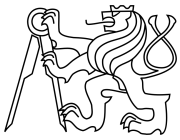
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF MASTER'S THESIS

Title:	Routing in Mobius Cubes
Student:	Bc. David Kocík
Supervisor:	prof. Ing. Pavel Tvrdík, CSc.
Study Programme:	Informatics
Study Branch:	System Programming
Department:	Department of Theoretical Computer Science
Validity:	Until the end of winter semester 2017/18

Instructions

Perform a survey of the Mobius cube theory [1]. Focus on the analysis of routing algorithms for Mobius cubes published in [1] and describe their properties and drawbacks. Design then variations of routing algorithms with respect to the minimality and deadlock freeness. In the C/C++ language, design and implement a library simulating the Mobius cube routing and using this library perform an analysis of properties of the designed routing algorithms and a comparison with routing in [1]. For the designed routing algorithms, study the problem of round-optimal multicast operation in 1-port Mobius cubes and formulate conclusions.

References

[1] P.Cull and S.M.Larson. THe Mobius Cubes, IEEE Transactions on Computers, vol. 44, No.5, pp. 647-659.

L.S.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague March 1, 2016

Czech Technical University in Prague
Faculty of Information Technology
Department of theoretical computer science

Master's thesis

Routing in Möbius Cubes

Bc. David Kocík

Supervisor: prof. Ing. Pavel Tvrdík, CSc.

30th June 2016

Acknowledgements

I would like to thank to prof. Ing. Pavel Tvrdík, CSc. for helping me and advising me during writing this thesis. Also my thanks belongs to Emi Do for language corrections.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 30th June 2016 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Kocík, David. *Routing in Möbius Cubes.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2016.

Abstrakt

Möbius cube je zajímavou topologií, která vznikla z topologie hypercube. Největší výhoda oproti hypercube je v přibližně polovičním průměru Möbius cube. V této práci je popsán algoritmus nejkratšího routování a jsou popsány i jeho klady a zápory. Velkou nevýhodou je možnost pádu do stavu uzamknutí (deadlock). Proto je v práci představen nový deadlock-free algoritmus a porovnán s předchozím algoritmem. Dále je v práci popsána možnost použití hypercubického multicast 1-portového wormhole algoritmu na Möbius cube.

Klíčová slova Möbius cube, routing algoritmus, shortest path routing, deadlock-free routing, multicast

Abstract

The Möbius cube is an interesting topology created from the hypercube. Its main advantage is the which that is around one half of the diameter of the hypercube. In this thesis, the shortest path algorithm is described as well as its properties and drawbacks. One major drawback is the possibility of a deadlock. Therefore, a new deadlock-free routing algorithm is introduced and compared to the previous algorithm. Later, usage of hypercube's multicast 1-port wormhole algorithm on the Möbius cube is described.

Keywords Möbius cube, routing algorithm, shortest path routing, deadlock-free routing, multicast

Contents

List of Figures

Introduction

Motivation and objectives

In the field of multiprocessor machines, one of the most debated topics are the interconnection networks. There are many uncountable topologies that can be used as a graph of a connections between processors. The research of these topologies has gone in many different directions as there are many requirements for these topology. Furthermore, some requirements are not possible to attain whe regarded with other requirements. For example, there is research (and also existing computers) based on the easiest and most basic topologies such as meshes and tori, while more complicated mathematically describable topologies or irregular (sometimes even fully random) graphs.

Among the most common structures is the hypercube. It was one of the first topologies to be used in the supercomputers, but later it completely disappeared, being overrun by more effective topologies. It disappeared from use in actual computers, but not from the theory and there is a lot of research with the goal to improve some of the attributes of the hypercube and hold qualities of the others.

One of the attributes that is possible to improve is the diameter of the hypercube. The theory is simple: in the cube, for each node, there is always just one node that is further than all other nodes. Thus, it amy be possible to modify a few edges to reach that node faster. That is the simplified theory behind the topology called the Twisted 3-cube.

The Möbius cube is very similar to the Twisted 3-cube. It takes the original design of the 3-dimensional cube and expands it into the recursively defined topology. This topology was introduced in "The Möbius cubes" article in 1995 by doctors Cull and Larson. It also introduces the shortest path routing algorithm for the Möbius cube and since the algorithm is relatively complicated, the first goal of this thesis will be to clarify why and how the algorithm work.

For any topology it is necessary to have a deadlock-free routing algorithm.

To establish multiple communications between nodes, we must be sure that the messages does not fall into a situation where all messages are waiting for another in such way that they will stay stuck forever. The next goal of this thesis is to analyze if a deadlock-free routing is possible and to introduce such an algorithm.

Another communication problem is the multicast. We have situation where one node needs to send message to a group of other nodes that are scattered across the whole graph. That is a common situation if we use a master & slaves system. The hypercube has very effective and easy algorithm for this problem, we will see if the same or a similar algorithm exist also for The Möbius cube.

Goals

This thesis has following goals:

- Perform a survey of the Möbius cube theory.
- Analyze the routing algorithm proposed in [1] and describe its properties and drawbacks.
- Design a routing algorithm with focus on a deadlock-free routing and describe its properties and drawbacks.
- Design a round-optimal multicast algorithm for 1-port Möbius cube using a wormhole routing.

