* networks.

4. NETWORK ISOMORPHISM

In this chapter, we examine some basic isomorphism properties of general *LE* networks. We show that a number of these properties are *NP*-complete or *NP*-hard. We then show that it is possible to determine if a *LE* network is isomorphic to a *LTLE* network. Lastly, we examine minimum-weight isomorphisms of a *LE* network.

4.1. ISOMORPHISM OF *LE* NETWORKS

In this section, we consider when two networks described by the formal model are *isomorphic* to each other as graphs – that is, whether one network can be transformed to a second with a renumbering of its nodes.

The *LE* network model is too general for uniquely describing Twisted Cube networks. For a given Twisted Cube network, there can be more than one *LE* network that describes it. For instance, the Twisted 3-Cube can be described by :

$$B^0 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} B^1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} A = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

It can also be described by:

$$B^0 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} B^1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} A = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

And even described by:

$$B^0 = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} B^1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} A = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix}$$

While the addressing of the nodes may differ between the networks, the underlying networks are isomorphic as graphs to one another. Unfortunately, the matrix descriptions of two networks give little clue as to whether they are isomorphic.

Network isomorphism is an important problem for at least two reasons. First, we already know a number of results for published networks. If we can show that one network is isomorphic to a second network, then the results for the first network will hold for the second network. Second, *LE* networks are divided into several sub-classes (for instance, *LTLE* networks) which have properties that not all *LE* networks share. We can use network isomorphisms to show that a network has membership in a sub-class, and so shares the sub-class's properties.

The problem of network isomorphism is closely related to the problem of graph isomorphism:

Definition 4.1.1 ***Graph Isomorphism Problem:***

Instance: *$G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ are two graphs of $|V| = n$ nodes.*

Question: *Is there a permutation $f : V \rightarrow V$ so that:*

$$\forall v_1, v_2 \in V : (v_1, v_2) \in E_1 \iff (f(v_1), f(v_2)) \in E_2$$

Though the original problem statement of graph isomorphism considered only connected undirected graphs, it is clear that the variant problems of unconnected graphs and directed graphs are computationally equivalent. It is important to know that these two variants are equivalent, because our networks are defined with directed channels and can potentially be unconnected. The polynomial-time transformations are outlined below:

A pair of unconnected graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ can be transformed to connected graphs by adding one vertex v_{n+1} to V_1 and V_2 and for every

i, $1 \leq i \leq n$, add edges (v_i, v_{n+1}) to E_1 and E_2 to make them connected. Since v_{n+1} is the only vertex in each graph with out-degree n, then $v_{n+1} \in E_1$ must map to $v_{n+1} \in E_2$. The constructed graphs are then clearly isomorphic iff G_1 is isomorphic to G_2.

Connected graphs are a trivial instances of potentially unconnected graphs, so no transformation in the other direction is needed.

A pair of undirected graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ can be transformed to directed graphs by replacing each undirected edge (v_i, v_j) with two directed edges (v_i, v_j) and (v_j, v_i). The constructed graphs are clearly isomorphic iff G_1 is isomorphic to G_2.

A pair of directed graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ can be transformed to undirected graphs, by encoding direction information into each undirected edge. One construction is to hang a chain of $n + 1$ new vertices from each original vertex, and replace each original directed edge (u, v) with 3 new vertices t_1, t_2, t_3, and 4 new undirected edges: (u, t_1), (t_1, t_2), (t_2, v), and (t_2, t_3). The construction take $O(n^2)$ steps. The constructed graphs are then isomorphic iff G_1 is isomorphic to G_2, because the chains can only map to each other, and the edge constructions will map to each other iff the original directed edges in G_1 map to directed edges in G_2.

The problem of network isomorphism is similar to the problem of graph isomorphism. However, rather than using the graph representation of our network, we will use the matrix representation of our networks.

Definition 4.1.2 ***Network Isomorphism Problem:***

Instance: *Two n-dimensional* LE *networks* $G_1 = (B^0, B^1, A)$ *and* $G_2 = (\check{B}^0, \check{B}^1, \check{A})$.

Question: *Is there is a 1-1 mapping* $f : \mathcal{Z}_2^n \rightarrow \mathcal{Z}_2^n$ *so that:*

$$\forall \vec{X}, \vec{Y} \in \mathcal{Z}_2^n : \vec{X} + \vec{Y} = B_i^{(A\vec{X})_i} \iff f(\vec{X}) + f(\vec{Y}) = \check{B}_j^{(\check{A}f(\vec{X}))_j}$$

The problems of network isomorphism and graph isomorphism differ in the size of the instance. An algorithm for graph isomorphism can be used to solve an instance of network isomorphism by expanding the network definition into an entire graph, but since an n-dimensional network has n nodes of degree n, the problem size for graph isomorphism problem would be an exponential $O(n2^n)$. We would like to find an algorithm that can find network isomorphisms in time polynomial in n. We will see strong evidence that such an algorithm does not exist.

4.1.1. Basic Network Isomorphisms

There are a number of "basic" or" standard" network isomorphisms that we can use to transform one network to another. These isomorphisms operate on the matrix descriptions, so they can be used to show that one network is isomorphic to another without expanding the network descriptions into entire graphs.

First, certain substitutions in the matrix description can produce networks that are identical address-for-address to the original, though the matrix descriptions are different.

Theorem 4.1.1 *Let* $G = (B^0, B^1, A)$ *be an* n*-dimensional* LE *network. If:*

$$\forall \vec{U}, \vec{V}, \vec{W} \in \mathcal{Z}_2^n : B_i^0 = B_i^1 = \vec{U}, \mathrm{row}_i(A) = \vec{V}$$

then substituting:

$$B_i^0 = \vec{U}, B_i^1 = \vec{W}, \mathrm{row}_i(A) = 0$$

creates a new network $\check{G} = (\check{B}^0, \check{B}^1, \check{A})$*, which is identical to the original network.*

Proof: This is because for any $\vec{X} \in \mathcal{Z}_2^n$, $B_i^{(\vec{V}^T\vec{X})} = B_i^0 = \vec{U}$. This substitution can also be reversed. ∎

One simple isomorphism allows the translation of all addresses of a cube by a constant amount. This isomorphism is a "reflection" of the network across a vector value $\vec{W}$.

Theorem 4.1.2 *Let $G = (B^0, B^1, A)$ be an n-dimensional* LE *network. G is isomorphic to $\check{G} = (\check{B}^0 = \{1 \leq i \leq n : B_i^{(A\vec{W})_i}\}, \check{B}^1 = \{1 \leq i \leq n : B_i^{\overline{(A\vec{W})_i}}\}, A)$, under the function $f : \mathcal{Z}_2^n \to \mathcal{Z}_2^n$ defined by $f(\vec{X}) = \vec{X} + \vec{W}$.*

Proof: Let $\vec{X}, \vec{Y} \in E$. Then:

$$
\begin{aligned}
(\vec{X}, \vec{Y}) \in E &\Leftrightarrow \vec{X} + \vec{Y} = B_i^{(A\vec{X})_i} \\
&\Leftrightarrow \vec{X} + \vec{Y} + \vec{W} + \vec{W} = B_i^{[A(\vec{X}+\vec{W}+\vec{W})]_i} \\
&\Leftrightarrow (\vec{X} + \vec{W}) + (\vec{Y} + \vec{W}) = B_i^{[A(\vec{X}+\vec{W})]_i + (A\vec{W})_i)} \\
&\Leftrightarrow f(\vec{X}) + f(\vec{Y}) = B_i^{[Af(\vec{X})]_i + (A\vec{W})_i} \\
&\Leftrightarrow f(\vec{X}) + f(\vec{Y}) = \check{B}_i^{(Af(\vec{X}))_i} \\
&\Leftrightarrow (f(\vec{X}), f(\vec{Y})) \in \check{E}
\end{aligned}
$$

∎

There are a number of other isomorphisms that exist for any *LE* network. These transformations depend on the fact that vector addition and permutation form groups over $\mathcal{Z}_2^n$.

Theorem 4.1.3 *Let $G = (B^0, B^1, A)$ be an n-dimensional* LE *network. If f is an automorphism over the group $\langle \mathcal{Z}_2^n, + \rangle$ and $\exists \check{A} : \forall \vec{X} : \forall i, 1 \leq i \leq n : (A\vec{X})_i = (\check{A}f(\vec{X}))_i$, then G is isomorphic to $\check{G} = (f(B^0), f(B^1), \check{A})$.*

Proof: Let $\vec{X}, \vec{Y} \in E$. Then:

$$\begin{aligned}
(\vec{X}, \vec{Y}) \in E &\Leftrightarrow \vec{X} + \vec{Y} = B_i^{(A\vec{X})_i} \\
&\Leftrightarrow f(\vec{X} + \vec{Y}) = f(B_i^{(A\vec{X})_i}) \\
&\Leftrightarrow f(\vec{X}) + f(\vec{Y}) = f(B_i^{(A\vec{X})_i}) \\
&\Leftrightarrow f(\vec{X}) + f(\vec{Y}) = f(B)_i^{(A\vec{X})_i} \\
&\Leftrightarrow f(\vec{X}) + f(\vec{Y}) = f(B)_i^{(\check{A}f(\vec{X}))_i} \\
&\Leftrightarrow (f(\vec{X}), f(\vec{Y})) \in \check{E}
\end{aligned}$$

∎

Using this theorem, we can show that row permutation, column permutation, and row addition on the matrix descriptions are network isomorphisms:

Theorem 4.1.4 *Networks are isomorphic under row permutation.*

Let $G = (B^0, B^1, A)$ *be an* n*-dimensional* LE *network.* G *is isomorphic to* $\check{G} = (\check{B}^0, \check{B}^1, \check{A})$*, where:*

$$\begin{aligned}
\check{B}^0_{i,j} &= B^0_{\pi(i),j} \\
\check{B}^1_{i,j} &= B^1_{\pi(i),j} \\
\check{A}_{j,i} &= A_{j,\pi(i)}
\end{aligned}$$

Proof: First, $f(X) = (X_{\pi(1)} X_{\pi(2)} \ldots X_{\pi(n)})$ is an automorphism, because it is a bijection from $\mathcal{Z}_2^n$ to itself (by definition of permutation), and $f(X + Y) = f(X) + f(Y)$, because $\forall i : \pi((X + Y)_i) = \pi(X_i) + \pi(Y_i)$.

Second, the selector functions are equivalent:

$$(\check{A}f(X))_i = \sum_{k=1}^{n} \check{A}_{i,k} f(X)_k$$

$$= \sum_{k=1}^{n} A_{i,\pi(k)} X_{\pi(k)}$$
$$= \sum_{k=1}^{n} A_{i,k} X_k$$
$$= (AX)_i$$

∎

Theorem 4.1.5 *Networks are isomorphic under column permutation.*

Let $G = (B^0, B^1, A)$ *be an* n*-dimensional* LE *network.* G *is isomorphic to* $\check{G} = (\check{B}^0, \check{B}^1, \check{A})$*, where:*

$$\check{B}^0_{i,j} = B^0_{i,\pi(j)}$$
$$\check{B}^1_{i,j} = B^1_{i,\pi(j)}$$
$$\check{A}_{j,i} = A_{\pi(j),i}$$

Proof: First, the permutation of columns of B^0 and B^1 is simply the reordering the edges out of a vertex $\vec{X}$, so the automorphism is the identity function $f(\vec{X}) = \vec{X}$. Second, the selector functions are equivalent under permutation:

$$(\check{A}f(X))_i = \sum_{k=1}^{n} \check{A}_{i,k} f(\vec{X})_k$$
$$= \sum_{k=1}^{n} A_{\pi(i),k} X_k$$
$$= (A\vec{X})_{\pi(i)}$$

So we have:

$$(\vec{X}, \vec{Y}) \in E \Leftrightarrow (\vec{X} + \vec{Y}) = B_i^{(A\vec{X})_i}$$
$$\Leftrightarrow (\vec{X} + \vec{Y}) = B_{\pi(i)}^{(\check{A}\vec{X})_{\pi(i)}}$$
$$\Leftrightarrow (\vec{X}, \vec{Y}) \in \check{E}$$

∎

Theorem 4.1.6 *Networks are isomorphic under row addition.*

Let $G = (B^0, B^1, A)$ *be an* n*-dimensional* LE *network.* G *is isomorphic to* $\check{G} = (\check{B}^0, \check{B}^1, \check{A})$*, where:*

$$\check{B}^0_{i,j} = \begin{cases} B^0_{i,j} & i \neq r \\ B^0_{r,j} + B^0_{s,j} & i = r \end{cases}$$

$$\check{B}^1_{i,j} = \begin{cases} B^1_{i,j} & i \neq r \\ B^1_{r,j} + B^1_{s,j} & i = r \end{cases}$$

$$\check{A}_{j,i} = \begin{cases} A_{j,i} & i \neq r \\ A_{j,r} + A_{j,s} & i = s \end{cases}$$

Proof: First, the mapping $f(\vec{X}) = (X_1, \ldots, X_r + X_s, \ldots, X_n)$ is an automorphism. The function f is 1-1, because if $X_s = 0$, then $f(\vec{X}) = \vec{X}$, and if $X_s = 1$, then $f(\vec{X}) = \vec{X} + e_s$. Also, $f(\vec{X} + \vec{Y}) = f(\vec{X}) + f(\vec{Y})$, because if $i \neq r$, then $f(\vec{X}+\vec{Y})_i = f(\vec{X})_i + f(\vec{Y})_i$, and if $i = r$, then $f(\vec{X}+\vec{Y})_i = (\vec{X}+\vec{Y})_r + (\vec{X}+\vec{Y})_s = \vec{X}_r + \vec{Y}_r + \vec{X}_s + \vec{Y}_s = f(\vec{X})_i + f(\vec{Y})_i$.

Second, the selector functions of both networks are equivalent. For $i \neq r$, $(\check{A}f(\vec{X}))_i = (A\vec{X})_i$. For $i = r$:

$$\begin{aligned}
(\check{A}f(\vec{X}))_i &= \sum_{k=1}^{n} \check{A}_{i,k} f(\vec{X})_k \\
&= \check{A}_{i,1}\vec{X}_1 + \ldots + (\check{A}_{i,r} + \check{A}_{i,s})\vec{X}_s + \ldots + \check{A}_{i,r}(\vec{X}_r + \vec{X}_s) + \ldots + \check{A}_{i,n}\vec{X}_n \\
&= \sum_{k=1}^{n} A_{i,k}\vec{X}_k + A_{i,r}\vec{X}_s + A_{i,r}\vec{X}_s \\
&= \sum_{k=1}^{n} A_{i,k}\vec{X}_k \\
&= (A\vec{X})_i
\end{aligned}$$

∎

Input: An n-dimensional LE network $G = (B^0, B^1, A)$, where B^0 contains n linearly independent column vectors.

Output: An n-dimensional LE network $\check{G} = (\mathbf{I}, \check{B}^1, \check{A})$ that is isomorphic to G.

```
GaussianReduceNetwork( G = (B^0, B^1, A) )
    for i = n down to 2 do { elimination
        j = i
        while j ≥ 1 and B^0_{i,j} ≠ 1 do
            j = j − 1
        end while
        if j ≥ 1 then
            swap( B^0_i, B^0_j )
            swap( B^1_i, B^1_j )
            swap( row_i(A), row_i(A) )
        end if
        for j = i − 1 to 1 do
            if B^0_{j,i} = 1 then
                row_j(B^0) ← row_j(B^0) + row_i(B^0)
                row_j(B^1) ← row_j(B^1) + row_i(B^1)
                A_i ← A_i + A_j
            end if
        end for
    end for
    for i = 1 to n do { back-substitution }
        for j = i + 1 to n do
            if B^0_{j,i} = 1 then
                row_j(B^0) ← row_j(B^0) + row_i(B^0)
                row_j(B^1) ← row_j(B^1) + row_i(B^1)
                A_i ← A_i + A_j
            end if
        end for
    end for
end procedure
```

Figure 4.1. Algorithm GaussianReduceNetwork.

The three isomorphisms above are useful by giving us a standard matrix description for a network.

Theorem 4.1.7 *Let $G = (B^0, B^1, A)$ be an n-dimensional* LE *network. Let B^0 have n linearly independent columns. Then G is isomorphic to a network $\check{G} = (\boldsymbol{I}, \check{B}^1, \check{A})$.*

Proof: We can apply a method similar to a standard Gaussian reduction to convert B^0 to I, using the three isomorphisms listed above. Algorithm GaussianReduceNetwork is listed in Figure 4.1.

B^0 must have n linearly independent columns, or the algorithm will produce a network with the first matrix not equal to **I**, but with ones and zeros on the diagonal and zeros elsewhere. ∎

We can use the transformations above to show that some of the published networks isomorphic to others.

Theorem 4.1.8 *The Flip MCube is isomorphic to the 1-Möbius cube.*

Proof: Begin with an n-dimensional MCube. Transform it to the isomorphic network with $B^0 = \mathbf{I}$ using Algorithm GaussianReduceNetwork. Use the row addition isomorphism to add row 1 to row 2, add row 2 to row 3, ..., and add row $n-1$ to row n, in that order. Then, because row 1 of A is zero, substitute column B^1_1 with $100\ldots0$.

The resulting network is identical to the n-dimensional 1-Möbius cube, except that B^0_i and B^1_i, $2 \leq i \leq n$, are exchanged. Use Theorem 4.1.2 to map each node $\vec{X}$ to $\vec{X} + 100\ldots0$. The matrix description of the resulting network is the 1-Möbius cube's description. ∎

Theorem 4.1.9 *The YAT cube is isomorphic to the 0-Möbius cube.*

Proof: The proof is the same as for the Flip MCube to 1-Möbius cube. ∎

In fact, after we discovered that the Flip MCube is isomorphic to the 1-Möbius cube, we constructed the YAT cube as the Flip MCube's analog of the 0-Möbius cube.

4.1.2. The Complexity of Network Isomorphism

The general problem of Graph Isomorphism has not been shown to be *NP*-complete. In fact, Graph Isomorphism is usually cited as a problem in *NP-P* which is not *NP*-complete [11]. There is no known efficient polynomial algorithm for determining if two graphs are isomorphic, nor is there a transformation from an instance of an *NP*-complete problem to an instance of graph isomorphism. As we will see, the problem of network isomorphism is at least as difficult as graph isomorphism.

Theorem 4.1.10 ***Graph Isomorphism*** $\leq$ ***Network Isomorphism***

Proof: To show this, we transform an instance of Graph Isomorphism to an instance of Network Isomorphism using a polynomial number of operations. Assume that we have a pair of connected digraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ with no self-looping edges (v_i, v_i) and assume that $|V_1| = |V_2| = n$.

(The restriction of no self-looping edges in an instance of the Graph Isomorphism problem will not affect the complexity of the problem. We can transform an instance of Graph Isomorphism to Graph Isomorphism without self-loops in polynomial time. We replace each self-loop in the two graphs with a directed chain of $n+1$ nodes. This guarantees that each chain in one transformed graph will be mapped to a chain in the other transformed graph. Hence the two transformed graphs are isomorphic iff the original graphs are isomorphic.)

Notice that G_1 and G_2 can be represented by $n \times n$ adjacency matrices M_1 and M_2, where each $a_{v_i,v_j} \in M_1$ equals 1 iff $(v_i, v_j) \in E_1$ and 0 otherwise. This construction takes polynomial time, because M_1 and M_2 are $n \times n$ matrices and can be constructed in $O(|V|^2)$ operations. Clearly if G_1 is isomorphic to G_2, then some permutation of vertex numbering can be done on G_1 to transform it to G_2. This permutation can be done by a series of exchanges of vertex numbers, which corresponds to a series of simultaneous row/column exchanges in M_1.

Construct two networks $\check{G}_1 = (\mathbf{I}, M_1, \mathbf{I})$ and $\check{G}_2 = (\mathbf{I}, M_2, \mathbf{I})$. This construction takes only $O(n^2)$ steps, the size of the matrix descriptions for G_1 and G_2. If G_1 is isomorphic to G_2, then clearly $\check{G}_1$ is isomorphic to $\check{G}_2$, because simultaneous row/column exchanges on the matrices will not affect the identity matrix $\mathbf{I}$.

If $\check{G}_1$ is isomorphic to $\check{G}_2$, then there is some mapping $f : \mathcal{Z}_2^n \rightarrow \mathcal{Z}_2^n$ from the nodes of $\check{G}_1$ to the nodes of $\check{G}_2$.

We note several properties of $\check{G}_1$ and $\check{G}_2$. The linear selector function for $\check{G}_1$ is simple: $(\mathbf{I}\vec{X})_i = X_i$. Further, the i-th channel $(\vec{X}, \vec{Y})$ with $X_i = 0$ has no restricted reciprocal channel $(\vec{Y}, \vec{X})$, because $(AB_i^0)_i = (\mathbf{I}e_i)_i = 1$. Also,the i-th channel $(\vec{X}, \vec{Y})$ with $X_i = 1$ has a restricted reciprocal channel because G_1 has no edge loops, M_1 has $a_{i,i} = 0$ and so $(AB_i^0)_i = (\mathbf{I}(M_1)_i)_i = 0$. These same properties hold true for $\check{G}_2$.

Consider only the set of edges generated by columns of B^0. Such edges only come from a node $\vec{X}$ when $X_i = 0$, and together form the edges of a directed n-dimensional hypercube, as in Figure 4.2. If $\check{G}_1$ is isomorphic to $\check{G}_2$, then this subgraph in $\check{G}_1$ will map to the same subgraph in $\check{G}_2$. The mapping f must be a homomorphism which corresponds to a rotation of the n-dimensional directed hypercube about the node with address $\mathbf{0}$. This limits f to a simultaneous permutation of the rows/columns in $\check{G}_1$ and hence a simultaneous permutation of rows/columns

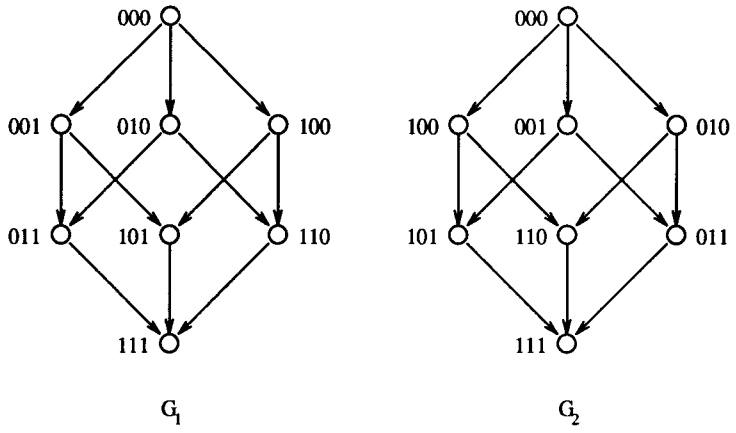

Figure 4.2. Two directed hypercubes.

in M_1. So $\check{G}_1$ isomorphic to $\check{G}_2$ implies that G_1 is isomorphic to G_2, and vice versa. ∎

While Graph Isomorphism is in *NP-P*, we have been unable to show that the general problem of Network Isomorphism is in either *NP-P* or *NP*. The difficulty lies in computing the mapping $f : \mathcal{Z}_2 \rightarrow \mathcal{Z}_2^n$ from vertices in a network G_1 to vertices in another network G_2. This mapping may require up to $O(n2^n)$ bits of space to describe completely and hence could take exponential time.

If we limit the applicable isomorphisms to the ones given above, we can show that these restricted versions of the Network Isomorphism are in *NP-P*. These two restrictions can be expressed as:

Definition 4.1.3 ***Network Permutation Isomorphism Problem:***

Instance: *Two n-dimensional* LE *networks* G_1 *and* G_2.

Question: *Is* G_1 *isomorphic to* G_2 *through a series of simultaneous row/column permutations?*

Definition 4.1.4 ***Network Automorphic Isomorphism Problem:***

Instance: *Two n-dimensional* LE *networks G_1 and G_2, where the columns of B^0 in G_1 and the columns of B^0 in G_2 are each n-linearly independent.*

Question: *Is G_1 isomorphic to G_2 through a series of row exchanges, column exchanges and row additions?*

We can now show that these restricted forms of Network isomorphism are equivalent to Graph Isomorphism, and hence in *NP-P*:

Theorem 4.1.11 ***Network Permutation Isomorphism = Graph Isomorphism***

Proof: The proof is in two parts:

Graph Isomorphism $\leq$ Network Permutation Isomorphism: Theorem 4.1.10 trivially applies to this restricted version of Network Isomorphism, because it uses only simultaneous row/column permutations.

Network Permutation Isomorphism $\leq$ Graph Isomorphism: Let G_1 and G_2 be an instance of Network Permutation Isomorphism. We will transform them to an instance of Weighted Graph Isomorphism, a generalization of the Graph Isomorphism problem.

(Weighted Graph Isomorphism is equivalent to Graph Isomorphism, because there are polynomial-time transformations from instances of one problem to instances of the other. We can transform an instance of Weighted Graph Isomorphism, where the edge weights are integers $0 \leq P[n]$, to an instance of Graph Isomorphism. We can replace each weighted edge with a chain of two edges. The middle node of the chain has hanging from it a chain of length $(n+1)+w$ edges that correspond to the weight of the original edge. This transformation takes $O(n^2P[n])$ steps. An

instance of Graph Isomorphism is trivially transformed to an instance of Weighted Graph Isomorphism by setting the weights of all the edges to 1.)

Use G_1 to construct an $n \times n$ integer array M_1, where each element $a_{i,j} = B^0_{i,j} + 2B^1_{i,j} + 4A_{i,j}$. Use G_2 to construct M_2 in the same manner. Then M_1 and M_2 are the adjacency arrays of two weighted graphs $\check{G}_1$ and $\check{G}_2$ respectively. These graphs are clearly isomorphic iff networks G_1 and G_2 are isomorphic. ∎

Theorem 4.1.12 ***Network Automorphic Isomorphism = Graph Isomorphism***

Proof: Because we can use Algorithm GaussianReduceNetwork to simplify the B^0 matrix for the two networks, we only have to consider simultaneous row/column permutations. Thus the results of Theorem 4.1.11 hold for Network Automorphic Isomorphism too.

(This problem also includes the use of the reflection isomorphism, because that isomorphism can be emulated with a series of row additions.) ∎

This shows that at least restricted versions of the the problem of Network Isomorphism are computationally equivalent to the problem of Graph Isomorphism.

4.2. ISOMORPHISM OF *LTLE* NETWORKS

We will consider two isomorphism problems of *LTLE* networks. First is the complexity of computing whether a general *LE* network is isomorphic to a *LTLE* network. Second is the complexity of computing whether two *LTLE* networks are isomorphic.

First, we consider the current published *LE* networks. By applying the isomorphisms given above to the published *LE* networks, we can show that all the published *LE* networks are isomorphic to *LTLE* networks.

Theorem 4.2.1 *The n-dimensional hypercube, the Möbius Cubes, the Bent Cube and the YAT Cube are all* LTLE *networks.*

Proof: This is trivially true, because they already meet the definition of *LTLE* networks. ∎

Theorem 4.2.2 *The Generalized Twisted Cube is isomorphic to a lower triangular network.*

Proof: The Generalized Twisted Cube is built by graph composition of Twisted 3-Cubes and a 1 or 2 dimensional hypercube. Since each of these networks are isomorphic to *LTLE* networks, their network composition will be a *LTLE* network, too. ∎

Theorem 4.2.3 *The Twisted Cube is isomorphic to a* LTLE *network.*

Proof: Assume that we have a n-dimensional Twisted Cube as defined in 2.2.5. Use the row exchange isomorphism to exchange each row $2j$ and $2j+1$. Then use column exchange isomorphism to exchange each column $2j$ and $2j+1$. The Twisted Cube is now in a *LTLE* configuration. ∎

Theorem 4.2.4 *The Flip MCube is isomorphic to a* LTLE *network.*

Proof: Begin with the n-dimensional Flip MCube as defined in 2.2.6. Use the row permutation isomorphism to move each row i, $1 \leq i \leq n-1$, to row $i+1$ and row n to row 1, then use the column exchange isomorphism to move each column i, $1 \leq i \leq n-1$, to column $i+1$ and column n to column 1. The network is now in a *LTLE* configuration. ∎

All of the above examples are instances of a more general problem – whether a network can be represented as a *LTLE* network.

Definition 4.2.1 LTLE ***Network Inclusion Problem (LTNI):***

Instance: *An n-dimensional* LE *network* $G = (B^0, B^1, A)$.

Question: *Is G isomorphic to a* LTLE *network?*

We show that *LTNI* has an efficient algorithm:

Theorem 4.2.5 $\boldsymbol{LTNI \in P}$.

Proof: To show that this is possible, we need to specify some sequence of polynomial-time isomorphic transformations that can simultaneously rearrange each of the matrices B^0, B^1, and A to a *LTLE* configuration. Note that we don't have to show that G is isomorphic to a specific *LTLE* network $\check{G}$. Instead, we only have to show that G is isomorphic to any *LTLE* network.

One method to show that a *LE* network cannot be *LTLE* is to use Algorithm GaussianReduceNetwork to transform it into a network $\check{G} = (\mathbf{I}, \check{B}^1, \check{A})$. If B^0 cannot be reduced to $\mathbf{I}$, then it is not a *LTLE* network. Otherwise, we can assume that $B^0 = \mathbf{I}$ and do other tests.

The two matrices B^1, A can be interpreted as an adjacency matrix for a digraph of n nodes. If the *LE* network is a *LTLE* network, then each of the matrices are acyclic digraphs (with the exception of the self-looping edges (i, i) that occur because $B^0_{i,i} = B^1_{i,i} = 1$). The matrices B^1 and A can be examined in $O(n^2)$ time to find if they are (separately) DAGs with self-loops. This check can be done using depth-first search on the matrix to find cycles. If B^0 and A are not DAGs with self-loops at each vertex, then the network cannot be a *LTLE* network.

Now since B^0 has 1's only down the diagonal, the only isomorphisms that will maintain this property are simultaneous row/column permutations. Through a series of such interchanges, we can generate any permutation of rows/columns

we wish, including permutations that are in a particular order, say a topologically sorted order.

Definition 4.2.2 *Let $G = (V, E)$ be a directed acyclic graph. A* ***topological sort*** *is the assignment of integers $a(1)$, $a(2)$, ..., $a(|V|)$ to the vertices of G so that if there is a directed edge from v_i to v_j, then $a(v_i) < a(v_j)$.*

Clearly, there can be more than one possible topological sort. One trivial example is a graph $G = (V, \{\})$, where no directed edges are in the graph. Then any numbering of the vertices is a topological sort.

Once a DAG is topologically sorted, its adjacency matrix is lower triangular. We can use topological sort on either B^1 and A to find a correct simultaneous row/column permutation that will make either one lower triangular.

Our problem is that we have to find some permutation of vertices that is simultaneously a topological sort for both B^1 and A. We can solve this problem by computing the graph union of B^1 and A, that is the union of the edges of the graphs defined by B^1 and A. If the graph union has a cycle, then there is no topological sort that can make B^0 and A both lower triangular matrices. If the union has no cycle, then it is a DAG and can be topologically sorted. Since B^1 and A are subgraphs of the graph union, they are also DAGs, and the permutation computed by this topological sort can be applied to G so that B^1 and A are simultaneously lower triangular matrices.

The algorithm for *LTLE* network inclusion is straightforward. For a network G, first use the Gaussian elimination algorithm to set $B^0 = \mathbf{I}$. If this cannot be done, then the network is not lower triangular. Then take the union of B^1 and A, remove the self-loops of B^1 from it, and check if the union is the adjacency matrix of a DAG. If it is not, then the network is not a *LTLE* network. If it is, topologically

sort it and use this ordering to simultaneously permute the rows and columns so that B^1 and A are simultaneously lower triangular matrices. The Gaussian elimination algorithm has a run time of $O(n^3)$, the union of the two DAGs can be done in $O(n^2)$ run time and the standard algorithm for topological sorting has asymptotic run time order of $O(|E| + |V|) = O(n^2)$. The total run time for *LTNI* is polynomial in n.

∎

Like the *LE* networks, the *LTLE* networks can use all the isomorphisms listed in Section 4.1.1. There are some trivial isomorphisms between *LTLE* networks. Theorem 4.1.1 leads to an trivial isomorphism property of lower triangular networks:

Lemma 4.2.1 *Let $F = (B^0, B^1, A)$ be an n-dimensional lower triangular network. Then for G, B^1_1 and $row_n(A)$ can be changed to any value without changing the network's topology.*

Proof: This is obvious, because $row_1(A) = \vec{0}$ and $B^0_n = B^1_n$. ∎

LTLE networks can also be treated as a special case of Theorem 4.1.7:

Corollary 4.2.1 *Let $G = (B^0, B^1, A)$ be an n-dimensional* LTLE *network. Then G is isomorphic to a network $\check{G} = (\boldsymbol{I}, \check{B}^1, \check{A})$.*

Proof: Because G is *LTLE* , B^0 and B^1 meet the conditions of Theorem 4.1.7. In fact, we need only to do the back-substitution portion of Algorithm GaussianReduceNetwork, because the elimination portion is already completed for us.

∎

We noticed before that Network Isomorphism is at least as hard as the Graph Isomorphism. However, *LTNI* has an algorithm with a strictly polynomial run time. We should also ask if two *LTLE* networks can be found to be isomorphic in polynomial time, too. This is not the case, because Graph Isomorphism on DAGs is at least as hard as Graph Isomorphism.

Definition 4.2.3 ***Graph Isomorphism on Directed Acyclic Graphs (GI-DAG):***

Instance: *Two directed graphs* $G_1 = (V, D_1)$ *and* $G_2 = (V, D_2)$.

Question: *Is* G_1 *isomorphic to* G_2*?*

Theorem 4.2.6 Graph Isomorphism $\leq$ GI-DAG.

Proof: Given two undirected networks $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$, we use a polynomial-time construction to make two directed graphs $\check{G}_1 = (\check{V}, D_1)$ and $\check{G}_2 = (\check{V}, D_2)$.

To construct $\check{V}$, construct three vertices $\check{v}_{i,1}$, $\check{v}_{i,2}$ and $\check{v}_{i,3}$ for each vertex $v_i \in V$. To construct D_1, include the edges $(\check{v}_{i,1}, \check{v}_{i,2})$, and $(\check{v}_{i,2}, \check{v}_{i,3})$ for each $v_i \in V$. Then include the edge $(\check{v}_{i,1}, \check{v}_{j,3})$ for each edge $(v_i, v_j) \in E_1$. To construct D_2, include the edges $(\check{v}_{i,1}, \check{v}_{i,2})$, and $(\check{v}_{i,2}, \check{v}_{i,3})$ for each $v_i \in V$. Then include the edge $(\check{v}_{i,1}, \check{v}_{j,3})$ for each edge $(v_i, v_j) \in E_2$.

Now, clearly if G_1 is isomorphic to G_2, then $\check{G}_1$ is isomorphic to $\check{G}_2$. For each vertex v_i of G_1 that is mapped to v_j of G_2 in the undirected graphs, we map $\check{v}_{i,1}$ of $\check{G}_1$ to $\check{v}_{j,1}$ of $\check{G}_2$, $\check{v}_{i,2}$ of $\check{G}_1$ to $\check{v}_{j,2}$ of $\check{G}_2$ and $\check{v}_{i,3}$ of $\check{G}_1$ to $\check{v}_{j,3}$ of $\check{G}_2$.

However, we must show that if $\check{G}_1$ is isomorphic to $\check{G}_2$, then G_1 is isomorphic to G_2. Assume that $\check{G}_1$ is isomorphic to $\check{G}_2$. Note that in the constructed digraph $\check{G}_1$ only the vertices $\check{v}_{i,1}$ where $1 \leq i \leq |\check{V}|$ have in-degree 0. From each $\check{v}_{i,1}$, there is exactly one path of length 2. All others are length 1. This means if any isomorphism between $\check{G}_1$ and $\check{G}_2$ exists, then every $\check{v}_{i,1}$ of $\check{G}_1$ must map to some $\check{v}_{j,1}$ of $\check{G}_2$, and $\check{v}_{i,2}$ and $\check{v}_{i,3}$ of $\check{G}_1$ must then map to $\check{v}_{i,2}$ and $\check{v}_{i,3}$ of $\check{G}_2$, respectively.

Then the vertex v_i in G_1 can map to v_j in G_2, and so G_1 is isomorphic to G_2. ∎

This result is hardly surprising. After typing the proof, I found it given as an exercise in a textbook [4].

4.3. MINIMUM-WEIGHT ISOMORPHISMS FOR *LTLE* NETWORKS

In Subsection 2.2, we saw that the Flip MCube and YAT cube are isomorphic to the Möbius cubes. This is an interesting isomorphism, because while the Möbius cubes have twisted edges that span up to n dimensions, the Flip Mcube and the YAT cube have twisted edges that span only 2 dimensions each. Using the standard isomorphisms, we can transform a network to an isomorphic network that has fewer twisted channels, or channels that twist across fewer dimensions.

For limited classes of the *LTLE* networks, we can sometimes reduce the number of twists by transforming a *LTLE* network to another *LTLE* network. Theorem 4.3.1 below gives some idea of how many twisted edges we can remove from such a network.

Theorem 4.3.1 *Let* $G = (B^0, B^1, A)$ *be an* n*-dimensional* LTLE *network with* $B_i^0 + B_i^1 \in \{0, B_{i-1}^0, B_{i+1}^1\}$. *There is a* n*-dimensional* LTLE *network* $\check{G} = (\check{B}^0, \check{B}^1, \check{A})$ *which is isomorphic to* G, *and which has weights* $W_H(\check{B}_i^0) \leq 2$ *and* $W_H(\check{B}_i^1) \leq 2$ *for* $1 \leq i \leq n$. *Further, the 1's in each column* i *of* $\check{B}^0$ *and* $\check{B}^1$ *will only appear in rows* i *and* $i+1$.

Proof: This can be done by applying row addition transformations to each column from n to 1. This is computed by Algorithm MinimumWeightIsomorphism and is given in Figure 4.3.

Because the initial network is defined with lower triangular matrices, B_n^0 and B_n^1 each have weight 1. For each iteration $i, n-1 \geq i \geq 1$ through the loop, the

Input: A n-dimensional *LTLE* network $G = (B^0, B^1, A)$, where for $1 \leq i \leq n$, $B_i^0 + B_i^1 \in \{0, B_{i+1}^0, B_{i+1}^1\}$.

Output: A n-dimensional *LTLE* network $\check{G} = (\check{B}^0, \check{B}^1, \check{A})$, where for $1 \leq i \leq n$, $\check{B}_i^0 \leq 2$ and $\check{B}_i^1 \leq 2$.

```
Procedure MinimumWeightIsomprhism( G = (B^0, B^1, A) )
    for i = n − 1 down to 1 do
        if W(B^1_{i+1}) = 2 and B^0_i + B^1_i = B^1_{i+1}
            Add row i + 1 to row i + 2
        end if
        for j = i + 1 to n
            if B^0_{j,i} = 1 then
                Add row i to row j
            end if
        end for
    end for
end procedure
```

Figure 4.3. Algorithm MinimumWeightIsomorphism.

algorithm modifies columns i and $i+1$ to guarantee that the weights of both B_i^0 and B_i^1 are each at most two.

The algorithm first checks if column B_{i+1}^1 has weight 2. If it does and $B_{i+1}^0 = B_i^0 + B_i^1$, the algorithm changes the weight of B_{i+1}^1 to 1. In this way, each iteration forces B_i^0 and B_i^1 differ by at most a single bit in row $i+1$. The algorithm then forces B_i^0 to weight 1, with the 1 in row i, and forces B_i^1 to weight 2, with the 1's in rows i and $i+1$.

At any point in the algorithm, the row addition will maintain the isomorphism to G and keep the *LTLE* network property. The addition of row i to higher rows does not affect columns $i+1$ through n, because row i is zeroed in those columns – the matrices are lower triangular. ∎

Notice that all of the published *LE* networks are not only *LTLE* networks, but they also meet the conditions of the theorem above and so are isomorphic to networks with columns that have maximum Hamming weight 2.

An advantage of using such a minimum-weight *LTLE* networks is that the physical layout of the network will be simpler, because channel twists will now occur across at most two dimensions. Fewer twisted channels also means that we can use a circuit layout that closely resembles the circuit layout for the hypercube.

We can extend this algorithm to more general cases. Using a similar algorithm, a *LTLE* network $G = (B^0, B^1, A)$ with $B_i^0 + B_i^1 \in \{0, B_{i+1}^0, B_{i+1}^1, \ldots B_{i+k}^0, B_{i+k}^1\}, 1 \leq k \leq n$, can always be transformed to an isomorphic *LTLE* network $\check{G} = (\check{B}^0, \check{B}^1, \check{A})$ where $W(\check{B}_i^0) \leq k$ and $W(\check{B}_i^1) \leq k$, and where the 1's in each column i of B^0 and B^1 appear only in rows i through $i+(k-1)$, for $i+(k-1) \leq n$. However, this may not be the absolute minimal weight network.

4.4. SUMMARY

In this chapter, we have again demonstrated that the *LE* network model is too general for describing interconnection networks. It allows one network to have more than one matrix description. There is even strong evidence that there is no efficient algorithm for determining if two matrix descriptions describe isomorphic networks. This is true for matrix descriptions of either *LE* networks or *LTLE* networks.

However, we did show a number of standard isomorphisms that could be used to to show that two networks are isomorphic. We also showed there is a efficient algorithm to determine if a *LE* network is isomorphic to a *LTLE* network, so that *LTLE* network results hold for these networks. Finally, we showed that at least some *LTLE* networks can be transformed to isomorphic *LTLE* networks with a minimal number of twisted edges.

5. ROUTING ALGORITHMS FOR *DM* NETWORKS

In this chapter, we devise a method for computing a minimal routing path between a source node $\vec{X}$ and destination node $\vec{Y}$ on any *LTDM* network $G = (B^0, B^1)$. We do this by computing a minimal expansion of $\vec{X} + \vec{Y}$ over the column vectors of B^0 and B^1.

We show that the problem of computing a minimal expansion is intractable, even for *DM* networks with lower triangular B^0 and B^1 matrices. We then show that certain subclasses of the *DM* networks have efficient algorithms for computing minimal expansions.

These results on *DM* networks will be used in computing a general minimal routing algorithm for *LE* networks.

5.1. DEFINITION OF MINIMAL EXPANSIONS

Consider $\mathcal{Z}_2^n$, the n-dimensional vector space over $\{0,1\}$ with vector addition and scalar multiplication mod 2. Assume also that the set $\{B_i^0, B_i^1 : 1 \leq i \leq n\}$ contains a basis for this vector space. It is possible, but not necessary, that B^0 and B^1 are each a basis.

Because $B^0 \cup B^1$ contain at least one basis over $\mathcal{Z}_2^n$, any vector $\vec{X}$ can be represented as a linear sum of these basis vectors. Let each $\alpha_i \in \{0,1\}$ and $\beta_i \in \{0,1\}$. Then:

$$\vec{X} = \sum_{i=1}^{n} (\alpha_i B_i^0 + \beta_i B_i^1), \tag{5.1}$$

Clearly, we can represent a vector $\vec{X}$ by the set of vectors B_i^0 and B_i^1 which have nonzero coefficients in this sum.

Definition 5.1.1 *$S(\vec{X}) \subset \{B_i^0, B_i^1 : 1 \leq i \leq n\}$, is an **expansion** of $\vec{X}$ iff the equality in Equation 5.1 is true, with $\alpha_i = 1$ iff $B_i^0 \in S(\vec{X})$ and $\beta_i = 1$ iff $B_i^1 \in S(\vec{X})$. For convenience, $S(\vec{X})$ is denoted S when $\vec{X}$ is obvious. Any $t \in S(\vec{X})$ is called a **term** of S.*

Because there can be more than one subset of B^0, B^1 that can form a basis over $\mathcal{Z}_2^n$, there can be more than one expansion of a vector. Several different linear sums, each with possibly a different number of vectors, can add up to the same vector $\vec{X}$. For this reason, we define the following:

Definition 5.1.2 *For a vector $\vec{X}$, the **weight** of an expansion $S(\vec{X})$ is the the cardinality of $S(\vec{X})$.*

Equivalently, the *weight* of an expansion $S(\vec{X})$ is the number of nonzero α's and β's in that expansion. An expansion can have a weight of up to $2n$. But $B^0 \cup B^1$ contains a basis over $\mathcal{Z}_2^n$, and so every vector over $\mathcal{Z}_2^n$ clearly has an expansion of weight of n or less.

Definition 5.1.3 *For any vector $\vec{X}$, let **a minimal expansion** of $\vec{X}$ be an expansion with least weight. For any vector $\vec{X}$, we let $W(\vec{X})$ denote the weight of a minimal expansion of $\vec{X}$.*

Certainly $W(\vec{X})$ is well-defined since there are only a finite number of expansions. However, there may be more than one minimal expansion for a particular vector $\vec{X}$ under B^0 and B^1.

5.2. USING MINIMUM EXPANSIONS TO ROUTE ON *DM* NETWORKS

Assume that for a connected *DM* network $G = (B^0, B^1)$, $\vec{X}$ and $\vec{Y}$ are the source and destination nodes of a message. Any expansion $S(\vec{X} + \vec{Y})$ is all that is necessary to compute a routing path.

At each step k with $1 \leq k \leq |S|$ in the routing path, we only add $B_{i_k}^{\phi_k} \in S$ to $\vec{X}_k = \vec{X} + \sum_{j=1}^{k} B_{i_j}^{\phi_j}$ to go to the next node $\vec{X}_{k+1}$. Every step is legal, because every node $\vec{X}_k$ would have the channels $(\vec{X}_k, \vec{X}_k + B^0)$ and $(\vec{X}_k, \vec{X}_k + B^1)$ defined, and the routing path correctly terminates at $\vec{Y}$ through mod 2 addition because:

$$\vec{X} + \sum_{k=1}^{|S|} B_{i_k}^{\phi_k} \in S = \vec{Y}$$

To route minimally, we only need to compute a minimal expansion. For a *DM* network $G = (B^0, B^1)$, a minimal expansion $S(\vec{X} + \vec{Y})$ defines one *or more* minimal routing paths between $\vec{X}$ and $\vec{Y}$, because addition over $\mathcal{Z}_2^n$ is commutative.

The length of the computed path will be $|S| \leq n$. A term will not appear more than once in a routing path, so we can represent the expansion S in as little as $2n$ bits, with each bit representing the inclusion or the exclusion of one term.

Let the time to compute a minimal expansion be $T(n)$ bit operations. We can compute a minimal expansion at the source node, and forward it along with the message to keep from repeatedly recomputing S at each address. We can also forward the last term used in routing, so that the routing algorithm doesn't have to check all $2n$ bits at each routing step. If we do both of these, the distributed run time at each node will be at worst $O(T(n))$ bit operations, because we compute S at the first node. The total run time will be $O(T(n) + n)$ bit operations – at most a constant number of bit operations at each succeeding node.

We have not yet examined an algorithm to efficiently compute a minimal expansion $S(\vec{X}+\vec{Y})$, so we don't yet know what $T(n)$ is.

5.3. INTRACTABILITY OF THE MINIMAL EXPANSION PROBLEM

The problem of finding a minimal expansion is important if we wish to quickly compute a minimal routing path on a given *DM* network. We define the minimal expansion problem below:

Definition 5.3.1 ***ME-DMN: Minimal Expansions in*** **DM** ***Networks:***

Instance: *An n-dimensional* DM *network* $G=(B^0,B^1)$, *source and destination addresses* $\vec{X},\vec{Y}\in\mathcal{Z}_2^n$, *and an integer* K *with* $1\leq K\leq 2n$.

Question: *Is there a expansion* $S(\vec{X}+\vec{Y})\subset\{B^0,B^1\}$ *with* $|S|\leq K$*?*

This problem is very similar to another problem, that of finding a minimal-weight solution to a set of linear equations over $\mathcal{Z}_2$.

Definition 5.3.2 ***COSET WEIGHT: [9] Coset Weights over*** $\mathcal{Z}_2^n$***:***

Instance: *A binary matrix* $\check{A}$, *a binary vector* y, *and a non-negative integer* w.

Question: *Is there a vector* x *of Hamming weight* $\leq w$ *such that* $xA=y$.

Theorem 5.3.1 CO SET WEIGHT *is* NP*-complete.*

Proof: (See [9]). ∎

Theorem 5.3.2 ME-DMN *is* NP*-complete.*

Proof: *ME-DMN* $\in$ *NP*, because we can guess an expansion S and verify that each $B_i^\phi\in S\subset\{B^0,B^1\}$, and that $\sum B_i^\phi\in S=\vec{X}+\vec{Y}$. These operations can be done in a polynomial number of bit operations.

COSET WEIGHT $\leq$ *ME-DMN*, because we can choose the dimension of G to be $\max(n, m)$, then set B^1 to $\vec{0}$, and set B^0 to $\check{A}$, filling the extra columns and rows with zeros, and finally choose $\vec{X} + \vec{Y} = y$ and $K = w$. Then a solution to *ME-DMN* of weight K is also clearly a solution to *COSET WEIGHT* of weight w. ∎

A minimal expansion is always a minimal routing path on a *DM* network, so the problem of computing a routing path on a *DM* network is also *NP*-complete.

5.4. A MINIMAL EXPANSION ALGORITHM FOR *LTDM* NETWORKS

Although the problem of computing minimal expansions is *NP*-complete, we can still design an efficient algorithm for a large number of *DM* networks. We will show that breadth-first search techniques will require at most a polynomial number of search states in many cases.

The obvious method for finding the shortest path between two vertices in a graph is to find the destination vertex using breadth-first search from the source vertex. This takes in the worst case $O(|V| + |E|)$ operations for a general directed graphs $G = (V, E)$, because every vertex and edge in the graph might be examined in the search. If we try this approach for a *DM* network, we may require up to $O(2^n + n2^n) = O(n2^n)$ search operations, which might each take $O(n^2)$ bit operations to compute (the time to compute the address of a neighboring node).

If we restrict ourselves to only lower triangular matrices for B^0 and B^1, where $B^0_{i,i} = B^1_{i,i} = 1$, then the search space can be reduced considerably. With lower triangular B^0 and B^1, only the terms B^0_1, B^1_1, ..., B^0_i, B^1_i can set $X_i = Y_i$. All terms B^0_j and B^1_i with $j > i$ cannot affect X_i.

This affects our search for an expansion $\vec{X} + \vec{Y}$. Consider the search space as a tree structure that we construct, with a root node $\vec{X}$. Each vertex of the tree

at level i is labeled with the address of a node $\vec{X}_i$ in G. Each edge $(\vec{X}_i, \vec{X}_{i+1})$ is labeled with either 0, B_i^0, B_i^1, or $B_i^0 + B_i^1$, which will be the mod 2 sum $\vec{X}_i + \vec{X}_{i+1}$). The edge will be weighted with the number of terms in its label, which is either 0, 1, or 2.