Structure

All the necessary and useful definitions are in the chapter 1. The thesis then continues with description of the shortest path routing algorithm in the chapter 2. It starts with explaining the minimal expansion algorithm and then follows with the actual routing. At the end of the chapter there is a mathematical evaluation of the distances in the Möbius cube. The following chapter 3 focuses on another new routing that is deadlock-free. Finally the algorithm for the multicast is described in the final chapter 4.

Chapter 1

Interconncetion networks

1.1 Basic Notions and terminology

In this section we first define the basic terms of the graph theory. Later, we describe the characteristics of both hypercube and the Möbius cube and define the deadlock-freeness and multicast algorithm.

1.1.1 Letters, alphabets and strings

For integer $d \geq 2$, let $Z_d = \{0, 1, .., d-1\}$ be an alphabet of d letters and $Z_d^n = \{x_{n-1}...x_0; x_i \in Z_d\}$, $n \geq 1$, be the set of all d-ary strings of n letters. The i-th letter x_i of a string x is also written $x[i]$, $0 \leq i \leq n$–1. The length of a string x is denoted by $len(x)$. For integer $i \geq 1$, the i-fold concatenation of a string x is denoted by x^i. For example, a string of $i \geq 1$ letters $a \in Z_d$ is written a^i. The binary alphabet $Z = \{0, 1\}$ is simply denoted by β. If $b = b_{n-1}..b_{i+1}b_ib_{i-1}..b_0 \in \beta^n$, then the string $b_{n-1}..b_{i+1}\overline{b_i}b_{i-1}..b_0$, where $\overline{b_i}$ is the inversion of b_i, is denoted by $neg_i(b)$. The complement of string b, denoted $\overline{b} = \overline{b}_{n-1}...\overline{b}_1\overline{b}_0$.

1.1.2 Hamming distance

Let the operation + between two binnary strings be defined as operation XOR. The hamming distance of two given binary strings, $b, c \in \beta^n$, is the number of bits in which b and c differs. $\varrho(b, c) = \sum_{i=0}^{n} yb_i + c_i$.

1.1.3 Graphs and subgraphs

Let $G = (V(G), E(G))$ be an undirected graph consisting of nodes $V(G)$ and edges $E(G)$. An edge incident on nodes u and v is said to be incident with the edge $\langle u, v\rangle$. Vice versa, nodes u and v are said to be incident with the edge $\langle u, v\rangle$. The Edge $\langle u, v\rangle$ is also said to connect nodes u and v and nodes u and v are then said to be adjacent (or neighbors). We also say that u and v are

end-nodes of $\langle u, v\rangle$. Similarly, edges $\langle u, v\rangle$ and $\langle v, w\rangle$ are adjacent. A graph H is the induced subgraph of G if it is the maximal possible subgraph of G with nodes $V(H)$. If $V(H) = V(G)$ and $E(H) \subset E(G)$, then H is a spanning subgraph of G. If H is a subgraph of G, we write simply $H \subset G$. A walk in graph is any sequence of adjacent distinct edges. A path is a walk with all nodes distinct. A cycle is a walk wit all nodes distinct except for the first and last.

1.1.4 Node degree and regularity

The degree of a node $u \in V(G)$, $deg_G(u)$, is the number of edges incident on u. The set of degrees of all nodes of a graph G, $\{deg_G(u); u \in V(G)\}$, is denoted by $deg(G)$. The (maximal) degree of G is $\Delta(G) = max(deg(G))$ and analogously, the minimal degree of G is $\delta(G) = min(deg(G))$. If $\Delta(G) = \delta(G) = k$, then G is said to be k-regular graph. A graph G is sparse if $|E(G)| = O|V(G)|$, i.e., $\Delta(G)$ is constant , and it is dense otherwise.

1.1.5 Isomorphism and automorphism

G_1 is isomorphic to G_2, $G_1 \equiv G_2$, if there exist a 1-1 mapping $f : V(G_1) \leftrightarrow V(G_2)$ preserving the adjecency, i.e., $f : V_1 \mapsto V_2$ such that $\langle u, v\rangle \in E(G_1) \Leftrightarrow \langle f(u), f(v)\rangle \in E(G_2)$. A permutation of nodes of G preserving the adjacency is called an automorphism. Since an isomorphism f preserves adjacency, any path $P_1 \subset G_1$ is mapped to a path $f(P_1)$ in G_2. More importantly, any cycle is preserved.

1.1.6 Union and cartesian product

Graphs are mathematical objects for which we can define operations and algebras. For our purposes, we will need only 2 operations on our graphs. The union of G_1 and G_2, $G_1 \cup G_2$, is a graph with nodes $V(G_1) \cup V(G_2)$ and edges $E(G_1) \cup E(G_2)$. The Cartesian product of 2 distinct graphs G_1 and G_2 is a graph $G = G_1 \times G_2$ such that $V(G_1), y \in V(G_2)\}$ and $E(G) = \{\langle [x_1, y], [x_2, y]\rangle; \langle x_1, x_2\rangle \in E(G_1)\} \cup \{\langle [x, y_1], [x, y_2]\rangle; \langle y_1, y_2\rangle \in E(G_2)\}$. The Cartesian product is commutative and associative operation: $G_1 \times G_2 \equiv G_2 \times G_1$ and $(G_1 \times G_2) \times G_3 \equiv G_1 \times (G_2 \times G_3)$.