From a node $\vec{W}$ in the i-th level of the search, we need to examine only how B_i^0 and B_i^1 can be included in the expansion from $\vec{W}$.

If $(\vec{W} + \vec{Y})_i = 1$, then there must be an odd number of B_i^0 and B_i^1 in the expansion. The only minimal choices are B_i^0 and B_i^1 themselves. We compute the addresses of $\vec{W} + B_i^0$ and $\vec{W} + B_i^1$, insert them as nodes at level $i+1$, and recursively search from them to $\vec{Y}$ using only terms B_{i+1}^0, B_{i+1}^1, ..., B_n^0, B_n^1.

If $(\vec{W} + \vec{Y})_i = 0$, then there must be an even number of B_i^0 and B_i^1 in the expansion. The only minimal choices are 0 and $B_i^0 + B_i^1$. We compute the addresses of $\vec{W} + 0$ and $\vec{W} + B_i^0 + B_i^1$, insert them as nodes at level $i + 1$, and recursively search from them to $vecY$ using only B_{i+1}^0, B_{i+1}^1, ..., B_n^0, B_n^1.

This approach takes $O(2^n)$ search operations, which can each take $O(n)$ bit operations to compute. If we construct this "expansion search tree" while we are searching it, we can reduce the number of vertices we construct by querying whether a vertex with a given address already exists in the tree before we insert it. If it doesn't exist, we insert a new vertex in the graph and add a new edge from our current vertex to it. If it does exists, we don't insert a new vertex; instead, we only add an edge from the current vertex to the existing vertex.

The constructed expansion search tree will be a directed graph of depth $n+1$. This graph will have label $\left\langle \vec{X}, 1 \right\rangle$ as the "root" vertex and one "leaf" vertex with address $\left\langle \vec{Y}, n+1 \right\rangle$. Algorithm ExpansionTree in Figure 5.1 will compute an expansion search tree D. Figure 5.2 shows an expansion search tree between addresses $X = (11011)$ and $Y = (10110)$ for the Folded Hypercube of [6] of dimension 5.

Input: A n-dimensional *LTDM* network $G = (B^0, B^1)$, a starting address $\vec{X}$ and a destination address $\vec{Y}$.

Output: A weighted, directed graph $D = (V, E)$, where each path from the vertex $\langle \vec{X}, 1 \rangle$ to $\langle \vec{Y}, n+1 \rangle$ corresponds to an expansion of $\vec{X} + \vec{Y}$.

Procedure ExpansionTree(G, $\vec{X}$, $\vec{Y}$)
    $V \leftarrow \{\langle \vec{X}, 1 \rangle\}$
    $E \leftarrow \{\}$
    for each $i : 1 \leq i \leq n$ do
        for each $\langle \vec{W}, i \rangle \in V$ do
            if $W_i = Y_i$ then begin
                $V \leftarrow V \cup \{\langle \vec{W}, i+1 \rangle, \langle \vec{W} + B_i^0 + B_i^1, i+1 \rangle\}$
                $E \leftarrow E \cup \{(\langle \vec{W}, i \rangle, \langle \vec{W}, i+1 \rangle, 0),$
                    $(\langle \vec{W}, i \rangle, \langle \vec{W} + B_i^0 + B_i^1, i+1 \rangle, 2)\}$
            end if
            if $W_i \neq Y_i$ then begin
                $V \leftarrow V \cup \{\langle \vec{W}, i+1 \rangle, \langle \vec{W} + B_i^1, i+1 \rangle\}$
                $E \leftarrow E \cup \{(\langle \vec{W}, i \rangle, \langle \vec{W} + B_i^0, i+1 \rangle, 1),$
                    $(\langle \vec{W}, i \rangle, \langle \vec{W} + B_i^1, i+1 \rangle, 1)\}$
            end if
        end for
    end for
    return $D \leftarrow (V, E)$
end procedure

Figure 5.1. Algorithm ExpansionTree.

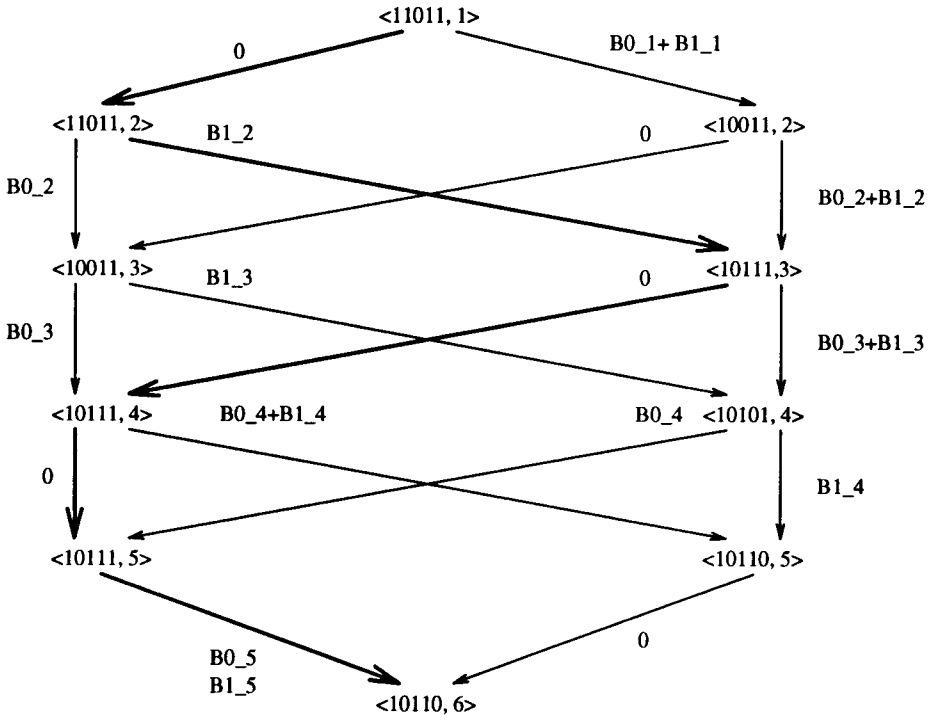

Figure 5.2. An expansion search tree between addresses $X = (11011)$ and $Y = (10110)$ on a 5-dimensional Folded Hypercube. The minimal path is shown in bold.

Theorem 5.4.1 *The algorithm ExpansionTree(G, $\vec{X}$, $\vec{Y}$) correctly computes a minimal expansion search tree D.*

Proof: The algorithm will eventually terminate, because there are at most $n+1$ levels in the tree, and for each level i, there are at most 2^{n-i+1} vectors $\vec{W}$ which have $W_j = Y_j$ with $i \leq j \leq n$, and the algorithm generates at most one vertex $\langle \vec{W}, i \rangle$ for any one of these.

The algorithm terminates with the correct result. The algorithm's loop invariants are for any vertex $\langle \vec{W}, i \rangle \in V$:

1. There is a path from $\langle \vec{X}, 1 \rangle$ to $\langle \vec{W}, i \rangle$.
2. This path corresponds to an expansion $S(\vec{X} + \vec{W})$ with a weight equal to the path's weight.
3. For $\langle \vec{W}, i \rangle$, $W_j = Y_j$ with $1 \leq j \leq i-1$.

It is not hard to show that if the algorithm inserts the edge $(\langle \vec{W}, i \rangle, \langle \vec{V}, i+1 \rangle, d)$, for some $\vec{V} \in \vec{W} + \{0, B_i^0, B_i^1, B_1^0 + B_i^1\}$, then each of these conditions hold for the new vertex $\langle \vec{V}, i+1 \rangle$. The only vertex that meets the third condition above when $i = n+1$ is $\langle \vec{Y}, n+1 \rangle$.

It is also clear that all paths in D correspond to some expansion $S(\vec{X} + \vec{Y})$ and each path has a weight equal to its corresponding expansion's weight.

There are no expansions that $S(\vec{X} + \vec{Y})$ that do not define a path in D. Assume that there is such an expansion $\check{S}$.

At some i with $1 \leq i \leq n$, there is a path from $\langle \vec{X}, 1 \rangle$ to $\langle \vec{W}, i \rangle$ that is defined by the set of terms $\{B_j^\phi \in S : 1 \leq j \leq i-1\}$, but either:

1. S contains either B_i^0 or B_i^1 (but not both), but there are edges from $\langle \vec{W}, i \rangle$ to $\langle \vec{W}, i+1 \rangle$ and $\langle \vec{W} + B_i^0 + B_i^1, i+1 \rangle$.

2. S contains neither B_i^0 nor B_i^1, but there are edges from $\langle \vec{W}, i \rangle$ to $\langle \vec{W} + B_i^0, i+1 \rangle$ and $\langle \vec{W} + B_i^1, \ i+1 \rangle$.

But in the first case $(\vec{X} + \vec{W})_i = 0$, so S cannot contain just B_i^0 or B_i^1 because S is an expansion of $\vec{X} + \vec{Y}$. In the second case $(\vec{X} + \vec{W})_i = 1$, so S must contain either B_i^0 or B_i^1 (but not both) because again S is an expansion of $\vec{X} + \vec{Y}$. ∎

We can find a minimal expansion from D, by finding a minimally weighted path from $\langle \vec{X}, 1 \rangle$ to $\langle \vec{Y}, n+1 \rangle$. This can be done using a breadth-first search from $\langle \vec{X}, 1 \rangle$. Each edge in D corresponds to zero, one or two terms in one or more expansions of $\vec{X} + \vec{Y}$, so a minimally weighted path corresponds to a minimal expansion of $\vec{X} + \vec{Y}$. Once we have found a shortest path in D, we can derive its corresponding minimal expansion quickly, by looking up which terms correspond to each channel in the path.

What is the largest size that the search tree D can attain? At any level i, there can be at most 2^n different nodes, because D has two branches from every node. The vertex $\langle \vec{W}, i \rangle$ has the property that $W_j = 0$ for $1 \leq j \leq i-1$, so that the vector $\vec{W}$ is also limited to 2^{n+1-i} different values. Maximizing under these two constraints this shows the largest possible number of vertices at any level is $2^{\lfloor n/2 \rfloor}$ – and summing over all possible vertices gives an upper bound of $|D| = O(2^{n/2})$.

Clearly this is an exponential number of vertices, and will lead to an exponential run time on any shortest path algorithm. We should consider any conditions that limits $|D|$ to a polynomial size, for instance, the number of linear combinations that can be made with B^0 and B^1.

Theorem 5.4.2 *Let $G = (B^0, B^1, A)$ be an n-dimensional lower triangular matrix, let $D = (V, E)$ be the expansion search tree and let the matrix H_i be defined as:*

$$\begin{bmatrix} B^0_{i,1} & B^0_{i,2} & \cdots & B^0_{i,i} & B^1_{i,1} & B^1_{i,2} & \cdots & B^1_{i,i} \\ B^0_{i+1,1} & B^0_{i+1,2} & \cdots & B^0_{i+1,i} & B^1_{i+1,1} & B^1_{i+1,2} & \cdots & B^1_{i+1,i} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ B^0_{n,1} & B^0_{n,2} & \cdots & B^0_{n,i} & B^1_{n,1} & B^1_{n,2} & \cdots & B^1_{n,i} \end{bmatrix}$$

Then $|V|$ is less than the sum of the sizes of the column spaces of H_i for $1 \leq i \leq n$, that is:

$$|V| \leq \sum_{i=1}^{n} \left| \left\{ \sum_{j=1}^{2i} \alpha_j (H_i)_j : \alpha_j \in \{0,1\} \right\} \right| + 1$$

Proof: For a given i, any vertex $\left\langle \vec{W}, i \right\rangle$ has for $\alpha_j, \beta_j \in \{0,1\}$:

$$\vec{W} = \vec{X} + \sum_{j=1}^{i} \alpha_j B^0_j + \beta_j B^1_j$$

$\vec{W}$ has the property that $W_j = 0$ for $1 \leq j \leq i-1$, so all the vectors in vertices at level i will be the same in the first $i-1$ indices. If we ignore the first $i-1$ indices of the columns of B^0 and B^1, then we only need to know how many different values a linear combination of the remaining indices can make. This is exactly the size of the column space of H_i above. Finally, there is only one vertex $\left\langle \vec{Y}, n+1 \right\rangle$. ∎

This is a rather loose upper bound, useful only for bounding the number of vertices in D to order.

For most networks, the number of distinct vertices appearing at each level of D is a fairly small constant. The next theorem uses that fact to limit the run time of a minimal expansion algorithm to a constant times $O(n^2)$, for most *LTDM* networks.

Theorem 5.4.3 *Let $G = (B^0, B^1)$ be an n-dimensional* LTDM *network. If the expansion tree can have at most $O(k)$ vertices at any depth, for some constant k, then a minimal expansion algorithm takes $O(kn^2)$ bit operations to compute.*

Proof: The algorithm has three stages: generating the expansion search tree D, computing a minimal weight path in D, and deriving a minimal expansion from the minimal weight path.

The first stage generates the D. Each vertex in D takes $O(n + \log n)$ bit operations to construct (a mod 2 vector addition and an integer increment). We can then divide the vertices into $n+1$ groups or "buckets" by their depth in D (the second component of each node's label). Each group then always has at most $O(k)$ vertices (by the assumptions in theorem's conditions). Inserting a new vertex into D takes at most $O(n)$ bit operations, because we need $O(k)$ vector comparisons to prevent redundant vertices from being inserted. Inserting a new edge into D takes $O(\log k)$ bit operations, because there are at most $O(k)$ vertices to point to. The total number of bit operations to construct the tree is:

$$O(kn) \times O(n) + O(kn) \times O(\log k) = O(kn^2)$$

The second part of the algorithm uses a breadth-first search algorithm on D to find a minimal weight path from $\left\langle \vec{X}, 1 \right\rangle$ to $\left\langle \vec{Y}, n+1 \right\rangle$. This can be done quickly level-by-level. Assume that we have already computed the distance of every node in level i from $\left\langle \vec{X}, 1 \right\rangle$. Consider an edge $\left(\left\langle \vec{W}, i, \right\rangle, \left\langle \vec{V}, i+1 \right\rangle, w \right)$. We can compute distance of $\vec{V}$ from $\vec{X}$ by adding the distance of $\vec{W}$ and the edge weight w. We compare this value to any previously computed distances of $\vec{V}$, and keep the minimal for $\vec{V}$.

This operation takes $O(\log k)$ bit operations for a pointer dereference to find $\vec{V}$, plus $O(\log n)$ bit operations to do a constant to integer addition, and an integer comparison. The size of the integer is bounded by $\log n$ because the maximum distance is n. Because there are at most $2k$ edges per level, the total number of bit operations is:

$$O(kn) \times [O(\log n) + O(\log k)] = O(kn \log n)$$

We can keep track of a minimal path to each node by having each node keep a $(\log k)$-bit pointer to its parent node that is closest to the root. When the minimal distance to each node has been computed, the chain of back-pointers from $\langle \vec{Y}, n+1 \rangle$ to $\langle \vec{X}, 1 \rangle$ describes a minimal weight path (in reverse). This record-keeping does not change the complexity of the algorithm.

The final part of the algorithm computes and stores the terms that correspond to edges on the minimal weight path. The terms can be looked up by following the path found by breadth-first search. The terms can be looked up by pointer dereference – $\log k$ bit operations per term – and compactly stored using 1 bit per term. The total number of bit operations to compute the expansion is:

$$O(n) \times [O(1) + O(\log k)] = O(n \log k)$$

The total number of bit operations overall is:

$$O(kn^2) + O(kn \log n) + O(n \log k) = O(kn^2)$$

∎

5.5. A *LTDM* NETWORK ROUTING ALGORITHM

Now we use the results from the minimal expansion algorithm to produce a routing algorithm for *LTDM* networks.

We present the steps to route minimally on a *DM* network in Algorithm DoubleMatrixRoute in Figure 5.3. We assume that the call to BreadthFirstSearch (not listed) does a standard breadth-first search on D and returns a minimal path P. We also assume that the call to PathToExpansion (not listed) converts P to its corresponding expansion S. Because we know the run time to compute the minimal expansion, we can compute the run time of Algorithm DoubleMatrixRoute.

Input: A n-dimensional *DM* network $G = (B^0, B^1)$, the current address $\vec{W}$, a starting address $\vec{X}$ and a destination address $\vec{Y}$.

Output: The next neighbor to route to from $\vec{W}$.

```
DoubleMatrixRoute( G = (B^0, B^1), W, X, Y)
    if W = Y then
        accept message
    if W = X then
        D ← ExpansionTree(G, X, Y)
        P ← BreadthFirstSearch(D)
        S ← PathToExpansion(P)
    else
        remove B_i^φ from head of S
        route from X to X + B_i^φ
    end if
end procedure
```

Figure 5.3. Algorithm DoubleMatrixRoute.

Theorem 5.5.1 *Let $G = (B^0, B^1)$ be an n-dimensional* LTDM *network. If the expansion search tree D has at most $O(k)$ vertices at any level, for some constant k, then the distributed run time of Algorithm DoubleMatrixRoute is at most $O(kn^2)$ bit operations per processor and the total run time is at most $O(kn^2)$ bit operations.*

Proof: This is trivially true, because the computation of D dominates the run time of the algorithm, and forwarding through the network will take $O(1)$ bit operations per node. Since the routing computation is done entirely at the source node, the routing computation will dominate both the total and distributed computation times. ∎

This gives a strict linear bound on a minimal expansion algorithm's run time, for all of the published *LTLE* networks.

Corollary 5.5.1 *For the Folded Hypercube of [44] and [6] and and the Enhanced Hypercube of [54] of dimension n, the distributed run time of Algorithm DoubleMatrixRoute is at most $O(4n^2)$ bit operations per processor and the total run time is at most $O(4n^2)$ bit operations.*

Proof: Examining the lower triangular matrix description of either of these networks reveals that the size of the column space of any H_i, $1 \leq i \leq n$ is never more than 4, by Theorem 5.5.1. Both networks have at most $O(4n)$ vertices in the expansion search tree, and at most 4 vertices at any level of the expansion search tree.

By Theorem 5.5.1, the algorithm will take at most $O(4n^2) = O(n^2)$ bit operations to find a minimal expansion on either the Folded Hypercube and the Enhanced Hypercube. ∎

The best algorithms for the Folded Hypercube and the Enhanced Hypercube can compute a routing in $O(n)$ and $O(n^2)$ bit operations, respectively. Though the

asymptotic run time order of our algorithm is larger, it has the advantage that it can efficiently route on a large number of *LTDM* networks.

5.6. NONREDUNDANT MINIMAL EXPANSIONS

Sometimes a minimal expansion of some vector $\vec{X}$ can contain both B_i^0 and B_i^1 for some i. Since this kind of expansion may be the only minimal expansion, any algorithm for finding minimal expansions must check for such an occurrence.

If we could show conditions for which B_i^0 and B_i^1 would never occur together in a minimal expansion, then the expansion tree algorithm would be simpler, because it would never have to check for these "redundant" terms, and the resulting expansion tree would be smaller.

If an expansion $S(\vec{X})$ is a multi-set, where the same element can appear in S more than once, then we can define "redundancy" as:

Definition 5.6.1 *For an n-dimensional* DM *network* $G = (B^0, B^1)$, *an expansion of a vector* $S(\vec{X})$ *is* ***redundant*** *iff* S *contains both* B_i^0 *and* B_i^1 *for some index* i, *or if* S *contains more than one occurrence of* B_i^0 *or* B_i^1. *Otherwise,* S *is* ***nonredundant***. *The terms* B_i^0 *and* B_i^1 *are called* ***redundant terms*** *if they appear together or each appear more than once in* S.

For some *DM* networks, the only minimal expansion is a redundant one. For instance, for the matrices:

$$B^0 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad B^1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

The only minimal expansion of (0111) is $B^0 + B^1$.

Because some *DM* networks have redundant minimal expansions, we need to ask what necessary and sufficient conditions guarantee that no redundant expansions can exist in a network.

Theorem 5.6.1 *A* DM *network* $G = (B^0, B^1)$ *contains no redundant minimal expansions iff for all* i, $W(B_i^0 + B_i^1) \leq 1$.

Proof: Assume that $S(\vec{X})$ is a expansion that contains redundant terms at index i. Then either B^0 or B^1 appear more than once each, or B^0 and B^1 appear once each together.

If B^0 or B^1 appear more than once each, we can immediately remove pairs of B_i^ϕ that occur more than once, because $B_i^\phi + B_i^\phi = 0$. So S cannot be minimal.

If B^0 and B^1 appear once each together, then either $W(B_i^0 + B_i^1) = 0$ or $W(B_i^0 + B_i^1) = 1$.

If $W(B_i^0 + B_i^1) = 0$, then $B_i^0 = B_i^1$ and the expansion $\check{S} = S - \{B_i^0, B_i^1\}$ is smaller by two terms.

If $W(B_i^0 + B_i^1) = 1$, then:

$$B_i^0 + B_i^1 = t \in \{B_{i+1}^0, \ldots, B_{i+1}^0, B_{i+1}^0, \ldots, B_{i+1}^0\}$$

and the expansion $\check{S} = S - \{B_i^0, B_i^1\} + \{t\}$ is smaller by one term. So S cannot be minimal.

Now assume G contains no redundant minimal expansions. The minimal expansion of $B_i^0 + B_i^1$ must have weight less than 2, or $B_i^0 + B_i^1$ would be its own redundant minimal expansion. If it has weight 0 or 1, then the only minimal expansions must be in the set $\{0, B^0, B^1\}$. ∎

For *LTDM* networks, the necessary and sufficient conditions can be restricted further to the condition that for all i, $B_i^0 + B_i^1 = t \in \{0, B_{i+1}^0, B_{i+1}^1, \ldots, B_n^0, B_n^1\}$.

Notice that for these networks $B^0_{i,i} + B^1_{i,i} = 1$, which implies that for $\alpha_j, \beta_j \in \{0,1\}$:

$$B^0_i + B^1_i = \sum_{j=i+1}^{n} \alpha_j B^0_i + \beta_j B^1_i \tag{5.2}$$

There are some *LTDM* networks that have redundant minimal expansions, but always have at least one alternate nonredundant minimal expansion for every source and destination address. The summation in Equation 5.2 allows us to specify which *LTDM* networks always have *at least* one nonredundant minimal expansion between any two addresses. Such networks will allow a nonredundant expansion tree algorithm to be used on them even if redundant minimal expansions exist.

Theorem 5.6.2 *Let $G = (B^0, B^1)$ define a n-dimensional* LTDM *network. For any $\vec{X}$, there will always be a minimal expansion $S(\vec{X})$ with no redundant terms iff for all i with $1 \leq i \leq n$ in Equation 5.2:*

$$\sum_{j=i+1}^{n} \alpha_j + \beta_j \leq 2$$

Proof: The proof is by induction on index i.

Base Case: For $i = n$, $B^0_n = B^1_n$. Then:

$$\sum_{j=n+1}^{n} \alpha_j + \beta_j = 0$$

And the only nonredundant minimal expansions possible are $S \in \{\{\}, \{B^0_n\}, \{B^1_n\}\}$.

Inductive Hypothesis: For any $\check{i} = \min\{k : \check{X}_k \neq 1\}$ with $\check{i} > i$ and $W_{\check{i}} \leq 2$, there is an minimal expansion $\check{S}(\vec{X})$ with no redundant terms.

Inductive Step: Let $i = \min\{k : X_k \neq 1\}$ and $W_i \leq 2$. Find a minimal expansion $S(\vec{X})$. S cannot have more than one occurrence of the term B^0_i, because duplicate terms of B^0_i can be paired off and removed without cost. The same is true of B^1_i.

A single term B_i^0 or B_i^1 in S is nonredundant. We can inductively find a nonredundant minimal expansion and so $S(\vec{X})$ would be equal to $B_i^0 + \check{S}(\vec{X} + B_i^0)$ or $B_i^1 + \check{S}(\vec{X} + B_i^1)$, respectively.

If $B_i^0 \in S \wedge B_i^1 \in S$, then by 5.2:

$$W_i(B_i^0 + B_i^1) = W_i(\sum_{j=i+1}^{n} \alpha_j B_i^0 + \beta_j B_i^1) = 2$$

S cannot be minimal if $W_i(B_i^0 + B_i^1) < 2$, because B_i^0 and B_i^1 could then be replaced by the fewer terms in a minimal expansion of their sum. So $B_i^0 + B_i^1 = B_j^\psi + B_k^\gamma$, where $i < j \leq k \leq n$.

We can produce another expansion $\check{S}(\vec{X}) = S(\vec{X}) - \{B_i^0, B_i^1\} + \{B_j^\psi, B_k^\gamma\}$. $X_i = 0$, so the smallest nonzero component of $\vec{X}$ is $\check{i} > i$. Thus we can then inductively find a nonredundant minimal expansion $\hat{S}$ for $\check{S}$, and then set $S(\vec{X}) = \hat{S}(\vec{X})$. ∎

If we restrict *LTDM* networks to have only nonredundant minimal expansions, then we can show several properties that these networks have:

Theorem 5.6.3 *Let $G = (B^0, B^1)$ be a* LTDM *network, and for all i with $1 \leq j \leq n$, let $W(B_i^0 + B_i^1) \leq 1$. Then for a vector $\vec{X}$, a minimal expansion $S(\vec{X})$ has:*

1. *If $i = \max\{k : X_k = 1\}$ then $\forall j, 1 \leq j < i : B_j^0 \notin S(\vec{X})$ and $B_j^1 \notin S(\vec{X})$.*
2. *If $i = \max\{k : X_k = 1\}$ then either $B_i^0 \in S(\vec{X})$ or $B_i^1 \in S(\vec{X})$.*
3. *If $i = \max\{k : X_k = 1\}$ and $\phi \in \{0, 1\}$, then $W(\vec{X}) - 1 \leq W(\vec{X} + B_i^\phi) \leq W(\vec{X})$.*

Proof: We deal with each property in turn:

1. Assume that there are one or more terms with indices less than i in a minimal expansion $S(\vec{X})$. The lowest indexed of these is at some position j. Then only

the terms B_j^0 and B_j^1 can affect the component X_j. But because $X_j = 0$, there must be an even number of terms B_j^0 and B_j^1, by mod 2 addition. But because $W(B_j^0 + B_j^1) \leq 1$, two or more terms at index j can be paired off and each pair replaced by fewer terms, by Theorem 5.6.1 above. Then any expansion of $\vec{X}$ that uses a term at position j has *at least* $W(\vec{X}) + 1$ terms and so is non-minimal.

2. By the arguments above, no term with index less than i is in S. Any term $B_j^\phi \in S$ with index $j > i$ will have $B_{i,j}^\phi = 0$ and cannot affect index i in the sum, by definition of the *LTDM* networks. So either $B_i^0 \in S$ or $B_i^1 \in S$.

3. We can express $\vec{X}$ as $B_i^0 + B_i^0 + \vec{X} = B_i^0 + (\vec{X} + B_i^0)$, so a minimal expansion for $\vec{X}$ cannot have more terms than 1 term for B_i^0, plus the number of terms in a minimal expansion for $(\vec{X} + B_i^0)$. The same holds true for B_i^1.

∎

We will use these properties to construct a expansion search tree algorithm for a nonredundant *LTDM* network. This will be similar to Algorithm ExpansionTree in Figure 5.1, but will reduce the number of vertices in the search space.

Consider a minimal expansion on the *LTDM* network $G = (B^0, B^1)$ with source address $\vec{X}$ and destination address $\vec{Y}$. By Theorem 5.6.3, the smallest indexed term B_i^ϕ with $i = \min(\{k : X_i \neq Y_i\})$ must affect $(\vec{X}+\vec{Y})_i$ in the expansion. Further, no B_j^ϕ with $1 \leq j < i$ can be used. When we search for a minimal expansion $S(\vec{X} + \vec{Y})$, this observation limits us to only two choices for the smallest indexed term in the expansion: B_i^0 and B_i^1. We can then recursively search from both $\vec{X}+B_i^0$ and $\vec{X} + B_i^1$ for the shortest path.

Because there are at most two choices at every vertex, we have a binary search tree. Because adding B_i^0 or B_i^1 to X corrects the smallest bit $(X + Y)_i = 1$,

the depth of any branch of this search tree is at most n. Thus an upper bound on the number of vertices in the nonredundant expansion search tree is $O(2^n)$.

As before, this upper bound can be reduced considerably, by querying if we already have constructed a vertex and re-using it if we have.

Note that the address of every vertex at level i of the tree has the bits $1,\ldots,i$ corrected to bits $1,\ldots,i$ of Y. There are 2^{n-i} possible combinations of bits $i+1,\ldots,n$, so from level 1 to level $n/2$, there are at most 2^i distinct vertices at each level, and from level $n/2+1$ to level n, there can be at most 2^{n-i} distinct vertices. The total number of vertices in the tree is then $O(2^{n/2})$ instead of $O(2^{2n/3})$.

Algorithm NonredundantExpansionTree in Figure 5.4 will compute a nonredundant expansion tree. The the proof of correctness is similar to the one in Theorem 5.4.1, so it won't be repeated here. However some differences should be noted.

This algorithm generates only vertices $\left\langle \vec{W}, i \right\rangle$ where $i \geq \min\left(\{k : X_k \neq Y_k\}, n+1\right)$ and $W_j = Y_j$ for $1 \leq j < i$, and $W_i \neq Y_i$. Since our previous algorithm, Algorithm ExpansionTree, allowed nodes with either $W_i \neq Y_i$ or $W_i = Y_i$, Algorithm NonredundantExpansionTree generates at most half the vertices that ExpansionTree does. This will often be even smaller than half, because the starting vertex is now $\left\langle \vec{X}, \min\left(\{k : X_k \neq Y_k\}, n+1\right)\right\rangle$.

All the edges now correspond to only one term and so all edges have the same weight. This will simplify the depth-first search algorithm for a minimal expansion in the expansion search tree.

Algorithm NonredundantExpansionTree can be used in place of Algorithm ExpansionTree in computing minimal routing paths on a nonredundant *LTDM* network. The only change to the routing algorithm will be the generation of the search tree. Though Algorithm NonredundantExpansionTree has the same asymptotic run

Input: A n-dimensional *LTDM* network $G = (B^0, B^1)$, a starting address $\vec{X}$ and a destination address $\vec{Y}$.

Output: A directed acyclic graph $D = (V, E)$, where each path from the vertex $\left\langle \vec{X}, \min\left(\{k : X_k \neq Y_k\}, n+1\right)\right\rangle$ to $\left\langle \vec{Y}, n+1\right\rangle$ corresponds to a nonredundant expansion of $\vec{X} + \vec{Y}$.

Procedure NonredundantExpansionTree(G, $\vec{X}$, $\vec{Y}$)
    $V \leftarrow \left\{\left\langle \vec{X}, \min\left(\{k : X_k \neq Y_k\}, n+1\right)\right\rangle\right\}$
    $E \leftarrow \{\}$
    for each $i : 1 \leq i \leq n$ do
        for each $\left\langle \vec{W}, i\right\rangle \in V$ do
            $j \leftarrow \min\left(\left\{k : (\vec{W} + B_i^0)_k \neq \vec{Y}_k\right\}, n+1\right)$
            $V \leftarrow V \cup \left\{\left\langle \vec{W} + B_i^0, j\right\rangle\right\}$
            $E \leftarrow E \cup \left\{\left(\left\langle \vec{W}, i\right\rangle, \left\langle \vec{W} + B_i^0, j\right\rangle\right)\right\}$
            $j \leftarrow \min\left(\left\{k : (\vec{W} + B_i^1)_k \neq \vec{Y}_k\right\}, n+1\right)$
            $V \leftarrow V \cup \left\{\left\langle \vec{W} + B_i^1, j\right\rangle\right\}$
            $E \leftarrow E \cup \left\{\left(\left\langle \vec{W}, i\right\rangle, \left\langle \vec{W} + B_i^1, j\right\rangle\right)\right\}$
        end for
    end for
    return $D \leftarrow (V, E)$
end procedure

Figure 5.4. Algorithm NonredundantExpansionTree.

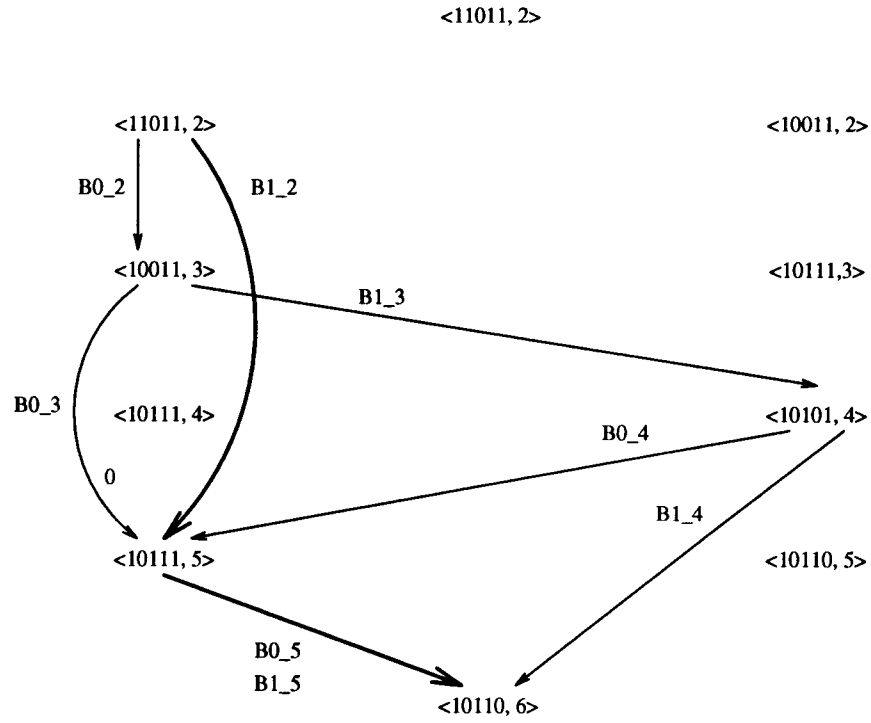

Figure 5.5. A nonredundant expansion search tree between addresses $X = (11011)$ and $Y = (10110)$ on a 5-dimensional Folded Hypercube. We include the vertices of the redundant expansion for comparison.

time order (in the worst case) as Algorithm ExpansionTree, it can be used to reduce the number of bit operations by half or often more.

Both the Folded Hypercube and the Enhanced Hypercube can be represented using a nonredundant *DM* network description, and so can use Algorithm NonredundantExpansionTree. Figure 5.5 shows a nonredundant expansion search tree for the Folded Hypercube cube of dimension 5.

5.7. SUMMARY

The problems of minimal point-to-point routing on *DM* networks and on *LTDM* networks are both *NP*-complete. This strongly suggests that a polynomial run time minimal routing algorithm does not exist.

For *LTDM* networks, we were able to devise routing algorithms that had a polynomial run time in a large number of common cases. These algorithms have total run time of at worst $O(kn^2)$ bit operations, compared to the hypercube's total run time of $O(n)$ bit operations. While polynomial, our minimal algorithms are not highly efficient, because they require a large amount of pre-computation at the source node.

We were able to show that for a subclass of *LTDM* networks – those with nonredundant minimal expansions – we could simplify the algorithm and reduce the size of the search tree by half. This still required a large amount of pre-computation at the source node, though reduced the run time by a constant factor.

The results in this chapter will be used in the next chapter, where we consider minimal routing for the *LE* networks.

6. ROUTING ALGORITHMS FOR *LE* NETWORKS

In the preceding chapter, we showed that a minimal expansion of $\vec{X} + \vec{Y}$ can be used to compute a minimal routing path from $\vec{X}$ to $\vec{Y}$ on a *LTDM* network $G = (B^0, B^1)$. In this chapter, we expand those results to include *LTLE* networks.

In general, the computation of a minimal expansion not only allows us to compute a minimal path between $\vec{X}$ and $\vec{Y}$ for a *DM* network $G = (B^0, B^1)$, but it also allows us to find a lower bound on the length of a path on a *LE* network $\check{G} = (B^0, B^1, A)$, as the theorem below shows:

Theorem 6.0.1 *Let $\check{G} = (B^0, B^1, A)$ be a connected n-dimensional* LE *network. Further, let $\vec{X}$ be a source address and $\vec{Y}$ be a destination address. Then the weight of a minimal expansion $S(\vec{X} + \vec{Y})$ is a lower bound on length of the path from $\vec{X}$ to $\vec{Y}$ on $\check{G}$.*

Proof: Because the *DM* network $G = (B^0, B^1)$ contains the *LE* network $\check{G} = (B^0, B^1, A)$ as a sub-network – with the same number of nodes but fewer channels, the path from $\vec{X}$ to $\vec{Y}$ on $\check{G}$ will be as long as or longer than the path from $\vec{X}$ to $\vec{Y}$ on G. ∎

Unfortunately, we may not be able to directly apply a minimal expansion $S(\vec{X} + \vec{Y})$ as a routing path between $\vec{X}$ and $\vec{Y}$ on a *LE* network $G = (B^0, B^1, A)$. The expansion S may not define a path if the terms of the expansion are applied in the wrong order, or the expansion may define no legal path at all.

In this chapter, we consider the complexity of routing on a *LE* network. We show that the general problem of routing on a *LE* network is at least *NP*-hard, and for restricted cases is *NP*-complete. We also use a modified version of minimal

expansions to apply the *LTDM* routing algorithm in the previous chapter to *LTLE* networks.

6.1. THE COMPLEXITY OF ROUTING IN *LE* NETWORKS

In this section, we consider the complexity of minimally routing on a general *LE* network. We will show that the general problem is *NP*-hard, and that a restricted version of the problem is *NP*-complete. We first define the problem of minimal routing:

Definition 6.1.1 ***MR-LEN: Minimal Routing over* LE *Networks:***

Instance: *An n-dimensional* LE *network* $G = (B^0, B^1, A)$, *source and destination addresses* $\vec{X}, \vec{Y} \in \mathcal{Z}_2^n$ *and an integer bound* $K \geq 1$.

Question: *Is there a path R from* $\vec{X}$ *to* $\vec{Y}$ *in G, where* $|R| \leq K$?

As in Chapter 2, we are unable to put a polynomial upper bound on the diameter of a *LE* network. Our inability to show that the maximum length of all minimal paths is polynomially bounded makes it impossible for us to show that *MR-LEN* $\in$ *NP*. We can avoid this problem by arbitrarily limiting the number of times each term can appear as a step in the routing path:

Definition 6.1.2 ***LMR-LEN: Limited Minimal Routing Over* LE *Networks:***

Instance: *An n-dimensional* LE *network* $G = (B^0, B^1, A)$, *source and destination addresses* $\vec{X}, \vec{Y} \in \mathcal{Z}_2^n$, *and an integer bound* K *with* $1 \leq K \leq 2n$.

Question: *Is there a path R from* $\vec{X}$ *to* $\vec{Y}$ *on G, where each term from* B^0 *and* B^1 *defines at most one channel in R and* $|R| \leq K$?

We show that LMR-LEN is *NP*-complete by showing that there is a polynomial-time transformation from an instance of the dominating vertex set problem. This problem was shown to be *NP*-complete in [33]:

Definition 6.1.3 ***DVS: Dominating Vertex Set:***

Instance: *A directed graph* $G = (V, E)$, *and an integer* B *with* $1 \leq B \leq |V|$.

Question: *Is there a set* $U \subset V$, $|U| \leq B$ *so that* $\forall v \in \{V - U\} : \exists u \in U :$ $(u, v) \in E$*?*

The original problem statement states that G is an undirected graph. But the case where G is a directed graph is trivially proved to be *NP*-complete by taking an instance of G and replacing each undirected edge $(\vec{X}, \vec{Y})$ with two directed edges $(\vec{X}, \vec{Y})$ and $(\vec{Y}, \vec{X})$. This transformation of an instance of *DVS* on undirected graphs into an instance of *DVS* on directed graphs takes $O(|E|)$ operations and the resulting directed graph clearly has a dominating vertex set of size B or less iff the undirected graph does.

We can use this result to show that there is probably no polynomial-time algorithm for limited minimal routing:

Theorem 6.1.1 LMR-LEN *is* NP*-Complete.*

Proof: Proof is by transformation from *DVS*.

LMR-LEN $\in$ NP: Guess a routing path R from $\vec{X}$ to $\vec{Y}$. Verify first that each adjacent pair of nodes in the routing path are joined by channels in the network. Second, verify that the starting node of R has address $\vec{X}$ and the ending node of R has address $\vec{Y}$ and $|R| \leq K$. There are at most $2n$ routing steps, and the verification steps on each take $O(n^2)$ bit operations (a matrix-vector multiply, a vector addition and a vector comparison), so *LMR-LEN* $\in$ *NP*.

DVS $\leq$ LMR-LEN: Let the directed graph $G = (V, E)$ and the bound $B \leq |V|$ be an instance of *DVS*.

Also let B^0, B^1 and A to be $2|V| \times 2|V|$ matrices. Then define B^0 as:

$$B^0_{2i,2j} = B^0_{2i-1,2j-1} = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}$$

$$B^0_{2i-1,2j} = B^0_{2i,2j-1} = 0$$

For $1 \leq i \leq |V|$ and $1 \leq j \leq |V|$.

Define B^1 as:

$$B^1_{2i,2j} = B^1_{2i,2j-1} = B^1_{2i-1,2j-1} = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}$$

$$B^1_{2i-1,2j} = 0$$

For $1 \leq i \leq |V|$ and $1 \leq j \leq |V|$.

Define A as:

$$A_{2i,2j-1} = A_{2i-1,2j-1} = \begin{cases} 1, & (v_i, v_j) \in E \\ 0, & (v_i, v_j) \notin E \end{cases}$$

$$A_{2i,2j} = A_{2i-1,2j} = 0$$

For $1 \leq i \leq 2|V|$ and $1 \leq j \leq 2|V|$. Let $\vec{X} = (1111\ldots11)$ and let $\vec{Y} = (0000\ldots00)$, and let $K = L + |V| \leq 2|V|$.

The transformation is complete. Now we show that a solution for *LMR-LEN* exists iff a solution for *DVS* exists.

Note that for i and j, B^0_{2i-1} and B^1_{2i-1} both depend on B^0_{2j-1} and B^0_{2j} iff $(v_i, v_j) \in E$, because $(AB^0_{2j-1})_{2i-1} = (AB^0_{2j})_{2i-1} = 1$. But B^0_{2i-1} and B^1_{2i-1} never depend on B^1_{2j-1}, because $(AB^1_{2j-1})_{2i-1} = 0$, always.

Initially each B^1_{2i-1} does not define a channel from either $\vec{X}$ or $\vec{Y}$, and so cannot be used as a routing step. Let $(AB^0_{2j-1})_{2i-1} = (AB^0_{2j})_{2i-1} = 1$. Then B^1_{2i-1}

can be used as a routing step iff either B^0_{2j-1} or B^0_{2j} is first used as a routing step. This corresponds to the idea that a vertex v_i can be a dominated vertex iff a dominating vertex v_j is adjacent to it. Let $(AB^0_{2j-1})_{2i-1} = (AB^0_{2j})_{2i-1} = 0$. Then B^1_{2i-1} cannot be used as a routing step after either B^0_{2j-1} or B^0_{2j} are used. This corresponds to the idea that a vertex v_i can never be dominated by vertex v_j if v_j is not adjacent to v_i.

If there is a dominating vertex set over G of size $\leq K$ vertices, then there is a path from $\vec{X}$ to $\vec{Y}$ of length less than $|V| + K$. This can be constructed by Algorithm ConstructTransformPath in Figure 6.1.

This algorithm takes $K = |V| + B$ steps. This is two steps for each vertex in U and one step for each vertex not in U. We mark each vertex not in U as it is used in a routing step, so that each vertex is never considered more than once.

If there is no dominating vertex set over G of size $\leq K$ vertices, then there is no way to construct a path. No matter which set of vertices we choose for U, there will be at least one vertex v_i that is not dominated by any vertex in U. Hence B^0_{2i-1} will not define any channel $(\vec{W}, \vec{W} + B^0_{2i-1})$ on the path R and 2 steps instead of 1 will be required to correct indices $2i - 1$ and $2i$. ∎

Example: We apply the transformation to the instance of *DVS* to graph G defined by the adjacency matrix below, and the bound $L = 2$:

$$G = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

We define the matrices B^0, B^1 and A as:

Input: A dominating vertex set U for a graph $G = (V, E)$.

Output: A path R from $\vec{X} = (1111\ldots11)$ to $\vec{Y} = (0000\ldots00)$ on the network $G = (B^0, B^1, A)$ constructed by the transformation in Theorem 6.1.1.

```
Algorithm ConstructTransformPath( U )
    W⃗ ← X⃗
    R ← {}
    for i = 1 to |V|
        mark[i] ← 0
    end for
    for each j : v_j ∈ U do
        W ← W⃗ + B^0_{2j-1}
        R ← R + {2j − 1}
        for each i : v_i ∈ {V − U} ∧ mark[i] = 0 do
            W ← W⃗ + B^1_{2i-1}
            R ← R + {2j − 1}
            mark[i] ← 1
        end for
        W ← W⃗ + B^0_{2j}
        R ← R + {2j}
    end for
    return R
end procedure
```

Figure 6.1. Algorithm ConstructTransformPath.

$$B^0 = \begin{bmatrix} 1&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0 \\ 0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&1 \end{bmatrix} \quad B^1 = \begin{bmatrix} 1&0&0&0&0&0&0&0 \\ 1&1&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&1&1&0&0&0&0 \\ 0&0&0&0&1&0&0&0 \\ 0&0&0&0&1&1&0&0 \\ 0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&1&1 \end{bmatrix} \quad A = \begin{bmatrix} 0&0&1&1&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 1&1&0&0&1&1&1&1 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&1&1&1&1 \\ 0&0&0&0&0&0&0&0 \\ 1&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \end{bmatrix}$$

We also define $\vec{X} = (11111111)$, $\vec{Y} = (00000000)$ and $K = L + n = 6$.

There is a dominating set of G of two vertices (in this case, it is $\{1, 3\}$). There is at least one routing path of length 6 from $\vec{X}$ to $\vec{Y}$, which is $R = (1, 3, 7, 2, 5, 6)$. This path works because:

$$\begin{aligned} (11111111) &\rightarrow (01111111) \\ &\rightarrow (01001111) \\ &\rightarrow (01001100) \\ &\rightarrow (00001100) \\ &\rightarrow (00000100) \\ &\rightarrow (00000000) \end{aligned}$$

The answer to both problem instances is "Yes".

If the bound is changed to $L = 1$, then there is no dominating set of size 1 for G, and no path of length 5 between $\vec{X}$ to $\vec{Y}$. The answer to both problem instances is now "No". ∎

From the above proof of *NP*-completeness, we can conclude that a number of special cases in *LMR-LEN* are also *NP*-complete. This is because the transformation above results in an instance of these special cases.

Corollary 6.1.1 *Minimal routing on networks where for all* $1 \leq i \leq n$, $B_i^0 + B_i^1$ *is limited to the set* $\{0, B_{i+1}^0, \ldots, B_{i+k}^0, B_{i+1}^1, \ldots, B_{i+k}^1, \}$ *for* $1 \leq k \leq n$ *is* NP*-complete.*

Corollary 6.1.2 *Minimal routing on networks where* B^0 *and* B^1 *are lower triangular is* NP*-Complete.*

We notice in the corollary above that B^0 and B^1 are lower triangular matrices, but that A is not necessarily lower triangular. If we restrict the problem instances to only *LTLE* networks, *LMR-LEN* is still *MP*-complete. The proof of this stems from the fact that *DVS* remains *NP*-complete when restricted to directed acyclic graphs (DAGs):

Definition 6.1.4 *Dominating Vertex Set on Directed Acyclic Graphs (*DVS-DAG*):*

Instance: *A directed acyclic graph* $G = (V, D)$, *and an integer* B *with* $1 \leq B \leq |V|$.

Question: *Is there a set* $U \subset V$ *so that* $|U| \leq B$ *and for every* $v \notin U$, *there is at least one* $u \in U$ *so that* $(u, v) \in D$*?*

This restricted version remains *NP*-complete:

Theorem 6.1.2 DVS-DAG *is* NP*-complete.*

Proof: DVS-DAG $\in$ NP: Guess U and verify that $|U| \leq B$. Then, for each $v \in V$, verify that either $v \in U$, or that there is a $u \in U$ with $(u, v) \in D$. This verification takes at most $O(|U||D|) < O(|V||D|)$ steps for each v, and so takes at

most $O(|V|^2|D|)$ steps total to check – clearly polynomial time in the size of the input.

DVS-DAG $\leq$ DVS: Assume that we have an instance of *DVS* with a graph $\check{G} = (V, E)$ and a bound $1 \leq B \leq |\check{V}|$. We will use a polynomial-time transformation to create a new directed acyclic graph $G = (\check{V}, \check{D})$ and a bound $\check{B}$.

For each $v_i \in V$, add two vertices $\check{v}_{i,1}$ and $\check{v}_{i,2}$ to $\check{V}$. Add one new vertex $\check{v}_0$ to $\check{V}$. Have $\check{D}$ initially contain the directed edges $(\check{v}_0, \check{v}_{1,1})$, $(\check{v}_0, \check{v}_{2,1})$, ..., $(\check{v}_0, \check{v}_{|V|,1})$, and the directed edges $(\check{v}_{1,1}, \check{v}_{1,2})$, $(\check{v}_{2,1}, \check{v}_{2,2})$, ..., $(\check{v}_{|V|,1}, \check{v}_{|V|,2})$. Then, for each $(v_i, v_j) \in E$, add the edge $(\check{v}_{i,1}, \check{v}_{j,2})$ to $\check{D}$. Finally, set the bound $\check{B} = B + 1$.

The transformation is complete. This transformation can be computed in polynomial time, because $|\check{V}| = 2|V| + 1$ and $|\check{D}| = 2|V| + 2|E|$.

Now to show that *DV* has a solution iff *DV-DAG* does.

It is clear that any solution to *DVS* gives a solution to *DV-DAG*. If $U = \{ u_{i_1}, u_{i_2}, \ldots, u_{i_k} \}$, $k \leq B$ is a solution to *DVS*, then we can use it to construct $\check{U} = \{ \check{v}_0, \check{v}_{i_1,1}, \check{v}_{i_2,1}, \ldots, \check{v}_{i_{k+1},1} \}$. The bound is then $\check{B} = B+1$. The construction works for three reasons: first, $\check{v}_0$ dominates all $\check{v}_{i,1}$; second, $u_{i_r} \in U$ with $1 \leq r \leq k$ implies that $\check{u}_{i_r,1}$ dominates $\check{u}_{i_r,2}$; and third, u_{i_r} with $1 \leq r \leq k$ dominates v_s with $1 \leq s \leq k$ implies that $\check{u}_{i_r,1}$ dominates $\check{v}_{s,2}$. So if U dominates all V, then $\check{U}$ dominates all $\check{V}$.

It also is clear that $\check{v}_0$ must be in any solution to *DV-DAG* because it has in-degree 0. If $\check{U} = \{\check{v}_0, \check{u}_{i_1,j_1}, \check{u}_{i_2,j_2}, \ldots, \check{u}_{i_k,j_l}\}, k \leq B$ is a solution to *DVS-DAG*, then we can use it to construct $U = \{u_{i_1}, u_{i_2}, \ldots, u_{i_k}\}$. The bound is $B = \check{B} - 1$. This is because $\check{u}_{i_1,1}$ dominates $\check{v}_{i_2,2}$ implies that u_{i_1} dominates v_{i_2}, and because $\check{u}_{i_1,2} \in \check{U}$ implies that u_{i_1} is not dominated by any other vertex and so must be included in U. ∎

Example: The graph in Figure 6.2(a) is a general, undirected graph. We can replace each undirected edge with two directed edges and create the directed

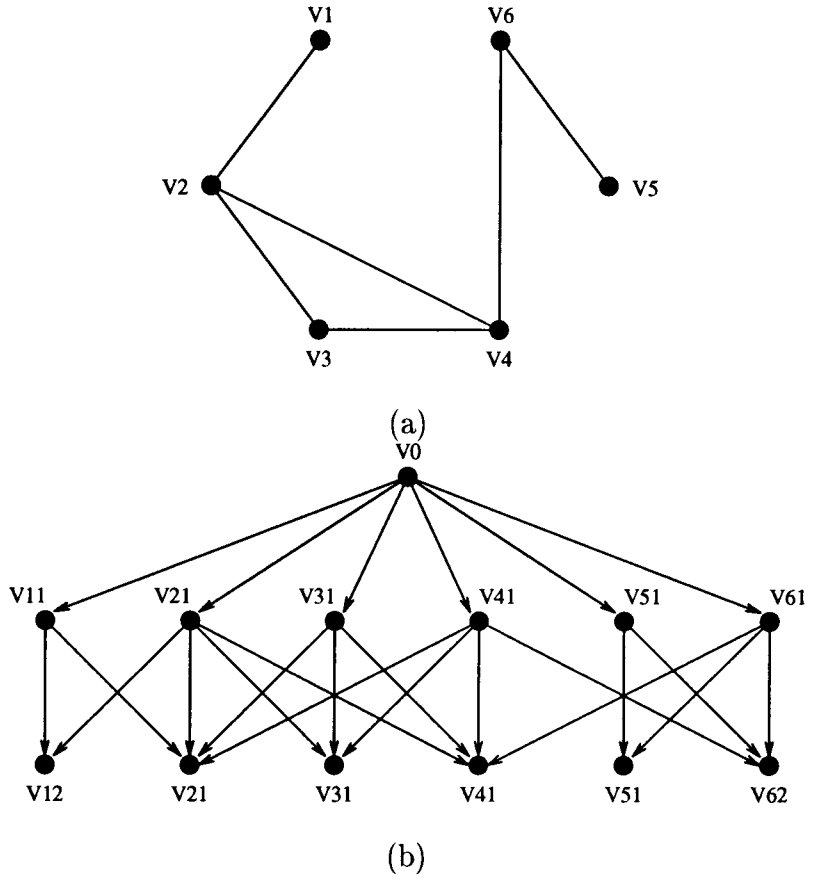

Figure 6.2. The transformation of an instance of *DV* to *DV-DAG*.

graph in Figure 6.2(b), using the transformation in Theorem 6.1.2. The first graph has the dominating vertex set $U = \{v_2, v_6\}$ and the second has a dominating vertex set $\check{U} = \{v_0, v_{2,1}, v_{6,1}\}$. However, neither has a smaller dominating vertex set. ∎

Any DAG can have its vertices topologically sorted in polynomial time, as mentioned in Chapter 4, which makes the graph's adjacency matrix lower triangular. Let G be an instance of *DVS-DAG*. If we topologically sort G and then use the transformation in Theorem 6.1.1 to create a *LE* network $\check{G} = (B^0, B^1, A)$, it will create a lower triangular A matrix and so $\check{G}$ is a *LTLE* network. Hence, we have:

Lemma 6.1.1 *Minimal routing on* LTLE *networks is* NP*-complete.*

Though most of the variations on routing problems for *LE* networks are intractable, this will not prevent us from writing an algorithm that is efficient in many cases.

6.2. LEGAL EXPANSIONS

Like the *DM* networks, the *LE* networks can use a minimal expansion $S(\vec{X} + \vec{Y})$ over B^0 and B^1 to find a minimal path between $\vec{X}$ and $\vec{Y}$. All such paths must correspond to some expansion, so we only have to find a minimal expansion that can be used.

Unfortunately, there are two problems to using expansions to find a routing path on a *LE* network $G = (B^0, B^1, A)$. The first is that for a given expansion S, a term in S may not define a channel from a particular node in G. We may have to use the terms of S in a completely or partially specified order, so that each term used defines a channel from an intermediate node in the routing path.

Second, a given expansion S may not correspond to *any* routing path in a network G. If G is connected, then there must be some expansion whose terms

define the channels in a routing path, but there are possibly some expansions have terms that do not define channels in any routing path.

For this reason, we have to distinguish between a "legal" expansion, for which a corresponding routing path exists in G, and an "illegal" expansion, for which no routing path exists:

Definition 6.2.1 *For a network $G = (B^0, B^1, A)$ and vectors $\vec{X}$ and $\vec{Y}$, an expansion $S(\vec{X} + \vec{Y})$ is a **legal expansion** iff:*

1. *S is an expansion.*

2. *There is an permutation π of the terms of S, so that for every $B_{i_{\pi(k)}}^{\phi_{\pi(k)}} \in S$ with $1 \leq k \leq |S|$, we have:*

$$\left[A\left(\vec{X} + B_{i_{\pi(1)}}^{\phi_{\pi(1)}} + \ldots B_{i_{\pi(k-1)}}^{\phi_{\pi(k-1)}}\right)\right]_{i_{\pi(k)}} = \phi_{\pi(k)}$$

This definition describes the existence of some permutation of terms in $S(\vec{X} + \vec{Y})$ so that each term $t_{\pi(k)} \in S$ defines a channel $(\vec{W}, \vec{W} + t_{\pi(k)})$ at step k of the routing path. We can again use an expansion $S(\vec{X} + \vec{Y})$ to find a routing path on $G = (B^0, B^1, A)$. The only difference is that now S must be a legal (and ordered) expansion over B^0, B^1 and A instead of an expansion over B^0 and B^1.

If the legal expansion between a pair of nodes is minimal, then clearly a routing path corresponding to that legal expansion is also minimal. Further, if we can quickly find the smallest (in weight) legal expansion that corresponds to a routing path, then we have an efficient, minimal routing algorithm.

The task of finding a minimal legal expansion requires finding an expansion that meets two possibly conflicting criteria. First, the expansion must meet the the conditions in Definition 6.2.1 above. Second, it must be the smallest such expansion meeting those criteria.

We define the problem of finding a legal minimal expansion below:

Definition 6.2.2 ***LME-LEN: Legal Minimal Expansions in* LE *Networks Problem:***

Instance: *An n-dimensional* LE *network $G = (B^0, B^1, A)$, source and destination addresses $\vec{X}, \vec{Y} \in \mathcal{Z}_2^n$, and an integer K with $1 \leq K \leq 2n$.*

Question: *Is there a legal expansion $S(\vec{X} + \vec{Y})$ in $\{B^0, B^1\}$ with $|S| \leq K$?*

Unfortunately, the problem *LME-LEN* is intractable:

Theorem 6.2.1 LME-LEN *is* NP*-hard for general* LE *networks, and is* NP*-complete for* LTLE *networks.*

Proof: We use a polynomial-time transformation of an instance of *MIN-COSETS* to *LME-LEN*, similar to the transformation of *MIN-COSETS* to *ME-DMN* Theorem 5.3.2. The only difference is that we also set A to $\vec{0}$.

LME-LEN remains *NP*-hard for *LE* networks, because we cannot bound the maximum path length polynomially in n. *LME-LEN* is *NP*-complete for *LTLE* networks, because the maximum path length is $O(n)$ steps, by Corollary 3.3.2. We can guess an expansion and a permutation of it and verify it against the definition in a polynomial time. ∎

For *LTLE* networks, we have a way to verify that an expansion is legal:

Theorem 6.2.2 *Let $G = (B^0, B^1, A)$ be an n-dimensional* LTLE *network. An expansion $S(\vec{X} + \vec{Y})$ corresponds to a routing path from $\vec{X}$ to $\vec{Y}$ on G iff for every $B_i^\phi \in S$, one of the following conditions are true:*

1. $(A\vec{Y})_i = \phi$
2. $(A\vec{Y})_i = \overline{\phi}$ *and* $\exists j < i, B_j^\rho \in S : (AB_j^\rho)_i = 1$

Input: A n-dimensional network $G = (B^0, B^1, A)$, a starting address $\vec{X}$, an ending address $\vec{Y}$, a legal minimal expansion $S(\vec{X} + \vec{Y})$ with the terms ordered by index from smallest to largest.

Output: An ordered list R of the terms in S so that each term i correspond to a legal channel in the i-th step of the routing path from $\vec{X}$ to $\vec{Y}$. If no such path exists, then the algorithm returns "Failure".

```
Procedure ConstructRoutePath( G, X⃗, Y⃗, S )
    R ← {}
    for k = 1 to |S| do
        B_i^φ ← S_k
        if (AY⃗)_i = φ then
            insert B_i^φ into R after R_|R|
        else if (AY⃗)_i = φ̄ then
            j = max({m : (AR_m)_i = 1}, 0)
            if j > 0 then
                insert B_i^φ into R before R_j
            else
                return "Failure"
            end if
        end if
    end for
    return R
end procedure
```

Figure 6.3. Algorithm ConstructRoutePath

Proof: Sufficient: Assume that conditions 1 and 2 are true for an expansion S, and let a routing path R be represented as an ordered list of terms. Algorithm ConstructRoutePath in Figure 6.3 will correctly compute R from S, if the terms in S are listed in order of increasing index.

In the loop with index variable k, we have a loop invariant for the beginning of each repetition. R is a routing path from $\vec{X} + S_k \in S + \ldots + S_{|S|} \in S$ to $\vec{Y}$. For the first iteration $k = 1$, this is trivially true.

Let B_i^ϕ be the k-th element of S. If $(A\vec{Y})_i = \phi$ then we append B_i^ϕ after $R_{|R|} \in R$, because the channel $(\vec{Y} + B_i^\phi, \vec{Y}) \in G$.

If $(A\vec{Y})_i = \overline{\phi}$ then insert B_i^ϕ into R immediately before the last $R_j \in R$ with $j < i$ and $(AR_j)_i = 1$. This $R_j \in R$ exists because condition 2 guarantees that such a term initially exists in S and the algorithm has already inserted all terms with index $j < i$ into R. For the node $(R_j + \ldots + R_{|R|} + \vec{Y})$ we have:

$$\begin{aligned}[A(R_j + \ldots + R_{|R|} + \vec{Y})]_i &= [(AR_j)_i + \ldots + (AR_{|R|})_i + (A\vec{Y})_i \\ &= 1 + 0 + \ldots + 0 + \overline{\phi} \\ &= \phi\end{aligned}$$

And so the channel $(R_j + \ldots + R_{|R|} + \vec{Y}, \vec{Y}) \in G$.

Because G is a *LTLE* network, G guarantees that $(AB_i^\phi)_j = 0$ for $j \leq i$. Inserting B_i^ϕ into the prescribed position in R will not affect the other terms already in R. This means that after the insertion, each term in R still defines a channel.

So, at then end of iteration k of the loop, R is a routing path from $\vec{X} + S_{k+1} \in S + \ldots + S_{|S|} \in S$ to $\vec{Y}$. The end of the final iteration will then trivially have R contain a routing path from $\vec{X}$ to $\vec{Y}$.

Necessary: Assume that $(A\vec{Y})_i = \overline{\phi}$ and that $\forall j, 1 \leq j < i, B_j^\rho \in S : (AB_j^\rho)_i = 0$. Then because G is lower triangular, $\forall j, 1 \leq j \leq n, B_j^\rho \in S : (AB_j^\rho)_i = 0$.

Let $\vec{W}$ be any linear combination of $B_j^\rho \in S$ added to $\vec{Y}$. Then:

$$
\begin{aligned}
(A\vec{W})_i &= \left[A(\vec{Y} + \sum_{\alpha_j \in \{0,1\}, B_j^\rho \in S} \alpha_j B_j^\rho) \right]_i \\
&= \left[A\vec{Y} + \sum_{\alpha_j \in \{0,1\}, B_j^\rho \in S} \alpha_j A B_j^\rho \right]_i \\
&= \left[A\vec{Y} \right]_i + \sum_{\alpha_j \in \{0,1\}, B_j^\rho \in S} \alpha_j \left[A B_j^\rho \right]_i \\
&= \overline{\phi} + \sum_{\alpha_j \in \{0,1\}, B_j^\rho \in S} \alpha_j 0 \\
&= \overline{\phi}
\end{aligned}
$$

Any such node $\vec{W}$ that can reach $\vec{Y}$ will have $(A\vec{W})_i = \overline{\phi}$) and so will have no legal channel $(\vec{W} + B_i^\phi, \vec{W})$. Therefore, the expansion S corresponds to no path from $\vec{X}$ to $\vec{Y}$. ∎

There are two corollaries that follow from this theorem:

Corollary 6.2.1 *Let $G = (B^0, B^1, A)$ be an n-dimensional* LTLE *network. The time to compute if an expansion $S(\vec{X} + \vec{Y})$ corresponds to some routing path from $\vec{X}$ to $\vec{Y}$ on G is at worst $O(n^2)$.*

Proof: This can be computed directly from the conditions in Theorem 6.2.2. For each $B_i^\phi \in S$ the product (AB_i^ϕ) can be precomputed. For a given $\vec{Y}$, each $(A\vec{Y})_i$ can be computed in $O(n)$ bit operations for a total of $O(n^2)$ bit operations. Then for each term of S, verifying Condition 1 takes $O(1)$ bit operations, and verifying Condition 2 takes at worst $|S| = O(n)$ bit operations. There are $|S| = O(n)$ terms so the run time is at worst $O(n^2)$. ∎

Corollary 6.2.2 *Let $G = (B^0, B^1, A)$ be an n-dimensional* LTLE *network. The time to compute a routing path R from $\vec{X}$ to $\vec{Y}$ on G using a legal expansion $S(\vec{X}+\vec{Y})$ is at worst $O(n^2 \log n)$.*

Proof: This can be computed directly from Algorithm ConstructRoutePath. For each $B_i^{\phi} \in S$ the product (AB_i^{ϕ}) can be precomputed. For a given $\vec{Y}$, each $(A\vec{Y})_i$ can be computed in $O(n)$ bit operations for a total of $O(n^2)$ bit operations.

There are $|S| = O(n)$ terms to insert into R. If we use a linked list, at most $O(\log n)$ bits will be needed to represent a pointer. Then inserting into the list can take at most $O(n \log n)$ bit operations for each term, because comparisons at each element of the list will take $O(1)$ bit operations. Therefore, the run time of the algorithm is $O(n^2 \log n)$ bit operations. ∎

Although Algorithm ConstructRoutePath gives only one path, a number of paths can be created that meet this condition. The algorithm needs only to insert the current term B_i^{ϕ} before an even number of terms where $(AR_j)_i = 1$ if $(A\vec{Y})_i = \phi$ (or before an odd number of terms with $(AR_j)_i = 1$ if $(A\vec{Y})_i = \overline{\phi}$). This will result in an equally correct routing path, but one in which the terms may be used in a different order.

We give an outline of the algorithm we will call LowerTriangularRoute. Given a *LTLE* network G, and nodes $\vec{X}$ and $\vec{Y}$, we can compute a routing path using the following steps:

1. Generate an legal expansion search tree D for the addresses $\vec{X}$ and $\vec{Y}$.
2. Use breadth-first search in D to find a minimal legal expansion $S(\vec{X} + \vec{Y})$.
3. Order the terms of S into a routing path R.
4. Use R to route the message from $\vec{X}$ to $\vec{Y}$.

We have an algorithm to compute all but the first step. The next section will present an algorithm to compute this.

6.3. A MINIMUM LEGAL EXPANSION ALGORITHM FOR *LTLE* NETWORKS

Algorithm ExpansionTree in Figure 5.1 will correctly generate expansions, but will have both legal and illegal expansions represented in the tree. To create an expansion tree that contains only legal minimal expansions, we now need some way to "prune" paths from the expansion tree so that edges corresponding to terms in an illegal expansion are not included.

There is a problem with simply pruning edges from the expansion tree. Terms that appear in legal expansions may also appear in illegal expansions, so simply an edge from the expansion tree may remove perfectly legal expansions from the tree. Instead, we may have to change the search tree to include a branch for the case where a term can be used in a legal expansion, and another branch for the case where the term cannot be used.

For a *LTLE* network $G = (B^0, B^1, A)$, the terms B_i^0 and B_i^1 can depend on another term B_j^ρ only if $j < i$ and $(AB_j^\rho)_i = 1$. Say that B_i^0 depends on B_j^ρ. By placing B_i^0 either before or after B_j^ρ in an expansion, we can guarantee that B_i^0 defines an channel in the routing path. Thus B_i^0 can be a part of a legal expansion if B_j^ρ is also part of the expansion. The same is also true for B_i^1.

The legal expansion tree algorithm generates a search tree of expansions, starting with edges that correspond the terms B_1^0 and B_1^1. Any term with index i is examined as part of the expansion only after terms with indices $1, \ldots, i-1$ have been examined. Because G is lower triangular, we can tell if terms B_i^0 and B_i^1 can be used in a legal expansion just by back-tracing up the search tree for any B_j^ρ with $j < i$ and $(AB_j^\rho)_i = 1$.

This back-tracing up the search tree could be done explicitly, but would increase the run time order of the search tree algorithm. Instead, we can can place

additional information in the label of each vertex in the tree. This additional information is an n-bit mask. If, for a vertex at level i, this mask has a 1 in index i, then both B_i^0 and B_i^1 are usable terms in a legal expansion and edges for both terms can be added to the expansion search tree at that vertex. If the mask has a 0 in index i, then only the one term $B_i^{(A\vec{X})_i}$ is usable, and only the edge for that one term can be added.

The terms that depend on B_j^ρ can be computed quickly by the product AB_j^ρ. If $(AB_j^\rho)_i = 1$, then B_i^0 and B_i^1 depend on B_j^ρ. Otherwise, they do not. In constructing new vertices, we can compute the new vertex's mask quickly by taking the union of the current vertex's mask and AB_j^ρ.

Algorithm LegalExpansionTree is listed in Figure 6.4. This algorithm is a variation of Algorithm ExpansionTree. A legal expansion tree for $\vec{X} = (1000000)$ and $\vec{Y} = (1101010)$ on the 7-dimensional Möbius Cube is shown in Figure 6.6. Algorithm LegalExpansionTree is very similar to Algorithm ExpansionTree and can be proved using similar arguments.

The major difference between Algorithm LegalExpansionTree and Algorithm ExpansionTree is that Algorithm LegalExpansionTree labels each vertex with three components instead of two. To simplify the algorithm, and reduce the number of vertices, any vertex with a given i has $M_j = 1$ for $1 \leq j \leq i-1$. This will also guarantee that there is exactly one vertex with address $\vec{Y}$ in its label – the one with $\vec{M} = \vec{1}$ and $i = n+1$.

Theorem 6.3.1 *Algorithm LegalExpansionTree(G, $\vec{X}$, $\vec{Y}$) is correct.*

Proof: The algorithm can be shown to be correct by a proof very similar to Theorem 5.4.1, and will not be duplicated here. This algorithm will generate only paths that correspond legal minimal expansions, because for each vertex, we now

Input: A n-dimensional network $G = (B^0, B^1, A)$, a starting address $\vec{X}$ and a destination address $\vec{Y}$.

Output: A weighted directed acyclic graph $D = (V, E)$, where each path from the vertex $\langle \vec{X}, \vec{0}, 1 \rangle$ to $\langle \vec{Y}, \vec{1}, n+1 \rangle$ corresponds to a legal expansion of $\vec{X} + \vec{Y}$.

Procedure LegalExpansionTree(G, $\vec{X}$, $\vec{Y}$)
$V \leftarrow \{\langle \vec{X}, \vec{0}, 1 \rangle\}$
$E \leftarrow \{\}$
for $i = 1$ to n do
for each $\langle \vec{W}, \vec{M}, i \rangle \in V$ do
if $W_i = Y_i$ then
if $M_i = 1$ or $(A\vec{W})_i = 0$ then
$V \leftarrow V \cup \{\langle \vec{W}, \vec{M} \vee AB_i^0 \vee e_i, i+1 \rangle\}$
$E \leftarrow E \cup \{(\langle \vec{W}, \vec{M}, i \rangle, \langle \vec{W}, \vec{M} \vee AB_i^0 \vee e_i, i+1 \rangle, 2)\}$
end if
if $M_i = 1$ or $(A\vec{W})_i = 1$ then
$V \leftarrow V \cup \{\langle \vec{W}, \vec{M} \vee AB_i^1 \vee e_i, i+1 \rangle\}$
$E \leftarrow E \cup \{(\langle \vec{W}, \vec{M}, i \rangle, \langle \vec{W}, \vec{M} \vee AB_i^1 \vee e_i, i+1 \rangle, 2)\}$
end if
if $M_i = 1$ then
$V \leftarrow V \cup \{\langle \vec{W}, \vec{M} \vee AB_i^0 \vee AB_i^1 \vee e_i, i+1 \rangle\}$
$E \leftarrow E \cup \{(\langle \vec{W}, \vec{M}, i \rangle, \langle \vec{W}, \vec{M} \vee AB_i^0 \vee AB_i^1 \vee e_i, i+1 \rangle, 2)\}$
end if
$V \leftarrow V \cup \{\langle \vec{W}, \vec{M} \vee e_i, i+1 \rangle\}$
$E \leftarrow E \cup \{(\langle \vec{W}, \vec{M}, i \rangle, \langle \vec{W}, \vec{M} \vee e_i, i+1 \rangle, 0)\}$
end if
if $W_i \neq Y_i$ then
if $M_i = 1$ or $(A\vec{W})_i = 0$ then
$V \leftarrow V \cup \{\langle \vec{W} + B_i^0, \vec{M} \vee B_i^0 \vee e_i, i \rangle\}$
$E \leftarrow E \cup \{(\langle \vec{W}, \vec{M}, i \rangle, \langle \vec{W} + B_i^0, \vec{M} \vee B_i^0 \vee e_i, i \rangle, 1)\}$
end if

(continued)

Figure 6.4. Algorithm LegalExpansionTree.

(continued from Figure 6.4)

```
                if M_i = 1 or (A W⃗)_i = 1 then
                    V ← V ∪ { ⟨W⃗ + B_i^1, M⃗ ∨ B_i^0 ∨ e_i, i⟩ }
                    E ← E ∪ { (⟨W⃗, M⃗, i⟩, ⟨W⃗ + B_i^1, M⃗ ∨ B_i^1 ∨ e_i, i⟩, 1) }
                end if
                if M_i = 1 then
                    V ← V ∪ { ⟨W⃗ + B_i^0, M⃗ ∨ B_i^0 ∨ B_i^1 ∨ e_i, i⟩ }
                    E ← E ∪ { (⟨W⃗, M⃗, i⟩, ⟨W⃗ + B_i^0, M⃗ ∨ B_i^0 ∨ B_i^1 ∨ e_i, i⟩, 3) }
                    V ← V ∪ { ⟨W⃗ + B_i^1, M⃗ ∨ B_i^0 ∨ B_i^1 ∨ e_i, i⟩ }
                    E ← E ∪ { (⟨W⃗, M⃗, i⟩, ⟨W⃗ + B_i^1, M⃗ ∨ B_i^0 ∨ B_i^1 ∨ e_i, i⟩, 3) }
                end if
            end if
        end for
    end for
    return D ← (V, E)
end procedure
```

Figure 6.5. Algorithm LegalExpansionTree (continued).

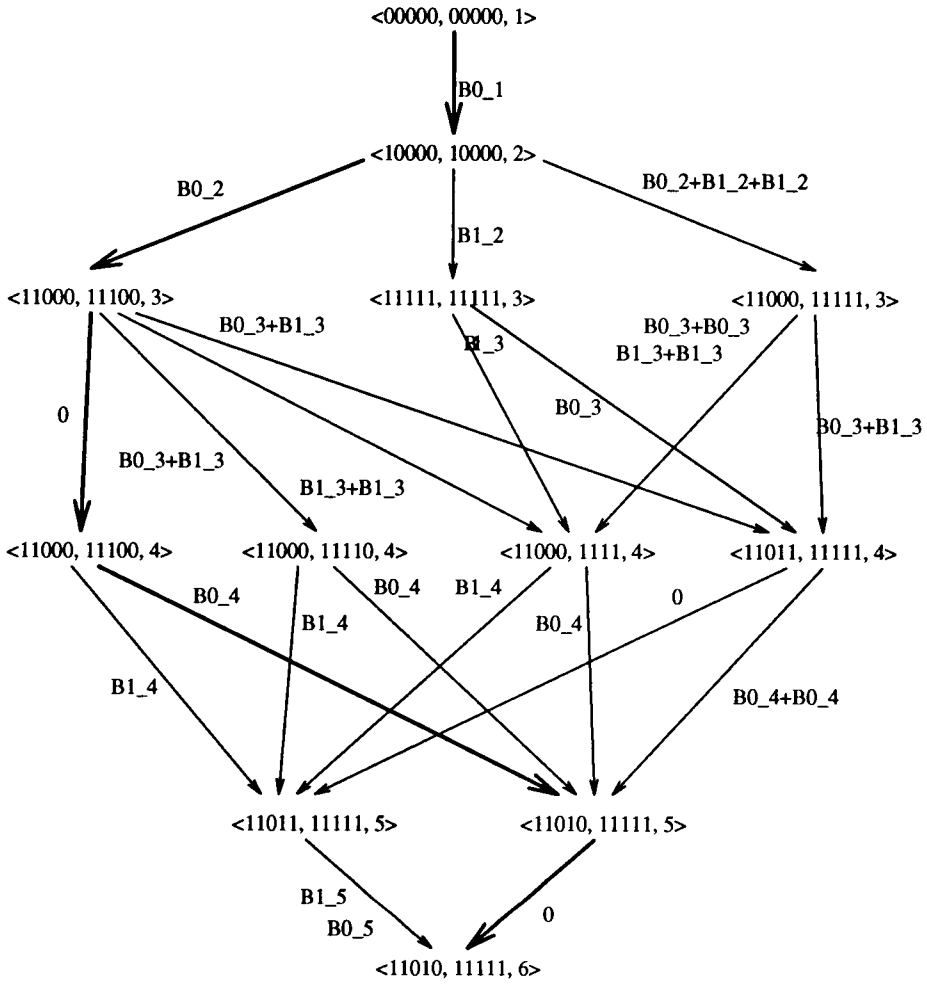

Figure 6.6. A legal expansion search tree between addresses $\vec{X} = (1000000)$ and $\vec{Y} = (1101010)$ on a 7-dimensional 1-Möbius Cube.

only add edges that correspond to terms that can be used in a legal expansion. All legal expansions will be generated, because each edge added corresponds to a term that depends only on terms that have already been included in the expansion at an earlier level.

In the algorithm, we have included all of the branches that can possibly give a minimal expansion. If $W_i = Y_i$, then we generate branches for:

$$\vec{W}, \vec{W} + B_i^0 + B_i^0, \vec{W} + B_i^1 + B_i^1, \vec{W} + B_i^0 + B_i^1$$

If $W_i \neq Y_i$, then we generate branches for:

$$\vec{W} + B_i^0, \vec{W} + B_i^1, \vec{W} + B_i^0 + B_i^1 + B_i^1, \vec{W} + B_i^1 + B_i^0 + B_i^0$$

All other possible summations of B^0 and B^1 to $\vec{W}$ will lead to non-minimal expansions and can be ignored. ∎

How large will the legal expansion search tree D get? For Algorithm LegalExpansionTree, the number of edges generated per vertex is a larger constant than for Algorithm ExpansionTree. We have also added an orthogonal component to the label of each vertex, so the number of vertices can be much larger. Where before we considered the vector space of the columns in Algorithm ExpansionTree, we now need to consider the union of the columns in the matrix products AB^0 and AB^1. The theorem below limits the number of expansion search tree vertices generated.

Theorem 6.3.2 *Let $G = (B^0, B^1, A)$ be an n-dimensional* LTLE *network and and let the matrix H_i be defined as in Theorem 5.4.2, and let the matrix K_i be defined as:*

$$\begin{bmatrix} AB^0_{i,1} & AB^0_{i,2} & \cdots & AB^0_{i,i} & AB^1_{i,1} & AB^1_{i,2} & \cdots & AB^1_{i,i} \\ AB^0_{i+1,1} & AB^0_{i+1,2} & \cdots & AB^0_{i+1,i} & AB^1_{i+1,1} & AB^1_{i+1,2} & \cdots & AB^1_{i+1,i} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ AB^0_{n,1} & AB^0_{n,2} & \cdots & AB^0_{n,i} & AB^1_{n,1} & AB^1_{n,2} & \cdots & AB^1_{n,i} \end{bmatrix}$$

Then for $\alpha_j \in \{0,1\}$ *with* $1 \leq j \leq 2i$, *the number of vertices in* D *is bounded above by the sum:*

$$\sum_{i=1}^{n} \left(\left| \left\{ \sum_{j=1}^{2i} \alpha_j (H_i)_j \right\} \right| \left| \left\{ \bigcup_{j=1}^{2i} \alpha_j (K_i)_j \right\} \right| \right) + 1$$

Proof: The proof is an extension of the proof of Theorem 5.4.2. In addition, we have that for a given i with $1 \leq i \leq n$, any vertex $\left\langle \vec{W}, \vec{M}, i \right\rangle$ has for $\alpha_j, \beta_j \in \{0,1\}$:

$$\vec{M} = \bigvee_{j=1}^{2i} \alpha_k B_j^0 + \beta_j B_j^1$$

Also, $\vec{M}$ has the property that $M_j = 1$ with $1 \leq j \leq i-1$, so that any vector $\vec{M}$ in vertices with a fixed i will be the same in the first $i-1$ indices. If we simply ignore the first $i-1$ indices of the columns of AB^0's and AB^1's, then we only need to know how many different values a union of the remaining indices can make. Finally there is only one final vertex $\left\langle \vec{Y}, \vec{1}, n+1 \right\rangle$. ∎

This is a rather loose upper bound on the number of vertices in D, but it is useful for bounding the size of D to order.

6.4. A MINIMAL *LTLE* NETWORK ROUTING ALGORITHM

Now we use the results from the legal minimal expansion algorithm to produce a routing algorithm for *LTLE* networks.

Algorithm LinearEquationRoute in Figure 6.7 will correctly route from source to destination on a *LE* network G. The algorithm is similar to Algorithm DoubleMatrixRoute, with two differences: A call to LegalExpansionTree is made in place of ExpansionTree, and call to ConstructRoutePath is made to correctly order the terms of the legal expansion before routing. The algorithm to minimally route on a *LTLE*

Input: A n-dimensional *LTLE* network $G = (B^0, B^1, A)$, the current address $\vec{W}$, a starting address $\vec{X}$ and a destination address $\vec{Y}$.

Output: The next neighbor to route to from $\vec{W}$.

```
LinearEquationRoute( G = (B^0, B^1, A), W, X, Y)
    if W = Y then
        accept message
    if W = X then
        D ← LegalExpansionTree(G, X, Y)
        P ← BreadthFirstSearch(D)
        S ← PathToExpansion(P)
        R ← ConstructRoutePath(G, X, Y, S)
    else
        remove B_i^φ from head of R
        route from X to X + B_i^φ
    end if
end procedure
```

Figure 6.7. Algorithm LinearEquationRoute.

networks is slightly worse than the algorithm to minimally route on a *LTDM* network.

Theorem 6.4.1 *Let $G = (B^0, B^1, A)$ be an n-dimensional* LTLE *network. If the expansion tree has at most $O(k)$ vertices at any level, for some constant k, then Algorithm LinearEquationRoute takes $O(n^2 \log n)$ bit operations to compute a minimal routing path.*

Proof: The algorithm has four parts: generating the expansion tree, computing a minimal weight path in the expansion search tree, deriving a minimal expansion from a minimal weight path, and ordering the terms of S into a routing path R.

The proof is similar to that in Theorem 5.5.1. This time, however, a legal expansion search tree is computed using Algorithm LegalExpansionTree instead of ExpansionTree. Since these two algorithms take the same number of bit operations (to order) to insert each new vertex and edge in the search tree, and since they will both generate the same number of vertices and edges (to order), Algorithm LegalExpansionTree will take $O(kn^2)$ bit operations.

The second part of the algorithm is the same breadth-first search and will take $O(kn \log n)$ bit operations to do.

The third part of the algorithm is also the same expansion construction and will take $O(n \log n)$ bit operations to compute and store the terms in the legal minimal expansion.

The final part of Algorithm LinearEquationRoute is the only part that is different from Algorithm DoubleMatrixRoute. By Corollary 6.2.2, ordering the terms of the expansion will take $O(n^2 \log n)$ bit operations.

The total number of bit operations overall is:

$$O(kn^2) + O(kn \log n) + O(n \log k) + O(n^2 \log n) = O(n^2(k + \log n))$$

∎

For the published *LE* networks, a legal routing path can be computed very quickly.

Corollary 6.4.1 *For all the published n-dimensional networks that can be described using* LE *networks, including the Twisted Cube, the Möbius Cubes, the Generalized Twisted Cube, the M-Cubes, and the Bent Cube, Algorithm LinearEquationRoute will have a total run time of* $O(n^2(12 + \log n))$ *bit operations and a distributed run time of* $O(n^2(12 + \log n))$ *bit operations.*

Proof: From Theorem 6.3.2, each network has at most 4 linear combinations of $\vec{W}$ each level i, where $W_i = 1$. If we examine the possible values of the mask $\vec{M}$ for each network, we see that there are at most only 3 possible unions at each level i. Each level will have will have at most $4 \times 3 = 12$ vertices. By Theorem 6.4.1, Algorithm LinearEquationRoute takes $O(n^2(12 + \log n))bitoperations$.

Since the routing computation is done entirely at the source node, the routing computation will dominate both the total and distributed computation times. ∎

This result shows that Algorithm LinearEquationRoute will compute minimal routing paths in $O(n^2 \log n)$ bit operations for all of the published *LE* networks. However, the Möbius cubes, and the Twisted Cube (for example) have routing algorithms that use at most a linear number of bit operations. This is asymptotically smaller than our algorithm can achieve.

There is an approach we can use to modify the algorithm LegalExpansionTree. Rather than compute all the bits of $\vec{W}$ and $\vec{M}$ in each vertex generated, we can use "place holders", or bits that indicate the condition for a whole set of indices.

If we use place holders, we no longer need to compute and copy the results of an entire mod 2 sum or union over up to n bits - just a constant number of bits.

For instance, on the Möbius cubes we notice that we need to keep track of possible cases for the values of $\vec{W}$. First, if no indices of $\vec{W}$ have been complemented, second, if only index i has been complemented (by adding B_i^0), or third, if indices i through n have been complemented (by adding B_i^1). These possibilities can be represented using only two bits, and so only 2 bits of $\vec{W}$ need computing and copying per vertex. Similarly, we can use place holders to tell if none of bits i through n of $\vec{M}$ are set, if only bit i of $\vec{M}$ is set, or if all of bits i through n of $\vec{M}$ are set. These possibilities can also be represented in only 2 bits. Finally, if the component i of each vertex is only implicitly represented in the structure of the graph by its depth from the root vertex, we can skip any computation for incrementing i. So, only a constant number of bit operations per vertex need be computed, and the algorithm to compute a legal expansion search tree now takes $O(n)$ bit operations.

The same approach can be taken with the other published networks. However, the approach clearly requires writing a different legal expansion tree algorithm for each network, because the place holders would have different meanings for each network. The more general algorithms that we have developed have the advantage that they can work with the algorithms unmodified.

This approach will not reduce the asymptotic run time of Algorithm LinearEquationRoute, because the last step of the algorithm takes $O(n^2 \log n)$ bit operations. If ordering the terms of the legal expansion can be done more quickly, then it might be possible to reduce the run time of Algorithm LinearEquationRoute to a linear number of bit operations in cases of specific networks.

6.5. NONREDUNDANT MINIMAL LEGAL EXPANSIONS

Just as minimal expansions can have redundant terms, legal minimal expansions can also have redundant terms. The *DM* networks have a simple condition for guaranteeing that no redundant minimal expansions exist in the network: $A(B_i^0 + B_i^1) \leq 1$. Unfortunately, this condition alone is not sufficient to guarantee that a *LE* network will have no redundant legal minimal expansions.

It is even possible that a *DM* network $G = (B^0, B^1)$ will have a nonredundant minimal expansion for two nodes $\vec{X}$ and $\vec{Y}$, while the related *LE* network $G = (B^0, B^1, A)$ will have only a redundant legal minimal expansion for the same nodes.

Example: Sometimes a cube definition makes it necessary to "un-correct" a certain term to route minimally. In the cube below, the first component controls whether a weight 1 or a weight 2 term is corrected at even positions:

$$B^0 = \begin{bmatrix} 1&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0 \\ 0&0&1&0&0&0&0 \\ 0&0&0&1&0&0&0 \\ 0&0&0&0&1&0&0 \\ 0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&1 \end{bmatrix} \quad B^1 = \begin{bmatrix} 1&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0 \\ 0&1&1&0&0&0&0 \\ 0&0&0&1&0&0&0 \\ 0&0&0&1&1&0&0 \\ 0&0&0&0&0&1&0 \\ 0&0&0&0&0&1&1 \end{bmatrix} \quad A = \begin{bmatrix} 0&0&0&0&0&0&0 \\ 1&0&0&0&0&0&0 \\ 1&0&0&0&0&0&0 \\ 1&0&0&0&0&0&0 \\ 1&0&0&0&0&0&0 \\ 1&0&0&0&0&0&0 \\ 1&0&0&0&0&0&0 \end{bmatrix}$$

For the *DM* network $G = (B^0, B^1)$, a minimal expansion for $\vec{X} = (0000000)$ and $\vec{Y} = (0111111)$ is B_2^1, B_4^1, B_6^1. For the *LE* network $G = (B^0, B^1, A)$, a minimal legal expansion is B_1^0, B_2^1, B_4^1, B_6^1, B_1^0 (in order). All other legal expansions will have weight 6. The term B_1^0 is used twice in the minimal legal expansion – in a network that is lower triangular.

This example network meets the conditions for nonredundant minimal expansions, yet still fails to have a nonredundant legal minimal expansion. ∎

To avoid redundant minimal legal expansions, we need to impose additional conditions on the network. The theorem below states the sufficient conditions to ensure that at least one minimal nonredundant legal expansion exists between any two nodes on a *LTLE* network:

Theorem 6.5.1 *Let* $G = (B^0, B^1, A)$ *is an n-dimensional* LTLE *network, where* $\forall i : W(B_i^0 + B_i^1) \leq 1$. *Then the following conditions are together sufficient to guarantee that a nonredundant legal minimal expansion* $S(\vec{X}, \vec{Y})$ *always exists for every pair of nodes* $\vec{X}$ *and* $\vec{Y}$*:*

- $B_j^\phi = B_i^0 + B_i^1 \Rightarrow (AB_i^0)_j = (AB_i^1)_j = 1, i < j$
- $((AB_i^\phi)_j = ((AB_i^\phi)_k = 1 \Rightarrow (AB_j^0)_k = (AB_j^1)_k = 1, i < j < k$

Proof: Condition 1 states that $B_i^0 + B_i^1 = B_j^\phi$, then B_j^ϕ depends on B_i^0 and B_i^1. Condition 2 states that if B_j^ψ and B_k^ρ, with $j < k$, both depend on some B_i^ϕ, then B_k^ρ also depends on B_j^ψ.

Assume that we have an legal minimal expansion S with redundant terms. Consider a pair of redundant terms with the smallest index i. If B_i^0 or B_I^1 appear in S, then there are a series of terms $B_{k_1}^{\psi_1}, B_{k_2}^{\psi_2}, \ldots, B_{k_n}^{\psi_n}$ that depend on B_i^0 or B_i^1, where $k_1 < k_2 < \ldots < k_n$.

There can be one of three cases:

Case 1: B_i^0 appears twice. We simply remove the two occurrences of B_i^0 from S, and transform the remaining terms.

If $B_{k_1}^{\psi_1}$ can be used in a legal expansion, then Condition 2 guarantees that $B_{k_2}^{\psi_2}, \ldots, B_{k_n}^{\psi_n}$ depend on $B_{k_1}^{\psi_1}$ and so can be arranged before or after $B_{k_1}^{\psi_1}$ to make a legal expansion $\check{S} = S - B_i^0 - B_i^0$ that is smaller than S.

If $B_{k_1}^{\psi_1}$ cannot be used in a legal expansion, then $B_{k_1}^{\psi_1} + B_{k_1}^{\overline{\psi_1}} = B_j^{\phi}$ for some j and Condition 1 allows us to replace $B_{k_1}^{\psi_1}$ with $B_{k_1}^{\overline{\psi_1}} + B_j^{\phi}$. Since B_j^{ϕ} depends on $B_{k_1}^{\psi_1}$, we can place it before or after to make it part of a legal expansion. Condition 2 then guarantees that $B_{k_2}^{\psi_2}, \ldots, B_{k_n}^{\psi_n}$ depend on $B_{k_1}^{\overline{\psi_1}}$ or B_j^{ϕ}, and so can be arranged before or after these terms to form a legal expansion. This replacement expansion $\check{S} = S - B_i^0 - B_i^0 - B_{k_1}^{\psi_1} + B_j^{\phi}$ is as small as S.

If there are any other redundant terms in the resulting $\check{S}$, they have an index greater than or equal to i and so can be removed recursively. This recursive process will eventually end, because is removes one redundant pair of terms at index i, and introduces no redundant pairs of terms at any index less than i.

Case 2: B_i^1 appears twice in S. We can use arguments similar to Case 1 to transform the expansion to a nonredundant expansion of the same or fewer terms.

Case 3: B_i^0 and B_i^1 each appear once in S. If $B_i^0 + B_i^1 = 0$, then we can replace B_i^1 with B_i^0 and use Case 1. However, if $B_i^0 + B_i^1 = B_j^{\phi}$, then we can replace B_i^0 and B_i^1 with B_j^{ϕ}. Then Condition 1 states that B_j^{ϕ} depends on B_i^0 and B_i^1, and so Condition 2 states that $B_{k_1}^{\psi_1}, B_{k_2}^{\psi_2}, \ldots, B_{k_n}^{\psi_n}$ depend on B_j^{ϕ}.

We can use an argument similar to the one in Case 1 to get a nonredundant expansion $\check{S}$ that is equal to or smaller than S. Here, however, we use B_j^{ϕ}, $B_{k_1}^{\psi_1}$, $B_{k_2}^{\psi_2}, \ldots, B_{k_n}^{\psi_n}$ instead of $B_{k_1}^{\psi_1}, B_{k_2}^{\psi_2}, \ldots, B_{k_n}^{\psi_n}$ in creating an expansion of equal or smaller weight. ∎

The published *LE* networks, including the Möbius cubes, the Flip MCube, the Twisted Cube, and the Generalized Twisted Cube, all meet the criteria of Theorem 6.5.1, so they are all nonredundant.

Algorithm NonredundantLegalExpansionTree in Figure 6.8 will compute a nonredundant legal expansion tree. A nonredundant legal expansion search tree for

Input: A n-dimensional nonredundant *LTLE* network $G = (B^0, B^1, A)$, a starting address $\vec{X}$ and a destination address $\vec{Y}$.

Output: A directed acyclic graph $D = (V, E)$, where each path from the vertex $\left\langle \vec{X}, \bigvee_{k=1}^{j-1} e_k, j \right\rangle$ with $j = \min\left(\{k : X_k \neq Y_k\}, n+1\right)$ to $\left\langle \vec{Y}, n+1 \right\rangle$ corresponds to a nonredundant legal expansion of $\vec{X} + \vec{Y}$.

Procedure NonredundantLegalExpansionTree(G, $\vec{X}$, $\vec{Y}$)
    $V \leftarrow \left\{ \left\langle \vec{X}, \bigvee_{k=1}^{j-1} e_k, j \right\rangle : j = \min\left(\{k : X_k \neq Y_k\}, n+1\right) \right\}$
    $E \leftarrow \{\}$
    for $i = 1$ to n do
        for each $\left\langle \vec{W}, \vec{M}, i \right\rangle \in V$ do
            $j \leftarrow \min(\{k : (\vec{W} + B_i^0)_k \neq \vec{Y}_k\}, n+1)$
            if $\vec{M}_i = 1$ or $(A\vec{W})_i = 0$ then
                $V \leftarrow V \cup \left\{ \left\langle \vec{W} + B_i^0, \vec{M} \vee AB_i^0 \vee e_i \vee \ldots \vee e_{j-1}, j \right\rangle \right\}$
                $E \leftarrow E \cup \left\{ \left(\left\langle \vec{W}, \vec{M}, i \right\rangle, \left\langle \vec{W} + B_i^0, \vec{M} \vee AB_i^0 \vee e_i \vee \ldots \vee e_{j-1}, j \right\rangle \right) \right\}$
            end if
            $j \leftarrow \min(\{k : (\vec{W} + B_i^1)_k \neq \vec{Y}_k\}, n+1)$
            if $\vec{M}_i = 1$ or $(A\vec{W})_i = 1$ then begin
                $V \leftarrow V \cup \left\{ \left\langle \vec{W} + B_i^1, \vec{M} \vee AB_i^1 \vee e_i \vee \ldots \vee e_{j-1}, j \right\rangle \right\}$
                $E \leftarrow E \cup \left\{ \left(\left\langle \vec{W}, \vec{M}, i \right\rangle, \left\langle \vec{W} + B_i^1, \vec{M} \vee AB_i^1 \vee e_i \vee \ldots \vee e_{j-1}, j \right\rangle \right) \right\}$
            end if
        end for
    end for
    return $D \leftarrow (V, E)$
end procedure

Figure 6.8. Algorithm NonredundantLegalExpansionTree.

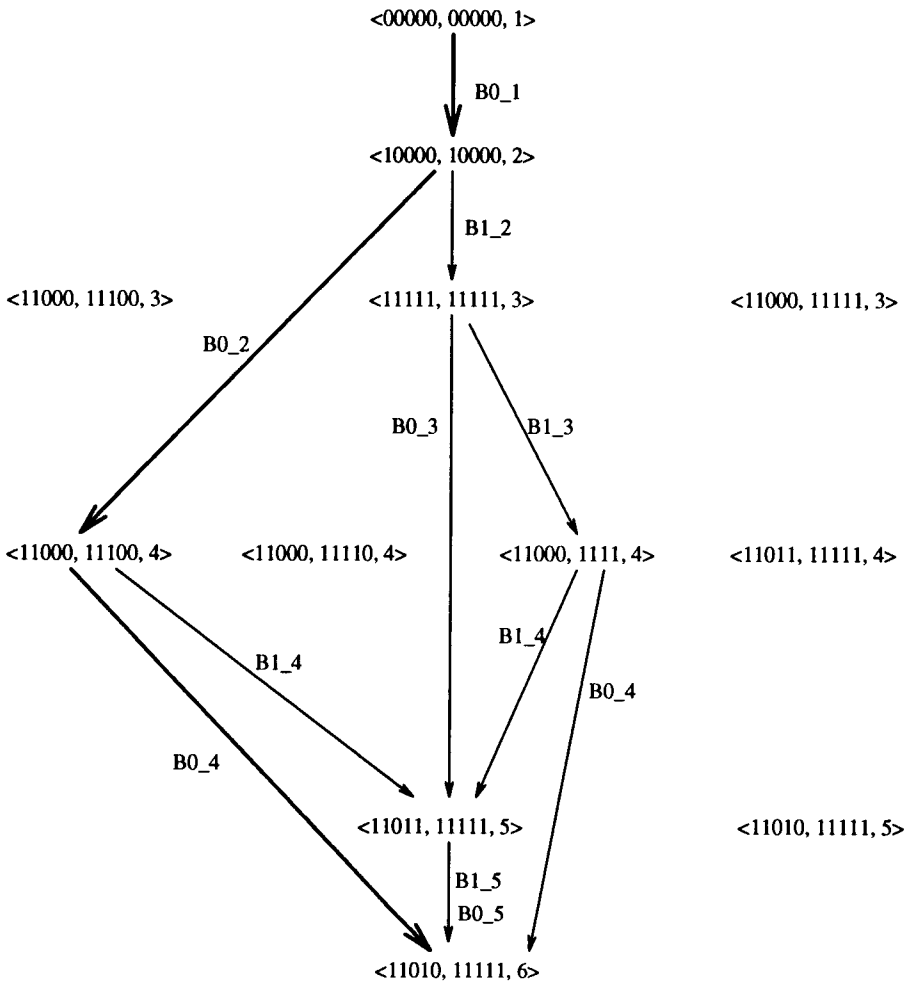

Figure 6.9. A nonredundant legal expansion search tree between addresses $\vec{X} = (1000000)$ and $\vec{Y} = (1101010)$ on a 7-dimensional Möbius Cube. The minimal path is shown in boldface.

$\vec{X} = (1000000)$ and $\vec{Y} = (1101010)$ on the 7-dimensional Möbius Cube is shown in Figure 6.9.

Algorithm NonredundantLegalExpansionTree is an extension of Algorithm NonredundantExpansionTree in Figure 5.4, in the same way that Algorithm Legal-ExpansionTree is an extension of Algorithm ExpansionTree. The same arguments given in Theorem 6.3.1 can be used to show that the algorithm is correct, and to show that the algorithm can execute in $O(kn^2)$ bit operations.

This algorithm will of course will execute the same asymptotic number of bit operations as the algorithm in Figure 6.4, because it generates the same or fewer number of nodes. It is essentially the same algorithm, with all but two of the cases removed. Because all the networks published in the literature to date meet the sufficient conditions of nonredundant networks in Theorem 6.5.1, they can all use Algorithm NonredundantExpansionTree to calculate nonredundant legal expansions.