1.1.7 Node and edge symmetry

G is node symmetric if $\forall u_1, u_2 \in V(G) \exists$ automorphism f such that $f(u_1) = u_2$. G is edge symmetric if $\forall e_1, e_2 \in E(G) \exists$ automorphism f such that $f(e_1) = e_2$.

1.1.8 Paths, distances and diameters

The length of a path $P(u,v)$ between nodes u and v, $len(P(u,v))$, is the number of its edges. The distance between u and v, $dist_G(u,v)$, is the length of a shortest path $P(u,v)$. The average distance in an G, $|E(G)| = N$, is

$$\overline{dist}(G) = \frac{1}{N(N-1)} \sum_{u,v,u \neq v} dist_G(u,v).$$

Given $u \in V(G)$, the eccentricity of u is $exc(u) = max_{v \in V(G)} dist_G(u,v)$. The diameter of G is the greatest possible distance: $\varnothing(G) = max_{u,v \in V(G)} dist_G(u,v) = max_{u \in V(G)} exc(u)$. 2 paths are node disjoint if they share at most of the end nodes: $V(P(u,v)) \cap V(P(x,y)) = \{u,v\} \cap \{x,y\}$. 2 paths are edge disjoint if they share no edge: $E(P(u,v)) \cap E(P(x,y)) = \emptyset$.

1.2 The Hypercube

The hypercube of dimension n, denoted as Q_n, is a strictly orthogonal topology defined as a Cartesian product of two hypercubes of dimension $n-1$, where Q_0 is 1 node. The nodes of Q_n are all n-bit strings $x_{n-1}...x_1x_0$, corresponding to points in n-dimensional Boolean space: $V(Q_n) = \beta^n$. $|V(Q_n)| = 2^n$. 2 nodes of hypercube are neighbors if they differ in one single bit: $E(Q_n) = \{\langle x, neg_i(x)\rangle; x \in V(Q_n), 0 \leq i < n\}$ and $|E(Q_n)| = n2^{n-1}$. Each node of Q_n has n neighbors, and so, Q_n is regular with $deg(Q_n) = \{n\}$. Q_n is hierarchically recursive topology. It consists of subcubes that can be specified by strings $S = s_{n-1}...s_1s_0$, where s_i is $\{0,1,*\}$ and $*$ is the don't care symbol.

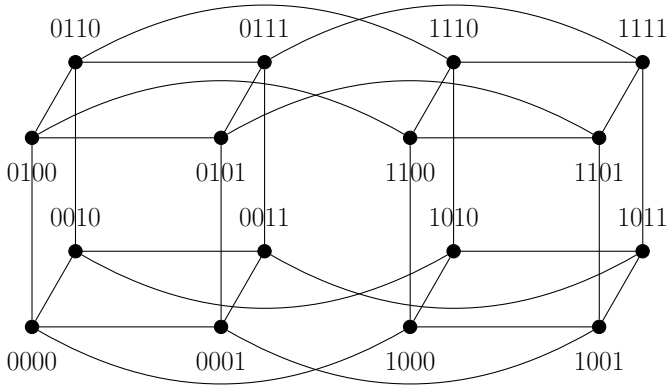


Figure 1.1: Q_4.

Any n-bit string can be produced from any other n-bit string by at most n inversions. So, the diameter of Q_n is logarithmic: $\varnothing(Q_n) = n$. The distance

between 2 nodes ($x, y \in V(Q_n)$) is the Hamming distance $\varrho(x, y)$ between the strings. The number of nodes in distance i from a given node is the number of ways to select i bits from n, which is $\binom{n}{i}$. Since $\binom{n}{i} = \binom{n}{n-i}$, It follows that the average distance is $\overline{dist}(Q_n) = \lceil \frac{n}{2} \rceil$. 2 nodes differ in k bits and there are $k!$ permutations of these bits, so the number of different shortest paths between x and y is $k!$. The standard greedy routing algorithm is called *e-cube* routing. It checks the bits in fixed order, typically from the least significant bit to the most significant bit. Because the hypercube grpah is edge and node symmetric, any selected order will provide routing with the same abilities: it is the shortest possible and deadlock-free.

```
e-cube(x, y, dim)

{
  if(x != y)
  {
    if(x + y & 2^dim)
    {
      x = neg_dim(x);
      print(x);
    }
    ecube(x, y, dim + 1);
  }
}
```

Figure 1.2: Routing algorithm in Q_n from LSB to MSB. The variable dim is equal to 0 on start.

Hypercube has proven to be very popular in parallel computation due to its abilities. It has relatively small diameter and it has a small number of connections per node (processor). An optimal algorithm exists for collective communication operations in almost all communication models. Also the hypercube can simulate efficiently almost any other topology. Thanks to those reasons, the hypercube is commonly considered to be the best topology from an algorithmic and communication viewpoint, however there also exists drawbacks that have lead to no commercial hypercube multiprocessors being produced nowadays. One such drawback is the logarithmic degree of every node (other topologies have a constant degree) and consequently high number of communication channels and poor scalability.