6.6. DETERMINISTIC MINIMAL LEGAL EXPANSIONS

Finally, we briefly consider removing non-determinism from the search for minimal legal expansions. There is the possibility that a minimal legal expansion is not unique. When faced with more than one nonredundant minimal expansion, the breadth-first search may return arbitrarily either path. A simple heuristic can make the breadth-first search algorithm always return the same path.

Our heuristic is: If two branches from one node have the same distance to the destination node, choose the branch that leads to the node with the smallest indexed component that differs from the destination. We used this heuristic to determine a unique and deterministic legal minimal expansion from the legal expansion search tree. We chose this algorithm somewhat empirically, because it gives

the best (by far) network performance for optimal routing. Other rules for making choices (including randomly choosing a minimal path) did not work nearly so well. This approach appears to work best because it forces the terms of the legal minimal expansion S to be as low-indexed as possible.

6.7. SUMMARY

The problem of minimal routing on *LE* networks is *NP*-hard, and the problem of minimal routing to *LTLE* networks is *NP*-complete. This is strong evidence that a minimal routing algorithm with polynomial run time in n does not exist.

For *LTLE* networks, we were able to devise routing algorithms that had a polynomial run time in a large number of common cases. These algorithms have total run time of at worst $O(n^2 \log n)$ bit operations, compared to the $O(n)$ bit operations of the hypercube. While polynomial, our algorithms are not highly efficient, because they require a large amount of pre-computation at the source node.

We were able to show that for certain subclasses of the *LTLE* networks (nonredundant subclasses), we could simplify the algorithm and reduce the total run time by a constant amount. This still required a large amount of pre-computation at the source node.

Though these algorithms are important, because they are our most successful effort at producing a general minimal routing algorithm, they leave a lot to be desired. They do not have a small distributed run time (say, $O(1)$ bit operations per node). The following chapter will discuss efficient, though non-minimal routing algorithms for *DM* networks and *LE* networks.

7. NON-MINIMAL AND WORMHOLE ROUTING ALGORITHMS FOR *LE* NETWORKS

In this chapter, we discuss variations on point-to-point routing, including non-minimal routing, and wormhole routing. In particular, we will discuss an efficient routing algorithm that produces non-minimal routing paths, but captures most of the behavior of the minimal algorithm. This algorithm can be shown to be deadlock-free, which makes it usable under the wormhole routing strategy.

7.1. NON-MINIMAL ROUTING ALGORITHMS FOR *LE* NETWORKS

It is not always important that a routing algorithm generate minimal paths. The asymptotic run time complexity of the routing algorithm sometimes can be more important than a minimal path. A small (constant) number of bit operations per vertex may be more desirable when the communication time approaches the same order of magnitude as the routing computation time.

With the *LE* networks, we may need to trade minimality for simplicity. The minimal routing algorithms in the previous chapters may seem unnecessarily complicated for efficient message-routing. These algorithms also have a run time complexity that approaches a total $O(n^2 \log n)$ bit operations, which is not at all efficient when compared to the $O(n)$ bit operations of the hypercube's routing algorithm.

In this section, we examine some non-minimal routing algorithms for *LTLE* networks, and derive properties for each algorithm.

Input: An n-dimensional *LTLE* network $G = (B^0, B^1, A)$, a source address $\vec{X}$, a destination address $\vec{Y}$ and the current address $\vec{W}$.

Output: If the message needs to be forwarded, the index of the neighbor to route the message to. If the message is at its destination, a signal to accept the message at the current processor.

```
LeftRightBitCorrectRoute( G, X, Y, W )
begin
    If W = Y then
        Return "Accept"
    else
        i ← min{k : W_k ≠ Y_k}
        return i
    end if
end procedure
```

Figure 7.1. Algorithm LeftRightBitCorrectRoute.

7.1.1. The Left-Right Bit Correction Algorithm

The hypercube has a very standard point-to-point routing algorithm – the left-right (LR) bit correction algorithm. This is also known as the "greedy" algorithm or the "E-cube" routing algorithm [45]. This algorithm, which we will call Algorithm LeftRightBitCorrectRoute, "corrects" any components of the source address that differ from the destination address, starting with the smallest or left-most index.

A distributed version of Algorithm LeftRightBitCorrectRoute appears in Figure 7.1, and is modified to work on *LTLE* networks.

Theorem 7.1.1 *Let $G = (B^0, B^1, A)$ be an n-dimensional* LTLE *network. Algorithm LeftRightBitCorrectRoute correctly routes a message from $\vec{X}$ to $\vec{Y}$, and has a distributed run time of $O(n)$ bit operations per node, and a total run time of $O(n)$ bit operations.*

Proof: The routing algorithm will terminate when $\vec{W} = \vec{Y}$. At each processor it corrects the leftmost differing component between $\vec{W}$ and $\vec{Y}$, by routing to $\vec{W}$'s i-th neighbor. Because G is a *LTLE* network, routing from $\vec{W}$ to $\vec{W} + B_i^{(A\vec{W})_i}$ does not un-correct any component with index $j < i$, so progress to the destination is made at each step. There are at most n indices the algorithm has to correct, so it will terminate after a maximum of n iterations.

The total run time of $O(n)$ bit operations can be achieved by forwarding, with the message, the index of the most recently corrected component. The next node only has to examine components with indices greater than i for the next one to correct. ∎

Any *LTLE* network that uses Algorithm LeftRightBitCorrectRoute will have the same maximum and average routing distance as the hypercube:

Theorem 7.1.2 *Let $G = (B^0, B^1, A)$ be an n-dimensional* LTLE *network. The maximum routing distance of Algorithm LeftRightBitCorrectRoute is n steps and the average routing distance is $n/2$ steps.*

Proof: The proof is by induction on the dimension n:

Base Case: $n = 1$. The only *LTLE* network $G = ([1], [1], [0])$ has a maximum routing distance of 1 step and an average routing distance of $1/2$ step. (We include the zero distance from a node to itself in the average distance.)

Inductive Hypothesis: For $\check{n} < n$, the maximum routing distance is $\check{n}$ steps and the average routing distance is $\check{n}/2$ steps.

Inductive Step: By Theorem 3.3.2, G can be subdivided into 2 disjoint *LTLE* networks G_1 and G_2 of dimension $n-1$. G_1 and G_2 are joined by the channels in dimension 1.

First we show the maximum routing distance. Assume that $\vec{X}$ and $\vec{Y}$ are both in G_1 or G_2, respectively. By induction, the maximum routing distance between $\vec{X}$ and $\vec{Y}$ is $(n-1)$ steps.

Assume that $\vec{X}$ and $\vec{Y}$ are in G_1 and G_2, respectively. Then $X_1 \neq Y_1$. The algorithm will first route from $\vec{X}$ to some $\vec{W} \in G_2$. By induction, routing from $\vec{W}$ to $\vec{Y}$ on G_1 takes a maximum of $n-1$ steps, and so the maximum routing distance from $\vec{X}$ to $\vec{Y}$ is n steps.

The maximum routing distance, over all possible cases, is n steps.

Now we show the average routing distance. Assume a uniform distribution of $\vec{X}$ and $\vec{Y}$. Their mod 2 sum is then also uniformly distributed over $\mathcal{Z}_2^n$. The address pairs $\vec{X}$ and $\vec{Y}$ can be divided into two groups of 2^{n-1} pairs each, one group with $X_1 = Y_1$ and the other with $X_1 \neq Y_1$.

The address pairs with $X_1 = Y_1$ have mod 2 sums with a uniform distribution over $(0\ \mathcal{Z}_2^{n-1})$. By induction, they have an average routing distance of $(n-1)/2$.

The address pairs with $X_1 \neq Y_1$ have mod 2 sums with a uniform distribution over $(1\ \mathcal{Z}_2^{n-1})$. By Lemma 2.1.1 the neighbor function N_1 is 1-1, and for *LTLE* networks $[N_1(\vec{X}]_1 \neq \vec{X}_1$, so the mod 2 sums of $N_1(\vec{X}) + \vec{Y}$ will map uniformly to $(0\ \mathcal{Z}_2^{n-1})$. By induction, these node pairs have an average routing distance of $(n-1)/2+1$.

The two groups of node pairs are equal in size, so the average routing distance is then $0.5((n-1)/2) + 0.5((n-1)/2+1) = n/2$. ∎

Another concern of a routing algorithm is how the algorithm distributes the routing paths of messages traveling through the network. If even a slightly

higher fraction of the messages are routed through a single channel, that channel can quickly become a communications bottleneck for the entire network – it will slow the transmission of all messages.

One measure, the *channel utilization*, is defined as the number of routing paths (generated by a routing algorithm) that pass through a given channel. This is measured as a fraction of all routing paths generated between all source and destination nodes in a network. In our measures, the channels are considered to be unidirectional, so the utilization of a bidirectional channel would be the sum of channel utilization for its corresponding unidirectional channels. The channel utilization is related to the *channel utilization rate*, which measures the fraction of time that a channel is utilized in transmitting messages.

The channel utilization depends not only on the topology of the network, but also on the routing algorithm used. For the hypercube, the channel utilization of any given channel is 0.25 of all messages, assuming that Algorithm LeftRightBitCorrectRoute. This is not true of all hypercube variants or routing algorithms. Abraham and Padmanabhan [2] showed that the Twisted Cube of Hilbers [34] had an non-uniform distribution of channel utilization, with some channel utilizations exceeding 0.25. This caused the Twisted Cube to have a much worse overall network behavior than the hypercube under heavy message loads.

We examine the channel utilizations for *LTLE* networks using Algorithm LeftRightBitCorrectRoute and a uniform message distribution. Not surprisingly, the channel utilizations are the same as the hypercube's channel utilizations.

Theorem 7.1.3 *Let* $G = (B^0, B^1, A)$ *be an* n*-dimensional* LTLE *network. If Algorithm LeftRightBitCorrectRoute is used, then the channel utilization for all channels will be uniformly 0.25.*

Proof: The proof is by induction on the network's dimension n.

Base Case: For a 1-dimensional *LTLE* network, the channel utilization is 0.25. There are four source, destination pairs, only two of which will use one of the two channels.

Inductive Hypothesis: Assume that for all $(n-1)$-dimensional *LTLE* networks using Algorithm LeftRightBitCorrectRoute, all of the unidirectional channels each have a channel utilization of 0.25.

Inductive Step: By Theorem 3.3.2, G can be subdivided into 2 disjoint *LTLE* networks G_1 and G_2 of dimension $n-1$. G_1 and G_2 are joined by the channels in dimension 1.

The paths with source $\vec{X}$ and destination $\vec{Y}$ can be divided into two groups of 2^{n-1} pairs each, one group with $X_1 = Y_1$ and the other with $X_1 \neq Y_1$. These two groups are of equal size, or 0.5 of all paths.

If $X_1 = Y_1$, then the path is entirely in the sub-network G_1 or G_2. By induction, this set of paths will give channels in dimension 2 through n a utilization of $0.5 \times 0.25 = 0.125$.

If $X_1 \neq Y_1$, then the path crosses channels in dimension 1. All the channels in dimension 1 will each have a channel utilization of $0.5 \times 0.5 = 0.25$. The neighbor function is one to one, so after removing the first step of these paths, the remaining sub-paths will have uniformly distributed sources and destinations in G_1 and G_2. By induction, these sub-paths will give channels in dimension 2 through n an additional utilization of $0.5 \times 0.25 = 0.125$.

The total channel utilization of any channel is then 0.25. ∎

These results make the *LTLE* networks clearly comparable to the hypercube. Specifically, any *LTLE* network has a routing algorithm with performance measures that are *at worst* comparable to the hypercube. However, a *LTLE* network can often

from \ to	000	001	010	011	100	101	110	111
000	-	3	2	2	1	1	**2**	1
001	3	-	2	2	1	1	1	**2**
010	2	2	-	3	**2**	1	1	1
011	2	2	3	-	1	**2**	1	1
100	1	1	1	**2**	-	3	2	2
101	1	1	**2**	1	3	-	2	2
110	1	**2**	1	1	2	2	-	3
111	**2**	1	1	1	2	2	3	-

Table 7.2. A routing table for the Twisted 3-Cube.

achieve even better performance measures than the hypercube, if given a slightly more complex routing algorithm.

7.1.2. The Three Bit Lookahead Algorithm

Consider for a moment, a routing algorithm for the Twisted 3-Cube. The routing instructions for the Twisted 3-Cube can be written as a table, as shown in Table 7.2, because the network is of fixed size. This table tells each processor which neighbor in the Twisted 3-Cube to route to next. A minimal routing table for the Twisted 3-Cube appears in Table 7.2.

This routing table follows (for the most part) the routing paths specified by Algorithm LeftRightBitCorrectRoute, with the exceptions listed in boldface. Because there is exactly one exception for each source, and because that exception always has the message route in dimension 2, this table can be stored very compactly as a table of the eight exceptions in $3 \times 8 = 24$ bits:

$$[110, 111, 100, 101, 011, 010, 001, 000]$$

For a message at current node address $\vec{W}$ on the Twisted 3-Cube, the routing algorithm looks up table entry $\vec{W}$. If the table entry is equal to the message's destination address, the algorithm routes to $\vec{W}$'s second neighbor. If not, it follows Algorithm LeftRightBitCorrectRoute.

We can extend this approach to higher dimensional *LE* networks, if we break the address space of a vector $\vec{X}$ into triples of components, as below:

$$\vec{X} = (X_1X_2X_3)(X_4X_5X_6)\ldots(X_{n-2}X_{n-1}X_n)$$

We can treat each component triple with indices i, $i+1$, $i+2$ as a 3-dimensional *LTLE* network, by ignoring all other components. At each node, a routing table can be generated and stored for each group of three indices. We then use the $\lceil i/3 \rceil$-th routing table to route along dimensions i, $i+1$, or $i+2$, ignoring what happens to the components with indices $i+3,\ldots,n$. By routing each component triple from smallest to largest, all the indices will be corrected and the message will reach its destination.

Depending on the number of component triples that are isomorphic to the Twisted 3-Cube, this routing algorithm can give a maximum routing distance as small as $\lceil 2n/3 \rceil$, and an expected routing distance as small as $\frac{11}{8}\lceil \frac{n}{3} \rceil + \frac{1}{2}(n \bmod 3)$.

This closely follows the approach used with the Generalized Twisted Cube of Chedid and Chedid [12], because their networks are built by graph composition of Twisted 3-Cubes. This approach always routes minimally for the Generalized Twisted Cubes, but not for other networks.

In fact, arbitrarily breaking the node addresses into triples can sometimes ignore the twisted channels that exist in the network. For instance, if $B^0_{3k} = e_{3k}$ and $B^1_{3k} = e_{3k} + e_{3k+1}$ for any $1 \leq k < n/3$, then this algorithm will ignore the fact that a step might be saved by routing on an channel defined by B^1_{3k} instead of one

defined by B^0_{3k}, because the components are broken into triples between indices $3k$ and $3k+1$.

A slightly different approach, Algorithm ThreeBitLookaheadRoute, doesn't break the address into triples. Instead, it operates more like Algorithm LeftRightBitCorrectRoute. It first finds the leftmost differing component between the current address and the destination address, then routes the components with indices i, $i+1$, and $i+2$ as a 3-Cube, ignoring indices $i+3$ through n. This approach is more expensive than the approach mentioned above, because it uses $n-3$ routing tables instead of $\lceil n/3 \rceil$ routing tables, but it will be more minimal, because it will ignore fewer twisted channels.

A distributed version of Algorithm ThreeBitLookaheadRoute is shown in Figure 7.2.

Theorem 7.1.4 *Let $G = (B^0, B^1, A)$ be an n-dimensional* LTLE *network. Algorithm ThreeBitLookaheadRoute correctly routes a message from $\vec{X}$ to $\vec{Y}$. Further, it has a distributed run time of at worst $O(n)$ bit operations per node and has a total run time of at worst $O(n)$ bit operations.*

Proof: The algorithm terminates correctly because it terminates only when $\vec{W} = \vec{Y}$. Each routing step finds is the smallest index i where $W_i \neq Y_i$, and then corrects component W_i, except if correcting W_{i+1} is locally shorter. Because G is a *LTLE* network, correcting any component W_i will not "un-correct" components with indices $j < i$, so at each successive node, i increases or stays the same. Also, if the algorithm corrects component W_{i+1}, it will correct component W_i on the next step, because the condition that selected index $i+1$ will no longer be true. Thus the algorithm will eventually terminate, because there are at most n components to

Input: An n-dimensional *LTLE* network $G = (B^0, B^1, A)$, a source address $\vec{X}$, a destination address $\vec{Y}$, and the current address $\vec{W}$.

Output: If the message needs to be forwarded, the index of the neighbor to route the message to. If the message is at its destination, a signal to accept the message at the current processor.

```
ThreeBitLookaheadRoute( G, X⃗, Y⃗ )
begin
    if W⃗ = Y⃗ then
        return "Accept"
    else
        i ← min{k : X_i + Y_i = 1}
        if i ≤ n − 2 then
            U⃗ ← W⃗ + B_{i+1}^{(AW⃗)_{i+1}}
            V⃗ ← U⃗ + B_i^{(AU⃗)_i}
        end if
        if 1 ≤ i ≤ n − 2 and B⁰_{i+1} ≠ B¹_{i+1} and (AB_i^{(AX⃗)_i})_{i+1} = 1
        and V_iV_{i+1}V_{i+2} = Y_iY_{i+1}Y_{i+2} then
            return i + 1
        else
            return i
        end if
    end if
end procedure
```

Figure 7.2. Algorithm ThreeBitLookaheadRoute.

correct, and each component is selected as the smallest differing component at most twice and corrected at most once.

Consider each of the conditionals in the routing algorithm. The computation of $\vec{V}$ takes a constant number of bit operations, because only components $V_iV_{i+1}V_{i+2}$ need to be computed. If needed, the possible values of $V_iV_{i+1}V_{i+2}$ can be precomputed and stored in a table at each node. The comparison of $\vec{V}$ to $\vec{Y}$ can also be done in a constant number of bit operations. All the other conditionals are independent of the message destination and can be precomputed for each node $\vec{W}$ and index i. The only computation that takes $O(n)$ bit operations is finding the lowest index i where $W_i \neq Y_i$.

We can forward with the message the index i of the most recently corrected component. Then the next node only has to examine any components with indices $j > i$ for the next component to correct. This algorithm then has a distributed run time of $O(1)$ bit operations per node, and a total run time of $O(n)$ bit operations. ∎

Algorithm ThreeBitLookaheadRoute has the same asymptotic run time order as Algorithm LeftRightBitCorrectRoute. The relatively small number of bit operations per node gives this algorithm a simple hardware implementation that is not much more complicated than Algorithm LeftRightBitCorrectRoute.

The maximum and average routing distances of Algorithm ThreeBitLookaheadRoute both compare favorably to Algorithm LeftRightBitCorrectRoute, as the next theorem shows.

Theorem 7.1.5 *Let $G = (B^0, B^1, A)$ be an n-dimensional* LTLE *network. The maximum routing distance of Algorithm ThreeBitLookaheadRoute is less than or equal to n routing steps and greater than or equal to $\lceil 2n/3 \rceil$ steps. The expected routing distance is less than or equal to $n/2$ routing steps and greater than or equal to $E(n)$ steps, where:*

$$E(n) = \frac{3}{7}n + \frac{5}{49} + \frac{1}{6 \cdot 49}\left[(15 - 11\sqrt{3}i)\lambda_1^n + (15 + 11\sqrt{3}i)\lambda_2^n\right]$$

Where:

$$\lambda_1 = \frac{-1 + \sqrt{3}i}{4}$$
$$\lambda_2 = \frac{-1 - \sqrt{3}i}{4}$$

Proof: There are three possible conditions for the components of node $\vec{W}$ at index i:

1. $1 \leq i \leq n-2$, $B_i^0 \neq B_i^1$ and $(AB_i^0)_{i+1} = (AB_i^1)_{i+1} = 1$.

2. $1 \leq i \leq n-2$, $B_i^0 = B_i^1$ or $(AB_i^0)_{i+1} = (AB_i^1)_{i+1} = 0$.

3. $n-1 \leq i \leq n$.

If Condition 1 is true, then the algorithm routes on the components with indices i, $i+1$ and $i+2$ like a Twisted 3-Cube. If Condition 2 is true, then the algorithm routes on the components with indices i, $i+1$ and $i+2$ like a normal 3-cube. If Condition 3 is true, then the algorithm routes on the components with index i or indices i and $i+1$ like a normal 1-cube or 2-cube.

By Theorem 3.3.2, G can be decomposed into smaller dimension networks. We will use this property to compute the maximum and expected distance.

For condition 3, $n-1 \leq i \leq n$. The maximum and average routing distance can be calculated by enumeration:

$$D(0) = 0,\ D(1) = 1,\ D(2) = 2$$

$$E(0) = 0,\ E(1) = 0.5,\ E(2) = 1.0$$

Assume that for every triple of components starting at index i, $1 \leq i \leq n-2$ in network G meet Condition 1. There are eight cases, based on the leftmost differing components of the sum $\vec{W} + \vec{Y}$. We can join the cases together into three main cases, based partly on the possible values of V in the algorithm:

1. If $W_1 + Y_1 = 0$, then the algorithm will inductively look at $W_2 + Y_2$. By induction, each of the four cases that meet this condition will have a maximum distance $D(n-1)$ and an average distance $E(n-1)$.

2. If $W_1 + Y_1 = 1$ and $V_iV_{i+1}V_{i+2} \neq Y_iY_{i+1}Y_{i+2}$, then the algorithm corrects the first two components in one step, by routing to the neighbor of $\vec{W}$ in dimension 1. The third component may or may not be set. By induction, each of these two cases will have a maximum distance $1 + D(n-2)$ and an average distance $1 + E(n-2)$.

3. If $W_1 + Y_1 = 1$ and $V_iV_{i+1}V_{i+2} = Y_iY_{i+1}Y_{i+2}$, then the algorithm corrects the first components in two steps, by first routing to the neighbor of $\vec{W}$ dimension 2, the routing to the neighbor of that node in dimension 1. In these two steps, the third component is also corrected. By induction, each of these two cases will have a maximum distance $2 + D(n-3)$ and an average distance $2 + E(n-3)$.

The maximum and average routing distances are described by the recurrence relations:

$$D(n) = \max\left(D(n-1), 1 + D(n-2), 2 + D(n-3)\right)$$
$$E(n) = \frac{1}{2}E(n-1) + \frac{1}{4}E(n-2) + \frac{1}{4}E(n-3) + \frac{3}{4}$$

Solving these recurrences will give us the solutions:

$$D(n) = \lceil \frac{2n}{3} \rceil$$
$$E(n) = \frac{3}{7}n + \frac{5}{49} + \frac{1}{6 \cdot 49}\left[(15 - 11\sqrt{3}i)\lambda_1^n + (15 + 11\sqrt{3}i)\lambda_2^n\right]$$

Where:

$$\lambda_1 = \frac{-1 + \sqrt{3}i}{4}$$
$$\lambda_2 = \frac{-1 - \sqrt{3}i}{4}$$

Now assume every index i, $1 \leq i \leq n-2$ in network G meets Condition 2. The algorithm cannot choose to correct components with indices $i+1$ and $i+2$ in one step, so the algorithm then functions identically to Algorithm LeftRightBitCorrectRoute.

For average routing distance, the routing algorithm has 2 cases, based on the first component of the mode 2 sum of $\vec{W} + \vec{Y}$.

1. If $W_1 = Y_1$, then the algorithm will inductively look at $W_2 + Y_2$. By induction the maximum routing distance is $D(n-1)$ and the average distance is $E(n-1)$.

2. If $X_1 \neq Y_1$, then the algorithm routes to $\vec{X}$'s neighbor in dimension 1. By induction, the maximum routing distance is $1 + D(n-1)$ and the average distance in this case is $1 + E(n-1)$.

The maximum and average cases can be described by recurrence relations:

$$D(n) = \max(D(n-1), 1 + D(n-1))$$
$$E(n) = \frac{1}{2}E(n-1) + \frac{1}{2}[1 + E(n-1)]$$

Then the maximum routing distance is $D(n) = n$ and the average routing distance is $E(n) = n/2$.

We now show that these are the upper and lower bounds on the maximum and average routing distance. Assume that we have two networks G_1 and G_2 that

have each triple of components meet the same conditions above, except that the components starting at index 1 in G_1 meet Condition 1 and the components starting at index 1 in G_2 meet Condition 2. Clearly from the recurrence relations, G_1 will have a smaller maximum and average routing distance than G_2.

Further, if we use Theorem 3.3.2 to show that G_1 is a sub-network of a network $\check{G}_1$, we can replace G_1 with G_2 to create a new network $\check{G}_2$ that will have an equal or larger maximum and expected routing distance. Thus each time we change the definition of a network so that the components starting at index i meet Condition 2 instead of Condition 1, the maximum and expected routing distance will be equal or larger.

The maximum and expected routing distance can be maximized by ensuring that all triples of components starting at index i with $1 \leq i \leq n-2$ meet Condition 1, and minimized by ensuring that all triples of components starting at index i with $1 \leq i \leq n-2$ meet Condition 2. ∎

A list of average routing distances for Algorithm ThreeBitLookaheadRoute on the Bent Cube appears in Table 7.3. As can be seen, Algorithm ThreeBitLookaheadRoute provides a not insubstantial savings on the expected distance – about 11% for an 8-dimensional cube.

For a packet-switched network, this algorithm may not provide very much savings in communication time when compared to a minimal algorithm. But average routing distance is not a dominant factor in network message delay for circuit-switched networks. Instead, it is the message length that determines the message delay. In a circuit switched network, the rate at which channels are utilized can be more important than the average routing distance between nodes. In the theorem below, we bound the channel utilization of networks using Algorithm ThreeBitLookaheadRoute.

Dimension	Left-Right Exp. Dist.	Lookahead Exp. Dist.	Percent Savings
1.00	0.500000	0.500000	0.0000 %
2.00	1.000000	1.000000	0.0000 %
3.00	1.500000	1.375000	8.3333 %
4.00	2.000000	1.812500	9.3750 %
5.00	2.500000	2.250000	10.0000 %
6.00	3.000000	2.671875	10.9375 %
7.00	3.500000	3.101562	11.3839 %
8.00	4.000000	3.531250	11.7188 %
9.00	4.500000	3.958984	12.0226 %
10.00	5.000000	4.387695	12.2461 %
11.00	5.500000	4.816406	12.4290 %
12.00	6.000000	5.244873	12.5854 %
13.00	6.500000	5.673462	12.7160 %
14.00	7.000000	6.102051	12.8278 %
15.00	7.500000	6.530609	12.9252 %

Table 7.3. Expected Distances of the 3-bit lookahead algorithm.

Theorem 7.1.6 *Let $G = (B^0, B^1, A)$ be an n-dimensional* LTLE *network. If Algorithm ThreeBitLookaheadRoute is used, then the channel utilization for all channels in dimensions 1 and 2 will be 0.25 of all paths in the network, and the utilization for all channels in dimensions 3 through n will be between 0.1875 and 0.25 of all paths in the network.*

Proof: The proof is by induction on the dimension of the network. We will represent the channel utilization for channels in dimension i on an n-dimensional *LTLE* network as $L(i, n)$.

Base Case: $n = 1$, $n = 2$. The channel utilization can be computed by enumerating all the cases. For a 1-dimensional cube, the channel utilization is:

$$L(1, 1) = 0.25$$

and for a 2-dimensional cube, the channel utilization is:

$$L(1, 2) = L(2, 2) = 0.25$$

Inductive Step: $n \geq 3$. In Theorem 7.1.5, we listed the three conditions that could occur for the components of a node $\vec{W}$ at index i for a network G.

First assume that all triples of components starting at index i with $1 \leq i \leq n - 2$ meet Condition 2. The routing paths can be broken into two cases, based on the first differing component of $\vec{W}$ and $\vec{Y}$. If $W_1 = Y_1$, then the algorithm will inductively look at $W_2 + Y_2$. If $W_1 \neq Y_1$, then the algorithm will route to $\vec{W}$'s neighbor in dimension 1, and then inductively look at $W_2 + Y_2$. Because the neighbor function is 1-1, the result of routing across dimension 1 will uniformly distribute source and destinations of the remaining sub-paths across $\vec{W} + \vec{Y} = (0\mathcal{Z}_2^{n-1})$. By induction, the channel utilization can be written as a recurrence relation:

$$L(1,n) = \frac{1}{4}$$
$$L(i+1,n) = L(i,n-1),\ 1 \leq i \leq n$$

Solving the recurrence gives $L(i,n) = 0.25$ for all $1 \leq i \leq n$.

Now assume that all all triples of components starting at index i with $1 \leq i \leq n-2$ meet Condition 1. The routing paths can be broken into eight cases, based on the first three indices of the sum $\vec{W} + \vec{Y}$. We can join the eight cases together into three main cases:

1. If $W_1 + Y_1 = 0$, then the algorithm will inductively look at $W_2 + Y_2$. By induction, the four cases that meet this condition add 0 to $L(1,n)$, add 0.5 $L(1,n-1)$ to $L(2,n)$, add 0.5 $L(2,n-1)$ to $L(3,n)$ and add 0.5 $L(i,n-1)$ to $L(i+1,n)$ for $3 \leq i \leq n-1$.

2. If $W_1 + Y_1 = 1$ and $V_iV_{i+1}V_{i+2} \neq Y_iY_{i+1}Y_{i+2}$, then the algorithm corrects the first two components in one step, by routing to the neighbor of $\vec{W}$ in dimension 1. The third component may or may not be set. Because the neighbor function is 1-1, the result of routing across dimension 1 will uniformly distribute source and destinations of the remaining sub-paths across $\vec{W} + \vec{Y} = (0\mathcal{Z}_2^{n-1})$. By induction, the two cases add 0.125 to $L(1,n)$, add 0 to $L(2,n)$, and add 0.25 $L(i,n-2)$ to $L(i+2,n)$ for $1 \leq i \leq n-2$.

3. If $W_1 + Y_1 = 1$ and $V_iV_{i+1}V_{i+2} = Y_iY_{i+1}Y_{i+2}$, then the algorithm corrects the first three components in two steps, by first routing to the neighbor of $\vec{W}$ in dimension 2, then routing in dimension 1. Because the neighbor function is 1-1, the result of routing across dimension 2 and then dimension 1 will uniformly distribute source and destinations of the remaining sub-paths across

$\vec{W}+\vec{Y} = (000\{\mathcal{Z}_2\}^{n-4})$. By induction, the two cases add 0.125 to $L(1,n)$, add 0.125 to $L(2,n)$, and add 0.25 $L(i,n-3)$ to $L(i+3,n)$ for $1 \leq i \leq n-3$.

Summing the channel utilization for each $L(i,n)$, we get:

$$L(1,n) = 0.125 + 0.125$$

$$L(2,n) = 0.5\ L(1,n-1) + 0.125$$

$$L(3,n) = 0.5\ L(2,n-1) + 0.25\ L(1,n-2)$$

$$L(i,n) = 0.5\ L(i-1,n-1) + 0.25\ L(i-2,n-2) + 0.25\ L(i-3,n-3)$$

It is not even necessary to solve the recurrence relation set up in this proof. Inductively substituting in the values of the L's into the recurrence will give $L(1,n) = 0.25$, $L(2,n) = 0.25$, and $L(3,n) = 0.1875$, *etc.* The channel utilization $L(i,n)$ is an average over $L(i-1,n-1)$, $L(i-2,n-2)$ (twice) and $L(i-3,n-3)$. The smallest value for channel utilization will be 0.1875 and the largest value will be 0.25, and so $0.1875 \leq L(i,n) \leq 0.25$ for $1 \leq i \leq n$. ∎

Table 7.4 gives a list of channel utilizations for 1- through 15-dimensional Bent Cubes. The sequence eventually converges to 0.21428571 in the first 8 digits, or approximately 85.71% of the channel utilization of the same channel using Algorithm LeftRightBitCorrectRoute.

7.1.3. Extending the Three Bit Lookahead Algorithm

Algorithm ThreeBitLookaheadRoute computes which neighbor to route to by using only the three components with indices i, $i+1$ and $i+2$, where $i = \min(\{k : W_k \neq Y_k\})$. The algorithm, as specified, computes the next neighbor to route to "on the fly". If we store at each node $\vec{W}$ a table of which neighbor to route to for all possible values of components $W_i + Y_i$, $W_{i+1} + Y_{i+1}$, and $W_{i+2} + Y_{i+2}$, we don't

Dimension	Channel Utilization
1	0.25
2	0.25
3	0.1875
4	0.21875
5	0.21875
6	0.2109375
7	0.21484375
8	0.21484375
9	0.2138671875
10	0.21435546875
11	0.21435546875
12	0.2142333984375
13	0.21429443359375
14	0.21429443359375
15	0.2142791748046875

Table 7.4. Minimal Channel Utilization of the 3-bit lookahead algorithm.

even need to worry about computing the next neighbor to route to; we can simply look it up on the table. If we naively store the routing tables, (as in Table 7.2), we need to store $n-2$ tables of $2^3 \times 2^3 = 64$ bits, so the total storage takes $64(n-2)$ bits. We'll also need a table of 32 bits and a table of 4 bits to store which neighbors to route to for indices $i = n-1$ and $i = n$, respectively.

We can extend Algorithm ThreeBitLookaheadRoute to "look ahead" at any number of bits. If we extend the algorithm to k lookahead indices, then we need to precompute all locally shortest paths (using the first k components starting from the component with index i), and then store the first step of each. Using the minimal routing algorithm we distributively compute these shortest paths on each node $\vec{X}$ in $O((n-k)(k^2 \log k))$ bit operations. The results of these computations would have

to be stored in $n-k+1$ different routing tables at each node, and each table would store $2^k \times 2^k$ numbers of k bits each, plus one table each for each $1 \leq i < k$ which stores $2^i \times 2^i$ numbers of i bits each. The total number of bits to store this would be:

$$(n-k+1)2^{2k} + \sum_{i=1}^{k-1} 2^{2i}$$

This amount of space grows exponentially as k increases, so network designers will probably want to keep k small.

There are other reasons to avoid extending the lookahead of the algorithm. For Twisted Cube networks of dimension 4 or larger, following minimal paths can route channels asymmetrically across the network, as shown by the Twisted Cube [2], which can lead to network bottlenecks. Also, any lookahead algorithms using a lookahead of more than 2 components will also bring only exponentially growing storage and lookup costs for diminishing returns.

7.2. WORMHOLE ROUTING ALGORITHMS FOR *LE* NETWORKS

The claim that *LE* networks are better than the hypercube rests on the assumption that a reduced expected and maximum routing distance will lead to shorter expected communication times. This claim is justified if the average number of routing steps a message takes is a dominant factor in the message latency (the time a message's transmission takes from source to destination). This can happen in packet-switched message-passing strategies, such as the store-and-forward routing, but for circuit-switched strategies, like wormhole routing, have message latencies that are relatively independent of the expected and maximum routing distance. Such strategies have become increasingly preferred in multicomputer implementations.

Two questions arise about the *LE* networks. First, can circuit-switched strategies like wormhole routing be implemented on *LE* networks? Second, do these alternate strategies show better performance on the *LE* networks than on the hypercube? In this section, we show the answers for these two questions for the wormhole routing strategy.

7.2.1. An Introduction to Wormhole Routing

Currently, "wormhole" routing is receiving wide attention as a routing method that is preferable to store-and-forward routing [46]. In a typical store-and-forward communications algorithm, each node along the path of a message receives the message and stores it, then computes which neighbor to forward the message to next. The message's latency – the time to travel from source to destination – is dominated by the product of the message's length and the number of routing steps, at least when a relatively small number of messages are in the network. Store-and-forward routing requires storage buffers at each node, which can be expensive in terms of hardware. If a message is too long to fit the buffer length, it may be broken into packets, which are each sent separately – and so store-and-forward routing is known as a *packet-switched* strategy.

The wormhole routing approach avoids the problem of buffers. It allocates all the communication channels along the routing path, as the head of the message is sent. It breaks the message into *flits* – the largest number of bits that can be transmitted through a channel simultaneously – and sends the flits directly to the destination in pipeline fashion. If a communication channel is not immediately available, the message waits until it can allocate the channel. When relatively few messages are in the network, the message latency is dominated largely by the product of the message's length and the time to transmit one flit. Wormhole routing is not

only often faster than store-and-forward routing, but has no need for storage buffers at each intermediate node. Wormhole routing is usually implemented at the circuit level – hence it is known as a *circuit-switched* strategy.

One problem with the two routing strategies is that they can allow processes to hold some resources while waiting for other resources. (For store-and-forward routing strategies, the processes are messages and the resources are buffers. For wormhole strategies, the processes are also messages, but instead the resources are channels.) This holding of resources makes it possible for *deadlock* to occur. A group of two or more processes may try to allocate resources the others hold, creating a *cycle of dependency*, in which each process makes no progress.

It is important that a routing algorithm does not allow deadlock to occur. Some routing algorithms are already inherently deadlock-free and can be used unmodified. Other algorithms allow cycles of dependency. These algorithms must be modified to prevent deadlock from ever occurring.

One way to show that a wormhole routing algorithm is deadlock-free is to build a *channel dependency graph.* A channel dependency graph D consists of a set C of vertices, one vertex for each unidirectional channel of the network, a set E of edges, where for $c_1, c_2 \in C$ we have $(c_1, c_2) \in E$ iff c_1 and c_2 are consecutive channels in at least one of the routing algorithm's routing paths. Then a test on D will tell if G is deadlock-free:

Theorem 7.2.1 *[46] A routing algorithm is deadlock-free iff its channel dependency graph is acyclic.*

An acyclic channel dependency graph prevents the messages from creating a cycle of dependency. Since the channels of an acyclic channel dependency graph

are nodes in an DAG, they can be topologically sorted. The theorem below then follows:

Theorem 7.2.2 *[46] A routing algorithm is deadlock-free iff its channels can be assigned monotonic ordering so that the routing algorithm allocates the channels in strictly increasing or decreasing numerical order.*

The hypercube's standard left-to-right bit correction algorithm is deadlock-free. This algorithm always uses the channels in the first dimension before channels in the second, and so forth. Since the algorithm never routes using more than one channel in each dimension, the channels can be ordered by their dimension (called *dimension-ordered routing*). This "dimensional" monotonic ordering makes routing deadlock-free.

This same left-to-right bit correction scheme also works for *LTLE* networks:

Theorem 7.2.3 *Algorithm LeftRightBitCorrectRoute in Figure 7.1 is deadlock-free, but not always minimal for all* LTLE *networks.*

Proof: The arguments follow those for the hypercube. At most one channel in dimension i is routed across, and the channels are always allocated in order of increasing dimension. This ordering makes the algorithm deadlock-free. ∎

It should be obvious that the same algorithm will also work on *LTDM* networks without modification.

The algorithm LeftRightBitCorrectRoute always produces minimal paths for only a *very* small subset of the *LE* networks, including the the hypercube. Is it possible for *LE* networks to have a wormhole routing algorithm that is both minimal and deadlock-free? Unfortunately, the answer is "no".

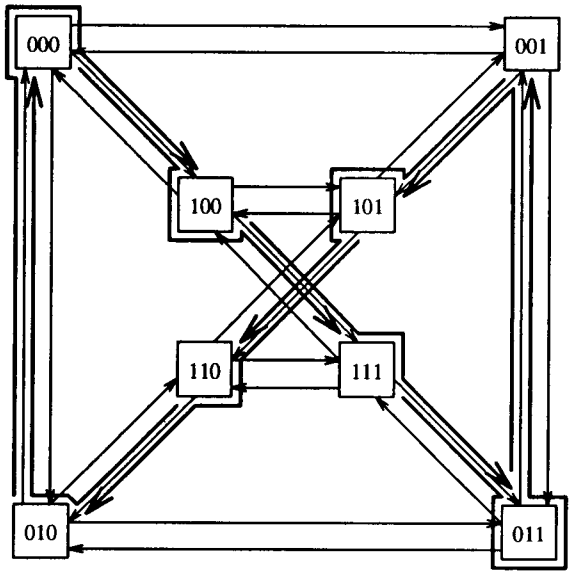

Figure 7.3. A cycle of dependency in the Twisted 3-Cube.

Theorem 7.2.4 *The Twisted 3-Cube has no minimal and inherently deadlock-free routing algorithm.*

Proof: The Twisted 3-Cube contains eight pairs of nodes that are each joined by only one path of length two. All other paths between them are longer. Any minimal routing algorithm will always choose this unique shortest path, if given one of these node pairs as source and destination. These paths are shown in Figure 7.3.

The union of these paths forms a cycle of length eight. Any minimal routing algorithm on the Twisted 3-Cube can have this as a cycle of dependency, if all nodes on the cycle simultaneously route messages to the nodes that are distance two away on the cycle. ∎

This excludes a large number of networks from the possibility of having a deadlock-free routing algorithm:

Theorem 7.2.5 *The Möbius Cubes, the Twisted Cube, the Twisted Hypercube, the Twisted N-Cube, the Flip MCube, the Crossed Cube, the Bent Cube and the Generalized Twisted Cube all have no deadlock-free routing algorithm.*

Proof: Consider the Twisted N-Cube first. It has a Twisted 3-Cube embedded at the single pair of twisted channel. There are alternate routes for the paths in the dependency cycle, but to step out of the Twisted 3-(sub)cube requires at least five steps – one step to jump out of the sub-network, three steps to correct the Twisted 3-Cube's bits, and one step to jump back in. This is longer than the minimal path's two steps.

Now consider the rest of the networks (assumed to be *LTLE* networks). All of these networks have the Twisted 3-Cube as a sub-network if we examine the last three components of the address vectors. Any alternate path between nodes of the sub-network requires at least three steps – one to set a component in dimension $1 \leq i \leq n-3$, one to correctly set the last three components, and one to reset the component in dimension $1 \leq i \leq n-3$. Again this is not minimal. So none of these networks have a routing algorithm that is minimal and deadlock-free ∎

Though the *LE* networks do not have minimal and deadlock-free routing algorithms, this does not exclude them from being used in wormhole routing architectures. For example, other networks whose minimal routing algorithms are not deadlock-free include the toroidal mesh and the k-ary n-cubes. These networks are still popularly used, despite the potential for deadlock.

7.2.2. Virtual Channels

In a network that has no inherently deadlock-free algorithm, there are two ways to make the algorithm deadlock-free. The first way is to examine the channel dependency graph and delete one edge from every cycle in the graph. This method is used to prevent deadlock in the k-ary n-cubes [46].

Unfortunately, this means the routing algorithm will need to be changed, so that it never uses any routing paths that use the removed dependency. It also means that for some networks, the lengths of some routing paths must be made non-minimal.

The second approach to removing deadlock from a routing algorithm involves adding more channels to the network, so that cycles in the channel dependency graph can be avoided. The channels are added between nodes that are already connected by a channel, so that routing distances and the connectivity of the network are unchanged. This approach will make the network a multigraph instead of a graph.

This approach has two advantages. First, the algorithm's routing paths are unchanged, so that the network still has the same characteristics. Second, the network does not need to have the additional channels physically added. Instead, they can be treated as *virtual channels* and be multiplexed across a single physical channel. The virtual channels take turns at using the physical channel, so at each time step, only one virtual channel sends a flit across the physical channel. In addition, multiplexing will need only a minimal amount of additional hardware.

The throughput of a multiplexed channel is an issue in network design. If we allow a maximum of m virtual channels per physical channel, and k of those virtual channels are allocated, then the multiplexer must alternate control between each of the k allocated channels. With a channel bandwidth of W, the effective bandwidth

from \ to	000	001	010	011	100	101	110	111
000	-	3, 1	2, 1	2, 1	1, 1	1, 1	2, 2	1, 1
001	3, 1	-	2, 1	2, 1	1, 1	1, 1	1, 1	2, 2
010	2, 1	2, 1	-	3, 1	2, 2	1, 1	1, 1	1, 1
011	2, 1	2, 1	3, 1	-	1, 1	2, 2	1, 1	1, 1
100	1, 1	1, 1	1, 1	2, 2	-	3, 1	2, 1	2, 1
101	1, 1	1, 1	2, 2	1, 1	3, 1	-	2, 1	2, 1
110	1, 1	2, 2	1, 1	1, 1	2, 1	2, 1	-	3, 1
111	2, 2	1, 1	1, 1	1, 1	2, 1	2, 1	3, 1	-

Table 7.5. A wormhole routing table for the Twisted 3-Cube.

of each virtual channel is W/k. The throughput of any message using the the channel then also drops to W/k, affecting the throughput of the message through other channels in its path. We must choose how to add and use virtual channels carefully, and we must consider how they affect the throughput of messages in the network.

We can use additional channels to make routing on the Twisted 3-Cube deadlock-free. If we allow virtual channels, then a maximum of 2 virtual channels per physical channel are needed to make an minimal and deadlock-free routing algorithm for the Twisted 3-Cube.

The output of a Twisted 3-Cube routing algorithm is a list R of ordered pairs (i, c) of neighbors i and channels c. For instance, if the first element of R is (i, c) the first step of the routing path uses virtual channel c between $\vec{X}$ and $\vec{X} + B_i^{(A\vec{X})_i}$.

Table 7.5 shows, for each current node and destination node, the number of the neighbor route to and the number of the virtual channel to route on. The only time the second virtual channel is used is for any exceptions to the left-right bit

correction algorithm, as shown in Table 7.2, so only dimension 2 requires virtual channels.

The routing algorithm encoded on this table correctly and minimally routes without deadlock on the Twisted 3-Cube. There are a small number of source and destination address pairs, that is, 2^6 pairs, so it is trivial to verify all the paths are correct and minimal. The routing paths of the algorithm will always allocate channels in the order: $(2,2)$, $(1,1)$, $(2,1)$, $(3,1)$. Again, this can be shown by verifying all paths. By Theorem 7.2.2, this network is deadlock-free.

This approach can be generalized to *LTLE* networks of higher dimensions, by examining only 3 adjacent components of the routing address at a time. The distributed routing algorithm appears in Figure 7.4. Notice that it is a slightly modified version of Algorithm ThreeBitLookaheadRoute.

Theorem 7.2.6 *Algorithm WormHoleThreeBitLookaheadRoute is correct, minimal and deadlock-free, and has a distributed run time of $O(n)$ bit operations and a total run time of $O(n)$ bit operations.*

Proof: The algorithm is correct and minimal by arguments given in Theorem 7.1.4.

The algorithm is deadlock-free because the network's channels can be grouped into the following order:

$$(2,2),(1,1),(3,2),(2,1),(4,2),(3,1),\ldots,(n,2),(n-1,1),(n,1)$$

and any path generated by the algorithm will use the channels in this order.

The algorithm has a distributed run time of $O(n)$ bit operations and a total run time of $O(n)$ bit operations, again by arguments given in Theorem 7.1.4. ∎

The channel utilization for Algorithm WormHoleThreeBitLookaheadRoute will be the same as the channel utilization for Algorithm ThreeBitLookaheadRoute,

Input: An n-dimensional *LTLE* network $G = (B^0, B^1, A)$, a source address $\vec{X}$, a destination address $\vec{Y}$ and the current address $\vec{W}$.

Output: If the message needs to be forwarded, the number of the neighbor and the number of the virtual channel to route the message to. If the message is at its destination, a signal to accept the message at the current node.

```
WormHoleThreeBitLookaheadRoute( G, X, Y, W )
begin
    if W = Y then
        return "Accept"
    else
        i ← min{k : W_i + Y_i = 1}
        if i ≤ n − 2 then
            U ← W + B_{i+1}^{(AW)_{i+1}}
            V ← U + B_i^{(AU)_i}
        end if
        if 1 ≤ i ≤ n − 2 and B_{i+1}^0 ≠ B_{i+1}^1 and (AB_i^{(AW)_i})_{i+1} = 1
        and V_i = Y_i and V_{i+1} = Y_{i+1} and V_{i+2} = Y_{i+2} then
            return ⟨i + 1, 2⟩
        else if 1 ≤ i ≤ n and B_i^0 ≠ B_i^1 and (AB_{i−1}^{(AW)_{i−1}})_i = 1 ) then
            return ⟨i, 1⟩
        else
            return either ⟨i, 1⟩ or ⟨i, 2⟩
        end if
    end if
end procedure
```

Figure 7.4. Algorithm WormHoleThreeBitLookaheadRoute.

because the two algorithms use the same routing paths. For the dimensions that use two virtual channels, the channel utilization of the second virtual channel will be 1/4 of the total channel utilization for the physical channel, because only 2 of the 8 possible conditions that can occur the algorithm will use the second virtual channel.

As in Section 7.1, the 3-bit lookahead algorithm can be extended to a k-bit lookahead algorithm. At the current time, we have bounded the number of virtual channels to a number linear in n.

Theorem 7.2.7 *Let $G = (B^0, B^1, A)$ be a n-dimensional non-redundant* LTLE *network. A k-bit lookahead algorithm for that network will require at most* $\min(2k-1, n)$ *virtual channels per physical channel.*

Proof: We can always route k terms in any given order if the indices of the terms are unique (nonredundant) and we allow k channels per physical channel. This is because we can arrange the dimension/virtual channel pairs as:

$$\begin{array}{cccc} (1,1) & (2,1) & \cdots & (k,1) \\ (1,2) & (2,2) & \cdots & (k,2) \\ \vdots & \vdots & \ddots & \vdots \\ (k,1) & (k,2) & \cdots & (k,k) \end{array}$$

If the i-th step has index j, then we route on the i-th virtual channel of physical channel j. It is deadlock-free because it uses channels in strictly increasing order by the dimension/virtual channel pairs.

Let R be the ordered terms that describe the routing path between source node $\vec{X}$ and destination node $\vec{Y}$, and let the index of the R_i-th step be j. The routing path R can always be broken into an ordered set of one or more sub-paths or "chains" C_1, C_2, ... so that:

$$\forall t_1 \in C_u, \forall t_2 \in C_v : u < v \Rightarrow t_1 < t_2$$

The length of any chain of terms is at most $k-1$ terms long, because there can be simultaneously at most $k-2$ terms that are either before R_i and have an index greater than j, or are after R_i and have an index less than j.

Further, the terms in a chain can have indices that differ by at most $k-1$, because the terms before R_i with index greater than j can only have indices between $j+1$ and $j+k$.

For each physical channel across dimension i, allocate $2k-1$ virtual channels numbered from $\max(1, i-k+1)$ to $\min(i+k-1, n)$. These dimension/virtual channel pairs can be ordered with dimension major, virtual channel minor. For instance, a network with $n=7$ and $k=3$, we have the channels ordered as:

$$\begin{array}{ccccccc}
(1,1) & (2,1) & (3,1) & & & & \\
(1,2) & (2,2) & (3,2) & (4,2) & & & \\
(1,3) & (2,3) & (3,3) & (4,3) & (5,3) & & \\
 & (2,4) & (3,4) & (4,4) & (5,4) & (6,4) & \\
 & & (3,5) & (4,5) & (5,5) & (6,5) & (7,5) \\
 & & & (4,6) & (5,6) & (6,6) & (7,6) \\
 & & & & (5,7) & (6,7) & (7,7)
\end{array}$$

(We break the channels into rows so the pattern is more clear.)

For any chain C, the last term $t_{|C|}$ will have the smallest index j (or the chain can be broken into two smaller chains). For $t_{|C|}$, assign the physical channel/virtual channel pair to be $(j, j+k-1)$. Then the terms $t_1, t_2, \ldots, t_{|C|-1} \in C$ can be routed deadlock-free no matter what order they appear in, because we have the channels in the following order:

$$\begin{array}{ccc} (j+1,j), & \cdots, & (j+k-1,j), \\ \vdots, & \ddots, & \vdots, \\ (j+1,j+k), & \cdots, & (j+k-1,j+k) \end{array}$$

As noted above, this is enough to route using all the terms with indices $j+1$ through $j+k$ in any order, without deadlock, before we route using a term with index j.

No two chains will not use the same physical channel/virtual channel pairs, because all the terms in one chain can be used as a routing step before any term in a following chain, and chains are always used as routing sub-paths in increasing order. Thus the algorithm is deadlock-free, because the chains will always use the channels in a strictly increasing order of dimension/virtual channel pairs.

So far, this approach still uses virtual channels numbered 1 through n. We can reduce this to at most $2k-1$ virtual channels by noting that we can assign the virtual channels modulo $2k-1$ without channel conflict. ∎

Theorem 7.2.7 only shows an upper bound on the number of virtual channels. For instance, the three-bit lookahead algorithm needs only two virtual channels and not three as the theorem suggests.

7.2.3. Minimal Wormhole Routing Algorithms

For the Twisted 3-Cube, only two virtual channels are needed to make the minimal routing algorithm deadlock-free. For larger LE networks, more virtual channels may be needed to make a minimal routing algorithm deadlock-free. By setting $k = n$ in Theorem 7.2.7, we can see that at most n virtual channels are needed. However, we may not need n virtual channels to make the minimal routing algorithm deadlock-free.

There are some special cases where minimal and deadlock-free routing is trivially possible. The algorithm LeftRightBitCorrectRoute is clearly minimal for the hypercube network. It is also trivially deadlock-free, by Theorem 7.2.3.

Another trivial algorithm uses the Twisted 3-Cube to create a minimal and deadlock-free routing algorithm for at least the Generalized Twisted Cube.

Theorem 7.2.8 *The Generalized Twisted Cube has a deadlock-free minimal routing algorithm that uses at most two virtual channels.*

Proof: In its definition, the Generalized Twisted Cube of dimension n is constructed by graph composition of $\lfloor n/3 \rfloor$ Twisted 3-cubes and at most one hypercube of dimension $n \bmod 3$. We can then route each Twisted 3-Cube sub-network using the routing table in Table 7.2, and the remaining hypercube using Algorithm LeftRightBitCorrectRoute. If we route through these composed graphs in a fixed order (say, from the smallest indexed one to the largest), then no more than 2 virtual channels are needed. ∎

Now consider the more general case of a *LTLE* network $G = (B^0, B^1, A)$. The algorithm for this case is more complicated than the algorithm for the Generalized Twisted Cube. We cannot use the same trick of decomposing the network into trivially routed sub-networks.

One problem in designing a deadlock-free minimal algorithm comes from the ordering of channels in the routing path. The minimal routing algorithm must sometimes route along the dimensions out-of-order. We have found no simple way restrict the possible orderings of the terms in the routing path, and so cannot put a bound on the number of virtual channels.

The simplest approach to modifying the minimal routing algorithm to be deadlock-free is to allow an arbitrary number of virtual channels. The simplest

Input: An n-dimensional *LTLE* network $G = (B^0, B^1, A)$, a source address $\vec{X}$, a destination address $\vec{Y}$ and a minimal routing path R from $\vec{X}$ to $\vec{Y}$.

Output: A wormhole routing path $\check{R}$, with neighbor and virtual channel number.

```
NaiveWormholeMinimalRoute( G, X⃗, Y⃗, R )
    i ← 1
    for each t ∈ R do
        Ř ← Ř + ⟨t, i⟩
        i ← i + 1
    end for
end procedure
```

Figure 7.5. Algorithm NaiveWormholeMinimalRoute.

method is to put each successive step of a routing path onto a different virtual channel. This is Algorithm NaiveWormholeMinimalRoute, shown in Figure 7.5.

Theorem 7.2.9 *Algorithm NaiveWormholeMinimalRoute is correct, minimal and deadlock-free and uses D_G virtual channels, where D_G is the diameter of the network.*

Proof: The algorithm is clearly correct and minimal, because it uses only the terms in R, a correct and minimal legal expansion.

The algorithm will use exactly D_G virtual channels because it uses a different virtual channel on each routing step of a routing path, and D_G is the length of the longest minimal path in G. The channels of the network can be identified as ordered pairs of dimension and channel number, as below:

$$(1,1),\ldots,(n,1),(1,2),\ldots,(n,2),\ldots,(1,D_G),\ldots,(n,D_G)$$

The list is ordered by channel number major, dimension minor. The algorithm allocates channel numbers in strictly increasing order, so the channels will be allo-

cated in left-to-right order from the list above. By Theorem 7.2.2, this is enough to guarantee that NaiveWormholeMinimalRoute is deadlock-free. ∎

There are problems with the algorithm NaiveWormholeMinimalRoute. The virtual channels will not be evenly utilized. For instance, the channels in dimension 1 will have their virtual channel 1 heavily utilized, because channel 1 is used most often as the first routing step in a path.

Since the diameter of the *LE* networks is (to some measure) a function of the dimension, using this algorithm means that increasing the dimension of the network will force an increase in the number of virtual channels per physical channel. This increase will require that the complexity of the hardware also increase.

One partial solution would be to find a routing algorithm that requires at most a constant number of virtual channels, independent of the network diameter. The Generalized Twisted Cube has already been shown to need only 2 virtual channels. Are there other network algorithms that use a constant number of channels? Unfortunately,we have not yet found a satisfactory solution for this problem.

7.3. SUMMARY

We have shown that the hypercube routing algorithm can be implemented on *LTLE* networks, and that it will cause *LTLE* networks to behave similar to the hypercube in performance. We have also developed an algorithm which is based on the hypercube routing algorithm, but which "looks ahead" three bits to see if there is a locally shorter path to route on. This lookahead algorithm compares favorably to the hypercube routing algorithm, and captures much of the performance of the minimal routing algorithm. Both of these algorithms are interesting because they use a constant number of bit operations per node to compute the step in the routing

path. This is a big advantage over the minimal routing algorithms before, which required that the path be precomputed at the source node.

We also examined the idea of wormhole routing on *LTLE* networks. In particular, we showed that the hypercube routing algorithm was deadlock-free and that the three bit lookahead algorithm could be made deadlock-free by adding a second, virtual channel to each physical channel. This allowed us to design wormhole routing algorithms for *LTLE* networks. Unfortunately, the minimal *LTLE* network routing algorithm can (to date) only be made deadlock-free by using a number of virtual channels equal to the diameter of the network.

In later chapters, we will compare the behavior of the wormhole routing algorithms to see if they compare well to store-and-forward routing algorithms.

8. EMBEDDINGS AND EMULATIONS FOR *LE* NETWORKS

One reason for the hypercube's popularity is that it can efficiently emulate any bounded-degree network [45]. The hypercube can route between any two of its processors in n steps. This allows the hypercube to simulate a single communication step for any arbitrary network of 2^n processors in at worst n communication steps, ignoring message collisions. Because of this, the hypercube can emulate algorithms written for these other networks with at worst a logarithmic slowdown or dilation (in the number of nodes).

The hypercube also directly embeds a number of well-studied networks, including ring networks, meshes, k-ary n-cubes and binomial trees. It can also "almost directly" embed several other networks, in particular, the binary tree. Because it can embed these networks, the hypercube can simulate a single communication step on these networks with no dilation, or at worst constant dilation.

The embeddings of the hypercube extend to many *LE* networks, because these networks share many of the hypercube's properties. In this chapter, we discuss several embeddings that can be done on *LTLE* networks, including Hamiltonian cycles, binomial trees and binary trees. We discuss the emulation of *LE* networks on the hypercube and vice versa.

8.1. PREVIOUS RESULTS

The literature on twisted hypercube variants contains several results on the embeddings and emulations for the twisted cubes. These results have been presented on an individual, network-by-network basis and occasionally contain errors. A summary of the embedding of ring networks, binary trees, and binomial trees,

Network Name	Hamiltonian Cycle	Binary Tree	Binomial Tree	Hypercube Emulation
Hypercube	Yes	No	Yes	1
Twisted Cube [34]	Yes	(?)	Yes	2
Twisted N-Cube [31]	Yes	Yes	Yes	2
Multiply Twisted Cube [29] Crossed Cube [30]	Yes	(?)	Yes	2 ‡
Flip MCube [51]	Yes	(?)	Yes	4 (2 ‡)
0-Möbius Cube [23]	Yes	(?)	Yes	4 (2 ‡)
1-Möbius Cube [23]	Yes	(?)	Yes	4 (2 ‡)
Generalized Twisted Cube [12]	Yes	(?)	Yes	2
Twisted Hypercube [28]	Yes	Yes	Yes	2

‡: Factor of dilation without considering channel collisions of messages.

Table 8.6. The embedding of networks for hypercube variants, and the constant factor of dilation for hypercube emulation.

and a summary of the dilation for emulation of the hypercube on *LE* networks is summarized in Table 8.6.

Clearly there are a number of results that remain to be shown for *LE* networks.

8.2. EMBEDDINGS

Many parallel algorithms have a communications pattern that is different from the structure of the hypercube. For example, matrix multiplication and discrete-space simulation algorithms are based on 2-dimensional meshes. These algorithms can be used on the hypercube by embedding a mesh through Gray-encoding. Many other algorithms are based on the divide-and-conquer paradigm,

which uses either binary or binomial trees. These algorithms can also be used on the hypercube, by embedding these tree networks. If we are to use these algorithms on a *LE* network, it is important to us to know which topologies the network can embed.

We will examine a number of common hypercube-embedded topologies and show that they can be embedded into *LE* networks. For the most part, we limit our discussion to *LTLE* networks. Some of our embedding results cannot be extended to *LE* networks, because the embeddings depend heavily on the properties of *LTLE* networks. This not a significant problem for us, because all published *LE* networks are also *LTLE* networks.

A network embedding requires that all channels of the embedded network be mapped to disjoint paths or channels. No two channels of the embedded network can map to the same channel, or two messages may have to compete for this channels. These "channel collisions" can result in a longer communication delay and so are to be avoided.

There are several types of embeddings. The first is a *direct embedding*, where each node of the embedded network is mapped to a unique node, and all the channels of the embedded network map to a single channel. A *squashed embedding* has two or more nodes of the embedded network map to the same node. A *stretched embedding* has each channel of the embedded network map to a path of two or more channels.

We define the dilation of a network embedding or emulation to be the maximum number of communication or computation steps needed to simulate one communication or computation step for the embedded network. For direct embeddings, the dilation is 1. For squashed embeddings, the dilation is the maximum number of nodes in the embedded network that are mapped to one node, and for stretched

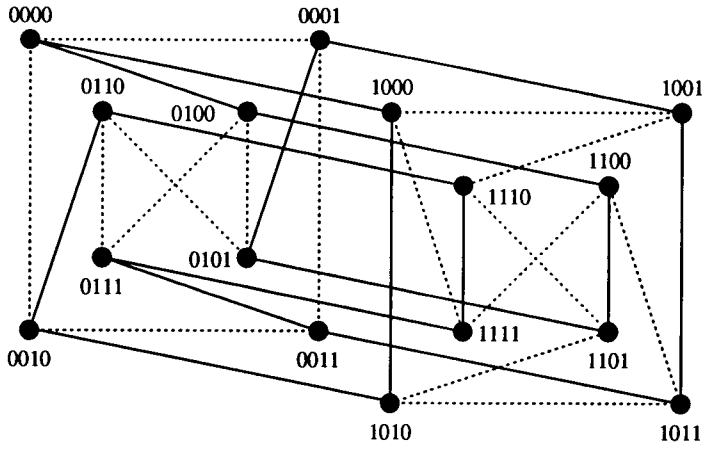

Figure 8.1. A Hamiltonian circuit on the 4-dimensional 0-Möbius cube.

embeddings, the dilation is equal to the length of the longest path that a channel in the embedded network is mapped to.

We examine in turn embeddings of ring networks, binomial and binary trees.

8.2.1. Hamiltonian Circuits and Ring Networks

If an n-dimensional *LE* network has a Hamiltonian circuit, then we can directly embed a ring network of length 2^n nodes into the network, and so use ring network algorithms on it. Theorem 8.2.1 shows that all *LTLE* networks have Hamiltonian circuits. It does so by showing that a Hamiltonian path with adjacent end points exists on every *LTLE* network.

Theorem 8.2.1 *Let* $G = (B^0, B^1, A)$ *be an* n*-dimensional* LTLE *network,* $n \geq 2$. G *contains a Hamiltonian circuit of length* 2^n.

Example: The Hamiltonian circuit for the 4-dimensional 0-Möbius cube is:

$$0000 \rightarrow 1000 \rightarrow 1111 \rightarrow 0111 \rightarrow 0100$$
$$\rightarrow 1100 \rightarrow 1011 \rightarrow 0011 \rightarrow 0010$$
$$\rightarrow 1010 \rightarrow 1101 \rightarrow 0101 \rightarrow 0110$$
$$\rightarrow 1111 \rightarrow 1001 \rightarrow 0001 \rightarrow 0000$$

and is shown in Figure 8.1. ∎

Proof: For a path P, let the path P^R denote the reverse of the path P. We show there is a Hamiltonian path H_n for G from the node w/ address $\vec{0}$ to the node B_n^0, where the path H_i is defined recursively by $H_i = H_{i-1}B_i^{\phi_i}H_{i-1}^R$ with $\phi_i \in \{0,1\}$ and $H_1 = B_1^0$. G then contains a Hamiltonian circuit, because $\vec{0}$ is adjacent to B_n^0.

First, we show by induction that H_i is a legal path, and that the end points of the path differ by $B_i^{\phi_i}$ for $\phi_i \in \{0,1\}$. Further, H_i contains 2^i unique nodes.

Trivially $H_1 = B_1^0$ is a path from $\vec{X}$ to $\vec{X} + B_1^0$. Further, it contains two uniquely addressed nodes.

Assume that H_{i-1} is a legal path of 2^i unique nodes, and that its end points differ by $B_{i-1}^{\phi_{i-1}}$ for $\phi_{i-1} \in \{0,1\}$. Then recursively follow the channels of H_{i-1} to reach $\vec{X}+B_{i-1}^{\phi_{i-1}}$. Only channels in dimensions 1 through $i-1$ are used in H_{i-1}. Route to the i-th neighbor. Because G is a *LTLE* network, no term $B_{1\leq j\leq i-1}^{\phi_j}$ depends on $B_i^{\phi_i}$, so we can trace the path H_{i-1}^R to get from $\vec{X} + B_{i-1}^{\phi_{i-1}} + B_i^{\phi_i}$ to $\vec{X} + B_i^{\phi_i}$. The first 2^{i-1} nodes are each visited only once by induction, as are the last 2^{i-1} nodes. Because every node in the first half of the path differs from every node in the second half by B_n^0, all 2^i nodes in the path H_i are visited only once.

Now we show a Hamiltonian cycle exists.

The path H_n will route from $\vec{X}$ to $\vec{X} + B_n^{\phi_n}$. Because the path H_n connects all 2^n unique nodes, it is a Hamiltonian path. Because G is a *LTLE* network,

$B_n^0 = B_n^1 = B_n^{\phi_n}$. Thus $\vec{X}$ is always adjacent to $\vec{X} + B_n^{\phi_n}$, and so a Hamiltonian cycle exists. ∎

From the proof above, we can show that a number of disjoint circuits with smaller lengths exist in the *LTLE* networks and so several rings can be simultaneously embedded in the *LTLE* networks.

Corollary 8.2.1 *Let $G = (B^0, B^1, A)$ be an n-dimensional* LTLE *network. G contains 2^{n-k} disjoint circuits of length 2^k.*

Proof: By the corollary of Theorem 3.3.2, G contains k disjoint *LTLE* networks, each of dimension $n - k$. By Theorem 8.2.1, a Hamiltonian circuit of length 2^{n-k} can be embedded into each of these networks.∎

Any two nodes in the same Hamiltonian cycle will have all of the first k components match, and any two nodes in different Hamiltonian cycles will have at least one of the first k components differ, so we can number the Hamiltonian cycles by the first k components of their nodes, and we tell which Hamiltonian circuit given node is in, simply by examining its address.

8.2.2. Binomial Trees

Another network that directly embeds into a *LE* network is the binomial tree. The binomial tree (not the binary tree) is a useful graph structure that is often embedded into the hypercube for divide-and-conquer parallel algorithms.

First, we define the binomial tree.

Definition 8.2.1 *([10]) The binomial class B_k of ordered trees is defined as:*

1. *Any tree of a single node is a B_0 tree.*

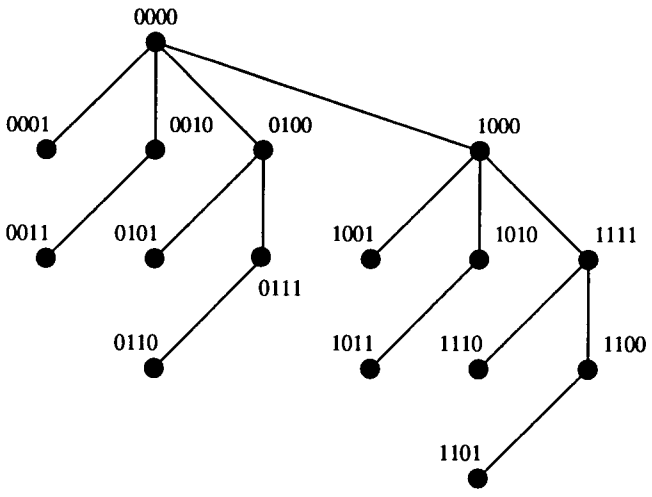

Figure 8.2. The binomial tree of order 4.

2. *If $\vec{X}$ and $\vec{Y}$ are disjoint B_{k-1} trees for $k \geq 1$, then the tree obtained by adding an edge to make the root of $\vec{Y}$ become the rightmost offspring of the root of $\vec{X}$ is a B_k tree. All binomial trees of order k are isomorphic in the sense that they have the same topology.*

The binomial tree B_4 appears in Figure 8.2. The binomial tree is aptly named, because the number of vertices at each distance from the root form a binomial distribution. It should be clear that for any $k \geq 0$, a B_k tree has 2^k nodes.

We can directly embed binomial trees into the *LTLE* networks, as shown in Theorem 8.2.2 below. The theorem depends on the decomposition of a *LTLE* network into two smaller networks.

Theorem 8.2.2 *Let $G = (B^0, B^1, A)$ be an n-dimensional* LTLE *network. Every node in G is the root of at least one binomial tree B_n.*

Proof: The proof is by induction on the dimension n of the network.

Base Case: For $n = 0$, the only 0-dimensional network G is a single node. The B_0 tree can be trivially embedded in G, because it is also a single node.

Inductive Hypothesis: Assume that any $(n-1)$-dimensional *LTLE* network $\check{G}$ has every node as the root of at least one B_{n-1} tree.

Inductive Step: Any node $\vec{X}$ in G has $\vec{X} + B_1^{(A\vec{X})_1}$ as its neighbor in the first dimension. By Theorem 3.3.2, G can be decomposed into two disjoint *LTLE* networks G_0 and G_1, which each have dimension $n-1$. Clearly either $\vec{X}$ lies in G_0 and $\vec{X} + B_1^{(A\vec{X})_1}$ lies in G_1, or vice versa. By induction, $\vec{X}$ is the root of a B_{n-1} tree in G_0 and $\vec{X} + B_1^{(A\vec{X})_1}$ is the root of a B_{n-1} tree in G_1. Because these two B_{n-1} trees are joined at the root, $\vec{X}$ is the root of a B_n tree. ∎

Binomial trees are used extensively in hypercube networks for a large class of parallel divide-and-conquer algorithms. This class of problems can also be efficiently computed on a *LTLE* network G. Assume that a problem P can be solved by dividing P into two equally sized subproblems P_1 and P_2 and then inductively solving P_1 and P_2. If we map a B_n tree into G with root $\vec{X}$ and place P at $\vec{X}$, then the subproblem P_1 can remain at $\vec{X}$ while other subproblem P_2 can be transmitted to $\vec{X} + B_1^{(A\vec{X})_1}$. The subproblems can then be recursively solved in parallel on the two B_{n-1} trees rooted at $\vec{X}$ and $\vec{X} + B_1^{(A\vec{X})_1}$, and the solutions recombined at $\vec{X}$. Each problem division and re-combination takes only one communication step, for a total of $2n$ communication steps.

8.2.3. Binary Trees

There are standard parallel divide-and-conquer algorithms that do not use a binomial tree structure. Instead, they have the parent node do divide and combine

operations in parallel while two child nodes recursively solve the sub-problems. For these algorithms, the ideal embedding is a binary tree.

We define the complete binary tree class as:

Definition 8.2.2 *The class T_k of complete binary trees is defined as:*

1. *Any tree of a single node is a T_0 tree.*

2. *If $\vec{X}$ and $\vec{Y}$ are disjoint T_{k-1} trees for $k \geq 1$, then the tree obtained by adding a root node R and one edge from R to the root of $\vec{X}$ and a second edge to the root of $\vec{Y}$ is a T_k tree. All full binary trees of order k are isomorphic in the sense that they have the same topology.*

It is clear that a T_k tree has $2^{n+1} - 1$ nodes, because it is constructed from two T_{k-1} trees, plus a root node. A T_k tree also has 2^d nodes that are a distance d from the root.

The binomial tree B_n has a squashed embedding of a binary tree T_n. [45]. This squashed embedding is achieved by mapping each parent node and its left child to the same node. This method maps n nodes of the binary tree to the root of the binomial tree.

There are several problems with this squashed embedding. Since every left child and its parent are mapped to the same node, the parent and child nodes cannot do computations simultaneously. This means that the squashed binary tree embedding has dilation n, because the root node must emulate n nodes. This result is not great, but it shows that a complete binary tree of $2^{n+1} - 1$ nodes can be squashed embedded on any *LTLE* network.

We now consider a direct embedding of a binary tree into a hypercube network. A T_{n-1} tree has $2^n - 1$ nodes. It might be thought that a T_{n-1} tree can be

embedded into an n-dimensional hypercube Q_n, but this is not the case, as can be shown by an red-black node coloring argument [45]. Color the nodes of Q_n with two colors, red and black, so that no two adjacent nodes have the same color. Then color the nodes of T_{n-1} the same way. Clearly all the nodes distance d from the root must all have the same color. The number of nodes of one color in the binary tree are either too large or too small to match the number of same-colored nodes in the hypercube.

At best, a T_{n-1} tree can be mapped into Q_n using a stretched embedding. This can be done by using a *double rooted* binary tree. Instead of a single node of degree 2 at the root, the double rooted T_{n-1} tree has 2 joined roots of degree 2. A communication between the two main subtrees then takes an extra routing step, which means that a T_{n-1} tree can be simulated on Q_n with a dilation of 2 [45].

Though the network Q_2 can trivially contain a T_1 tree, Q_3 cannot contain a T_2 tree. Oddly, the Twisted 3-Cube can directly embed a T_2 tree, as in Figure 8.3. Since the hypercube is trivially an LE network, it shows that not all LE networks can directly embed a binary tree.

Can we generalize the Twisted 3-Cube's embedding of a T_2 tree? Chedid and Chedid [12] attempted to show that a T_{n-1} tree can be embedded in an n-dimensional Generalized Twisted Cube. Their proof is incorrect.

They note that a complete T_2 tree of 7 nodes can be embedded in a Twisted 3-Cube, as in Figure 8.3. They attempt to construct a complete binary tree embedding by inductively embedding two trees in two disjoint $(n-1)$–dimensional sub-networks. They add a new root which joins these subtrees into a single tree, as in Figure 8.4.

The construction fails because it assumes that there will always be an unused node adjacent to the root of at least one subtree. This assumption is true in the base case. But in the inductive step, when the new root is inserted that joins two

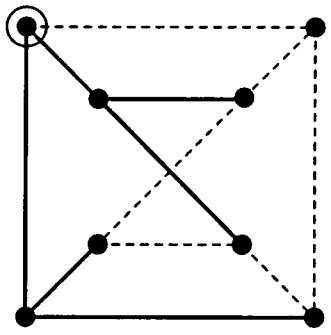

Figure 8.3. A Twisted 3-Cube with an embedded 7 node binary tree. The tree is indicated by the solid lines, and the root is circled.

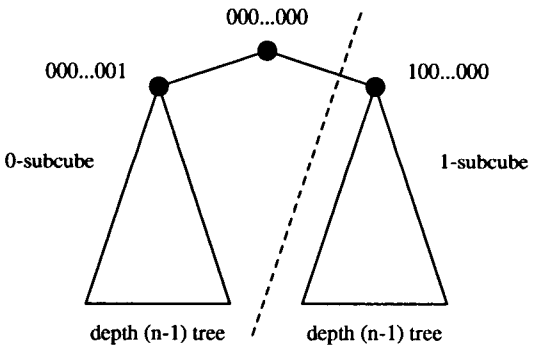

Figure 8.4. The inductive step of Chedid and Chedid's proof to embed a binary tree.

inductively constructed T_{n-1} trees, the proof fails to show that the remaining unused node of TQ_n is now adjacent to the root of the embedded T_n tree.

This error can be seen in the 6-dimensional cube example given in their Figure 11. When we filled in the addresses of the nodes that Chedid and Chedid claim should be in the unexpanded subtrees at the bottom of their figure, we found that these subtrees contain nodes that were already used to connect the subtrees together.

If we don't mind wasting a large number of network nodes, we can use the embedding of the T_2 tree into the Twisted 3-Cube (in Figure 8.3 as a basic unit for embedding a $T_{\lfloor 2n/3 \rfloor}$ tree into an n-dimensional Bent Cube or Generalized Twisted Cube network. Every three dimensions of the Bent Cube or Generalized Twisted Cube network allow us to add 2 levels of depth to a binary tree. This embedding uses a total of $2^{\lfloor 2n/3 \rfloor + 1} - 1$ nodes – a fairly small fraction of the total number of nodes in the network. However, we can do better.

The Twisted N-cube of Estafahanian *et al.* [31] can directly embed at least one T_{n-1} tree. The construction of this embedding is exactly the same as for embedding the double rooted binary tree into the hypercube, but with one change. One ordinary hypercube channel and one of the network's two twisted channels are used to connect the two binary subtrees to one root. In fact, this network was designed to embed a binary tree.

We can use a construction similar to the one given for Twisted N-cube to directly embed a T_{n-1} tree in at least one LE network. This construction starts with a double-rooted binary tree directly embedded into an ordinary hypercube $Q_n = (\mathbf{I}, \mathbf{I}, \mathbf{0})$. We use the embedding given in [45]. This construction is illustrated in Figure 8.5. It works by embedding double-rooted binary trees in the two $(n-1)$-dimensional sub-networks, then joining them at the roots. Channels along dimension

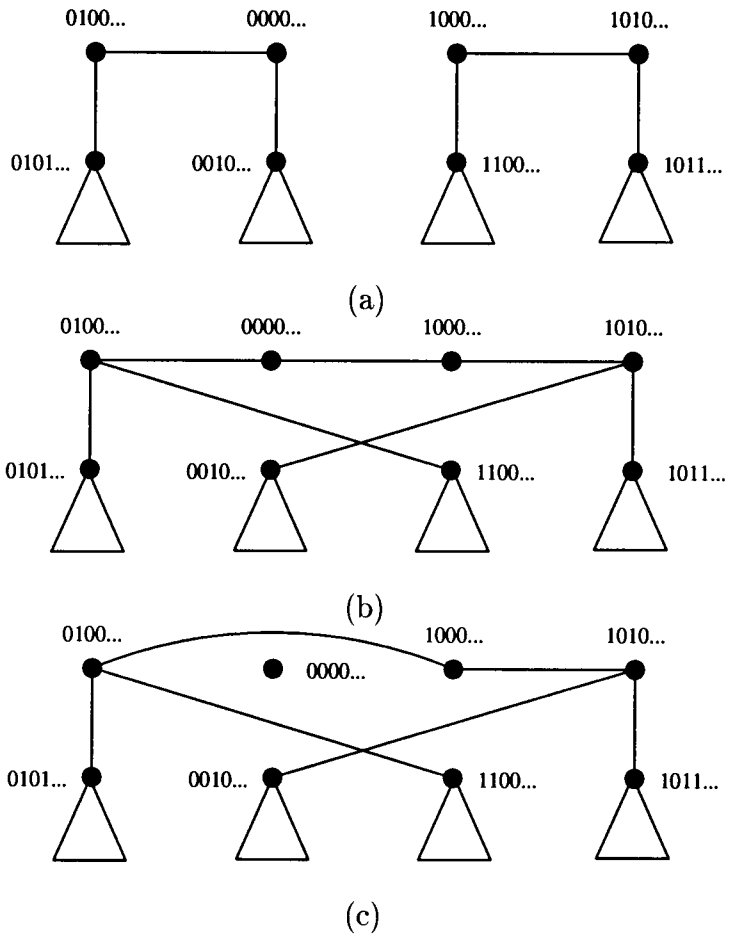

Figure 8.5. Construction of a single-rooted T_n tree from two double-rooted T_{n-1} trees on a *LE* network.

k on the hypercube are not used below level $k+2$ on the double-rooted binary tree, which means that dimension 2 channels will not be used below level 4 of the tree.

We then modify the definition of the hypercube Q_n to a new network G, so that $B_2^0 = e_1 + e_2$, and $row_2(A) = e_1 + e_3 + e_4$. Of the four channels of the binary tree that span dimension 2, the three channels $(0101\ldots, 0001\ldots)$, $(0010\ldots, 0110\ldots)$ and $(1011\ldots, 1111\ldots)$ remain unchanged, because $(A\vec{X})_2 = 1$ for each of these addresses. But the channel $(0000\ldots, 1000\ldots)$ no longer exists. We can remove the channels $(0000\ldots, 1000\ldots)$ and $(0000\ldots, 0100\ldots)$ and add the channel $(0100\ldots, 1000\ldots)$ to change the double-rooted binary tree into a T_{n-1} tree. The channels in levels 3 and 4 of T_n that cross dimension 2 are defined by B_2^1 and so are unchanged, and no channels in any other level of T_n are defined by B_2^0 or B_2^1. Thus T_n is embeddable in at least one *LE* network.

The general question of embedding a complete T_{n-1} tree into an arbitrary n-dimensional *LE* network is still unanswered. On the one hand, the hypercube is a *LE* network that cannot directly embed a binary tree. On the other hand, there is at least one *LE* network that does embed a binary tree. It is not known what the necessary and sufficient conditions are for an arbitrary network.

8.2.4. Meshes

Not all topologies that embed into the hypercube will embed as easily into *LE* networks. Mesh networks, which embed onto the hypercube by Gray-encoding, apparently cannot be directly embedded (in general) onto a *LE* network because the Gray-encoding no longer works.

If we have a *LTLE* network, we can embed the special case of a 2 by $2^n - 1$ mesh by noting that the i-th node in an embedded Hamiltonian circuit has the

$(n-i-1)$-th node as a neighbor in dimension n. This is because the second half of the circuit is the same as the first, with node addresses differing only in the last bit.

Cheng and Chuang [13] managed to show that their Varietal Hypercube (a.k.a. the Generalized Twisted Cube) can embed 2-dimensional meshes of size $2^p \times 2^q$ for $p+q=n$. Their proof rests on the fact that a Twisted 3-Cube can embed a 1×8 mesh or a 2×4 mesh, a 2-cube can embed a 1×4 mesh or a 2×2 mesh, and a 1-cube can embed a 1×2 mesh.

A Generalized Twisted Cube is built from the graph composition of $\lfloor n/3 \rfloor$ Twisted 3-Cubes and sometimes an additional n mod 3-Cube. Since each of these networks can embed a mesh, their graph composition can embed an arbitrary mesh of 2^n nodes and up to $\lceil 2n/3 \rceil$ dimensions, which can in turn embed a 2-dimensional mesh by Gray-encoding. Cheng and Chuang showed that the values of p and q in the dimensions above can be any two positive numbers that add to n.

It is not clear how we can extend this result to other *LE* networks. The problem is that their proof depends on the graph composition of networks to produce a Generalized Twisted Cube. Since most of the other networks are not produced by graph composition, this mesh embedding cannot be directly applied. The question of whether a *LE* network can directly embed a mesh network of arbitrary dimensions is still unanswered.

8.3. LE NETWORK EMULATIONS

In this section, we consider the emulation of one network by another. A network emulation is different from a network embedding. An embedding allows a mapping of each channel of one network to a path in another, but requires that these paths must be disjoint – they can contain no common nodes or channels. An

emulation allows each channel of the emulated network to be mapped to a path, but the set of set paths in the mapping do not necessarily have to be disjoint.

In an emulation two messages may attempt to use the same channel simultaneously, causing a *channel collision.* One message must wait while the other proceeds, possibly causing the emulation to have a larger dilation. In proving that an emulation can be done with a given dilation, we must consider the possibility of channel collisions.

Because the hypercube Q_n and an n-dimensional *LE* network G have the same number of nodes and the same node addressing scheme, we can write an algorithm for Q_n, then transfer it to G, replacing each of Q_n's communication steps with a series of communication steps on G. If it is possible to emulate a single hypercube routing step in a constant number of steps on G, then we could directly run hypercube algorithms on G with little modification and only a constant amount of "slowdown" or dilation. Conversely, we could execute *LE* network programs on a hypercube with constant dilation, by emulating each of the network's routing steps on the hypercube.

8.3.1. Emulating *LE* Networks on Hypercubes

How well a hypercube can emulate a *LE* network? As the theorem below shows, a hypercube can emulate a general *LE* network with a dilation at worst linear in n.

Theorem 8.3.1 *Let $G = (B^0, B^1, A)$ be an n-dimensional general* LE *network. The hypercube Q_n can simulate G with dilation n using the store-and-forward routing strategy.*

Proof: We assume a direct mapping of node addresses between Q_n and G, so that $\vec{X} \in Q_n$ maps to $\vec{X} \in G$.

A single term in G can have weight n, so that it takes a path of at least n steps in Q_n to emulate the single channel in G. We can emulate a single routing step of a single message in G, by using the hypercube's routing algorithm to correct the components one at a time.

There is at least one LE network (a trivial one) that must be emulated in at least n steps using our mapping. This is the network $G = (B^0, B^1, A)$ where $B_i^0 = B_i^1 = e_i$, except for $B_2^1 = e_2 + e_3 + \ldots + e_n$, and A is zeroed out, except for $A_{2,1} = 1$. A message that travels from $\vec{X}$ to $\vec{X} + B_1^2$ will require $n - 1$ steps to route on the hypercube. (We will ignore mapping isomorphisms of G to Q_n.)

We show a way to avoid channel collisions during the emulation of one routing step. We allow n synchronous routing steps on Q_n for each routing step of G. Consider a message routing along channel $(\vec{X}, \vec{X} + B_i^\phi)$ in G. At routing step j with $1 \leq j \leq n$, we route the message along dimension $((i + j - 2) \bmod n) + 1$ of Q_n iff $B^\phi_{i,((i+j-2) \bmod n)+1} = 1$. At the end of n steps the message will be at the correct destination. Two messages using $B_{i_1}^\phi$ and $B_{i_2}^\phi$ with $i_1 = i_2$ will not collide with each other, because the i-th neighbor function is one-to-one, and two messages using $B_{i_1}^\phi$ and $B_{i_2}^\phi$ with $i_1 \neq i_2$ will not collide, because they are routed in different dimensions of Q_n at each time step j. ∎

Though the hypercube cannot always emulate a general LE network with less than linear dilation, there are subclasses of LE networks that can be emulated with constant dilation.

Theorem 8.3.2 *Let $G = (B^0, B^1, A)$ be an n-dimensional* LTLE *network, where B^0 and B^1 are banded lower triangular matrices, and where k is the largest number*

such that $B^{\phi}_{i+k-1} = 1$ *with* $\phi \in \{0,1\}$ *and* $1 \leq i \leq n$. *The hypercube* Q_n *can simulate* G *with dilation* k *using the store-and-forward routing strategy.*

Proof: If B^0 and B^1 are banded lower triangular matrices, then the Hamming distance between any node $\vec{X}$ and $\vec{X} + B_i^{(A\vec{X})_i}$ is at most k. Q_n can simulate a single communication step in G in at most k steps. Consider a message routing along channel $(\vec{X}, \vec{X} + B_i^{\phi})$ in G. At routing step j with $1 \leq j \leq k$, we route the message along dimension $(i+j)-1$ of Q_n iff $(i+j)-1 \leq n$ and $B^{\phi}_{i,(i+j)-1} = 1$. Two messages using $B^{\phi}_{i_1}$ and $B^{\phi}_{i_2}$ with $i_1 = i_2$ will not collide with each other, because the i-th neighbor function is one-to-one, and two messages using $B^{\phi}_{i_1}$ and $B^{\phi}_{i_2}$ with $i_1 \neq i_2$ will not collide, because they are routed in different dimensions of Q_n at each time step j. ∎

Notice that most of the published *LTLE* networks have banded lower triangular B^0 and B^1 matrices. We can make the following inference about non-redundant *LTLE* networks:

Corollary 8.3.1 *Let* $G = (B^0, B^1, A)$ *be an non-redundant* n*-dimensional* LTLE *network, where for all* i, $B_i^0 + B_i^1 \in \{0, B_{i+1}^0, B_{i+1}^1, \ldots, B_n^0, B_n^1\}$. *Let* k *be the largest number of components with index* i *that all have* $B_i^0 + B_i^1$ *sum to* B_j^0 *or* B_j^1 *with the same index* j *where* $1 \leq i < j \leq n$. *The hypercube* Q_n *can simulate* G *with dilation* $k+1$ *using the store-and-forward routing strategy.*

Proof: By using an extension of Algorithm MinimumWeightIsomorphism in Figure 4.3, we can transform G into a network $\check{G} = (\check{B}^0, \check{B}^1, \check{A})$, where each $\check{B}_i^0$ and $\check{B}_i^1$ has weight of at most k. The network $\check{G}$ then has $\check{B}^0$ and $\check{B}^1$ be banded matrices with at most $k+1$ bands on and below the triangular *LE* . Then by Theorem 8.3.2, we can emulate $\check{G}$ on the hypercube with dilation k. ∎

This leads immediately to a conclusion about the dilation for emulating a number of published networks on the hypercube:

Corollary 8.3.2 *The Möbius Cubes, the Twisted Cube, the Flip MCube, the Generalized Twisted Cube, the Bent Cube and the YATC Cube of dimension n can all be emulated on the hypercube Q_n with dilation 2.*

Proof: Notice that all of these networks have $B_i^0 + B_i^1 \in \{0, B_{i-1}^0, B_{i-1}^1\}$ with $1 \leq i \leq n$. All but the Möbius cubes can be emulated by Theorem 8.3.2. The Möbius cubes can emulated by first using Algorithm MinimumWeightIsomorphism to an isomorphism where the matrix descriptions have B^0 and B^1 be banded lower triangular matrices with maximum weight 2. ∎

8.3.2. Emulating Hypercubes on *LE* Networks

If *LE* network family are going to be used, it is important that we show that *LE* networks can emulate the hypercube. The hypercube has a large number of parallel algorithms designed for its architecture. We would like to be able to use those algorithms directly on *LE* networks.

We have shown that the hypercube can emulate the *LE* network family with at worst linear dilation. The reverse, emulating Q_n on G, is somewhat more difficult to prove. The problem is that we have not proven a polynomial bound on the maximum number of steps needed to route in a *LE* network. Until we can, we cannot put a polynomial bound on the dilation, because the *LE* network may require an exponential number of communication steps to emulate one hypercube communication step.

Input: An n-dimensional *LTLE* network $G = (B^0, B^1, A)$, and a set of messages M.

```
Procedure HypercubeEmulate ( G, M )
begin
for each W⃗ ∈ Z_2^n do in parallel
    for each message M at W⃗ with source X⃗ and destination X⃗ + e_i do in parallel
        Send M from W⃗ to W⃗ + B_i^(AW⃗)_i
    end parallel
end parallel
for k = i + 1 to n
    for each W⃗ ∈ Z_2^n do in parallel
        for each message M at W⃗ with source X⃗ and destination X⃗ + e_i
        with i + k − 1 ≤ n do in parallel
            if W_{i+k−1} ≠ X_{i+k−1} then
                Send M from W⃗ to W⃗ + B_i^(AW⃗)_i
            end if
        end parallel
    end parallel
end for
end procedure
```

Figure 8.6. Algorithm HypercubeEmulate.

We can avoid this problem by (again) restricting the class of networks we use. An n-dimensional *LTLE* network can emulate the hypercube Q_n with at worst linear dilation, as shown in the theorem below:

Theorem 8.3.3 *Let $G = (B^0, B^1, A)$ be an n-dimensional* LTLE *network. Then G can emulate the hypercube Q_n with dilation n.*

Proof: Algorithm HypercubeEmulate effectively emulates one hypercube routing step, and is shown in Figure 8.6. Notice the similarity between this algorithm and Algorithm LeftRightBitCorrectRoute. This algorithm will use the same routing

paths as Algorithm LeftRightBitCorrectRoute, and so correctly routes messages from source to destination.

We assume that for a single communication step of the hypercube Q_n, there is at most one message crossing each unidirectional channel of Q_n. Since Algorithm HypercubeEmulate will use only channels in dimensions i through n of G to route from $\vec{X}$ to $\vec{X} + e_i$ on G, at worst n communication steps on G will be needed to emulate any routing step on Q_n.

Now we show that no two messages will collide.

Assume that message M_1 travels from $\vec{X}$ to $\vec{X}+e_i$ in Q_n and that message M_2 travel from $\vec{Y}$ $\vec{Y} + e_j$ in Q_n, where $i < j$. Let M_1 and M_2 be routed simultaneously at any time step k in the emulation algorithm. M_1 is routed from a node $\vec{X}^{(k)}$ to node:

$$\vec{X}^{(k+1)} = \vec{X}^{(k)} + B_{i+k-1}^{(A\vec{X}^{(k)})_{i+k-1}}$$

and M_2 gets routed from a node $\vec{Y}^{(k)}$ to node:

$$\vec{Y}^{(k+1)} = \vec{Y}^{(k)} + B_{j+k-1}^{(A\vec{Y}^{(k)})_{j+k-1}}$$

Message M_1 and M_2 will never have a channel collision, because they never route across the same neighbor at the same time step.

Now assume that message M_1 travels from $\vec{X}$ to $\vec{X} + e_i$ and that message M_2 travels from $\vec{Y}$ to $\vec{Y} + e_i$ in Q_n. The only way that M_1 and M_2 could cause a channel collision is if at some stage, they were routed to the same node $\vec{W}$ during the emulation algorithm.

From Theorem 3.3.2, G can be divided into 2^k sub-networks of 2^{n-k} nodes each. These sub-networks are differentiated by components 1 through k of the node addresses, and each node inside a sub-network has a unique address in components $k+1$ through n.

We state two loop invariants for Algorithm HypercubeEmulate and show that these remain true for the entire algorithm. The first invariant is that before stage k, M_1 is at some $\vec{X}^{(k)}$ and M_2 is at some $\vec{Y}^{(k)}$, where $\vec{X}^{(k)} \neq \vec{Y}^{(k)}$. The second invariant is that if before stage k, M_1 and M_2 are in the same $(n-k)$-dimensional sub-network, then:

$$\vec{X}^{(k)} + (\vec{X} + e_i) = \vec{Y}^{(k)} + (\vec{Y} + e_i)$$

These invariants hold before stage 1, because M_1 and M_2 are at nodes $\vec{X}^{(1)} = \vec{X} \neq \vec{Y}^{(1)} = \vec{Y}$ in the same n-dimensional cube G, and:

$$\vec{X}^{(1)} + (\vec{X} + e_i) = \vec{Y}^{(1)} + (\vec{Y} + e_i)$$

If after stage k, M_1 and M_2 are in different $(n-k)$-sub-networks, then $X_j^{(k)} \neq Y_j^{(k)}$ for at least some component with index $1 \leq j \leq k$. Then because Algorithm HypercubeEmulate will not affect indices 1 through k, M_1 and M_2 will not be in the same sub-network after step $k+1$, because their new addresses $\vec{X}^{(k+1)}$ and $\vec{Y}^{(k+1)}$ will also differ in index j.

If after stage k, M_1 and M_2 are in the same $(n-k)$-sub-network, then $X_j^{(k)} = Y_j^{(k)}$ for all $1 \leq j \leq k$. Either both messages M_1 and M_2 get routed in step $k+1$ or neither of them do, because the second invariant implies that for the $(k+1)$-th component:

$$(\vec{X}^{(k)} + (\vec{X} + e_i))_{k+1} = (\vec{Y}^{(k)} + (\vec{Y} + e_i))_{k+1}$$

So M_1 gets routed to node:

$$\vec{X}^{(k+1)} = \vec{X}^{(k)} + B_{i+k-1}^{(A\vec{X}^{(k)})_{i+k-1}}$$

and M_2 gets routed to node:

$$\vec{Y}^{(k+1)} = \vec{Y}^{(k)} + B_{i+k-1}^{(A\vec{Y}^{(k)})_{i+k-1}}$$

Since:

$$(A\vec{X}^{(k)})_{i+k-1} = (A\vec{Y}^{(k)})_{i+k-1}$$

we have $\vec{X}^{(k+1)} \neq \vec{Y}^{(k+1)}$ by mod 2 vector addition. So $\vec{X}$ and $\vec{Y}$ do not get routed to the same node $\vec{W}$. Also, if $\vec{X}^{(k+1)}$ and $\vec{Y}^{(k+1)}$ are in the same $(n-(k+1))$-dimensional sub-network, we have:

$$\vec{X}^{(k+1)} + (\vec{X} + e_i) = \vec{Y}^{(k+1)} + (\vec{Y} + e_i)$$

by mod 2 vector addition. So our loop invariants remain true in step $k+1$ and M_1 and M_2 do not collide. ∎

The above proof allows any *LTLE* network to emulate the hypercube with at worst linear dilation. We were unable to show that in general a *LTLE* network can emulate a hypercube with at most constant dilation. However, there is a restricted subset of networks that can emulate the hypercube with constant dilation, as the theorem below shows.

Theorem 8.3.4 *Let* $G = (B^0, B^1, A)$ *be an* n*-dimensional* LTLE *network. Let* $B_i^0 + B_i^1 \in \{0, B_{i+1}^0, B_{i+1}^1\}$ *and let* $(AB_i^0)_{i+1} = (AB_i^0)_{i+1} = 1$. *Then* G *can emulate the hypercube* Q_n *with dilation* 4.

Proof: The algorithm to generate the routing paths is Algorithm Limited-HypercubeEmulate, shown in Figure 8.7. For each Q_n routing step that a message takes from a node $\vec{X}$ to its i-th neighbor $\vec{X} + e_i$, G performs a series of routing steps along its channels to get the message to $\vec{X} + e_i$.

Input: An n-dimensional *LTLE* network $G = (B^0, B^1, A)$ where $B_i^0 + B_i^1 \in \{0, B_{i+1}^0, B_{i+1}^1\}$ and $(AB_i^0)_{i+1} = (AB_i^0)_{i+1} = 1$ with $1 \leq i \leq n$, a source node $\vec{X}$ and a hypercube dimension i to route in.

Procedure LimitedHypercubeEmulate (G)
for each $\vec{W} \in \mathcal{Z}_2^n$ do in parallel
  for each message M at $\vec{W}$ with source $\vec{X}$ and destination $\vec{X} + e_i$ do in parallel
    if $B_i^{(A\vec{X})_i} = e_i$ then
      Send M from $\vec{W}$ to $\vec{W} + B_i^{(A\vec{W})_i}$
    if $B_i^{(A\vec{X})_i} \neq e_i$ and $B_i^0 + B_i^1 \neq B_{i+1}^{(A\vec{X})_{i+1}}$ then
      Send M from $\vec{W}$ to $\vec{W} + B_i^{(A\vec{W})_i}$
    end if
  end parallel
end parallel
for each $\vec{W} \in \mathcal{Z}_2^n$ do in parallel
  for each message M at $\vec{W}$ with source $\vec{X}$ and destination $\vec{X} + e_i$ do in parallel
    if $B_i^{(A\vec{X})_i} \neq e_i$ and $B_i^0 + B_i^1 \neq B_{i+1}^{(A\vec{X})_{i+1}}$ then
      Send M from $\vec{W}$ to $\vec{W} + B_{i+1}^{(A\vec{W})_{i+1}}$
    end if
  end parallel
end parallel
for each $\vec{W} \in \mathcal{Z}_2^n$ do in parallel
  for each message M at $\vec{W}$ with source $\vec{X}$ and destination $\vec{X} + e_i$ do in parallel
    if $B_i^{(A\vec{X})_i} \neq e_i$ and $B_i^0 + B_i^1 = B_{i+1}^{(A\vec{X})_{i+1}}$ then
      Send M from $\vec{W}$ to $\vec{W} + B_{i+1}^{(A\vec{W})_{i+1}}$
    end if
  end parallel
end parallel
for each $\vec{W} \in \mathcal{Z}_2^n$ do in parallel
  for each message M at $\vec{W}$ with source $\vec{X}$ and destination $\vec{X} + e_i$ do in parallel
    if $B_i^{(A\vec{X})_i} \neq e_i$ and $B_i^0 + B_i^1 = B_{i+1}^{(A\vec{X})_{i+1}}$ then
      Send M from $\vec{W}$ to $\vec{W} + B_i^{(A\vec{W})_i}$
    end if
  end parallel
end parallel
end procedure

Figure 8.7. Algorithm LimitedHypercubeEmulate.

We assume that G has minimal-weight B^0 and B^1. If not, we can transform it to an isomorphic network that does, by using Algorithm MinimumWeightIsomorphism. In G, if $B_i^0 = B_i^1$, then $B_i^0 = B_i^1 = e_i$. If $B_i^0 \neq B_i^1$, then $B_i^0 + B_i^1 = e_{i+1}$, and so either $B_i^0 = e_i$ and $B_i^1 = e_i + e_{i+1}$, or $B_i^1 = e_i$ and $B_i^0 = e_i + e_{i+1}$.