1.3 The Twisted 3-cube

There exist many topologies that try to improve some of the abilities of the hypercube and keep the good results of the other abilities. In some literature such topologies are called the enhanced cubes. Most of them generalize the idea of the twisted 3-cube. The diameter of the hypercube is good but for its resources, it is not the best possible. The twisted 3-cube is almost identical to Q_3, it uses the same number of nodes and same number of edges as Q_3 but it changes just two edges so the diameter is reduced from 3 to 2. Note that with the same number of nodes and edges as the hypercube, there is no chance to improve the degree, the number of communication channels or the scalability. The twisted 3-cube exists only in the form of 8-nodes graph so there have been many attempts to generalize this idea of the twisted 3-cube into recursive topology such as the Twisted Cube, the Twisted N-cube, the Crossed Cube, the Flip MCube, the Generalized Twisted Cube, or the Möbius cube.

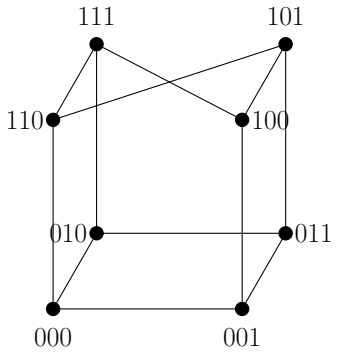


Figure 1.3: The twisted 3-cube.

1.4 The Möbius cubes

Out of many enhanced cube topologies, the Möbius cube, introduced in [1] is the one that is the most conceptually simple with very good results. The Möbius cube of dimension n, denoted as MC_n, has a topology where nodes of MC_n are all n-bit strings $x_{n-1}...x_1x_0$, corresponding to points in n-dimensional Boolean space: $V(Q_n) = \beta^n$. $|V(Qn)| = 2^n$, just like the hypercube. The address of the node y which is the neighbor node of the node x in i-th dimension is defined as:

$$y = x_{n-1}...x_{i+1}\,\overline{x}_i\,x_{i-1}...x_0 \text{ if } x_{i+1} = 0,$$

$$y = x_{n-1}...x_{i+1}\,\overline{x}_i\,\overline{x}_{i-1}...\overline{x}_0 \text{ if } x_{i+1} = 1.$$

We say that the edge that is created by the first case (where routing changes 1 single bit) is a "hypercubic" edge, denoted as e_i. And we say that the edge that is created by the other case (where routing changes all bits with index $\leq i$) is a "twisted" edge, denoted as E_i. Furthermore, we define these edges as binary strings, where x+edge= y. Than, $e_i = 0_{n-1}...0_{i+1}\,1_i\,0_{i-1}...0_0$ and $E_i = 0_{n-1}...0_{i+1}\,1_i\,1_{i-1}...1_0$.

As every edge depends on a bit in a dimension one higher than the dimension of the edge, the highest dimension edges depends on bits that are not included in the address of the node (x_n). Normally we would assume that $x_n = 0$ and we define that such topology that has 0 in front of address of nodes is denoted 0-MC$_n$. However, because of the recursive point of view we also need to define the topology where we assume that $x_n = 1$, we say that such topology is denoted as 1-MC$_n$. Because of the new imaginary highest bit, 1-MC$_n$ contains "twisted" edges in its highest dimension where 0-MC$_n$ has "hypercubic" edges so these two topologies are different. Once we have defined both Möbius cubes, we can see that recursively, any MC$_n$ is made by connecting 0-MC$_{n-1}$ with 1-MC$_{n-1}$. The highest dimension edges will be all "hypercubic" edges for 0-MC$_n$ and all "twisted" edges for 1-MC$_n$.

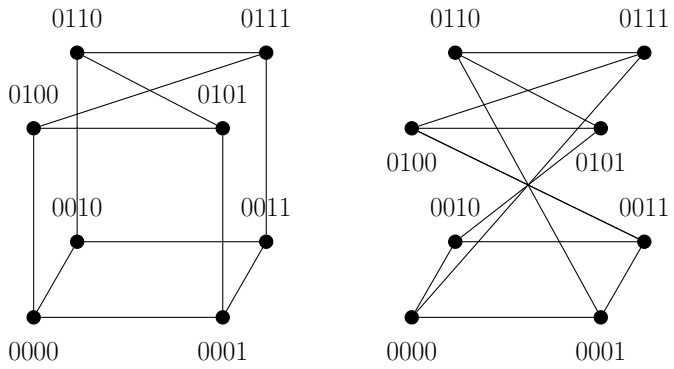


Figure 1.4: 0-MC$_3$ on the left and 1-MC$_3$ on the right.

1.4.1 Efficiency of the Möbius cubes

There are many requirements on any topology when it is meant to be successful in the field of multiprocessor computers. We will see how the Möbius cubes stands in every category.

Node degree. It is a technological requirement to have a small and fixed node degree. A small number of a communication links per procesing nodes makes the construction of a communication network simpler and cheaper. The

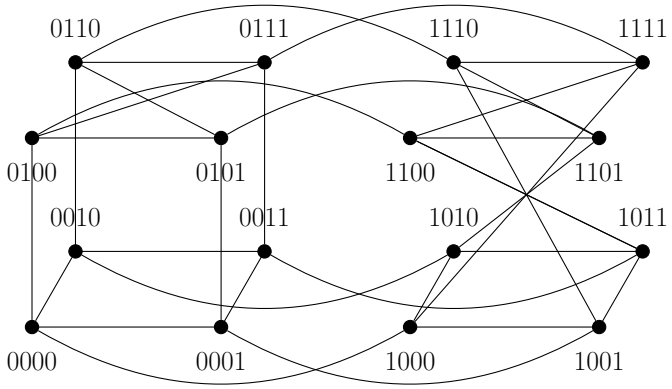


Figure 1.5: 0-MC_4.