If the conditions of the theorem are true, then the *LTLE* network will never need more than two routing steps to emulate one hypercube routing step. Assume $B_i^0 = e_i$. If B_i^0 defines a channel $\left(\vec{X}, \vec{X} + B_i^0\right)$ in G, we route in one step from $\vec{X}$ to $\vec{X} + e_i$. If B_i^0 does not define a channel $\left(\vec{X}, \vec{X} + B_i^0\right)$ in G, then $B_i^1 = e_i + e_{i+1}$ does and we route in two steps from $\vec{X}$ to $\vec{X} + e_i$ by using channels defined by B_i^1 and B_{i+1}^ϕ, where $B_{i+1}^\phi = B_i^0 + B_i^1$. Since B_{i+1}^ϕ depends on B_i^1 (by the theorem's conditions), we can ensure that B_{i+1}^ϕ defines a channel in the path. This is done by ordering the step across a channel of dimension $i+1$ either either before or after the step across a channel of dimension i.

This same argument is also true for $B_i^1 = e_i$.

Algorithm LimitedHypercubeEmulate simulates a single communication step of the hypercube. For a message that is routed along e_i in the hypercube Q_n, the algorithm determines if there is a corresponding legal channel in G and sends the message along channel. If there is no legal channel, the algorithm routes the message along two legal channels to reach the destination. If we ignore channel collisions, a dilation of 2 is sufficient to route a single hypercube message on G.

If we cannot ignore channel collisions, then we must guarantee that no two paths use the same channel simultaneously. There are three sets of paths used by Algorithm LimitedHypercubeEmulate for any dimension i with $1 \leq i \leq n-1$. These paths are listed below, by their conditions:

1. $e_i = (A\vec{X})_i$. The path is $\vec{X} \rightarrow \vec{X} + B_i^{(A\vec{X})_i}$ (and $\vec{X} \rightarrow \vec{X} + B_i^1$).

Path Type	Stage 1	Stage 2	Stage 3	Stage 4
1	$B_i^{(A\vec{X})_i}$			
2	$B_i^{(A\vec{X})_i}$	$B_{i+1}^{(A\vec{X})_{i+1}}$		
3			$B_{i+1}^{(A\vec{X})_{i+1}}$	$B_i^{(A\vec{X})_i}$

Table 8.7. Timing stages for each type of path.

2. $e_i = \overline{(A\vec{X})_i}$ and $e_{i+1} = (A\vec{X})_{i+1}$. The path is $\vec{X} \rightarrow \vec{X} + B_{i+1}^{(A\vec{X})_{i+1}} \rightarrow \vec{X} + B_{i+1}^{\phi} + B_i^{(A\vec{X})_i}$, for $i < n$.

3. $e_i = \overline{(A\vec{X})_i}$ and $e_{i+1} = \overline{(A\vec{X})_{i+1}}$. The path is $\vec{X} \rightarrow \vec{X} + B_i^{(A\vec{X})_i} \rightarrow \vec{X} + B_i^{(A\vec{X})_i} + B_{i+1}^{(A\vec{X})_{i+1}}$, for $i < n$.

Within each type of path, the individual paths do not cause channel collisions. Two paths of the same type that emulate e_i on $\check{G}$ do not collide, because by Lemma 2.1.1 the neighbor function N_i is 1-1. Two paths of the same type that emulate e_i and e_j with $i \neq j$ cannot collide, because at each step they use channels in different dimensions.

If each type of path is routed in separately, then the hypercube can be simulated without channel collisions with dilation 5. We can overlap the routing step for paths of type 1 with the first routing step for paths of type 2, because they use disjoint sets of channels. This gives us a dilation of at worst 4, as shown in Table 8.7. However, overlapping the routing steps for paths of types 2 and 3 will allow channel collisions. A message on a path of type 3 emulating e_i can have an collision with a message on a path of type 2 emulating e_{i+1}, and vice versa. So we cannot use this method to get a dilation of 2 or 3. ∎

If we join and transmit simultaneously messages that use the same channels, then the dilation can be effectively reduced to 2 steps. This might be considered "cheating," since the network hardware and software must be designed to allow message joining and splitting.

We can apply the results from the above theorem to the published *LE* networks.

Corollary 8.3.3 *The Möbius Cubes, the Flip MCube, the Bent Cube and the YATC Cube of dimension n can emulate a hypercube Q_n with dilation* 4.

Proof: This follows directly from the theorem above. ∎

Corollary 8.3.4 *The Twisted Cube and the Generalized Twisted Cube of dimension n can emulate a hypercube Q_n with dilation* 2.

Proof: This follows from the theorem above. However, because $W(B_i^0 + B_i^1) = 1$ implies that $W(B_{i+1}^0 + B_{i+1}^1) = 0$ in these networks, the paths of type 2 and 3 can now overlap without collision, because we no longer need to worry about whether a path of type 3 emulating e_i will collide with a path of type 2 emulating e_{i+1}, or vice versa. Because we can overlap all three types of paths, G can emulate the hypercube Q_n with dilation 2. ∎

So while *LTLE* networks apparently need linear dilation to emulate the hypercube, all of the published *LE* networks need only constant dilation.

8.3.3. Emulating Other Networks on *LTLE* Networks

The *LTLE* network emulation of the hypercube can be generally applied to a number of networks. The hypercube is a member of a family of networks that can emulate each other with a constant dilation [45]. These networks include the

Cube Connected Cycles network, the DeBruijn network, and the Butterfly network. Because the published twisted cube networks can emulate the hypercube with a constant dilation, they also belong to this family of networks. Any *LTLE* network can also emulate these networks with a constant dilation.

However, we are interested in emulating networks that we could not directly embed into the *LTLE* networks. We can emulate a toroidal mesh (or an ordinary mesh) on a *LTLE* network G, by emulating a hypercube Q_n on G.

Corollary 8.3.5 *Let $G = (B^0, B^1, A)$ be an n-dimensional* LTLE *network. If G can emulate Q_n with dilation t, then G can emulate a $2^{\lfloor n/2 \rfloor} \times 2^{\lfloor n/2 \rfloor}$ toroidal mesh with dilation t, and G can emulate a k-ary $(n-k)$-cube with dilation t.*

Proof: Trivially true, because Q_n can directly embed a $2^{\lfloor n/2 \rfloor} \times 2^{\lfloor n/2 \rfloor}$ toroidal mesh or a k-ary $(n-k)$-cube by Gray-encoding. ∎

In fact, any network that is directly embedded into the hypercube can be emulated on a *LTLE* network with the same dilation that emulating a hypercube requires. If the embedding is squashed or stretched, then the dilation may increase.

This is the case with emulating a binary tree on an arbitrary *LTLE* network. Though the direct embedding of a binary tree was shown possible for a given *LE* network, we would like to be able to emulate a binary tree on any *LTLE* network.

Corollary 8.3.6 *Let $G = (B^0, B^1, A)$ be an n-dimensional* LTLE *network. If G can emulate Q_n with dilation t, then G can emulate a $2^n - 1$ node binary tree with dilation at worst $t + 1$.*

Proof: Trivially true, because Q_n can directly embed a 2^n node double-rooted binary tree. This means Q_n can emulate the $2^n - 1$ node binary tree with dilation 2. We only have to rearrange the dimension so that the channel connecting

the double roots falls along dimension n. Because dimension n can always be routed in one step for both Q_n and G, communication between the first level children and the root can take at most 2 steps on Q_n and at most $t+1$ steps on G. The rest of the communications will take one step on Q_n and t steps on G. ∎

The binary tree emulation is a special case. In general, this technique of emulating a network A by emulating an emulation of A on Q_n will have a dilation of at worst $k_1 \times k_2$, where k_1 is the dilation of emulating Q_n on G, and k_2 is the dilation of emulating A on Q_n.

8.4. SUMMARY

In this chapter, we have dealt with several different aspects of *LE* networks, including embeddings, emulations. We have managed to show that *LTLE* networks are able to embed such networks as rings, binomial trees, and in some cases binary trees. We have also shown that all *LTLE* networks can emulate the hypercube with linear dilation, and that some of the published *LE* networks can emulate the hypercube with constant dilation. This allows the published *LE* networks to emulate a large number of networks with constant slowdown, including binary trees, meshes, and k-ary n-cubes.

9. ALGORITHMS FOR *LE* NETWORKS

One reason that the hypercube interconnection network remains popular is because it can efficiently compute a large number of standard parallel algorithms, including most algorithms with a divide-and-conquer structure. Coupled with efficient general communications algorithms, this makes the hypercube network ideal for writing simple and efficient parallel algorithms.

In this chapter, we will discuss broadcasting algorithms for the *LE* networks. We will examine some algorithms that can be computed using fewer communication steps than the hypercube. Finally, we consider the implementation of a hypercube and several *LE* networks by using a single reconfigurable network.

9.1. BROADCASTING ALGORITHMS

The typical Twisted Cube paper examines direct one-to-one routing, since it is essential to interconnection network study. However, there are very few papers on resource-preserving hypercube variants that examine other general communication algorithms for their particular networks. The only paper that considers any other communication algorithms is the Multiply Twisted Cube [29] (later the Crossed Cube [30]), which examined the one-to-many broadcast algorithm.

Communication algorithms are always designed for one of two communications models. The first, the single-channel model, allows each processor to send/receive only one message along a single channel at each communication step. The second, the multiple-channel model, allows a node to send/receive multiple messages across several or all channels at each communication step. Efe showed

that under the multiple-channel model, the Crossed Cube was able to broadcast to all processors in $\lceil (n+1)/2 \rceil$ steps.

In this section, we demonstrate that some broadcasting algorithms for the hypercube can be modified for use on *LTLE* networks. We show that hypercube's broadcasting algorithm under the single-channel model is minimal for *LE* networks. We demonstrate several broadcasting algorithms under the multiple-channel model that are at least as efficient as the same algorithms for the hypercube. These algorithms are nonredundant, in the sense that each node receives exactly one message and no channel is used more than once.

We begin with the single-channel model of communications:

Theorem 9.1.1 *Let* $G = (B^0, B^1, A)$ *be an n-dimensional* LTLE *network using single-channel communications. Algorithm SingleChannelBroadcast in Figure 9.1 broadcasts on an n-dimensional* LTLE *networks in n communication steps, which is minimal.*

Proof: Each node that has received the message in a previous communication step can send the message to at most one other node during the current step. This implies that the number of nodes that have received the message can at most double at each communication step. So the lower bound on any single channel broadcast algorithm is $\log_2 2^n = n$ communication steps.

The broadcasting tree of Algorithm SingleChannelBroadcast follows the channels of an embedded spanning binomial tree rooted at the source node $\vec{X}$. It is similar to the algorithm given in [38].

At step 0 of the algorithm, the source node $\vec{X}$ has the broadcast message. At step $0 < i \leq n$, each node that has the message forwards it to the neighbor in dimension i.

Input: An n-dimensional *LTLE* network $G = (B^0, B^1, A)$, the source node $\vec{X}$, and a message M.

```
Procedure SingleChannelBroadcast ( G, X, M )
for i = 1 to n do
    for all nodes W ∈ Z_2^n do in parallel
        if W has the message M
            send M from W to W + B_i^{(AW)_i}
        end if
    end for
end parallel
end procedure
```

Figure 9.1. Algorithm SingleChannelBroadcast.

On an n-dimensional cube, assume that after step $(i-1)$, there are 2^{i-1} nodes with the message, and that the addresses of the nodes with the message all differ somewhere in components 1 through $(i-1)$.

On step i, a node $\vec{W}$ with the message transmits the message to $\vec{V} = \vec{W} + B_i^{(A\vec{W})_i}$. By the definition of the *LTLE* networks, the addresses $\vec{W}$ and $\vec{V}$ differ in the i-th component, and are equal in components 1 through $(i-1)$. $\vec{V}$ differs from any $\vec{U} \neq \vec{W}$ that had the message in step $i-1$, because $\vec{V}$ and $\vec{U}$ must differ somewhere in components 1 through $i-1$. Every node $\vec{W}$ will produce a unique $\vec{V}$ because the neighbor function is 1-1, so there are $2(2^{i-1}) = 2^i$ nodes that have the message at step i.

At step n, 2^n unique nodes have the message, so the message has been broadcast to the entire network. ∎

Algorithm SingleChannelBroadcast is basically the same algorithm that the hypercube uses. In practice, the hypercube's single-channel broadcasting algorithm

is often used for multiple-channel broadcasting, because the diameter of the hypercube is the same as the bound on single-channel broadcasting. Because the *LTLE* networks have diameters that can be less than n, algorithm SingleChannelBroadcast may not be minimal under the multiple-channel model.

Theorem 9.1.2 *Let $G = (B^0, B^1, A)$ be an n-dimensional* LTLE *network using multiple-channel communications, and let $D(G)$ be the diameter of G. Algorithm MultipleChannelBroadcast in Figure 9.2 executes in at most $D(G)$ communication steps, which is minimal.*

Proof: The diameter is clearly a lower bound on the number of steps that a multiple-channel broadcasting. algorithm – a message cannot be transmitted from the source to a node a distance $D(G)$ in fewer steps.

The minimal point-to-point routing algorithm gives a unique path from a source node $\vec{X}$ to a destination $\vec{Y}$. For any path of length 1 or more, there is clearly a unique next-to-last node $\vec{W}$ on the path adjacent to $\vec{Y}$.

At step 0, the message is given to the source node $\vec{X}$. At each following step, and for each node $\vec{W}$ with the message, each neighbor $\vec{V}$ of $\vec{W}$ is examined. If $\vec{V}$ would receive a message from $\vec{X}$ through $\vec{W}$ (say, using Algorithm LinearEquationRoute from Figure 6.7), then the message is sent to $\vec{V}$ in the broadcast.

In the algorithm, a boolean variable "Forwarded" is used to have each node broadcast only once. $\vec{V}$ is the j-th neighbor of $\vec{W}$. A list P is used to store the path from $\vec{X}$ to $\vec{V}$. The final element of the list, $P_{|P|}$, will be $B_j^{(A\vec{W})_j}$ iff $\vec{W}$ is on the path from $\vec{X}$ to $\vec{V}$, because $\vec{W}$ is the j-th neighbor of $\vec{Y}$.

Messages must arrive at all nodes because all *LTLE* networks are connected and so a path exists between every pair of nodes in each network. Only one copy of the message arrives at each node because every path from $\vec{X}$ given by the routing

Input: An n-dimensional *LTLE* network $G = (B^0, B^1, A)$, the source node $\vec{X}$, and the message M.

```
Procedure MultipleChannelBroadcast ( G, X, M )
for all W ∈ Z_2^n do in parallel
    NotSent ← true
end parallel
for i = 1 to D(G) do
    for all W ∈ Z_2^n do in parallel
        if W has the message M and NotSent = true
            for j = 1 to n do
                V ← W + B_j^(AW)_j
                P ← MinimalRoute( G, X, V )
                if P_|P| = B_j^(AW)_j then
                    send message M from W to V
                end if
            end for
            NotSent ← false
        end if
    end for
end parallel
end procedure
```

Figure 9.2. Algorithm MultipleChannelBroadcast.

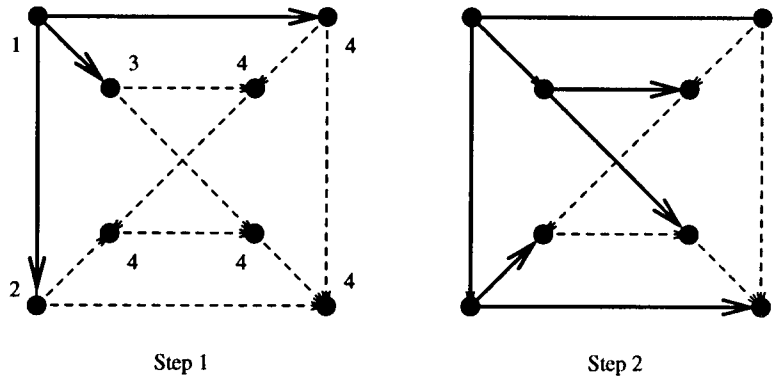

Figure 9.3. Broadcasting in two steps on a Twisted 3-Cube.

algorithm is unique. At most $D(G)$ communication steps are needed for any network G, because no longer routing path will be given by the minimal routing algorithm.

The algorithm takes $O(n) \times O(n^2 \log n) = O(n^3 \log n)$ bit operations per each routing step, by Theorem 6.4.1, because the routing algorithm is run for each neighbor of a single processor. The number of communication steps is $D(G)$, and so is minimal. ∎

Algorithm MultipleChannelBroadcast is a general method for multiple-channel broadcasting that is applicable not just to the *LE* networks, but also for other networks. There is another multiple-channel broadcasting algorithm that can take advantage of *some* of the structure in a *LE* network, and so can in general broadcast using fewer communication steps than the hypercube. This algorithm does not use an optimal number of communication steps, but does use asymptotically fewer bit operations than Algorithm MultipleChannelBroadcast above.

The method used by this algorithm is similar to one used by the 3-bit lookahead algorithm. In Figure 9.3, we note that the Twisted 3-Cube has a diameter of

2, and any message can communicate to the rest in 2 steps. In the first routing step the source node distributes the message to all three of its neighbors, then in the second routing step two neighbors of the source node distribute the message to the four remaining nodes.

This Twisted 3-Cube broadcast suggests a fairly simple broadcasting algorithm, as shown in Figures 9.4 and 9.5. We won't formally prove the behavior of Algorithm ThreeBitBroadcast, as it is similar to Algorithm ThreeBitLookahead-Route in Figure 7.2.

This broadcasting algorithm takes a maximum of $O(n)$ routing steps and a minimum of $O(\lceil 2n/3 \rceil)$ routing steps, depending on the network used. This algorithm also has a distributed run time of $O(1)$ bit operations and a total run time of $O(n)$ bit operations – mostly through pre-computing of matrix operations and simultaneous transmission of the broadcast message.

Algorithm ThreeBitBroadcast deals with the conditions that can occur locally in a twisted define sub-networks that are ordinary 3-Cubes, the algorithm behaves much the hypercube's multiple-channel broadcast algorithm. However, if the components i, $i+1$ and $i+2$ define sub-networks that are Twisted 3-Cubes, then each node must do different actions to broadcast to all nodes in the Twisted 3-Cube formed by components i, $i+1$ and $i+2$. If we group together different nodes that do the same action, we have four different cases to consider. The case corresponding to each node is shown by the numbers in Figure 9.3, and the actions required for each case are listed in Algorithm ThreeBitBroadcast.

Though Algorithm ThreeBitBroadcast does not use the minimal number of communication steps, it has the advantage that (in many cases) it takes fewer routing steps than the multiple-channel broadcasting algorithm of the hypercube, and takes fewer computation steps than the minimal multi-channel broadcasting algorithm.

Input: An n-dimensional *LTLE* network $G = (B^0, B^1, A)$, the source node $\vec{X}$, and message M.

Procedure ThreeBitBroadcast (G, $\vec{X}$, M)
For node $\vec{X}$, *state* $\leftarrow 1$
for all nodes $\vec{W} \in \mathcal{Z}_2^n$ do in parallel
    NotSent $\leftarrow$ true
end parallel
$i \leftarrow 1$
delay $\leftarrow 0$
while $i \leq n$ do
    for all nodes $\vec{W} \in \mathcal{Z}_2^n$ do in parallel
        if $\vec{W}$ has message M and NotSent = true then
            case *state* $= 1$:
                if $1 \leq i \leq n-2$ and $B^0_{i+1} \neq B^1_{i+1}$ and $(AB_i^{(A\vec{X})_i})_{i+1} = 1$
                    Route M to $\vec{W} + B_{i+1}^{(A\vec{W})_{i+1}}$ with *state* $= 2$
                    Route M to $\vec{W} + B_{i+2}^{(A\vec{W})_{i+2}}$ with *state* $= 3$
                    Route M to $\vec{W} + B_{i+3}^{(A\vec{W})_{i+3}}$ with *state* $= 4$
                    Route M to $\vec{W} + B_j^{(A\vec{W})_j}$ with *state* $= 1$ for $i+3 < j \leq n$
                else
                    Route M to $\vec{W} + B_j^{(A\vec{W})_j}$ with *state* $= 1$ for $i < j \leq n$
                end if
            case *state* $= 2$
                Route M to $\vec{W} + B_{i+2}^{(A\vec{W})_{i+2}}$ with *state* $= 4$
                Route M to $\vec{W} + B_{i+3}^{(A\vec{W})_{i+3}}$ with *state* $= 4$
                Route M to $\vec{W} + B_j^{(A\vec{W})_j}$ with *state* $= 1$ for $i+3 < j \leq n$
            case *state* $= 3$
                Route M to $\vec{W} + B_{i+1}^{(A\vec{W})_{i+1}}$ with *state* $= 4$
                Route M to $\vec{W} + B_{i+3}^{(A\vec{W})_{i+3}}$ with *state* $= 4$
                Route M to $\vec{W} + B_j^{(A\vec{W})_j}$ with *state* $= 1$ for $i+3 < j \leq n$
            case *state* $= 4$
                Route M to $\vec{W} + B_j^{(A\vec{W})_j}$ with *state* $= 1$ for $i+3 < j \leq n$
            end case
            NotSent $\leftarrow$ false
        end if
    end parallel

(continued)

Figure 9.4. Algorithm ThreeBitBroadcast.

(continued from Figure 9.4)

```
    if 1 ≤ i ≤ n − 2 and B^0_{i+1} ≠ B^1_{i+1} and (AB_i^{(AX)_i})_{i+1} = 1 then
        if delay = 0 then
            delay ← 1
        else
            i ← i + 3
            delay ← 0
        end if
    else
        i ← i + 1
    end if
end while
end procedure
```

Figure 9.5. Algorithm ThreeBitBroadcast (continued).

This algorithm should require at most a constant computation time per routing step, provided we precompute which sets of components define sub-networks that are Twisted 3-Cubes. The only disadvantage is that a constant number of bits have to be sent with the message to signal the type of broadcasting action the node should perform.

There are certain cases where this algorithm is minimal:

Theorem 9.1.3 *The Bent Cube and the Generalized Twisted Cube have a multi-channel broadcasting algorithm that takes* $\lceil 2n/3 \rceil$ *communication steps, and is minimal for both networks. The Twisted Hypercube has a multi-channel broadcasting algorithm that takes* $n-1$ *communication steps and is minimal.*

Proof: This result depends on the fact that a Twisted 3-Cube has a diameter of 2. We can group the components of the address vector into Twisted 3-Cubes and then broadcast in 2 steps to each 3-dimensional sub-network. Since the diameters

of these 2 networks are both $\lceil 2n/3 \rceil$, Algorithm ThreeBitBroadcast is minimal – we cannot broadcast to all nodes in fewer communication steps. The Twisted Hypercube has only the last three components of its address vector form sub-networks that are Twisted 3-Cubes and so must broadcast like the Hypercube for the first $n-3$ steps, and then broadcast like the Twisted 3-Cube in the last 2 steps. This is optimal because the Twisted Hypercube has a diameter of $n-1$. ∎

But there are also cases where this algorithm is not minimal:

Theorem 9.1.4 *The Möbius Cubes, the MCube, and the YAT Cube have a multi-channel broadcasting algorithm that takes* $\lceil 2n/3 \rceil$ *communication steps.*

Proof: The proof is the same as in Theorem 9.1.3. But with these networks, the algorithm is not minimal, because the diameter of all of these networks is $\lceil (n+1)/2 \rceil$. ∎

9.2. GENERAL ALGORITHMS

Only the Crossed Cube of Kemal Efe [30] has had any sort of parallel application written for it. Efe examined semi-group computations, matrix multiplication, and sorting. He showed that these problems could be computed using almost half the communication steps that the hypercube would use for the same algorithms, largely because of the reduced diameter of the network. (In reality, Efe "cheated" a little. The semigroup calculation will operate as he suggests, but will have the correct result at only the root node. The sorting algorithm used the rank sorting method, which isn't really the traditional problem of sorting, and which requires 2^{2k} processor nodes to compute 2^k values).

We consider the problem of implementing various algorithms on the *LE* networks. We begin with a few examples, which admittedly duplicate the results of Efe.

Efe defines a semigroup operation as a recurrence relation over a binary associative operation $\circ$ over a set of elements $a_1, \ldots, a_m$:

$$y_1 = a_1$$

$$y_i = a_i \circ y_{i-1} \text{ for } i = 2 \ldots m$$

Assume that each element a_i is at a different processor of a hypercube. All y_i can be computed on a hypercube in n communication steps, by using a binomial tree to collect and compute values. Because *LTLE* networks embed binomial trees, we can replicate the algorithm on a *LTLE* network in the same number of communication steps.

If we want to compute just y_1 at one node $\vec{X}$ in a *LTLE* network, then the communication time can be shortened. The algorithm assumes that each node either knows or can compute its parent and children in a broadcast tree from $\vec{X}$. Any leaf simply sends its value to its parent. Any interior node waits until receives all the results of its children's computations, and then computes the group sum of their results and sends the sum to its own parent. At the end of the computation, the root node contains the group sum of all elements.

This computation takes only as many communication steps as a broadcast algorithm, and so will take at most as many steps as the hypercube would. This algorithm allows us to compute many collection problems, such as global maximum, global sum, etc.

Another algorithm that can take advantage of the smaller broadcasting distances of *LTLE* networks is matrix-vector multiplication $A\vec{X}$ where A is a $2^{n/2} \times 2^{n/2}$

matrix. First, examine the algorithm on a $2^{n/2} \times 2^{n/2}$ mesh. The rows and columns of A are stored on the rows and columns of the mesh, and the elements of $\vec{X}$ are stored in the leftmost column. The algorithm first has the elements of the first column broadcast their X_i values to their rows, then computes the product $A_{i,j}X_i$ at each element. The final phase is adding the elements of each row into the first element of each row.

The hypercube can do this algorithm by Gray-encoding. It will take n communication steps, because the broadcast and summation can treated as broadcasts and group sums over dimension $n/2$ sub-networks, which will each take $n/2$ steps.

By Corollary 8.2.1, up to $2^{n/2}$ disjoint Hamiltonian cycles of length $2^{n/2}$ can be put onto one *LE* network. We can use these cycles to do simultaneous computation of each row in the matrix-vector product. Then the algorithm will take as little as $2D(G_{n/2})$ communication steps, where $D(G_{n/2})$ is the maximum diameter of the sub-networks. For instance, the algorithm will take $n + 2$ communication steps for the 0-Möbius cube.

When running parallel algorithms, the *LTLE* networks can show a improvement in communications delay over the hypercube in at least two ways. First, direct one-to-one communications (over a uniform source/destination distribution) have shorter paths on a *LTLE* network than on the hypercube, and so require fewer communication steps. Second, one-to-many broadcast trees generally have smaller height, and so broadcast and collect algorithms are also faster on an *LTLE* network.

The *LTLE* networks can compute many parallel algorithms with the same number of communications steps as the hypercube. Most parallel algorithms use a divide-and-conquer approach, where one of two recursive subproblems is transmitted to another processor and computed simultaneously. These algorithms have a binomial tree communication pattern and so map easily to the hypercube's re-

cursive, easily decomposable structure. The *LTLE* networks embedding of binomial trees, also allow us to implement a parallel divide-and-conquer algorithm on a *LTLE* network with the same communications complexity as on a hypercube.

A *LTLE* network will have a communications time improvement over a hypercube in the emulation of PRAM algorithms. Since the PRAM is a global memory model, and the *LTLE* network and hypercube are not, we are faced with two choices: Either the processors split the memory so each processor has one segment of the global memory, or each processor contains an image of the shared memory space. In the first case, a processor will require direct one-to-one communication with another processor if it accesses a memory location outside its own local space. In the second, broadcast and partial semigroup computations will be needed to update each image of the global memory. In either case, the *LTLE* network has an advantage over the hypercube if its diameter is smaller and if memory accesses are uniformly distributed.

However, there are situations where a *LTLE* network will have no advantage over a hypercube in communications time. When the communications patterns are heavily dependent on the hypercube's structure, the shorter routing distances offered by an *LTLE* network will be of little use. This is especially true if the *LTLE* network cannot efficiently embed or emulate a hypercube.

9.3. RECONFIGURABLE NETWORKS

An interesting property of the Crossed Cube architecture of Efe [30] is that it can be obtained from a hypercube of the same dimension by adding crossover switches to a small number of channels. We can control the switches' settings so that the channels are configured for either the hypercube or the Crossed Cube architecture at any given time.

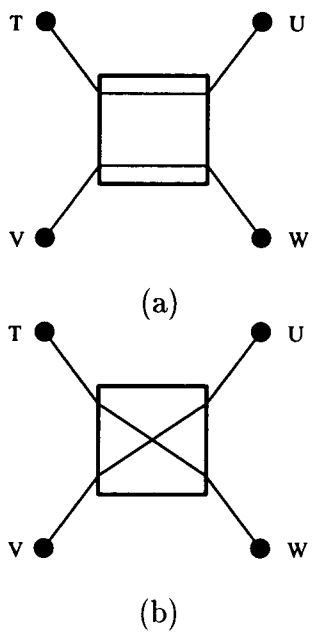

Figure 9.6. A reconfigurable switch.

We find four processors T, U, V, W where, (T, U) and (V, W) are channels only in the hypercube, and where (T, W) and (U, V) are channels only in the Crossed Cube. We then put in a switch that ties the two channels together. By setting the switch, we can either let the two channels pass straight through or cross each other, as in Figure 9.6.

Efe showed that the hypercube network requires only a fairly small number of switches to reconfigure into a Crossed Cube network. For a network of dimension $n = 2k$ or $n = 2k + 1$, the number of switches needed is:

$$s_{2k} = (k-2)2^{2k-1} + 2^k$$

$$s_{2k+1} = k2^{2k} - 3(2^{2k-1} - 2^{k-1})$$

This number is relatively small compared to the number of channels in the Crossed Cube network. Because the switches are relatively simple, the complexity of constructing the network will be dominated by the number of routing elements and not by the number of switches.

The advantages of this reconfigurable network are clear. The switches can initially set the network to a hypercube configuration. However, when the Crossed Cube has a better communication algorithm, we can temporarily turn the switches to configure the network as a Crossed Cube and communicate. The best of both networks exist in this "dynamically reconfigurable network".

The reconfigurable network concept can be applied to the *LTLE* networks. Here, each pair of twisted channels can be joined by a switch, so that the network is reconfigurable as a hypercube.

We will limit our discussion to *LTLE* networks for now.

The twisted channels of the *LTLE* network will always occur in pairs. Let $(\vec{X}, \vec{X}+B_i^{(A\vec{X})_i})$ be a twisted channel in a *LTLE* network. Then $(\vec{X}+e_i+B_i^{(A\vec{X})_i}, \vec{X}+e_i)$ is also a twisted channel in the *LTLE* network, because: $(Ae_i)_i = 0$ and $B_i^{(AB_i^{(A\vec{X})_i})_i} = 0$ by definition of a *LTLE* network, and so:

$$\begin{aligned}
&\vec{X} + e_i + B_i^{(A\vec{X})_i} + B_i^{[A(\vec{X}+e_i+B_i^{(A\vec{X})_i})]_i} \\
&= \vec{X} + e_i + B_i^{(A\vec{X})_i} + B_i^{(A\vec{X})_i} + B_i^{(Ae_i)_i} + B_i^{(AB_i^{(A\vec{X})_i})_i} \\
&= \vec{X} + e_i + 0 + 0
\end{aligned}$$

Also, $(\vec{X}, \vec{X}+e_i)$ and $(\vec{X}+e_i+B_i^{(A\vec{X})_i}, \vec{X}+B_i^{(A\vec{X})_i})$ are channels in a hypercube network. So the two *LE* network channels can be made reconfigurable into hypercube channels with one switch.

Assume that the addresses of the nodes in the reconfigurable network will be the same for both the hypercube setting and the *LE* network setting. The number

of switches needed is one-half of the total number of twisted channels in the *LTLE* network, as computed by Theorem 2.3.1. If we implement one of the published *LTLE* networks as a reconfigurable network using unidirectional channels, the number of switches will be half the number of twisted channels listed in Table 1.1.

All *LTLE* networks have reciprocal unidirectional channels. In real hardware implementations, the two unidirectional channels are often replaced by one bidirectional channel. This will cut hardware costs by cutting the number of switches needed in half, because the one bidirectional channel can work as the two reciprocal unidirectional channels. If we implement the network using bidirectional channels, the number of switches needed will be one-quarter of the total number of twisted channels, as computed by Theorem 2.3.1.

The logical generalization is to ask how many different *LE* networks can be implemented into a single reconfigurable network. The theorem below presents at least a partial result:

Theorem 9.3.1 *Let RQ_n be an n-dimensional reconfigurable hypercube network, with a switch between every pair of channels $(\vec{X}, \vec{X}+e_i)$ and $(\vec{X}+e_{i+1}, \vec{X}+e_i+e_{i+1})$ – a total of $(n-1)2^{n-1}$ switches. Then RQ_n can emulate every n-dimensional* LTLE *network which has $B_i^0 + B_i^1 \in \{0, B_{i+1}^0, B_{i+1}^1\}$.*

Proof: Let G be an n-dimensional *LTLE* network that meets the conditions of the theorem. We can use Theorem 4.3.1 to transform the network G to a banded *LTLE* network $\check{G} = (\check{B}^0, \check{B}^1, \check{A})$ where the only nonzero elements of $\check{B}^0$ and $\check{B}^1$ are on the main diagonal and the diagonal immediately below it. Then each channel of $\check{G}$ is either e_i or $e_i + e_{i+1}$ for dimension i.

For network $\check{G}$, we only have to compute whether each switch needs to be configured. For the switch controlling channels $(\vec{X}, \vec{X}+e_i)$ and $(\vec{X}+e_{i+1}, \vec{X}+e_i+$

e_{i+1}), we compute $(\check{A}\vec{X})_i$. If $W(\check{B}_i^{(\check{A}\vec{X})_i}) > 1$, the switch must cross the channels to make RQ_n emulate $\check{G}$. If $W(\check{B}_i^{(\check{A}\vec{X})_i}) = 1$, the switch must be set to leave the channels un-crossed. ∎

Note that the reconfigurable network RQ_n can be configured to emulate all of the published *LTLE* networks, because all of those networks meet the criteria of Theorem 9.3.1. RQ_n can also (with the right settings) emulate all of the MCubes of Singhvi and Ghose [51], and the Twisted N-Cube of Estafahanian [31], by setting exactly one switch to twist its channels. Because it can emulate Twisted N-Cube, it can also be reconfigured to embed a $2^n - 1$ node binary tree. Unfortunately, RQ_n cannot be configured to emulate the Crossed Cube of Efe.

It might be questioned if RQ_n can put some arbitrary setting on the switches to reduce the diameter of the reconfigurable network to less than that of the best published *LTLE* network. The answer is no, because each routing step can correct at most 2 components, and the routing step across the channel in the first dimension can always be forced to correct only one component, no matter what the settings on the switches in the first dimension. So RQ_n also has a diameter of at least $\lceil (n+1)/2 \rceil$ steps.

9.4. SUMMARY

In this chapter, we showed that a variety of broadcasting algorithms exist for the *LTLE* networks. We also showed that at least a few algorithms can be implemented more efficiently on a *LTLE* network than on a hypercube.

Possibly the most important part of this chapter was the section on reconfigurable networks. We could show that one reconfigurable network was able to configure not only to the hypercube, but also to most of the published *LE* networks

(or isomorphisms of them). This allows us to use "the best of both worlds", so to speak. If we need efficient point-to-point routing or single node broadcasting, we can configure the network to a LE network. If we need a more general communication, or want to implement a hypercube algorithm, we can configure the network to a hypercube. This can be done "on the fly", so to speak, with a control bus changing the network to whatever configuration is currently needed.

10. STATIC PERFORMANCE MEASURES FOR *LE* NETWORKS

The performance of a network depends on several factors: The topology of the network, the routing paths computed routing algorithm, the strategy used for forwarding messages, and the distribution of messages in the network. If the topology of the network is poorly designed, say in the case where only one channel joins two large components of the network, then the network performance will be poor, regardless of the other factors. To examine and compare the behavior of the interconnection networks, we need some measures of the performance of a network's topology.

Performance measures that are based on the graph-theoretic properties of the network's topology are typically called *static* performance measures, because they are independent of ephemeral conditions, such as the number and distribution of messages in the network.

Because they are relatively easy to derive, most published papers on resource-preserving hypercube variants have derived some static measures for their own networks. Static measures include:

Diameter The maximal routing distance between any two processors, as measured by the number of communications channels crossed. This is an estimate of the maximum time a message will be in transit through the network.

Expected Distance The expected minimal distance between any pair of processors, averaged over all pairs of processors. This is an estimate of the average time a message will be in transit in the network.

Bisection Width The number of channels that must be removed to disconnect the network. This is a measure of the fault tolerance, and also an estimate of the network's bandwidth.

There has been a careful examination of the static properties of nearly all the published Twisted Cube networks. This has usually been necessary because the major claim of most Twisted Cube networks is a reduced diameter and expected distance.

We summarize the diameter and expected distance results about these *LE* networks in Table 1.1. We also include information about the routing algorithm and number of twists. The diameter and expected distance are, of course, computed for all the published networks. The bisection width has been computed for the hypercube, the Möbius Cubes, and the MCubes – and is n in all cases.

In this chapter, we will compare the static performance measures of the *LE* networks. In particular, we will try to put bounds on the diameter and expected distance for the *LE* networks.

10.1. STATIC MEASURES FOR *LE* NETWORKS

In this section, we consider bounds on the performance measures of the general *LE* networks. We discuss mostly the bounds we have achieved on the minimal and maximal diameters.

We were unable to compute reasonable bounds on expected distances, and so give no results in this section. However, the minimal network bisection width is trivially 0, because a disconnected network is already bisected. The maximal bisection width is n, because a single node can be disconnected from the network only with the removal of n edges. This bound is tight because the hypercube has a bisection width of n [49].

10.1.1. Lower Bounds on Minimal Diameter

We first consider a trivial lower bound on the *LE* network diameter. We can find the lower bound on the diameter for not only the *LE* networks, but all possible networks formed by the rearranging the resources of a hypercube.

Lemma 10.1.1 *Let G be a graph on 2^n vertices in which each vertex has degree at most n. Then the diameter is:*

$$\Omega\left(\frac{n}{\log n}\right)$$

and the bound can be attained by a tree.

Proof: If the out-degree of each vertex is n, then the maximum number of vertices within distance D is at most $1 + n + n^2 + n^3 + \ldots + n^D$, which is $\frac{n^{D+1}-1}{n-1}$. For D to be the diameter, we must have

$$\frac{n^{D+1} - 1}{n - 1} \geq 2^n$$

So:

$$D \geq \frac{n}{\log n} + \frac{\log(n-1)}{\log n}$$

and $D = \Omega(\frac{n}{\log n})$. Clearly a tree with branching factor n and 2^n vertices has a diameter that meets this lower bound. ∎

This lower bound on the *LE* network diameter is not tight, because it is not clear how we can "twist" a cube to make such a tree.

We tried to raise the lower bound on the *LE* networks by finding a bound on the *DM* networks. The lower bound on the minimal *DM* network diameter can be used to put a lower bound on the minimal *LE* network diameter, by Theorem 6.0.1. However, we show that computing the lower bound on the minimal *DM*

network diameter does not raise our current lower bound on the minimal *LE* network diameter.

Consider a n-dimensional *DM* network $G = (B^0, B^1)$. In a minimal routing path, no term B_i^{ϕ} will be used more than once (if it was, the duplicate terms can be paired and removed from the path). However, B_i^0 and B_i^1 may be used together in a routing path. So there are at most $\binom{2n}{k}$ nodes a distance k from a node $\vec{X}$. Then the smallest maximal distance from node $\vec{X}$ is D, where:

$$\sum_{i=0}^{D} \binom{2n}{i} \geq 2^n$$

Since the *DM* network is symmetric, this is a lower bound on the distance from any node to all other nodes, and so is also a lower bound on the minimal diameter of a n-dimensional *DM* network.

The tree of possible routing paths based on the above arguments will be the binomial tree B_{2n} with 2^{2n} nodes. Clearly, there will be redundant paths to at least some of the 2^n network nodes. Even so, we can assume the tree has no redundant nodes in the first D levels.

If we can compute an upper bound on the summation, then we can bound the diameter from below, by underestimating k. However, it should already be clear that this lower bound on the minimal *DM* network diameter will not raise the lower bound on the minimal *LE* network diameter, because the B_{2n} tree can easily contain a full n-ary tree of depth $n/\log n$, and so the value of D obtained by this argument is equal to or less than our previous argument.

10.1.2. Upper Bounds on Minimal Diameter

Now we consider the upper bound on minimal *LE* network diameter. So far, we've only seen *LE* networks that have a diameter of at least $\lceil (n+1)/2 \rceil$. Are there networks that have a diameter that break the $n/2$ bound?

Consider the 2-dimensional *DM* network:

$$B^0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} B^1 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

It takes only 3 vectors to place any node at most one routing step away from any other. This means that for an n-dimensional *DM* network, we need at most $3n/2$ of the $2n$ vectors to correct all n components of a source address to the components of a destination address in $n/2$ steps. Can we use the other $n/2$ vectors to reduce the diameter of a n-dimensional *DM* network to less than $n/2$? Actually, we can.

Theorem 10.1.1 *Let $G = (B^0, B^1)$ be an n-dimensional* DM *network. The minimal diameter of G has an upper bound of*

$$\lfloor 2n/5 \rfloor + \lceil (n \bmod 5)/2 \rceil$$

which is less than $n/2$ for $n \geq 5$.

Proof: It takes 7 vectors to place any node at most one step away from any other in a 3-dimensional *DM* network. Though there are only 6 vectors available in a 3-cube, there are 10 vectors in a 5-cube. This means that we can correct 3 components in 1 step using 7 vectors, then correct the remaining 2 components in 1 step using the remaining 3 vectors. This *DM* network is:

$$B^0 = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \quad B^1 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

We can construct larger *DM* networks with dimension $n = 5k$ by graph composition, similar to the construction of the Generalized Twisted Cube. This makes the upper bound on minimal *DM* network diameter $2n/5$ for n divisible by five. For n not divisible by five, we can construct a network by using the graph composition of a $5k$-dimensional *DM* network with 1- and 2-dimensional *DM* networks that can be corrected (routed on) in 1 step each. This gives our upper bound on minimal diameter.

This is the smallest diameter that we can achieve by our method. We can prove this by extending the method. Let us subdivide the n-component vector address into r groups of components and try to correct each group of components in one step. These groups of components will have sizes k_1, k_2, ..., k_r, and must meet the following properties:

$$k_1 + k_2 + \ldots + k_r = n \qquad (2^{k_1} - 1) + (2^{k_1} - 1) + \ldots + (2^{k_1} - 1) \leq 2n$$

We can assume that all k_i (except one) must be equal to 1 or 2, because $2^{k_i} > 2k_i$ for $k_i > 2$. (if there are more than one group with more than 2 components, then we can break the groups into 2 sets, each with one group of more than 2 components. Let w_1 be the number of groups of 1 component and w_2 be the number of groups of 2 components. Then we have:

$$w_1 + 2w_2 + k_r = n$$

$$w_1 + 3w_2 + 2^{k_r} - 1 \leq 2n$$

We want to minimize:

$$\frac{r}{n} = \frac{w_1 + 2w_2 + 1}{n}$$

We can solve for n and w_1 and w_2:

$$\frac{r}{n} = \frac{2^{k_r} - 2k_r}{2^{k_r+1} - 3k_r - 1 + w_2}$$

To minimize this ratio, we must maximize w_2, so we set $w_1 = 0$ and $w_2 = r - 1$. When we do so, the ratio of r/n grows to $1/2$ as k_r approaches infinity. The ratio is thus minimal for $k_r = 3$. ∎

This bound may not be the smallest possible to achieve. We may be able reduce the bound by considering groups of components that we correct in two or more steps.

This lower bound may not seem very useful for *LE* networks, but we can create a *LE* network with a diameter that is only a constant larger than this *DM* network.

Theorem 10.1.2 *Let $G = (B^0, B^1)$ be an n-dimensional* DM *network with diameter at most D. Then there is an $(n+1)$-dimensional* LE *network $\check{G}$ that has a diameter of at most $D + 2$.*

Proof: The *LE* network $\check{G} = (\check{B}^0, \check{B}^1, \check{A})$ can be defined as:

$$\check{B}^0 = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & B^0_{1,1} & \cdots & B^0_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & B^0_{n,1} & \cdots & B^0_{n,n} \end{bmatrix} \quad \check{B}^1 = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & B^1_{1,1} & \cdots & B^1_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & B^1_{n,1} & \cdots & B^1_{n,n} \end{bmatrix} \quad \check{A} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & \cdots & 0 \end{bmatrix}$$

Note that the choice of $\check{A}$ ensures that all terms in $\check{B}_i^0$ and $\check{B}_i^1$ with $2 \leq i \leq n$ depend on $\check{B}_1^0$, and depend on no other term.

The routing algorithm on $\check{G}$ starts by finding a minimal expansion S of $(X_2 + \cdots X_n) + (Y_2 + \cdots Y_n)$ on the *DM* network G, and use that to construct a routing path on $\check{G}$.

If $X_1 = 0$, we route using all terms $\check{B}_i^0 : B_i^0 \in S$, then route using $\check{B}_1^0 = e_1$, then route using all terms $\check{B}_i^1 : B_i^1 \in S$. Otherwise if $X_1 = 1$, we route using all terms $\check{B}_i^1 : B_i^1 \in S$, then route using $\check{B}_1^0 = e_1$, then route using all terms $\check{B}_i^0 : B_i^0 \in S$.

At this point, all $X_i = Y_i$ for $2 \leq i \leq n$. If $X_1 \neq Y_1$, we route using $\check{B}_1^0 = e_i$. This will correctly get us from $\vec{X}$ to $\vec{Y}$. Since $|S| \leq D$, the length of the path generated is at most $D + 2$. ∎

This gives us an upper bound on the minimal *LE* network diameter:

Theorem 10.1.3 *Let $G = (B^0, B^1, A)$ be an n-dimensional* LE *network. The minimal diameter of G has an upper bound of:*

$$\lfloor 2(n-1)/5 \rfloor + \lceil ((n-1) \bmod 5)/2 \rceil + 2$$

Which is equal to $n/2$ when $n = 16$ and less than $n/2$ when $n = 21$.

Proof: We combine the results of Theorem 10.1.1 and Theorem 10.1.2 ∎

This network has a diameter that grows at a rate less than $n/2$, but not by much. There are probably several ways to reduce this upper bound on the minimal *LE* network diameter, but our point is that a *LE* network with diameter less than $n/2$ does exist.

10.1.3. Lower Bounds on Maximal Diameter

Now we consider the bounds on the maximal *LE* network diameter. For the lower bound on maximal *LE* network diameter, we have had some success in showing

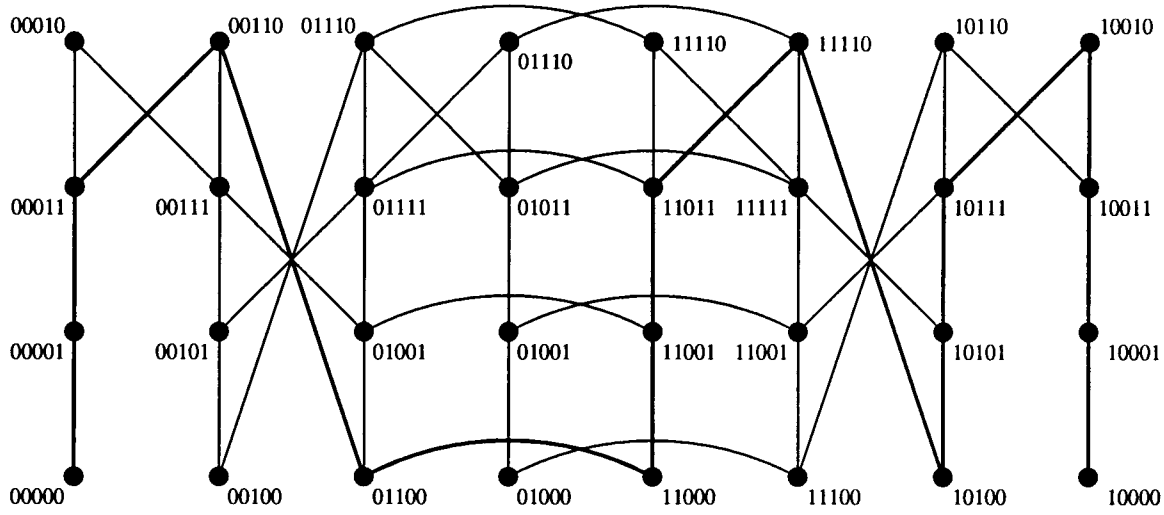

Figure 10.1. A five-dimensional network with an $O(n^2)$ diameter.

that the lower bound on maximal diameter is at least super-linear. However, we were unable to show that the lower bound is limited to a polynomial.

Theorem 10.1.4 *Let* $G = (B^0, B^1, A)$ *be an* n*-dimensional* LE *network. Then the lower bound on maximal diameter is at least* $[n(n+1)]/2$.

Proof: This bound is proved by giving an example network. This network is defined by $B^0 = (\vec{0}, \ldots, \vec{0}, e_n)$, $B^1 = (e_1 + e_3, e_2 + e_4, \ldots, e_{n-2} + e_n, e_{n-1}, e_n)$, and $A = (\vec{0}, e_1, \ldots, e_{n-1})$. The five-dimensional network is described by the matrices below, and is displayed in Figure 10.1.

$$B^0 = \begin{bmatrix} 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&1 \end{bmatrix} \quad B^1 = \begin{bmatrix} 1&0&0&0&0 \\ 0&1&0&0&0 \\ 1&0&1&0&0 \\ 0&1&0&1&0 \\ 0&0&1&0&1 \end{bmatrix} \quad A = \begin{bmatrix} 0&1&0&0&0 \\ 0&0&1&0&0 \\ 0&0&0&1&0 \\ 0&0&0&0&1 \\ 0&0&0&0&0 \end{bmatrix}$$

This network can be shown to have a diameter of $[n(n+1)]/2$ by recursive argument. Assume that the network $\check{G}$ of dimension $n-1$ has a diameter of $[(n-1)n]/2$. The network G can be constructed from two networks G_0 and G_1. G_0 has its addresses pre-pended with a zero and G_0 has its addresses pre-pended with a one. G_0 and G_1 are connected together with channels of the form $((a_1, 0, a_2, a_3, \ldots, a_n), (\overline{a_1}, 0, \overline{a_2}, a_3, \ldots, a_n))$.