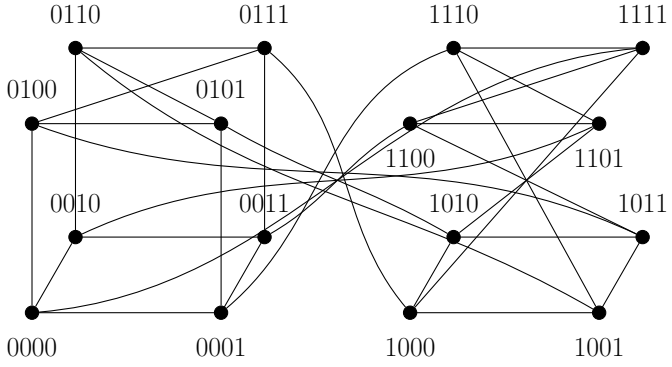


Figure 1.6: 1-MC_4

.

Möbius cube has the same degree as the hypercube and both are sparse graphs which is optimal. Also for the length of wires, MC_n and Q_n makes a small difference, around one half of the wires in MC_n are $\sqrt{2}$ times longer then their hypercubic counterparts.

Diameter and average distance. It is an algorithmic requirement to have as small of a diameter on topology as possible. The closer the nodes are to each other, the faster they can communicate and communication latency will be smaller. We already introduced the Möbius cubes as topology that is lowering the diameter and the average distance of the hypercube. Later in Chapter 2, after we introduce the shortest path routing algorithm, we will be also be able to count the exact value of both diameters and average distances of the Möbius cubes.

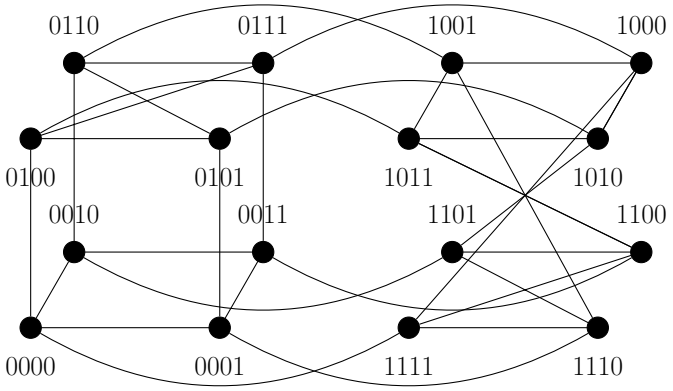


Figure 1.7: 1-MC_4. Notice different positioning of nodes in right subcube.

Symmetry. Both node and edge symmetry are important for designing or analyzing parallel algorithms, since it does not matter where the computation starts. Unfortunately, because of the non-equal number of "hypercubic" and "twisted" edges in any two subcubes of MC_n, the Möbius cubes have lost the perfect symmetry of the hypercube. On the other hand, there exists some partial symmetry that allows us to design some algorithms for all nodes universally.

Hierarchical recursivity. To have a topology that is nicely recursive is helpful for designing and manufacturing of interconnection networks. We know that we have recursive operations that create MC_n but it is not so nicely hierarchically recursive commpared to Qn that can divide into any subcubes by selecting any dimensions. On the other hand, a hierarchical recursivity is contradictory to scalability.

Scalability. Ideally, we wish to have incrementally and efficiently scalable topologies. There are only very few such topologies. Most topologies are partially scalable and only part of them is efficiently scalable. Unfortunately, similarly to the hypercube, the Möbius cubes are very poorly scalable, as we can only double the number of nodes with the set shape.

High connectivity and fault tolerance. High connectivity leads to the existence of many parallel disjoint paths between pairs of nodes, moreover if the fault average distance is small, these redundant paths are short. We know that Q_n has $k!$ different shortest paths for two nodes in distance k. That is not true for the Möbius cubes, but we have to mind that the diameter is lower and MC_n is not symmetric. So for two nodes with distance k in Q_nm we will be able to find $k!$ different paths, but they will be shorter, longer or equal to

k with sum of lenghts $\approx k \cdot k!$. For the fault tolerant algorithms in the Möbius cubes, they are mostly yet to be introduced and measured.

Embedding. With many existing topologies and multiprocessor computers based on them, it is important to be able to embed the topologies of one into another with good embedding measures. As the Möbius cube computer is very unlikely to be built, it would be very interesting to see how well it can be MC_n embedded into another topology such as mesh or tori.

1.5 Deadlock

The existence of a deadlock-free routing algorithm is another important requirement for parallel networks. Deadlock occurs mainly while using wormhole routing. The following text of this section is taken from [9].

1.5.1 Wormhole routing

The simplicity, low cost, and distance-insensitivity of wormhole switching are the main reasons behind its wide acceptance by manufacturers of commercial parallel machines. The packets are split to flits, which are snaked along the route. The routers do not have input and output buffers for whole packets, but only small buffers for 1 or several flits. Hence, the routers are small and cheap, but the price we must pay for that is blocking. If a header cannot proceed due to busy output channels, the whole chain of flits gets stalled and the flit buffers in routers along the path get blocked, see fig. Anyway, WH switching has been and still is a very popular switching technique used in commercial machines. One consequence of this solution is that packets do not have to be of same length, which of course, is an advantage. But the WH switching also has a great disadvantage. Due to the blocking feature, it is deadlock prone: One frozen chain of flits may block other chains and the snow-ball effect may lead to the collapse of the whole network or some its component.