The distance to route from $(0,1,1,1,\ldots,1)$ to $(0,0,1,1,\ldots,1)$ on G_0 and from $(1,0,1,1,\ldots,1)$ to $(1,1,1,1,\ldots,1)$ on G_1 is $n(n-1)/2$ steps each. But note that the step from G_0 to G_1 essentially saves up to $(n-1)(n-2)/2-1$ steps (the maximum distance between the corresponding addresses on G_0 and G_1). The total distance between $(0,1,1,1,\ldots,1)$ and $(1,1,1,1,\ldots,1)$ is:

$$\begin{aligned} D(G) &= 2\,[fracn(n-1)2] - \left[\frac{(n-1)(n-2)}{2} - 1\right] \\ &= \frac{2n(n-1) - (n-1)(n-2) + 2}{2} \\ &= \frac{n^2+n}{2} \\ &= \frac{n(n+1)}{2} \end{aligned}$$

So the diameter of G is $n(n+1)/2$. ∎

There is a variation of this network for four dimensions. This network is connected and has a diameter of 12 channels. It is shown in Figure 10.2, and its matrix description is:

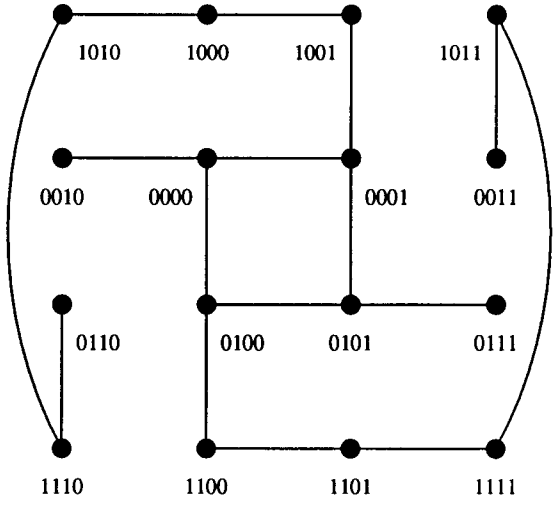

Figure 10.2. A four-dimensional network with a diameter of 12.

$$B^0 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad B^1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad A = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

This network is an extension of the maximal diameter network of dimension 3. 6 channels are added to the longest path by increasing the dimension by one. Unfortunately, this technique is not useful for $n \geq 5$, because the series of "large diameter" networks above have their diameter grow at a rate of at least n^2, while this extension adds only 6 channels to the longest path.

10.1.4. Upper Bounds on Maximal Diameter

The upper bound on maximal *LE* network diameter is (in effect) infinity, because an *LE* network can be disconnected. Even when we limit the problem to

instances of connected networks, the best upper bound on maximal diameter we could achieve is a trivial $2^n - 1$ – the minimal number of channels to connect all nodes in the network. To date, we cannot limit the maximal routing distance of any *LE* network to a polynomial number of channels in n.

This is discouraging, because the maximal diameter of a connected *DM* network is easy to bound. The only way that the *DM* network can be connected is if n of the vectors in B^0 and B^1 form a basis over n components, which means that all expansions can be written as a linear combination of those n vectors. So the upper bound on the maximal diameter of a *DM* network is n. This bound is tight, because we can choose $B^0 = B^1 = I$, which has diameter n.

The problem of computing the upper bound on maximal diameter for *LE* networks is that in our model, local linearity does not imply global linearity. Without global linearity, it is often difficult to compute the diameter of even a single *LE* network, let alone show the upper bound over all *LE* networks.

We can restrict the upper bound on the maximal diameter to $2n - 1$, if we restrict ourselves to a subset of networks that have the property that there can exist no group or cycle of terms that mutually depend on each other:

Theorem 10.1.5 *Let $G = (B^0, B^1, A)$ be an n-dimensional* LE *network where the terms of B^0 and B^1 can be ordered:*

$$B_{i_1}^{\phi_1}, B_{i_2}^{\phi_2}, \ldots, B_{i_n}^{\phi_n}$$

so that:

$$(AB_{i_u}^{\phi_u})_{i_v} = 1 \Rightarrow u > v$$

Then the maximal diameter is bounded above by $2n - 1$.

Proof: Assume that a (possibly non-minimal) path of any length exists. In this path, more than one copy of a term $B_{i_u}^{\phi_u}$ may exist. If an even number of copies exist, then the mod 2 sum of these copies is zero. If an odd number of copies exist, the mod 2 sum of these copies is $B_{i_u}^{\phi_u}$. Remove all but 2 copies of $B_{i_u}^{\phi_u}$ from the path if the number is even, and all but the leftmost copy if the number is odd. Order the remaining terms into a list as in the the statement of the lemma above.

Consider each single term or pair of terms $B_{i_u}^{\phi_u}$ in order. If $B_{i_u}^{\phi_u}$ defines a channel from node $\vec{X}$, insert single or double terms first in the new path. If $B_{i_u}^{\phi_u}$ does not define a channel from node $\vec{X}$, then insert it in the new path after the first term $B_{i_v}^{\phi_v}$ with $(AB_{i_v}^{\phi_v})_{i_u} = 1$. This will make term $B_{i_u}^{\phi_u}$ define a channel from node $\vec{X} + B_{i_1}^{\phi_1} + \ldots + B_{i_v}^{\phi_v}$. A term $B_{i_w}^{\phi_w}$ coming after $B_{i_u}^{\phi_u}$ will still define a channel, because $(AB_{i_u}^{\phi_u})_{i_w} = 0$, but it will now define a channel from node $\vec{X} + B_{i_1}^{\phi_1} + \ldots + B_{i_v}^{\phi_v} + B_{i_u}^{\phi_u} + B_{i_{v+1}}^{\phi_{v+1}} + \ldots + B_{i_{w-1}}^{\phi_{w-1}}$.

The leading term always defines a path from node $\vec{X}$, because it depends on no other term in the original path. This makes it always possible to insert it into the empty path. The entire construction will give us a path of at most $4n$, or at most two occurrences of each term.

We can shorten the path length even further. Note that any term B_i^{ϕ} can appear at most once or twice along a path from $\vec{X}$ to $\vec{Y}$. We split the set of terms that appear in the path into 2 sets. The set S_1 contains terms that occur exactly once in the path, and S_2 contains terms that appear exactly twice. The length of the path is:

$$D = |S_1| + 2|S_2|$$

We can specify some bounds on the size of S_1. First, S_1 must be nonempty if $\vec{X} \neq \vec{Y}$. Second, S_1 represents a linear expansion over n components of $\vec{X} + \vec{Y}$ and so can

be represented using no more than n vectors from B^0 and B^1. If $k > n$, then some terms have a linear dependence and can be removed. So, $1 \leq |S_1| \leq n$.

Similarly, the size of S_2 can be bounded. A term $B_i^{\phi} \in S_2$ is included because another term B_j^{ψ} in the path needed some $\vec{W}$ along the path to have $(A\vec{W})_j = \psi$, but $(A\vec{X})_j = \overline{\psi}$ and there is no term $B_k^{\zeta} \in S_1$ with $(AB_k^{\zeta})_j = 1$. However, $(AB_i^{\phi})_j = 1$, and can be placed before and after B_j^{ψ} in the path to make it a step across a legal channel.

The set S_2 can be empty, and because each nonzero term in the routing path can affect a minimal of 1 component each, the total sum of terms in both sets cannot be greater than n. So $0 \leq |S_2| \leq n - |S_1|$.

Maximizing the distance equation over these two constraints, we get:

$$D \leq 2n - 1$$

∎

10.2. STATIC MEASURES FOR *LTLE* NETWORKS

We now turn from *LE* networks to *LTLE* networks. These networks have a lot of properties that are much easier to prove, partly because the networks can be decomposed into smaller *LTLE* networks.

10.2.1. Bounds on Minimal Diameter

We could not put a tight lower bound on the minimal *LE* diameter except in limited cases. However, we were able to prove that that the upper bound on the minimal *LTLE* network diameter is less than $n/2$.

First consider *LTDM* networks. We can design a *LTDM* network with a diameter approaching $n/2$, using the same technique that was used on *DM* networks. This time, however, we have the limitation that $B_n^0 = B_n^1 = e_n$ for *LTDM* networks.

Theorem 10.2.1 *Let $G = (B^0, B^1)$ be an n-dimensional* LTDM *network. The minimal diameter of G has an upper bound of:*

$$\lfloor 2(n-1)/5 \rfloor + \lceil ((n-1) \bmod 5)/2 \rceil + 2$$

This is equal to $n/2$ when $n = 6$ and less than $n/2$ when $n = 11$.

Proof: We can rearrange the columns of the network in Theorem 10.1.1, and add 1's where required by the definition of *LTLE* networks. Then the network will be:

$$B^0 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad B^1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \end{bmatrix}$$

We can route on this network in three steps. First, we correct the last three components in one step by using B_3^0, B_4^0, B_5^0, B_1^1, B_2^1, B_3^1, and B_4^1. Then, we use B_1^0 and B_2^0 to correct the first two components in at most 2 steps. If we blindly use graph composition again to create network of dimension $n = 5k$, it will have a diameter of $3n/5$ for n divisible by five.

We can reduce the diameter to less than $n/2$ for larger *LTDM* networks, if we change every B_{5k}^1 with $k < n/5$. Originally $B_{5k}^1 = e_{5k}$ and so $B_{5k}^1 = B_{5k}^0$, but we can modify it to $e_{5k} + e_{5k+1} + e_{5k+2}$ so that we can use it to correct components $5k+1$ and $5k+2$ in one step.

The algorithm to correct all the components is then to repeatedly, from $k = n/5 - 1$ to $k = 0$, correct components $5k + 3$, $5k + 4$ and $5k + 5$ in one step using the vectors B^0_{5k+3}, B^0_{5k+4}, B^0_{5k+5}, B^1_{5k+1}, B^1_{5k+2}, B^1_{5k+3} and B^1_{5k+5}, then correct $5k+1$ and $5k+2$ in one step using B^0_{5k+1} B^0_{5k+2} and B^0_{5k} (or at most two steps using B^0_1 and B^0_2 if $k = 0$). This gives us the bound in our theorem. ∎

We can use Theorem 10.2.1 to get an upper bound on the minimal *LTLE* network diameter:

Theorem 10.2.2 *Let $G = (B^0, B^1, A)$ be an n-dimensional* LTLE *network. The minimal diameter of G has an upper bound of:*

$$\lfloor 2(n-2)/5 \rfloor + \lceil ((n-2) \bmod 5)/2 \rceil + 3$$

Which is equal to $n/2$ when $n = 22$ and less than $n/2$ when $n = 27$.

However, we note that the published *LTLE* networks have at best a diameter of $\lceil (n+1)/2 \rceil$. Notably, only the 1-Möbius cube, the Flip MCube and the Twisted Cube meet this bound. Why do none of these networks have a diameter smaller than $\lceil (n+1)/2 \rceil$?

Part of the answer lies in the the fact that for these networks, the condition $B^0_i + B^1_i \in \{0, B^0_{i+1}, B^1_{i+1}\}$ is held. This severely limits the choices possible for expansions over B^0 and B^1:

Theorem 10.2.3 *Let $G = (B^0, B^1)$ be an n-dimensional nonredundant* LTDM *network, where $B^0_i + B^1_i \in \{0, B^0_{i+1}, B^1_{i+1}\}$. The minimal diameter of G is $\lceil n/2 \rceil$.*

Proof: The proof of the *DM* network diameter is by induction on the dimension n.

Base Case: If $n = 1$ the minimal diameter is $D(1) = 1$, because we can construct $G = ([1], [1])$. If $n = 2$ the minimal diameter is $D(2) = 1$, because we can construct:

$$G = \left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \right)$$

Inductive Hypothesis Assume that for $\check{n} < n$, the minimal diameter is at least $\lceil \check{n}/2 \rceil$.

Inductive Step: Since G is a *DM* network, we can assume *wlog* that any path in the network can route on its routing steps in any order, so wlog we assume the routing steps are always in order of their increasing index in B^0 and B^1. Also, no term with any index i will appear in the routing path more than once, because $B_i^0 + B_i^1 \in \{0, B_{i+1}^0, B_{i+1}^1\}$ is sufficient to make G a nonredundant network. We also use Theorem 3.3.2 to decompose G into $(n-1)$-dimensional and $(n-2)$-dimensional sub-networks.

Assume we start at any node $\vec{X}$ in the network. $\vec{X}$ is in one $(n-1)$-dimensional sub-network of G.

If $B_1^0 + B_2^0 = 0$, then both paths $\vec{X} \to \vec{X} + B_1^0$ and $\vec{X} \to \vec{X} + B_1^1$ map to the same node in the opposite sub-network. This is the only way to cross to the opposite sub-network. This gives us a minimal diameter of:

$$D(n) = \max\{D(n-1), D(n-1)+1\} = \lceil (n-1)/2 + 1 \rceil = \lceil (n+1)/2 \rceil$$

If $B_1^0 + B_2^0 = B_{i+1}^0$, then the $(n-1)$-dimensional sub-network opposite $\vec{X}$ can be reached by $\vec{X} \to \vec{X} + B_1^0$ or $\vec{X} \to \vec{X} + B_1^1$. Because $B_{i,i}^0 = 1$, $\vec{X} + B_1^0$ and $\vec{X} + B_1^1$ will be in separate $(n-2)$-dimensional sub-networks in the sub-network opposite $\vec{X}$. This gives us a minimal diameter of:

$$D(n) = \max\{D(n-1), D(n-2)+1, D(n-2)+1\} = \lceil (n-2)/2 + 1 \rceil = \lceil n/2 \rceil$$

The smallest possible diameter is then the smaller of the above two cases, or $\lceil n/2 \rceil$.

No other paths are shorter, because to reach nodes in the $(n-1)$-dimensional sub-network opposite $\vec{X}$, we must use B_1^0 or B_1^1 in the path. All other paths using these terms can be replaced with paths of equal or lesser length because:

$$\vec{X} + B_1^0 + B_2^0 = \vec{X} + B_1^1$$

$$\vec{X} + B_1^1 + B_2^0 = \vec{X} + B_1^0$$

$$\vec{X} + B_1^0 + B_2^1 = \vec{X} + B_1^1 + t \in \{0, B_3^0, B_3^1\}$$

$$\vec{X} + B_1^1 + B_2^1 = \vec{X} + B_1^1 + t \in \{0, B_3^0, B_3^1\}$$

Finally, if $B_1^0 + B_2^0 = B_{i+1}^1$, we can use arguments similar to the ones above to show that the minimal diameter is $D(n) = \lceil n/2 + 1 \rceil$.

This minimal diameter is tight, because it is the diameter of the Enhanced Hypercube.∎

Corollary 10.2.1 *Let $G = (B^0, B^1, A)$ be an n-dimensional nonredundant* LTLE *network with the same B^0 and B^1 matrices. Then the lower bound on the minimal diameter of G is $\lceil (n+1)/2 \rceil$, and this bound is tight.*

Proof: For G, we note that B_1^0 (and never B_1^1) must always be chosen to reach the $(n-1)$-dimensional sub-network opposite $\vec{X}$. If we assume that we can always choose the terms in the opposite sub-network, we can treat the sub-network like a *DM* network and so the diameter is at least $\lceil (n-1)/2 + 1 \rceil = \lceil (n+1)/2 \rceil$. This lower bound on the minimal diameter is tight because it is the diameter of the 1-Möbius cube and the Flip MCube. ∎

10.2.2. Bounds on Maximal Diameter

Now we consider the maximal *LTLE* network diameter. The hypercube has a diameter of n, which is the maximal Hamming distance between two nodes in the hypercube. Calculating the diameters of the *LTLE* networks can be slightly more complicated, due to the asymmetries of the network. Fortunately, we can use one of our routing algorithms to bound the maximal routing distance.

Theorem 10.2.4 *Let $G = (B^0, B^1, A)$ be an n-dimensional* LTLE *network. The maximal diameter of G is n, and this bound is tight.*

Proof: The Algorithm LeftRightBitCorrectRoute always takes at most n routing steps. The hypercube, a *LTLE* network, has a proven diameter of n steps [49]. ∎

10.2.3. Bounds on Expected Distance

The diameter represents the worst-case behavior of a single message traveling in the network. The average-case behavior would be measured by the expected distance, or the average number of routing steps between nodes. The asymmetries and variance of the *LTLE* networks make a general calculation of the expected distance difficult. However, it is possible to bound the expected distance to a value below the hypercube's expected distance of $n/2$.

Theorem 10.2.5 *Let G be an n-dimensional* LTLE *network. The maximal expected distance of G is $\lceil n/2 \rceil$, and this bound is tight.*

Proof: If we choose a uniform distribution of source and destination address vectors, the mod 2 sums of the source and destination will be uniformly distributed

over $\mathcal{Z}_2^n$. This means that the probability of any component i differing between the source and destination is 0.5. At each step i of Algorithm LeftRightBitCorrectRoute, there is a probability of 0.5 that the algorithm will route to the i-th neighbor of a node. So Algorithm LeftRightBitCorrectRoute has an average routing distance of $\lceil n/2 \rceil$ routing steps.

The hypercube has a expected distance of $\lceil n/2 \rceil$ steps, so this upper bound is tight. ∎

In general, we can't put a tight lower bound on the minimal *LTLE* network expected distance, but we can make some reasonable statements about it. Because the expected distance is always less than the diameter, it is clear that the minimal expected distance has an upper bound of $2n/5 + 3$, for large n.

In restricted cases of *LTLE* networks, we can make a stronger statement on the bounds on minimal expected distance:

Theorem 10.2.6 *Let $G = (B^0, B^1, A)$ be an n-dimensional* LTLE *network, where $B_i^0 + B_i^1 \in \{0, B_{i+1}^0, B_{i+1}^1\}$. The lower bound on the minimal expected distance of G is:*

$$E(n) = \frac{1}{6} + \frac{n}{3} + \frac{1}{9}\left[1 - \left(-\frac{1}{2}\right)^{n-1}\right]$$

and the upper bound on the minimal expected distance is $E(n) + 1$.

Proof: We show the bounds by computing $E(n)$, the expected number of terms in the minimal expansion of a vector. If we choose a uniform distribution of source and destination address vectors, the mod 2 sums of the source and destination will be uniformly distributed over $\mathcal{Z}_2^n$.

Assume wlog that $B_i^0 \neq B_i^1$ for $1 \leq i < n$. If so, then $B_i^0 + B_i^1 \in \{B_{i+1}^0, B_{i+1}^1\}$. Assume also wlog that we can always choose which of B_i^0 or B_i^1 we want to use, so

that we can always correct at least 2 components per time step. This will guarantee that at least components with index i and $i+1$ are correct after correcting component i.

Then the expected distance is expressed by the recurrence:

$$T(1) = 1/2,\ T(2) = 3/4,\ T(n) = \frac{1}{2}T(n-1) + \frac{1}{2}(1 + T(n-2))$$

The solution of this recurrence relation is:

$$T(n) = \frac{n}{3} + \frac{1}{9}\left[1 - \left(-\frac{1}{2}\right)^n\right]$$

which can easily be verified by substitution.

Since the number of terms in the minimal expansion is a lower bound on the number of routing steps, the expected distance is greater than or equal to this value.

Since the highest indexed term can only correct at most one component, the expected distance is then bounded from below by:

$$\begin{aligned} E(n) &= \frac{1}{2} + T(n-1) \\ &= \frac{1}{2} + \frac{n-1}{3} + \frac{1}{9}\left[1 - \left(-\frac{1}{2}\right)^{n-1}\right] \\ &= \frac{1}{6} + \frac{n}{3} + \frac{1}{9}\left[1 - \left(-\frac{1}{2}\right)^{n-1}\right] \end{aligned}$$

For some qualifying networks (as in the case of the 1-Möbius cube), the routing path requires at most 1 extra routing step over the minimal expansion. This restricts the upper bound on expected distance to $E(n) + 1$. ∎

This result gives bounds on the minimal expected distance for all of the published *LE* networks, because they meet the theorem's conditions. The 1-Möbius cube, the Flip MCube and the Twisted Cube all have expected distances between $E(n)$ and $E(n)+1$ (these cubes sometimes require one "extra" step to route between some nodes. However, $E(n)$ is not a tight lower bound on the minimal *LTLE* network

expected distance, because it assumes we can *always* choose between B_i^0 and B_j^1 when routing, which is just not true.

10.2.4. Bisection Width

One measure of a network's performance for many messages is the network's bisection width, or the number of channels that must be removed to disconnect the network.

Theorem 10.2.7 *If $G = (B^0, B^1, A)$ is an n-dimensional* LTLE *network, then the bisection width of G is n.*

Proof: The proof is by induction on the dimension n of the network. For this proof, we assume that all the channels are bidirectional.

Base Case: n = 1 The 1-dimensional *LE* network $G = ([1], [1], [0])$ has only one edge and two nodes, so its bisection width is 1.

Inductive Hypothesis: Assume that for $\check{n} < n$, the bisection width of any $(\check{n})$-dimensional *LTLE* network $\check{G} = (\check{B}^0, \check{B}^1, \check{A})$ is $\check{n}$.

Inductive Step: We can clearly disconnect a single node from G by removing all n channels adjacent to the node. Remove any $n-1$ edges from G. By Theorem 3.3.2 G can be divided into two $(n-1)$-dimensional sub-networks, so these removed channels are distributed inside one or the other sub-network, and/or in the channels between the two sub-networks.

If the removed channels are taken from both of the two $(n-1)$-dimensional sub-networks and from the channels between the two sub-networks, then these sub-networks each remain connected by our inductive assumption, and they remain joined to each other by at least $2^n - (n-1)$ connections. If all the removed channels are taken from one sub-network, then that sub-network may be disconnected.

However, every node in that sub-network has a channel to the other connected sub-network, so a path still exists between each pair of nodes in the disconnected sub-network through the connected one. ∎

10.3. CONCLUSIONS

In this chapter we have attempted to put bounds on some of the static performance measures of *LE* networks, including diameter and expected distance. Though we were unable to put tight bounds on the diameters of the *LE* networks, we were able to show that minimal diameter is at most $2n/5 + 2$, and that the maximal diameter is at least $n(n+1)/2$.

For *LTLE* networks, we showed that the minimal diameter is at most $2n/5+3$, and that the maximal diameter is tightly bounded at n. Further, we were able to show why none of the published *LTLE* networks have a diameter less than $\lceil (n+1)/2 \rceil$, and we were able to put bounds on the minimal expected distance for these networks.

Lastly, we were able to show that the bisection width of all the *LTLE* networks is n. This means that all *LTLE* networks have the same "fault tolerance" as the hypercube.

There are still a number of open problems in bounding the static performance measures. The most important is that no tight bound on minimal or maximal diameter has been found for the *LE* networks. We currently suspect that the maximal diameter is polynomial in n, and that the minimal diameter is strictly less than $2n/5 + c$, though there is no proof that this is true.

There are also many more static measures and properties to derive and bound for the *LE* networks. One static property that we did not examine is distance distribution, or the number of nodes a given distance from a source node. The Flip

MCube has a uniform distance distribution [51] – the distance distribution is the same no matter which node we choose. Because the 1-Möbius cube is isomorphic to the Flip MCube, it must also have a uniform distance distribution. However, the distance distributions of other *LE* networks are still unknown.

11. DYNAMIC PERFORMANCE OF *LE* NETWORKS

Static measures describe the behavior of a single message traveling in the network, and so they are not always useful measures of network performance. To examine the actual performance of a network, we will need to use *dynamic* performance measures, or measures based not only on the network's topology, but also the particular routing algorithm, routing strategy, and the number of messages passing through the network at a given time. Even if the topology of the network is reasonable, the network performance may still be poor if the routing algorithm is poorly designed, as in the case where every message is first routed through one node before getting routed to its destination.

In this chapter, we will design a program that simulates routing algorithms on *LTLE* networks and examines their dynamic performance. This simulation will be run using both wormhole and store-and-forward strategies, and using both optimal and approximate routing algorithms.

11.1. DYNAMIC PERFORMANCE MEASURES

Dynamic performance measures examine the interaction of a message with the network and with other messages. Unlike static measures, dynamic measures can depend not only on the topology of the network, but also on the particular routing algorithm used and on the number and distribution of messages traveling in the network. For this reason, dynamic performance measures may give a better overall picture of the real behavior of the network.

Dynamic performance measures are difficult to derive analytically from the network structure of an asymmetrical Twisted Cube network, so not many papers

have any calculation of these measures. Seth Abraham has done a statistical calculation and derivation of performance measures of both the hypercube and Hilber's Twisted Cube [3] [2] [1]. Despite the complexity of deriving the performance of those networks, they are relatively simple compared to other *LTLE* networks, for instance, the Möbius cubes. Because we want to generate and test these performance measures for a large number of *LTLE* networks, simulating these networks is the preferable method for measuring dynamic performance.

Typical dynamic performance measures include:

Channel Utilization Rate The utilization of a particular channel, as measured by the average number of messages processed through that processor/channel per unit time step.

Message Latency If only one message per unit time step can be transmitted through a communications channel, then other messages that need that same channel must be buffered until a later time. Message latency is the mean number of time steps that a message takes to route between its source and destination.

Probability of Arrival Consider a system that uses store-and-forward routing and allows only a finite number of messages to be buffered at a busy communication channel. Some messages may be lost if a buffer becomes full. The probability of arrival is the probability that a transmitted message will reach its destination in a finite buffered system.

In this dissertation, we will examine the expected message latency, and expected channel utilization for several *LE* networks. We will not examine probability of arrival, because it is not valid in our simulation model (we assume no finite buffers in store-and-forward routing).

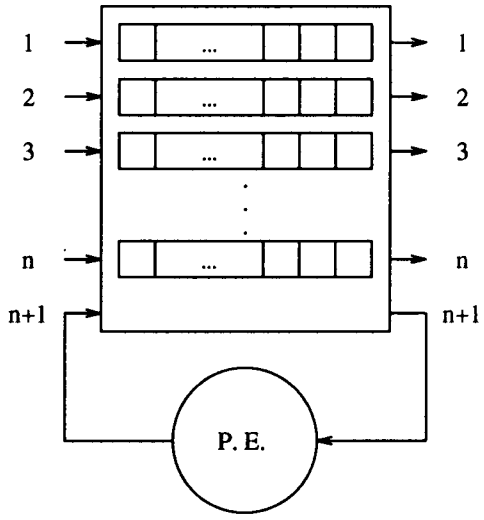

Figure 11.1. The model of a network node used in the simulation.

11.2. THE SIMULATION

We have programmed a discrete-time simulation of a Twisted Cube network, similar to the simulation described in Abraham [1]. The design of each network node element is shown in Figure 11.1. Each node in the network has two parts: a processing element, and a routing element. The processing element, or PE, is the element that does the actual computation of the network and so is the source and destination for all messages. The routing element, or router, simulates a crossbar switch with n unidirectional input and n unidirectional output channels. The processing and routing elements are joined together by 2 unidirectional channels. To complete the model, the n input and n output channels of each router are connected to the other nodes of the network by the interconnection rules.

The finished program can simulate any *LTLE* network, by taking as input a dimension n and three $n \times n$ matrices: B^0, B^1 and A. In addition, a number

of command-line arguments can be used to set various switches for the modeled network. These switches include:

- The network configuration file (containing n, B^0, B^1 and A).
- The simulation run time.
- The message generation rate.
- The mean and standard deviation of the message length (as measured in flits, the largest amount of information that can be sent across a channel in one time step).
- The routing algorithm. We have a choice of Algorithms LeftRightBitCorrectRoute, NonRedundantMinimalRoute and ThreeBitLookaheadRoute.
- The routing strategy, either wormhole or store-and-forward.
- The number of virtual channels. This is the maximum number of virtual channels that a physical channel can multiplex.
- The flit buffer size, or the number of flits that a channel can queue up for forwarding if the message stream gets interrupted. If there is more than one virtual channel, each virtual channel has a separate flit buffer of this size.
- A choice of whether the router-PE channels are single-accepting or multiple-accepting. A PE can accept one message at a time from the router, or up to n messages simultaneously.

The output of the simulation is a statistical summary. During the simulation, the program records: the number of messages in transit at each time step; the latency of each message; and the channel utilization for each channel. At the end of the

simulation, the maximum, mean and standard deviation of each measure is printed out. The channel utilization is averaged for all the channels of each dimension 0 through $n-1$, and is averaged for all the PE-router channels.

11.2.1. Simulation Messages

The main element of the simulation is the message. In a wormhole routing model, the message may be divided into several segments, each located at a different PE in the network. Because of this, we chose to represent each message as a list of segments, each knowing its own location and size. As the message travels through the network, it updates its head and tail segments, and processes its flits through the list of segments from tail to head. To keep track of all the messages, the simulation maintains a list of all the currently existing messages and always processes them in a FIFO order.

The simulation is discrete-time, with the unit of time measured as the time to transmit one flit over a channel. During each step, the simulation does a sequence of actions to each message. These steps are (in order):

1. Generate new messages with uniformly distributed random sources and destinations, and place one message segment at the source location.

2. Try to allocate the channel to the next location on the message's path. If the channel is allocated, create a new segment and make it the new head segment of the message.

3. For each segment of each message that has not filled its available buffer space and has a preceding segment with a nonempty buffer space, request the transmission of one flit.

4. For each segment of each message that requested a flit in the previous step, attempt to access the channel between the segment and its predecessor, and forward one flit if access is given.

5. For each message with an empty tail segment, delete the tail segment. For each message with only a segment at the message's destination location, delete the message, and update any message statistics.

Unfortunately, these steps must be performed on all messages synchronously, which means that the simulation must iterate through the list of messages five times per step.

The simulation allows use of both the wormhole and store-and-forward routing strategies. A message using wormhole routing tries to advance its head segment through the network until it reaches the destination router, then routes into the destination PE. Store-and-forward routing is identical, except that a message routes to each PE on the path (through the routers, of course) and will not advance its head segment to the next PE until the last flit has arrived at the current one. In this representation, store-and-forward routing is a special case of wormhole routing.

Since the model does not simulate message buffers at each PE, the store-and-forward model assumes unlimited storage at each PE, which allows messages to be blocked indefinitely while en route. In this way, the model avoids the possibility of deadlock in the store and forward model. This is admittedly an unrealistic assumption about store-and-forward processing, since all real PEs have finite storage.

11.2.2. Simulation Channels

Each channel in the network is unidirectional and allows the transmission of one flit per time step (i.e., single-accepting). The only exception to this are the channels between each PE and router

Each physical channel can control as many virtual channels as specified, though no more virtual channels than dimensions should ever be needed. At most one message can control a virtual channel at any one time step. Virtual channels are multiplexed over the physical channel by a round-robin scheduler. To maximize bandwidth, the scheduler only picks a virtual channel that has been requested to transmit a flit in the current time step.

As mentioned before, the channel between the PE and router is a special channel. It can either be made to accept one flit per time step (single-accepting), or any number of flits per time step (multiple-accepting). To simplify matters for the single-accepting option, the number of virtual channels is nv, where n is the dimension of the network and v is the number of virtual channels on channels between routers. This allows up to the maximum of n messages using the Pes channels through multiplexing. The multiple-accepting option does not use virtual channels. Instead, all incoming messages can routed simultaneously through the same router-PE channel.

11.2.3. Message Generation

The dynamic performance measurements will depend not just on the algorithm and routing strategy, but on the rate and distribution at which messages are being generated. Our simulation assumes that messages are being generated with

uniform source/destination addresses over the network. We define the probability that a message will be created:

Definition 11.2.1 *The* message generation rate *is the probability that a single processor will generate a message during one time step.*

At a high enough message generation rate, messages will be created faster than the network can transmit them, leading to a backlog of messages. The network will use every channel available to simultaneously route as many messages as possible.

Definition 11.2.2 *The* saturation rate *of a network is the message generation rate at which every channel is at 100% utilization.*

At message generation rates above the saturation rate, the number of messages in the network grows without bound.

The saturation rate is dependent not only on the network's topology, but also the routing algorithm, and the routing strategy involved. As an extreme example, consider a routing algorithm on the hypercube which routes every message through processor $000\ldots0$.

We can give several rough calculations on the upper bound for the saturation rate for the hypercube and other LE networks. A network of dimension n has 2^n nodes and $n2^n$ unidirectional channels. Assume that we have a uniform message generation rate of m_g, and that the message sources and destinations are uniformly distributed. Let the average message length be m_l and the expected distance of the network be $E(n)$.

Using the store-and-forward routing strategy, the product of the length of the message and the expected distance is the dominating factor in the duration of

the message, because the entire message has to be transmitted over each channel separately. The average number of messages that exist in the network at any time is:

$$2^n m_g \times m_l E(n)$$

Each message allocates at most one channel. At the saturation rate, all $n2^n$ channels are utilized at once, so:

$$2^n m_g \times m_l E(n) = n2^n$$

Then:

$$m_g m_l = \frac{n}{E(n)}$$

Using the wormhole routing strategy, the length of the message is the dominating factor in the duration of the message. The average number of messages that exist in the network at any time is:

$$2^n m_g \times m_l$$

Each message allocates approximately $E(n)$ channels. At the saturation rate, we again have:

$$2^n m_g m_l \times E(n) = n2^n$$

And:

$$m_g m_l = \frac{n}{E(n)}$$

So the saturation rate for the wormhole routing strategy is the same as for the store and forward routing strategy.

For example, the hypercube's expected saturation rate would be about $m_g m_l = 2$, from its expected distance of $n/2$. The *LE* networks would of course have *higher* saturation rates, theoretically.

When the processor channels are single-accepting, they create a network bottleneck. As before, the number of messages in the system at any time is:

$$2^n m_g M_l$$

There are 2^{n+1} channels between each processor and its router. Every message will utilize two of these channels for most of its transmission. These single-accepting channels will reach 100% utilization when:

$$2(2^n m_g M_l) = 2^n + 1$$

So the network will saturate at $m_g m_l = 1$. This is considerably less than the saturation rate calculated for networks with multiple-accepting processor channels. For this reason, we chose to avoid using single-accepting channels in our simulation tests.

11.3. SIMULATION RESULTS

In this section we examine the results of the simulation on some of the published *LTLE* networks. Since the store-and-forward and the wormhole routing strategies are substantially different approaches to routing messages, we will consider the dynamic behavior of the *LE* networks separately for the two routing strategies.

11.3.1. Store and Forward Routing Strategy

We first compared the published *LTLE* networks using the store-and-forward routing strategy. We used the following networks: the Hypercube, the Twisted

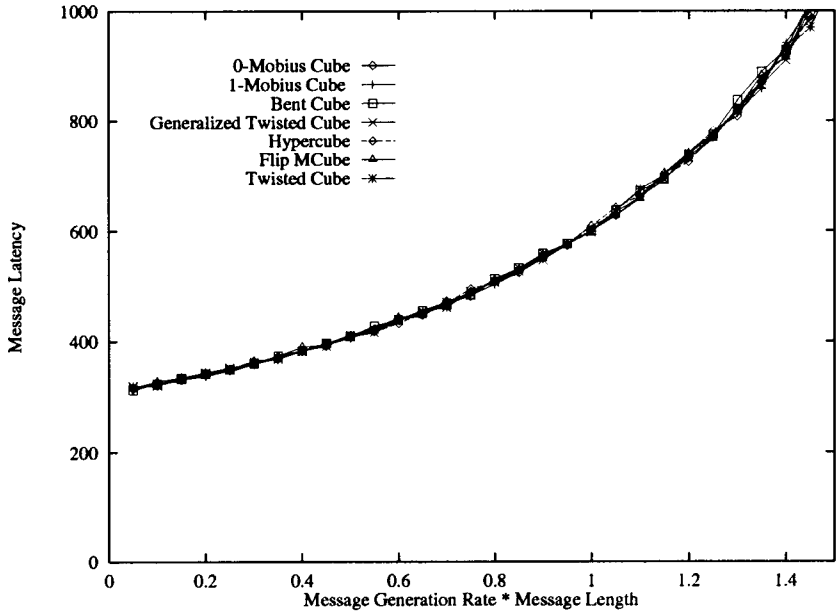

Figure 11.2. Message latencies for Algorithm LeftRightBitCorrectRoute, using the store-and-forward routing strategy.

Cube, the 0-Möbius Cube, the 1-Möbius Cube, the Flip MCube,the Generalized Twisted Cube, and the Bent Cube.

For the tests, we assumed that all networks are six-dimensional and lower triangular, that each virtual channel had a flit buffer length of 1 flit, and that all Pes were multiple-accepting. We also assumed that the messages averaged about 100 flits in length with a standard deviation of 10 flits. The simulation was run for 50,000 time steps, varying the message generation rate from 0.0 to approximately 0.0015 for each test run.

We compared the expected message latencies of networks using the three Algorithms LeftRightBitCorrectRoute, NonRedundantMinimalRoute, and Three-

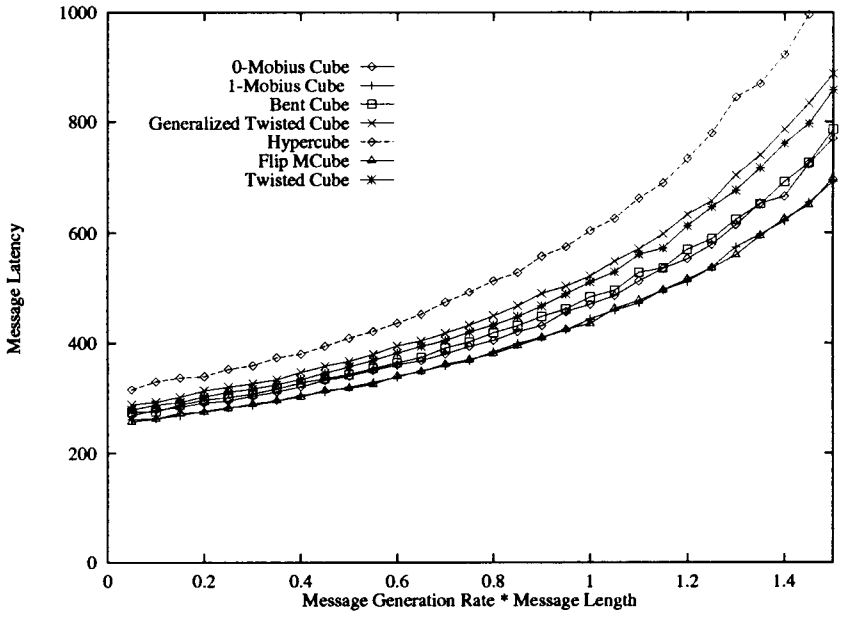

Figure 11.3. Message latencies for Algorithm NonRedundantMinimalRoute, using the store-and-forward routing strategy.

BitLookaheadRoute. The expected message latencies for the three algorithms are shown in Figures 11.2, 11.3, and 11.4, respectively.

Algorithm LeftRightBitCorrectRoute makes all the *LTLE* networks behave more or less like the hypercube, with no appreciable difference in message latency. At low message generation rates, the product of the message length and the expected routing distance is the dominant factor in the message latency. As the message generation rate increases, the *LTLE* networks show the same increase in message latency. This performance shows that all *LTLE* networks can perform at least as well as the hypercube.

Algorithm NonRedundantMinimalRoute produced the lowest message latencies of any algorithm at high message generation rates. The difference in message

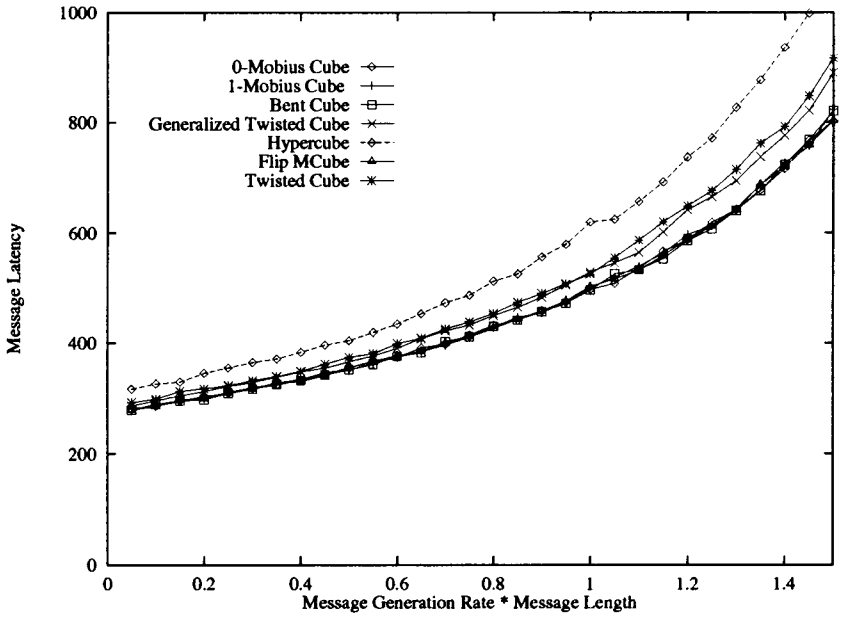

Figure 11.4. Message latencies for Algorithm ThreeBitLookaheadRoute, using the store-and-forward routing strategy.

latencies grows wider as the message generation rate grows, with the same approximate ratios between the networks. All of the *LTLE* networks produced notably lower message generation rates, with the 1-Möbius Cube and the Flip MCube showing the lowest latencies.

Networks using Algorithm ThreeBitLookaheadRoute showed expected message latencies that were similar to the same networks using Algorithm NonRedundantMinimalRoute, though slightly higher. The higher message latencies can be explained by the longer routing paths generated by Algorithm ThreeBitLookaheadRoute.

The networks fall into three groups: The first is the hypercube, which shows no change in message latencies from Algorithm LeftRightBitCorrectRoute. The

second is the Twisted Cube and the Generalized Twisted cube, which show approximately the same message latencies. The third group is formed by the rest of the networks. Notice that these networks all have both hypercube edges and twisted cube edges in dimensions 1 through 4.

Networks using Algorithm ThreeBitLookaheadRoute were able to capture most of their behavior under Algorithm NonRedundantMinimalRoute. This is especially true of the 1-Möbius cube, the Flip MCube, and the Bent Cube. This is important, because Algorithm ThreeBitLookaheadRoute requires only a constant number of bit operations per node, while Algorithm NonRedundantMinimalRoute requires heavy computation at the source node. For a slight reduction in performance, we can use this much simpler algorithm.

We also examined the channel utilization rates of networks using the three Algorithms LeftRightBitCorrectRoute, NonRedundantMinimalRoute, and ThreeBitLookaheadRoute. At a low message generation rate, the channel utilization rates are near zero for all networks. We compared the behavior of the Hypercube, the 1-Möbius cube, the Bent Cube, the Twisted Cube and the Generalized Twisted Cube.

The channel utilization rates for the hypercube appear in Figure 11.5. It shows that the hypercube uniformly utilizes all of the channels uniformly, which is due to the high symmetry of the hypercube. Not only is this typical of the hypercube for Algorithm LeftRightBitCorrectRoute, but also for the hypercube using Algorithms NonRedundantMinimalRoute and ThreeBitLookaheadRoute. Further, it is typical of channel utilization rates for the all *LTLE* networks using Algorithm LeftRightBitCorrectRoute. For this reason, we do not show results for any other network using Algorithm LeftRightBitCorrectRoute – it would be redundant.

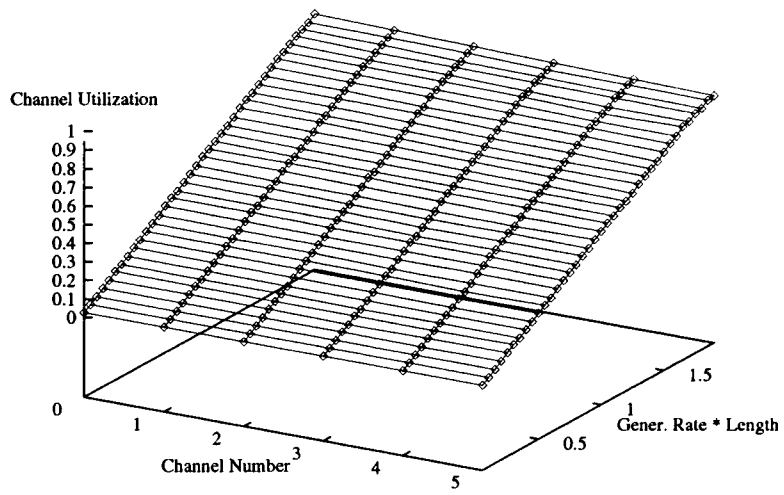

Figure 11.5. Channel utilization rates for the Hypercube, using Algorithm LeftRightBitCorrectRoute and the store-and-forward routing strategy.

Notice that the channel utilization grows linearly with the message generation rate. This indicates that the network has not reached a saturation rate in our graph, which we estimated before to be 2.

The channel utilization rates of the Twisted Cube using Algorithms NonRedundantMinimalRoute and ThreeBitLookaheadRoute are shown in Figures 11.6 and 11.7, respectively. Hilber's Twisted Cube shows some interesting results. The ratios of the channel utilizations in Figure 11.6 closely follow the measured channel utilizations given in Table 4 of [2].

We can derive the computed channel utilization for Algorithm ThreeBitLookaheadRoute, if we use the method outlined in Theorem 7.1.6. Then we get:

$$R(0) = 0.25$$

$$R(1) = 0.25$$

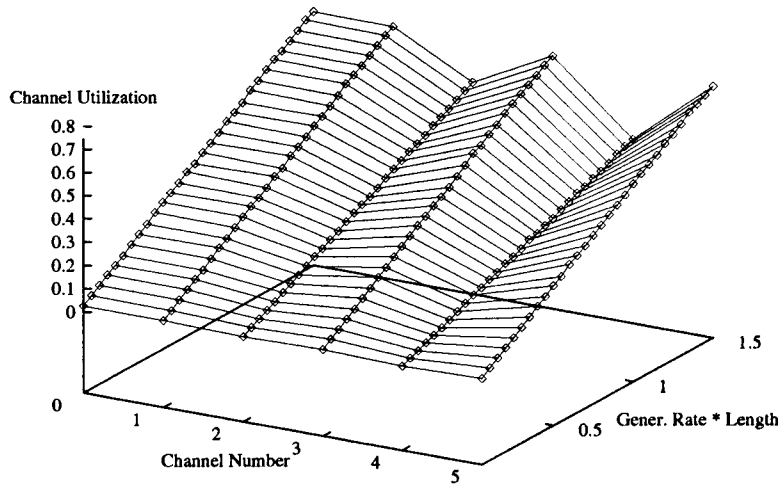

Figure 11.6. Channel utilization rates for the Twisted Cube, using Algorithm NonRedundantMinimalRoute and the store-and-forward routing strategy.

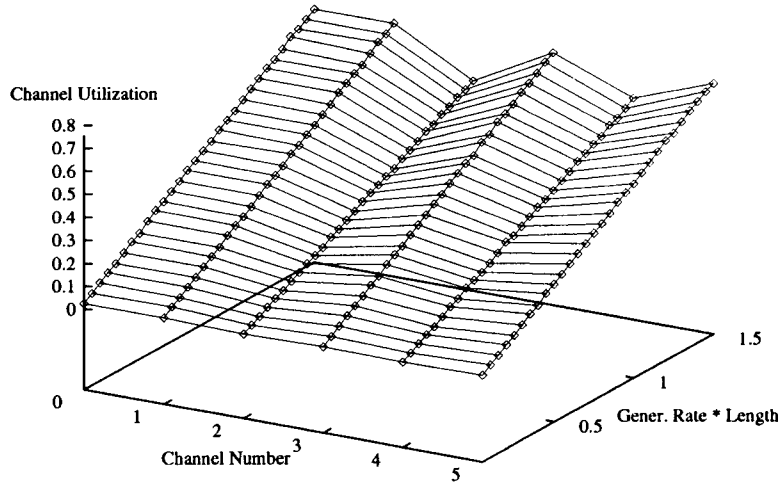

Figure 11.7. Channel utilization rates for the Twisted Cube, using Algorithm Three-BitLookaheadRoute and the store-and-forward routing strategy.

$$R(2) = 0.1875$$

$$R(3) = 0.25$$

$$R(4) = 0.21875$$

$$R(5) = 0.25$$

The ratios of the channel utilizations in Figure 11.6 closely match these values.

Notice that the dimensions which receive lower channel utilization rates are the ones immediately after dimensions that have both hypercube channels and twisted channels in the definition. This corresponds to what was predicted for Algorithm ThreeBitLookaheadRoute in Theorem 7.1.6.

We examined the standard deviation in channel utilization rates across each of the dimensions, and found that Algorithm NonRedundantMinimalRoute had the widest standard deviation .Algorithm ThreeBitLookaheadRoute had a quite low standard deviation on channel utilization rates for all channels, and was very comparable to the standard deviations generated by Algorithm LeftRightBitCorrectRoute. We conclude that this arises from the very uniform distribution of routing paths generated by both Algorithms ThreeBitLookaheadRoute and LeftRightBitCorrectRoute. This remained true, not just for the Twisted Cube, but for all of the LE networks.

The channel utilization rates of the Generalized Twisted Cube using Algorithms NonRedundantMinimalRoute and ThreeBitLookaheadRoute are shown in Figures 11.8 and 11.9, respectively. Because the Generalized Twisted Cube is a graph composition of the Twisted 3-Cube, dimensions 1, 2, 3 and dimensions 4, 5, 6 of the Generalized Twisted Cube respectively have the same expected channel utilization rates as dimensions 1, 2, 3 of the Twisted 3-Cube, or about $0.5m_g$, $0.5m_g$, and $0.325m_g$, respectively.

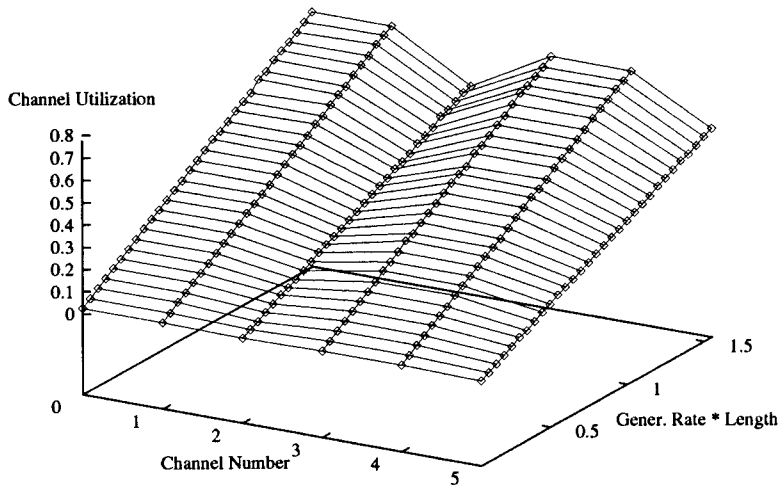

Figure 11.8. Channel utilization rates for the Generalized Twisted Cube, using Algorithm NonRedundantMinimalRoute and the store-and-forward routing strategy.