1.5.2 Deadlocks

Deadlocks are a serious problem in WH networks. Hence, we will show some basic methods how to deal with them. Consider 4 packets P_i and 4 destination nodes A, B, C, D. Consider that these packets have chosen the destinations as follows.

$P_0 \to C$	$P_0 \to C$	$P_0 \to C$	$P_0 \to C$

Figure 1.9 shows a typical deadlocked situation. Each packet has acquired some buffers in the attempt to cut-through to the destination, but it is blocked

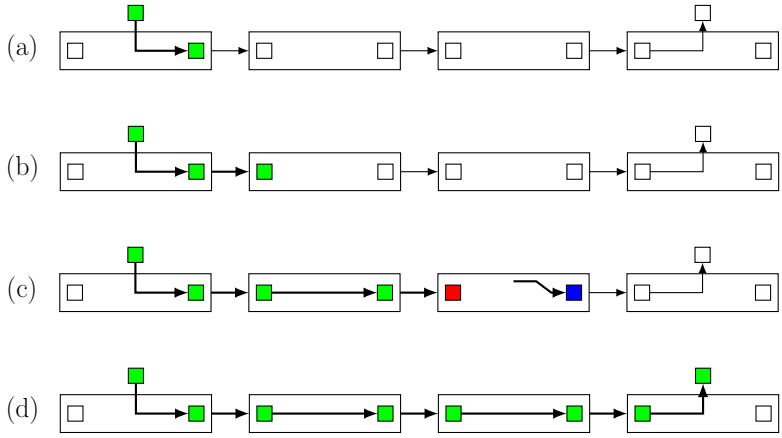


Figure 1.8: Wormhole switching. (a) The header flit is stored in the output buffer. (b) The chain of flits moves one step further. (c) The chain of flits has been blocked since the output channel is busy. (d) A conflict free pipeline of flits occupying all buffers of all routers on the path between source and destination nodes.

since the nest channel it needs is busy, and its further progress depends on a release of the next buffer. The trouble is that these dependencies form a cycle. In fact, each pocket waits for itself, and therefore, will wait forever.

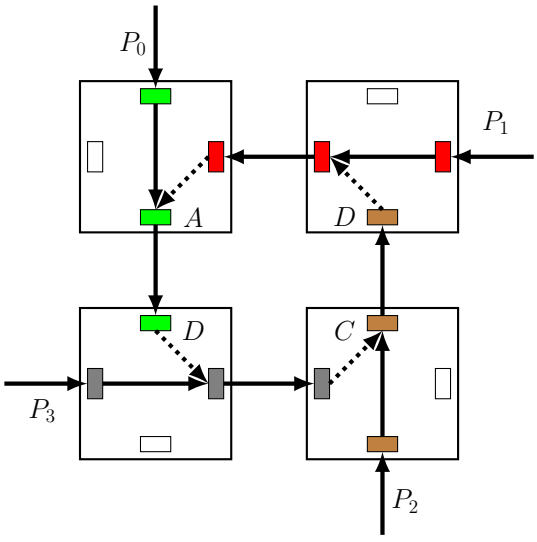


Figure 1.9: Deadlock in WH network with 4 packets P_i.

1.5.3 How to solve the deadlock problem

The communication deadlock, described above, is equivalent to deadlock problems in any system with shared resources. Typically, these problems appear in multitasking operating systems. The methods of how to deal with deadlocks here are the same as in any operating systems.

Deadlock prevention. It consists in the most conservative allocation of resources. A given resource is allocated only if it is able to be proved to be free. If we need more resources, first we reserve them and then allocate them all at once. A disadvantage is that it can cause low utilization of resources.

Deadlock detection and recovering. This is the other extreme. It is the least careful approach. We do not care about possible cyclic dependencies. If some appear, we must be able to detect them and recover from them, i.e., to interrupt somewhere the dependence cycle. This approach is useful only in systems with rare deadlocks, since the deadlock detection is a costly operation. Hence, it may give great gains in some systems, but also incur huge overhead in other systems.

Deadlock avoidance This is the middle way. We assign resources in a smart way so that a cycle cannot appear.

1.5.4 Channel dependence graph

We consider deterministic routing only.

Routing function. For a given node $e \in V(G)$, for its input channel c_1, and for a given destination node d, a routing function R determines the output channel $c_2 = R(u, c_1, d)$.

The basic tool for intelligent channel allocation in a given network with a given routing function is a graph hich shows all possible assigments of channels.

Channel dependence graph (CDG). Given a connected graph and routing function R, then $CDG(G, R)$ is a graph whose nodes are the channels c_i of network G and 2 such nodes are adjacent, i.e., $\langle c_1, c_2 \rangle \in E(CDG(G, R))$, if and only if the routing function R can route in network G a packet from input channel c_1 to output channel c_2, i.e., $R(u, c_1, d) = c_2$ for some 2 nodes u and d.