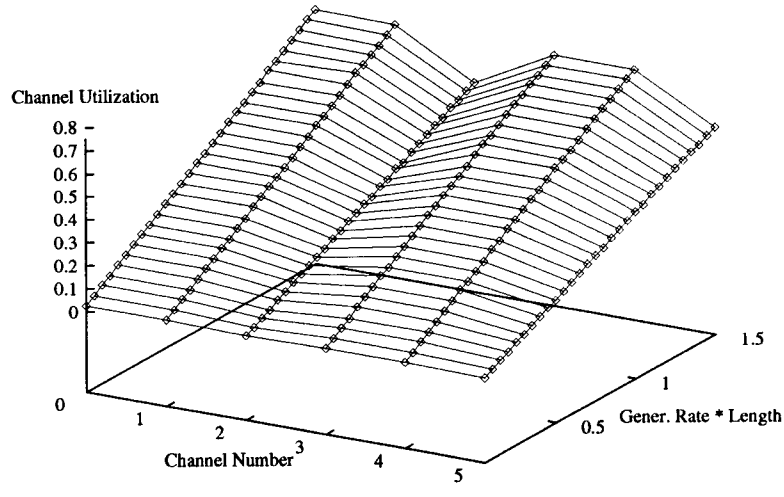

Figure 11.9. Channel utilization rates for the Generalized Twisted Cube, using Algorithm ThreeBitLookaheadRoute and the store-and-forward routing strategy.

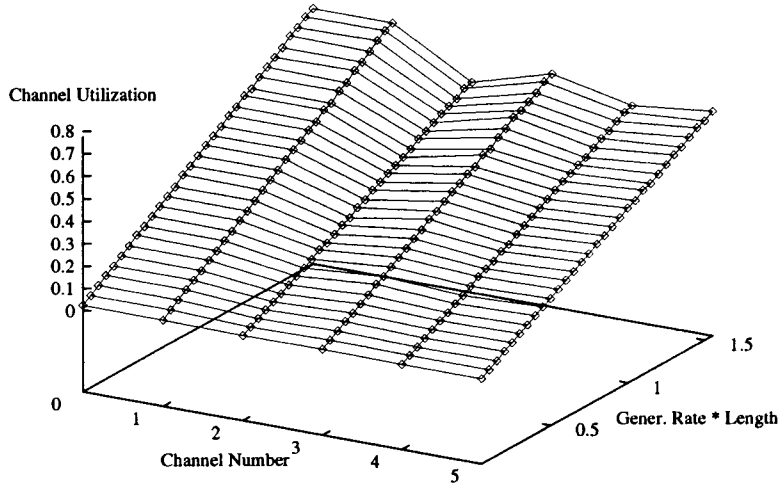

Figure 11.10. Channel utilization rates for the Bent Cube, using Algorithm NonRedundantMinimalRoute and the store-and-forward routing strategy.

The figures demonstrate that the Generalized Twisted Cube behaves the same under both algorithms. This is because looking ahead more than 3 components buys no distance savings on this network.

The channel utilization rates of the Bent Cube using Algorithms NonRedundantMinimalRoute and ThreeBitLookaheadRoute are shown in Figures 11.10 and 11.11, respectively. Similarly, the channel utilization rates of the 1-Möbius Cube are shown in Figures 11.12 and 11.13.

Note that for Algorithm ThreeBitLookaheadRoute, the ratios of the channel utilization rates of the Bent Cube and the 1-Möbius cube closely match the lower bounds on the channel utilization rates given in Theorem 7.1.6. This is no coincidence. The Bent Cube was designed from this lower bound, and the 1-Möbius cube inspired it.

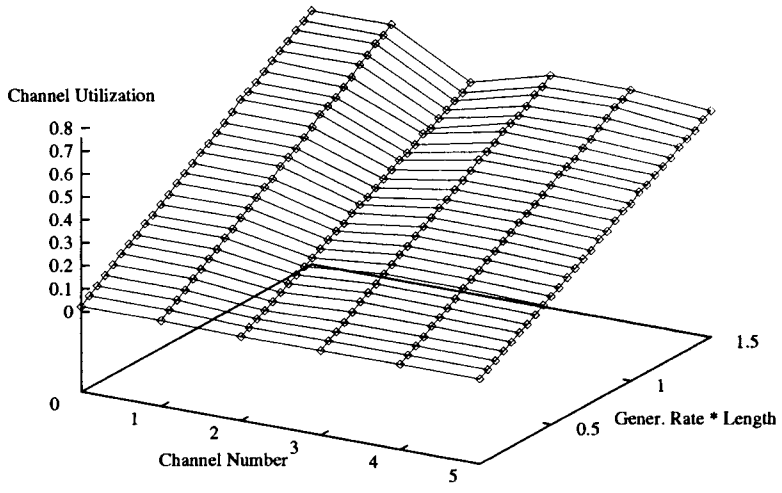

Figure 11.11. Channel utilization rates for the Bent Cube, using Algorithm Three-BitLookaheadRoute and the store-and-forward routing strategy.

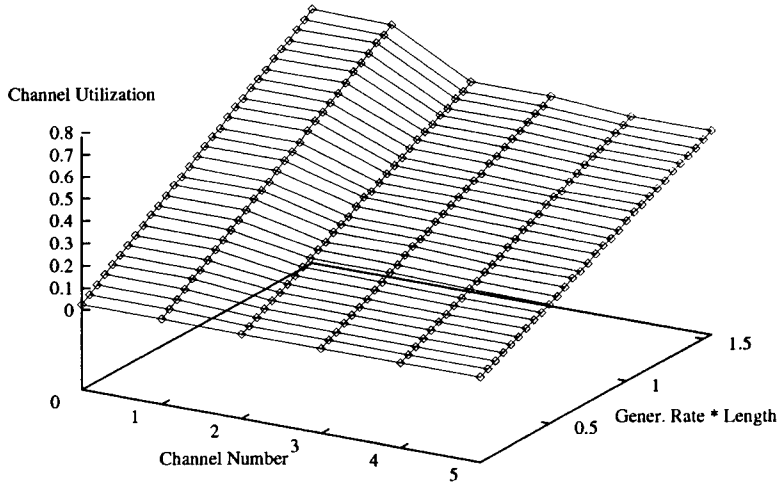

Figure 11.12. Channel utilization rates for the 1-Möbius Cube, using Algorithm NonRedundantMinimalRoute and the store-and-forward routing strategy.

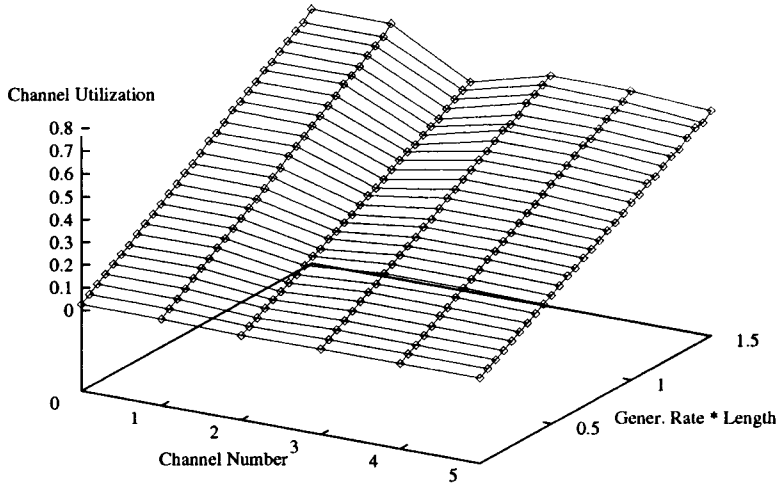

Figure 11.13. Channel utilization rates for the 1-Möbius Cube, using Algorithm ThreeBitLookaheadRoute and the store-and-forward routing strategy.

Also, note that for Algorithm ThreeBitLookaheadRoute, the channel utilization rates of the Bent Cube and the 1-Möbius cube are identical. This is because the Algorithm ThreeBitLookaheadRoute generates the same expected routing distance and the same distribution of channel utilizations for both networks.

Finally, note that for Algorithm NonRedundantMinimalRoute, the channel utilization rates of the Bent Cube and the 1-Möbius cube are still *very* similar, and only slightly better than the channel utilization rates for Algorithm ThreeBitLookaheadRoute. This helps explain why the Bent Cube has message latencies that are almost as small as the 1-Möbius cube, even though it has a maximum routing distance of about $2n/3$ instead of about $n/2$. This low channel utilization makes the Bent Cube a viable alternative to the 1-Möbius cube for wormhole routing.

The common factor here is that the Bent Cube and the 1-Möbius Cubes have twisted channels in every dimension but 0 and 5. This explains why the networks have lower utilizations in dimensions 2 through 5, which in turn help explain why the message latencies for these networks are smaller. Smaller channel utilization rates mean less delay for messages using those channels.

The channel utilization rates for the 0-Möbius cube and Flip MCube are not shown, because the 0-Möbius cube's rates are just slightly higher, and the Flip MCube's rates are identical.

11.3.2. Wormhole Routing Strategy

We next compared the behavior of the networks using the wormhole routing strategy. We again assumed that all networks are six-dimensional and lower triangular, that each virtual channel had a flit buffer length of 1 flit, and that all Pes were multiple-accepting. This time, we assumed that all physical channels had up to six virtual channels available for messages.

We again assumed that the messages averaged 100 flits in length with a standard deviation of 10 flits. The simulation was run for 50,000 time steps, varying the message generation rate from 0.0 to approximately 0.0015 for each test run.

We compared the expected message latencies of networks using the three Algorithms LeftRightBitCorrectRoute, NonRedundantMinimalRoute, and ThreeBitLookaheadRoute. The expected message latencies for the three algorithms are shown in Figures 11.16, 11.14, and 11.15, respectively.

With wormhole routing, the dominant factor in the message latency at low message generation rates is the message's length. This is borne out by the near-identical performance of all the networks at the lower rates. This makes the wormhole routing strategy preferable to the store-and-forward routing strategy, if the

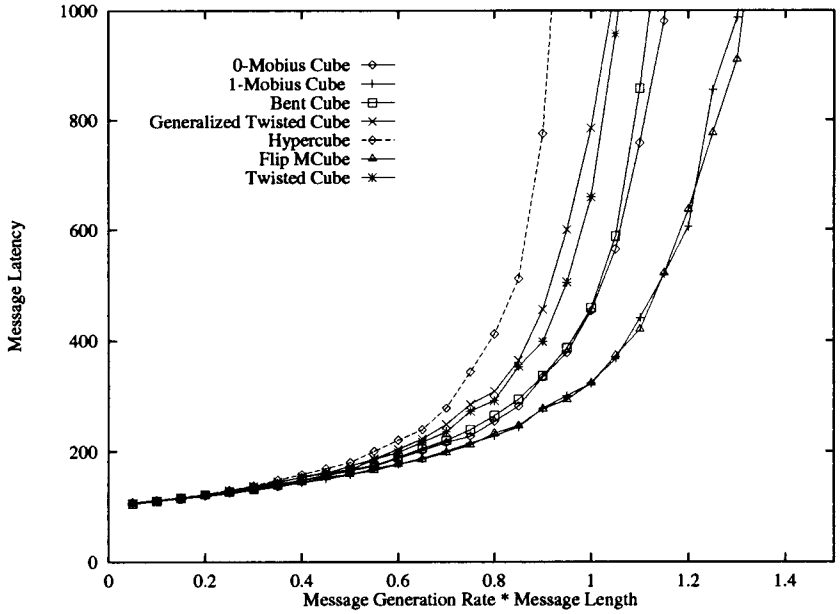

Figure 11.14. Message latencies for Algorithm NonRedundantMinimalRoute, using the wormhole routing strategy.

message-passing rates are kept suitably low. However, at higher message generation rates, the expected message latencies grow at a much faster rate for the wormhole routing strategy. All of the networks begin to saturate much sooner.

This lower saturation rate can be attributed to the nature of wormhole routing. Since under wormhole routing, a message can take multiple channels simultaneously, the chance of contention for a given channel will be greater. Since messages that are waiting for access to a channel will not release the channels they already have, the possibility of contention can increase considerably.

Wormhole routing will suffer more performance degradation at high message passing rates than will store-and-forward routing. Since all the channels in a message's path will be allocated at one time, the number of messages that can simultaneously coexist in the network without contention is much smaller than for

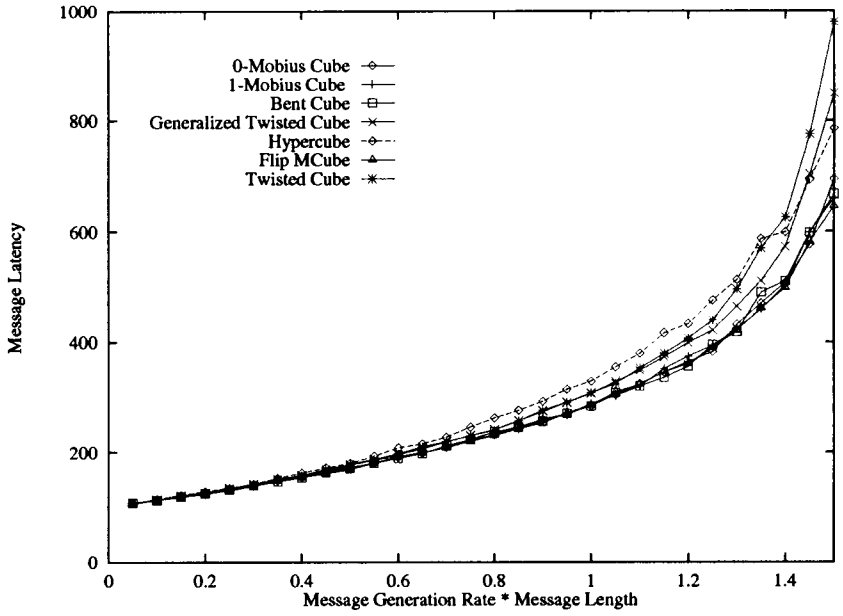

Figure 11.15. Message latencies for Algorithm ThreeBitLookaheadRoute, using the wormhole routing strategy.

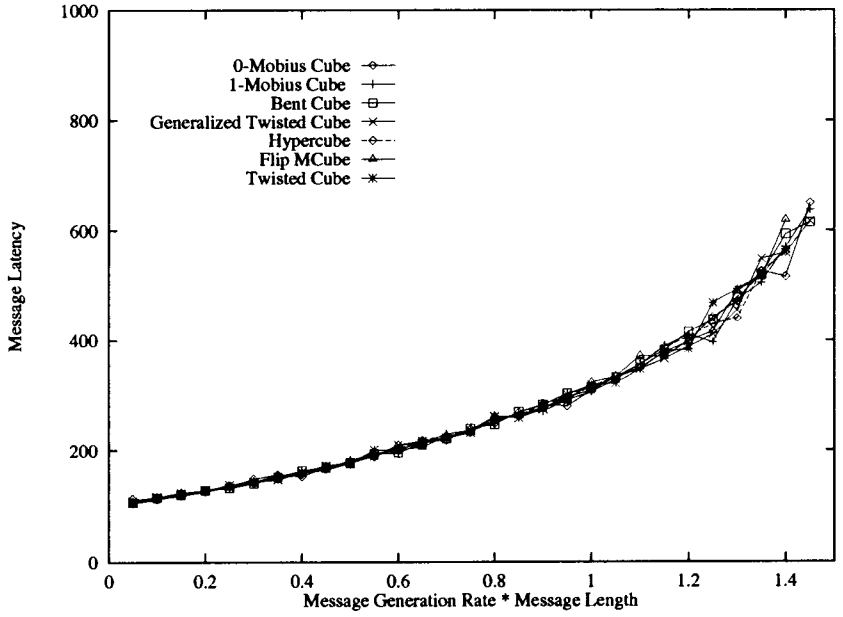

Figure 11.16. Message latencies for Algorithm LeftRightBitCorrectRoute, using the wormhole routing strategy.

the store-and-forward routing strategy. Under store-and-forward message passing, up to $n2^n$ can be transmitted simultaneously. Under wormhole message passing, up to approximately $n2^n/E(n) = O(2^n)$ messages can be transmitted simultaneously.

The network latencies for networks using Algorithm NonRedundantMinimalRoute are abysmally large, because the algorithm is using a naive method of assigning virtual channels.

Networks using Algorithm ThreeBitLookaheadRoute behave very similarly when using wormhole routing. The Twisted Cube and the Generalized Twisted Cube show by far the highest message latencies of the measured networks. The best behavior is by the 1-Möbius Cube and the Flip MCube, followed by the Bent Cube and the 0-Möbius Cube.

We again examined the channel utilization rates of networks using the three Algorithms LeftRightBitCorrectRoute, NonRedundantMinimalRoute, and ThreeBitLookaheadRoute. We compared the behavior of the Hypercube, the 1-Möbius cube, the Bent Cube, the Twisted Cube and the Generalized Twisted Cube. At a low message generation rate, the channel utilization rates are near zero for all networks.

For comparison with the store-and-forward routing strategy, we include the simulation results for all of the same networks, this time using the wormhole routing strategy.

The channel utilization rates for the hypercube appear in Figure 11.17. Again the hypercube uniformly utilizes all of the channels, and again the channel utilization grows linearly with the message generation rate.

The channel utilization rates of various Twisted Cubes using Algorithms NonRedundantMinimalRoute and ThreeBitLookaheadRoute are shown in Figures 11.18 through 11.25.

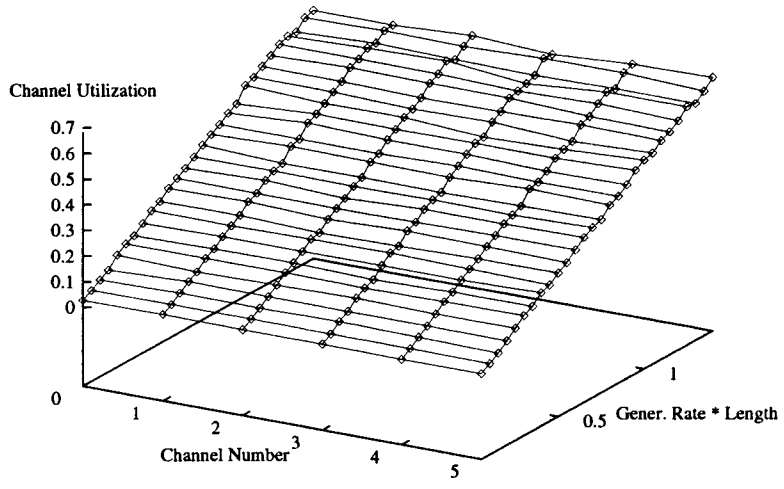

Figure 11.17. Channel utilization rates for the Hypercube, using Algorithm Left-RightBitCorrectRoute and the wormhole routing strategy.

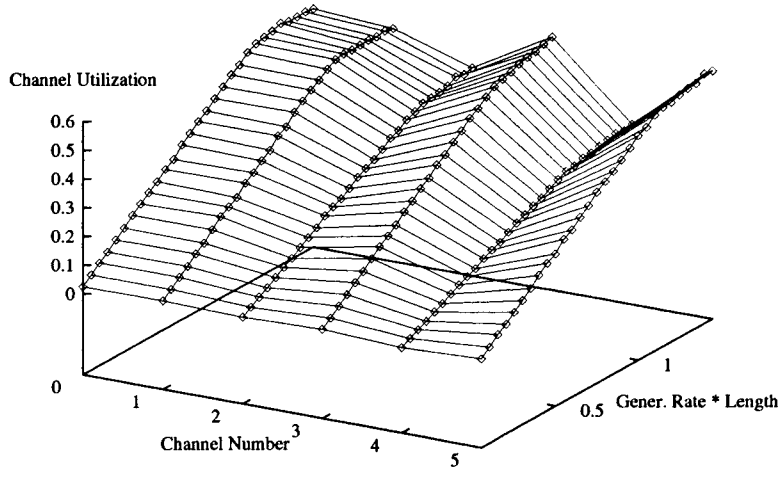

Figure 11.18. Channel utilization rates for the Twisted Cube, using Algorithm NonRedundantMinimalRoute and the wormhole routing strategy.

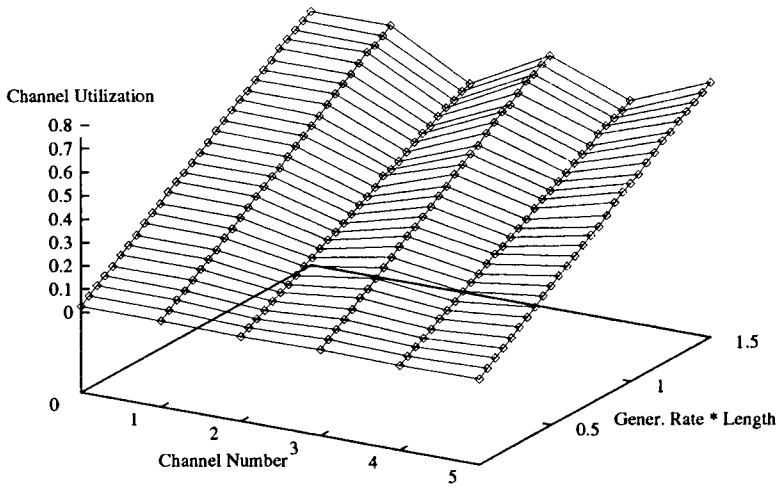

Figure 11.19. Channel utilization rates for the Twisted Cube, using Algorithm ThreeBitLookaheadRoute and the wormhole routing strategy.

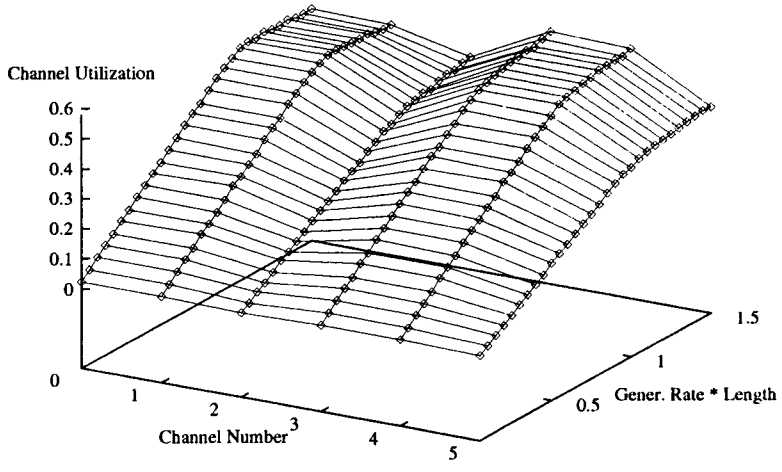

Figure 11.20. Channel utilization rates for the Generalized Twisted Cube, using Algorithm NonRedundantMinimalRoute and the wormhole routing strategy.

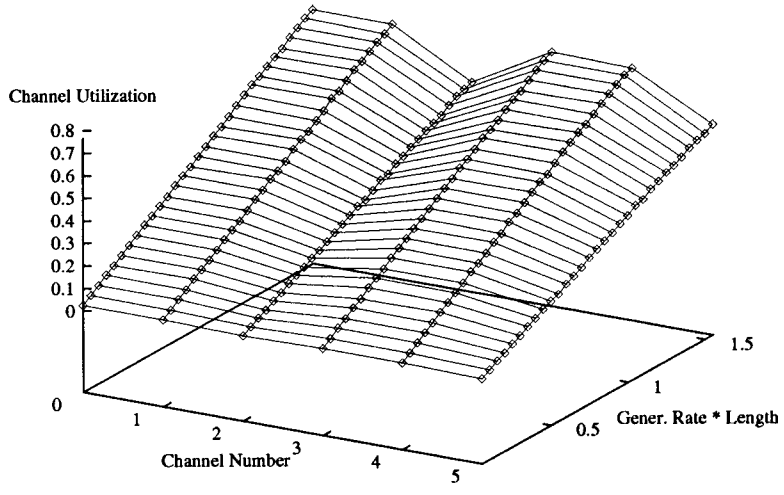

Figure 11.21. Channel utilization rates for the Generalized Twisted Cube, using Algorithm ThreeBitLookaheadRoute and the wormhole routing strategy.

For Algorithm ThreeBitLookaheadRoute the channel utilization rates are nearly identical to the channel utilization rates for store-and-forward routing, for every network. This indicates that the algorithm is suitable for both routing strategies.

For Algorithm NonRedundantMinimalRoute, the message utilization is much higher. This is because of the way that virtual channels were arbitrarily chosen for each message, which causes even higher message latencies. Notice that the channel utilization rates "flatten out" sharply for the Twisted Cube and the Generalized Twisted Cube. This explains why the message latencies rose so quickly for networks. Some of the channels in each network had begun to saturate at fairly low message passing rates. This "flattening" is less pronounced in the other networks, which have correspondingly lower message latencies.

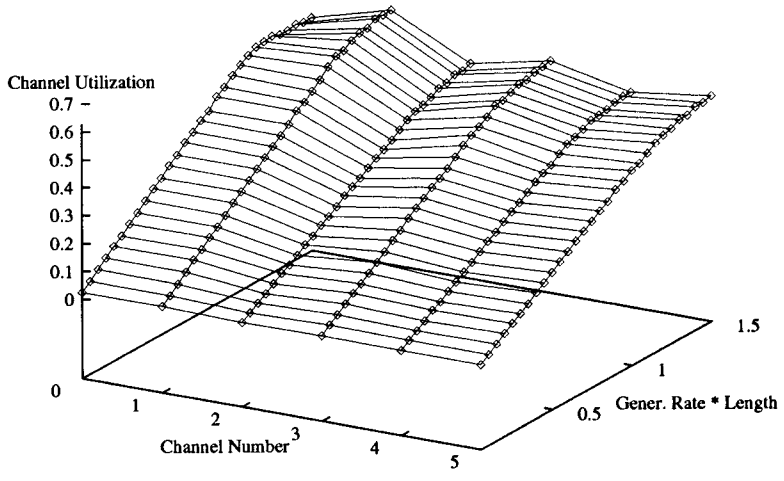

Figure 11.22. Channel utilization rates for the Bent Cube, using Algorithm NonRedundantMinimalRoute and the wormhole routing strategy.

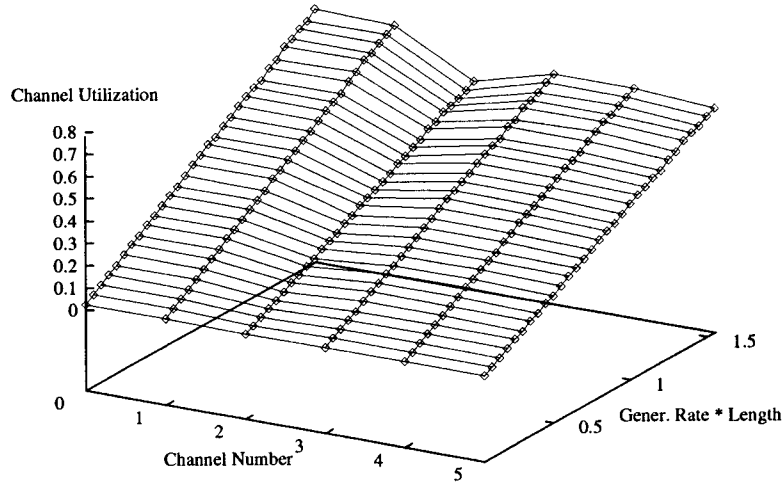

Figure 11.23. Channel utilization rates for the Bent Cube, using Algorithm ThreeBitLookaheadRoute and the wormhole routing strategy.

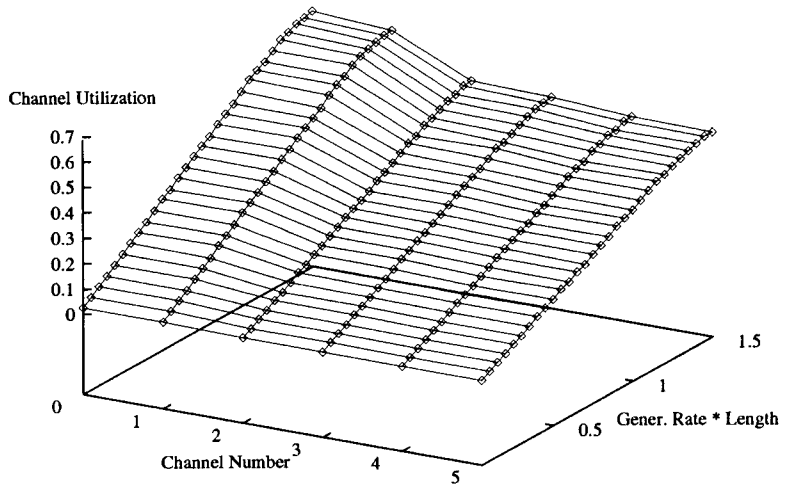

Figure 11.24. Channel utilization rates for the 1-Möbius Cube, using Algorithm NonRedundantMinimalRoute and the wormhole routing strategy.

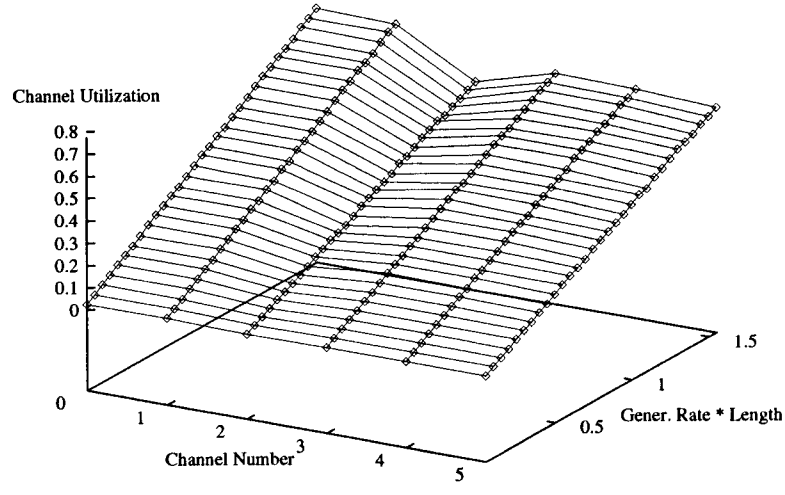

Figure 11.25. Channel utilization rates for the 1-Möbius Cube, using Algorithm ThreeBitLookaheadRoute and the wormhole routing strategy.

11.3.3. Conclusions

There are several conclusions we reach here. First, all of the *LTLE* networks perform approximately the same using Algorithm LeftRightBitCorrectRoute. This suggests that any *LTLE* network can be used in place of the hypercube for general routing, with about the same performance measures.

We also concluded that for store-and-forward routing, Algorithm NonRedundantMinimalRoute gives a dynamic performance comparable to Algorithm LeftRightBitCorrectRoute on the hypercube. The message latency for most networks is consistently lower than the message latency of the hypercube, except possibly at the very highest message generation rates. The only exceptions to this appear to be the Twisted Cube and the Generalized Twisted Cube, which have an asymmetric distribution of channel utilization rates.

The very best *LTLE* network appears to be the 1-Möbius cube (or its isomorphic twin, the Flip MCube). Its message latency using Algorithm NonRedundantMinimalRoute is much lower than the rest of the networks for store-and-forward message passing.

If wormhole routing is preferred, then the best approach appears to be using either the 1-Möbius Cube or the Bent Cube, and Algorithm ThreeBitLookaheadRoute. The Bent Cube is quite comparable to the 1-Möbius Cube in performance. Its larger diameter and expected distances do not add much to its expected message latency when using Algorithm ThreeBitLookaheadRoute. We conclude that having a large number of dimensions with both twisted and hypercube channels is more important than a small diameter when it comes to overall performance under wormhole routing.

We also feel that adapting Algorithm NonRedundantMinimalRoute for wormhole routing does not appear to be a practical consideration. This is partly because there does not seem to be an effective means of limiting the number of virtual channels to a constant, but more because a wormhole algorithm implicitly requires that each node on the routing path uses a small number of bit operations for computations. Algorithm NonRedundantMinimalRoute takes a relatively large number of bit operations and requires that the entire path be precomputed at the source node.

Finally, we conclude that Algorithm ThreeBitLookaheadRoute gives the best performance for the least amount of work. It requires a total of $O(n)$ bit operations to compute, which can be distributed across the nodes in the message's routing path. It also captures most of the dynamic behavior of Algorithm NonRedundant-MinimalRoute, which means that an algorithm with further lookahead would give only a diminishing return for the extra work done.

12. CONCLUSION

In this short chapter, we review what we have accomplished in our research. We also discuss open problems for *LE* networks, and where future research should be concentrated. Finally, we discuss the practicality and viability of implementing networks in the Twisted Cube family.

12.1. RESEARCH ACCOMPLISHED

In this dissertation, we have designed the linear equation networks (or *LE* networks), a new model for describing resource-preserving hypercube-variant interconnection networks, by using a system of linear equations to define the network's communications channels. We have shown that a number of existing networks can be expressed in our model, and have also designed a number of new networks using this model. We have also shown sufficient conditions for certain network properties, including connectedness and bidirectionality (reciprocality) of channels. We have also shown some conditions and methods for showing network isomorphism.

We have managed to show that a number of basic results were *NP*-complete or *NP*-hard, including:

- (Limited) path existence on *LE* networks
- Isomorphism of *LE* networks and *LTLE* networks
- (Limited) minimal routing on *LE* networks
- Minimal routing on *LTDM* networks and on *LTLE* networks

Many of these results seem to stem from one problem with the *LE* networks: Local linearity in the network connections did not lead to global linearity.

Despite these problems, we managed to show a number of results for network properties. In particular, by limiting ourselves to a subclass of the *LE* networks, the *LTLE* networks, we managed to solve many of our problems, including network connectedness, bidirectionality of channels, and routing.

We were able to produce results in several areas. These areas include:

Routing Algorithms We produced a minimal routing algorithm for the *LTDM* networks and the *LTLE* networks, and showed that it used a polynomial asymptotic run time for all of the published networks. We also designed non-minimal routing algorithms – the "3-bit lookahead" algorithm – for the *LTLE* networks that had the same run time order as the hypercube's "greedy" routing algorithm. We also designed deadlock-free routing algorithms for networks using the wormhole routing strategy. Finally, we designed broadcasting algorithms that take advantage of the *LTLE* networks topology and execute in fewer communication steps than on the hypercube.

Embeddings and Emulations We were able to show that all of the *LTLE* networks can embed Hamiltonian rings and binomial trees. We were also able to show that some of the *LTLE* networks are able to embed full binary trees and other are able to embed meshes. Finally, we were able to show that some *LTLE* networks are able to simulate the hypercube with constant dilation, and vice versa.

Performance We bounded the diameter and expected routing distance for some of the *LE* networks (and in the process showed that some *LTLE* networks are able to have a diameter of less than $n/2$, though not much less).

We also wrote a program for simulating the behavior of routing algorithms on the *LTLE* networks. Using this program, we were able to show that the 3-bit lookahead captures most of the behavior of minimal routing on the *LTLE* networks, and saves about 13 to 15 percent in message latency. We also discovered that it is more important to have many dimensions with both hypercube channels and twisted channels, than it is to have a low diameter.

12.2. OPEN PROBLEMS

There are a number of problems that remain open for the *LE* networks. Though we spent many weeks on these problems, we were not able to find any satisfactory solutions. These problems include:

Connectedness of *LE* networks This is one of the most important problems we were unable to solve. Currently, the only way to tell if a *LE* network is connected is to construct its graph from a matrix description and run a connected components algorithm on the graph. Though we were able to prove some necessary and some sufficient conditions to make the matrix description produce a connected *LE* network, we were unable to show simultaneous necessary and sufficient conditions.

Upper bound on diameter for connected *LE* networks Though a trivial upper bound of $2^n - 1$ exists for *LE* networks, we were not able to design a network that had a diameter of more than $O(n^2)$ steps. We were also unable to show that a polynomial upper bound on the diameter exists, though we believe that the bound is polynomial. The solution to this problem may show that the connectedness problem is *NP*-complete.

Lower bound on diameter for *LE* networks Though we were able to show that a network with a diameter of approximately $3n/5$ does exist, we did not show the lower bound on diameter for *LE* networks was tight. If there is a smaller diameter possible on *LE* networks, what is it?

Minimal and Deadlock-Free Wormhole Routing We would like to be able to do minimally routing on any *LTLE* networks with a small (constant) number of virtual channels. The naive approach does not work very well. A minimal wormhole algorithm that shows good dynamic performance would go a long distance towards gaining acceptance for *LE* networks.

12.3. FUTURE WORK

Besides the problems above that we did consider and did not solve, there were a number of problems we did not consider, partly due to time, and also partly due to the amount of work already done. These problems included:

Other Network Embeddings We considered only the most common networks for embedding into *LE* networks. It would be worthwhile to explore embedding other networks, into either the entire *LE* network family, or into a specific *LE* network.

Other Communications Algorithms Parallel algorithms are often not written specifically for the architecture they are run upon. Instead, they use a "library" of standard communications routines. These routines include a number of communications patterns, such as:

- **Single Node Broadcast:** Also known as one-to-many routing. One processor has a single message which is sent to all other processors in the network.

- **Single Node Scatter:** One processor has 2^n messages, and each message is sent to a different processor in the network.
- **Multiple Node Broadcast:** Every processor has one message, and each processor broadcasts its message to all other processors in the network.
- **Total Exchange:** Every processor has 2^n messages, and each processor scatters its messages to all other processors in the network.

We did not consider the implementation of these general communications algorithms for the *LE* networks. Using multiple-channel communications, the minimal hypercube algorithms for these communication patterns require the computation of the maximal number of independent paths from a single node [32], [38], [39]. Though the problem of computing independent paths is well-known for the hypercube, we did not find an simple, general method for computing independent paths on a general *LE* network. Thus the minimal algorithms for these problems remain unknown, and these other communications algorithms are outside the scope of this dissertation.

General Parallel Algorithms Probably the most practical outstanding task is finding algorithms that can effectively use the properties of the *LE* networks. We considered mapping only one or two of the simpler parallel algorithms onto the *LE* networks. The problem is that these problems are relatively simple and isolated. Are there any significant algorithms for *LE* networks?

There are several things that can be done in this direction: First, an algorithm could be mapped to a general *LE* network so that it could execute with the same or fewer communication steps than the hypercube. Second, a *LE* network could be specifically designed to run a given parallel algorithm in fewer communication steps than the hypercube.

Processor Layout We did not consider the problem of processor layout for the *LE* networks. The hypercube and other networks are often implemented as VLSI circuits. It is important to performance that the processors are arranged to minimize the longest physical channel length and minimize the total layout area. Though we probably could modify the hypercube's processor layouts for networks with a small number of twisted channels (each with a small Hamming weight), it is not clear we could do so for networks with a large proportion of twisted channels.

Extending the *LE* Network Simulation The simulation program was quite general, in that we covered several networks and algorithms using different routing strategies. However, the simulations are unrealistic in that they test only uniformly distributed message transmissions. Most algorithms use highly structured communication patterns. We could extend the program to simulate non-uniform communication distributions, including random "hot-spot" sources and destinations, and distributions based on actually parallel algorithms.

Though we do plan to eventually look at these other problems, we decided that each of the above projects was too involved to consider in this dissertation.

12.4. EVALUATION

There are a number of summary conclusions we have about the *LE* networks. These conclusions are mostly about the implementation of the routing algorithms on *LE* networks and the implementation of *LE* networks as scalable multicomputers.

Our first conclusion is that the *LE* model is too general for rigorously proving any properties. The full *LE* model allows networks that can be disconnected or

weakly connected, and can allow several matrix representations to describe the same network. Further, its generality makes it difficult to prove even the simplest of network properties.

We also conclude that minimal routing is not always the best way to do routing. We've already shown that the three bit lookahead routing algorithm captures most of the behavior of the minimal routing algorithm. Its worst case routing behavior (as measured by the diameter) is at most 16% worse than the minimal routing algorithm, and for store-and-forward routing, its average message latency is often better than the minimal algorithm's message latency. Since the three bit lookahead routing algorithm also has the advantages of having a faster asymptotic run time and not needing path pre-computation at the source node, we recommend it over the minimal routing algorithm.

Also, we recommend the implementation of the reconfigurable network given in Section 9.3. Though this network has more complicated hardware, it allows the emulation of several networks, rather than the implementation of one. It even allows the implementation of networks that are not in the LE model. Further, it allows "the best of both worlds", by using dynamic reconfiguration of the network. We can always run the best communication algorithm on the best network to solve a particular problem, even reconfiguring the network during an algorithm to optimize the communications.

Finally, we recognize that the LE networks will probably will not be used for interconnection network design. Its biggest draw, the reduced diameter, is rendered superfluous by the fact that most actual networks now use wormhole routing for point-to-point communications. Though there is some savings in message latency offered by the wormhole implementation of the three-bit lookahead algorithm, it may or may not be worth the asymmetry that is inherent in the LE network model.

Probably the results of this dissertation will be more useful in other areas of interconnection networks. Already, at least one *LE* network has been used in programming fault-tolerant behavior into Folded Hypercubes [40]. There is every reason to believe that the same results could be repeated with any of the networks here and the Enhanced Hypercubes. Finally, there is every reason to believe that the basic premise of our dissertation, that of choosing channel connections by using linear equations, can be extended to generate and describe new interconnection networks.

12.5. REFERENCES

[1] Seth Abraham. Issues in the architecture of direct interconnection schemes for multiprocessors. Technical Report 977, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL 61801-2392, 1990.

[2] Seth Abraham and Krishnan Padmanabhan. An analysis of the twisted cube topology. In *1989 International Conference on Parallel Processing*, volume 1, pages 116–120. Pennsylvania State Press, 1989.

[3] Seth Abraham and Krishnan Padmanabhan. Performance of the direct binary n-cube network for multiprocessors. *IEEE Transactions on Computers*, 38(7):1000–1011, Jul 1989.

[4] Alfred V. Aho and John E. Hopcroft amd Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Co., Reading, Massachusettes, 1974.

[5] Sheldon B. Akers and Balakrishnan Krishnamurthy. A group-theoretic model for symmetric interconnection networks. *IEEE Transactions on Computers*, 38(4):555–566, Apr 1989.

[6] Ahmed Al-Amaway and Sharam Latifi. Properties and performace of folded hypercubes. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):31–42, Jan 1991.

[7] Brian Alspatch. Cayley graphs with optimal fault-tolerance. *IEEE Transactions on Computers*, 41(10):1337–1339, Oct 1992.

[8] M. Baumslag. Processor-time tradeoffs for Cayley graph interconnection networks. In *The Sixth Distributed Memory Computing Conference Proceedings*, pages 630–636, Portland, Oregon, 1991. Springer-Verlag.

[9] Elwyn R. Berlekamp, Robert J. McEliece, and Henk C. A. Van Tilborg. On the inherent intractibility of certain coding problems. *IEEE Transactions on Information Theory*, 24(3):384–386, May 1978.

[10] M.R. Brown. Implementation and analysis of binomial queue algorithms. *SIAM Journal on Computing*, 7(8):298–319, Aug 1978.

[11] by J. Köbller, U. Schöning, and J. Torán. *The Graph Isomorphism Problem.* Birkhäuser, 1993.

[12] Fouad B. Chedid and Riad B. Chedid. A new variation on hypercubes with smaller diameter. *Information Processing Letters*, 46(7):275–280, July 1993.

[13] Shou-Yi Cheng and Jen-Hui Chuang. Varietal hypercubes - a new interconnection networks topology for large scale multicomputer. In *ICPADS'94: International Conference on Parallel and Distributed Systems*, pages 703–708. IEEE Computer Society Press, 1994.

[14] P. Cull and S. Larson. The Möbius cubes. *IEEE Transactions on Computers*, 44(5):647–659, May 1995.

[15] Paul Cull and Shawn Larson. The Möbius cube. Poster session at *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, Dec 1990.

[16] Paul Cull and Shawn Larson. The Möbius cube. Technical Report 91-20-02, Department of Computer Science, Oregon State University, Dec 1991.

[17] Paul Cull and Shawn Larson. The Möbius cube: An interconnection network for parallel processing. Technical Report 91-20-2, Department of Computer Science, Oregon State University, 1991.

[18] Paul Cull and Shawn Larson. The Möbius cubes. In *6th Distributed Memory Computing Conference*, pages 699–702, Portland, Oregon, 1991. IEEE Computer Society Press.

[19] Paul Cull and Shawn Larson. The Möbius cubes: Improved cubelike networks for parallel computation. In *6th International Parallel Processing Symposium*, pages 610–613. IEEE Computer Society Press, 1992.

[20] Paul Cull and Shawn Larson. The Möbius cubes: Improved cubelike networks for parallel computation. Technical Report 92-20-01, Department of Computer Science, Oregon State University, 1992.

[21] Paul Cull and Shawn Larson. The Möbius cubes. Technical Report 93-20-01, Department of Computer Science, Oregon State University, 1993.

[22] Paul Cull and Shawn Larson. The Möbius cubes: New twisted interconnection networks for parallel computation. Technical Report 92-20-03, Department of Computer Science, Oregon State University, 1993.

[23] Paul Cull and Shawn Larson. Static and dynamic performance of the Möbius cubes (long version). Technical Report 93-20-2, Department of Computer Science, Oregon State University, 1993.

[24] Paul Cull and Shawn Larson. Static and dynamic performance of the Möbius cubes (short version). In *PARLE'93: Parallel Languages and Architectures Europe*, pages 92–103. Springer-Verlag, 1993.

[25] Paul Cull and Shawn Larson. Static and dynamic performance of the Möbius cubes (short version). Technical Report 93-20-3, Department of Computer Science, Oregon State University, 1993.

[26] Paul Cull and Shawn Larson. A linear equation model for twisted cube networks. In *ICPADS'94: International Conference on Parallel and Distributed Systems*, pages 708–714. IEEE Computer Society Press, 1994.

[27] Paul Cull and Shawn Larson. Wormhole routing algorithms for twisted cube networks. In *Proceedings of the 1994 Symposium for Parallel and Distributed Processing*, pages 696–703. IEEE Computer Society Press, 1994.

[28] Rajib K. Das, Krishna Mukhopadhyaya, and Bhabani P. Sinha. A new family of bridged and twisted hypercubes. *IEEE Transactions on Computers*, 43(10):1240–1247, Oct 1994.

[29] Kemal Efe. A variation on the hypercube with lower diameter. *IEEE Transactions on Computers*, 40(11):1312–1316, Nov 1991.

[30] Kemal Efe. The crossed cube architecture for parallel computation. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):513–524, Sep 1992.

[31] Abdol-Hossein Estafahanian, Lionel M. Ni, and Bruce Sagan. The twisted N-cube with application to multiprocessing. *IEEE Transactions on Computers*, 40(1):88–93, Jan 1991.

[32] D.P. Bertserkas *et al.* Optimal communication algorithms for hypercubes. *Journal of Parallel and Distributed Computing*, 11(4):263–275, Apr 1991.

[33] M. Garey and D. Johnson. *Computers and Intractibility: A Guide to the Theory of* NP*-Completeness.* Freeman and Co., San Francisco, 1979.

[34] Peter A.J. Hilbers, R.J. Marion Koopman, and Jan L.A. Van de Snepscheut. The twisted cube. In J. deBakker, A. Numan, and P. Trelearen, editors, *PARLE: Parallel Architecture and Languages Europe, Volume 1: Parallel Architectures*, pages 152–158, Berlin, W. Germany, 1987. Springer-Verlag.

[35] Daniel W. Hillis. *The Connection Machine (Computer Architecture for the New Wave)*, Sep 1981.

[36] Ching-Tien Ho. An observation on the bisectional interconnection networks. *IEEE Transactions on Computers*, 41(7):873–877, July 1992.

[37] Jim Holloway. Private Communication, 1987.

[38] S.L. Johnsson and C.T. Ho. Optimum broadcasting and personalized communication in hypercubes. *IEEE Transactions on Computers*, 38(9):1249–1268, Sep 1989.

[39] S.L. Johnsson and C.T. Ho. Optimum all-to-all personalized communication with minimum span on Boolean cubes. In *The 6th Distributed Memory Com-*

puting Conference Proceedings, pages 299–305, Portland, Oregon, 1991. IEEE Computer Society Press.

[40] Jong Kim and Kang G. Shin. Operationally enhanced folded hypercubes. *IEEE Transactions on Parallel and Distributed Systems*, 5(12):1310–1316, Dec 1994.

[41] J. Mohan Kumar and L. M. Patnaik. Hierarchical network of hypercubes with folded connections. In *Proceedings of the 6th International Parallel Processing Symposium: Parallel Systems Fair*, pages 33–37, The Beverly Hilton, Beverly Hills, CA, Mar 1992.

[42] Shawn M. Larson. The Möbius cube: An interconnection network for parallel processing. Master's thesis, Department of Computer Science, Oregon State University, 1991.

[43] Shawn M. Larson. *A General Linear Equation Model for Hypercube-Variant Networks.* PhD thesis, Department of Computer Science, Oregon State University, 1994.

[44] Shahram Latifi. The efficiency of the folded hypercube in subcube allocation. In *1990 International Conference on Parallel Processing*, volume I, pages 218–221. Pennsyvania State University Press, 1990.

[45] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures.* Morgan Kaufman Publishers, San Mateo, CA, 1992.

[46] Lionel M. Ni and Philip K. McKinley. A survey of wormhole routing techniques in direct networks. *IEEE Computer*, 26(2):62–76, Feb 1993.

[47] Krishnan Padmanabhan. Cube structures for multiprocessors. *Communications of the ACM*, 33(1):43–52, Jan 1990.

[48] Franco F. Preparata and Jean Vuillemin. The cube-connected cycles: A versatile network for parallel computation. *Communications of the ACM*, 24(5):300–309, May 1981.

[49] Youcef Saad and Martin H. Schultz. Topological properties of hypercubes. *IEEE Transactions on Computers*, 37(7):867–872, Jul 1988.

[50] M.R. Samatham and D.K. Pradhan. The de Bruijn multiprocessor network: A versatile parallel processing and sorting network. *IEEE Transactions on Computers*, 38(4):567–581, Apr 1989.

[51] Mitin K. Singhvi and Kanad Ghose. The MCube. Technical report, State University of New York, 1992.

[52] M. A. Sridhar and C.S. Raghavendra. Fault-tolerant networks based on the de Bruijn graph. *IEEE Transactions on Computers*, 40(10):1167–1174, Oct 1991.

[53] Nian-Feng Tzeng, Hsing-Lung Chen, and Po-Jen Chuang. Embeddings in incomplete hypercubes. In *1990 International Conference on Parallel Processing*, volume III, pages 335–339. Pennsylvania State University Press, 1990.

[54] Nian-Feng Tzeng and Sizheng Wei. Enhanced hypercubes. *IEEE Transactions on Computers*, 40(3):284–294, Mar 1991.