Theorem 1. *A deterministic routing function R on a graph G is deadlock free if and only if $CDG(G, R)$ is acyclic.*

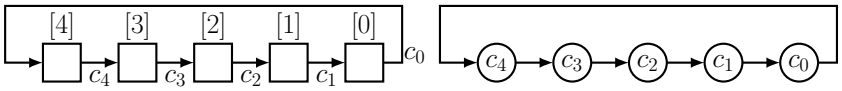


Figure 1.10: Torus K(5) and its channel dependence graph

Proof. Suppose $D = CDG(G, R)$ has a cycle S. It follows from our assumptions that D has no loops. Hence, S must be of length $l(S) \geq 2$. If S consists of two channels c_1 and c_2, then a pair of requests $R(c_1, w_1) = c_2$ and $R(c_2, w_2) = c_1$, where w_1 and w_2 are in distance at least 2 ahead from the packets, will deadlock. If $l(S) \geq 3$, then a deadlock configuration is even easier to construct, just by submitting $l(S)$ packets simultaneously, each injected in one node of S and destined to the node in G in distance two ahead following the channels is S.

Suppose D is acyclic. Then we can assign a total order to channels of G (= nodes of D) such that $c_i > c_j$ if and only if $\langle c_i, c_j \rangle \in E(D)$. Let c_m be the least channel in this order with a full buffer. Every channel c_n that is fed from c_m by the routing algorithm, must be less than c_m, i.e., $c_n < c_m$, and so it cannot be full. Therefore no flit waiting in the buffer of c_m is blocked and must eventually proceed further and the buffer of c_m will be freed. The argument is by induction on the length of any sequence of full channels. □

This gives us a tool. For a given G and R, we construct $CDG(G, R)$ and see if it is accyclic, If so, we are done, since no deadlock can appear. The question is what to do otherwise. There are 3 possibilities.

1.5.5 Deadlock avoidance by routing restrictions

The first method can be shortly defined as follows: Restrict the routing function R to R' so that $CDG(G, R')$ becomes acyclic and G remains (strongly) connected with respect to the restricted routing function R'.

The requirement of string connectedness is the key here. We cannot restrict the routing function to R' so that some nodes become unreachable if we use R'. Hence, this solution can be used only in some cases.

1.5.5.1 Dimension-ordered routing

This method is useful for meshes and hypercubes. The restricted routing R' is called the dimension-order routing. The dimension-ordered routing means that all routers use the dimension in the same fixed order and once a packet has used a channel in dimension i on its path to the destination, it cannot use channels of dimensions $j < i$ in the rest of the path. These routing functions have special names: *XY-routing* in 2-D meshes and *e-cube* routing in the hypercube. Figure 1.11 documents that if the underlying topology is

the hypercube, then the same situation that caused a deadlock in Figure 1.9 produces no deadlock if all routers use *e-cube* routing.

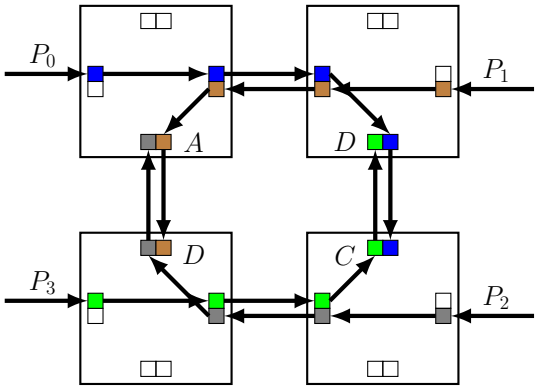


Figure 1.11: The request of 4 packets does not produce a cycle if *e-cube* routing is used.

1.5.6 Deadlock avoidance based on virtual channels

The restriction to dimension ordered routing does not work in 1-D tori (denoted $K(n)$). Each 1-D torus has trivially a cyclic CDG and any restriction on the routing function would disconnect a torus into a mesh. However, there exists a solution, based on the so-called virtual channels. The resources of a physical channel (buffers plus link controllers plus a physical communication medium) are split: buffers are installed in more copies, link controllers multiplex among them, and the physical medium is shared using the time multiplex. The WH implementing virtual channels must distinguish among data flits (remember that they are anonymous, they have no routing information nor sequencing numbers) of different packets that share the physical channel.

1.6 Muticast

The multicast is an extremely important one-to-many communication pattern that can be defined as a broadcast from the source node s to an arbitrary subset M of the nodes of a network (equivalently, it is a broadcast with the set $M \cup \{s\}$). The standard One-to-All broadcast algorithm is very inefficient for small M, especially in WH networks. The lower bounds are similar to those for the OAB. Since M can be located anywhere in the given topology, the optimal MC algorithms are substantially more complicated than OAB algorithms.

Chapter 2

Shortest path routing algorithms

The shortest path routing algorithm was proposed in [1]. And we will describe it in this chapter. Notations, definitions and proofs are recast with following changes: the address of nodes in the original text are vectors in the n-dimensional vector space over (0, 1) with addition and scalar multiplication mod 2, denoted Z_2^n. In this text they are strings of β^n. The other change is the change of direction in ordering dimensions of MC_n, while the most significant bit in the original text has index 0, we use index $n-1$. And oppositely, the least significant bit has index $n-1$ in the original text and in this text it has index 0.

2.1 Minimal expansion

Minimal expansion is an algorithm that creates a set of edges that combines into a binary string. Minimal expansion will be later used to provide set of edges that leads from one node to another so it is a crucial part of routing algorithm introduced in [1].

Definition 1. *A set S consisting of terms t, where every t is edge e_i or E_i, $0 \leq i < n$ is called expansion of string $b \in \beta$ iff $\sum_{i=1}^{|S|} t_i = b$.*

We have described the edges e_i and E_i in section 1.4. We know that adding an edge with a node address will create the address of neighbor node. We can also combine edges together to create a new strings. The expansion of node x is then a set of edges from node with the address 0^n to node x.

Definition 2. *For a string b, the weight of an expansion S of b is the cardinality of set S, denoted $|S|$.*

There is at least one expansion for any string. For example just by using every e_i where i is an index of every 1 in the string.

Lemma 1. *Any string $b \in \beta^n$ has an expansion, where no two terms has same index.*

Proof. The possible operations with edges of same index are:

$$e_i + E_i = E_{i-1}.$$

$$e_i + e_i = E_i + E_i = 0.$$

That shows that by using any edges e_i, E_j, we can not make string b^k where $k > i \wedge k > j$. It is also clear that using two edges with same index is useless as it can be substituted with just one or no edge. □

Definition 3. *For any string $b \in \beta^n$, let a minimal expansion of b be an expansion with the least weight. The minimal expansion set is denoted S_x.*

We are able to a find finite number of expansions of b and some of them will be smaller sets than others. Some strings have only one minimal expansion, while others can have more minimal expansions. For example $E_2 + e_1 = e_2 + E_0 = 101$.

Lemma 2. *If string b has its highest bit containing 1 on index j, there will be no e_i or E_i in the minimal expansion of b for any $i > j$.*

Proof. Making expansion of string b is the same as adding strings e_i or E_i to ab empty string 0...0. By adding e_i we will revert just the bit with index i and by adding E_i we are reverting bit i and all bits with smaller indices. Neither of them has an effect on bits indexed higher than i. Therefore, using any edges with index $i > j$ in the expansion would cause the expansion to have at least one more edge than is necessary and thus will not minimal. □

Only the bits with an index lower or equal than the index of the highest bit containing 1 are important to determine the number of edges in minimal expansion. The bits higher that the highest bit containing 1 can be skipped by any algorithm generating the minimal expansion.

Lemma 3. *For any string x, $|S_x| \leq 1 + |S_{\overline{x}}|$.*

Proof. The string x with the highest bit on index i can be written as

$$x = E_i + E_i + x = E_i + \overline{x}.$$

A minimal expansion of x cannot have more edges than one for E_i plus the number of edges of the minimal expansion of $\overline{x}$. □

2.1.1 Greedy minimal expansion algorithm

Now, we describe the greedy minimal algorithm from Figure 2.1. We will proof that it generates exactly the smallest possible number of edges. In the algorithm, we use the denotation $x = *x'$, where $*$ is a binary string, and x' stands for a string that contains the remaining bits of the string x. The algorithm is divided into 5 cases by content of string x and is recursively proceeding from the most significant bit.

Case 1 The first case is trivial. If the string x has length 0, the algorithm will stop.

Case 2 In the second case, the string x is containing only string 1. The algorithm will add edge E_0 to the minimal expansion of x. Note that it would be also possible to produce edge e_0.

Case 3 The third case skips bits containing 0. If the first bit is 0, the algorithm calls recursively itself on string x without its most significant bit.

Case 4 In this case, there is "10" string in the beginning of string x. Then, the algorithm will add the edge e_i to the minimal expansion of x and call recursively itself on x without first two bits.

Case 5 In the last case, there is "11" string in the beginning of string x. Then, the algorithm will add the edge E_i to the minimal expansion of x and call recursively itself on complement of x without the first two bits. The complement is used because using "twisted" edges reverts all bits with smaller indexes.

Proof. We will prove the algorithm by the induction of the length of x. If $x = ()$, then no terms are necessary in the expansion and the algorithm correctly returns nothing. If vector $x = 1$, then one term E_0, or e_0, is necessary and is sufficient in the expansion of x, and the algorithm correctly returns E_0.

Now assume that string x has its highest bit on index i and for any $x' = x_j, x_{j-1}...x_0$, $j < i$, ME(x', j) produces a minimal expansion of x'.

There can be 0 or 1 on the highest indexed bit of x. If $x = 0x'$ there can be no minimal expansion containing E_i or e_i by Lemma 2. Therefore, the algorithm is correctly skipping 0 on the highest indexed bit in case 3.

If $x = 1x'$, the algorithm has to include E_i or e_i into the mnimal expansion. There is no other option if the bit on index i is the most significant bit.

```
ME(x, i)
  {
    if (x == "")
                return {};
    if (x == 1)
                return {E_0};
    if (x == 0x')
                return ME(x', i − 1);
    if (x == 10x')
                return {e_i} ∪ ME(x', i − 2);
    if (x == 11x')
                return {E_i} ∪ ME(x̄', i − 2);
  }
```

Figure 2.1: Greedy minimal expansion algorithm of string x of length i.

To decide which edge is correct to use, the algorithm must read one more bit.

For $x = 10x'$, if E_i is used, the remaining string would be $1\overline{x}'$. Otherwise, if e_i is used, the remaining string would be $0x'$. Then we can easily calculate the weight of minimal expansion of x:

$$|S_x| = min\{1 + |S_{0x'}|, 1 + |S_{1\overline{x}'}|\}.$$

Where $|S_{0x'}|$ will remove 0 with case 3. And $|S_{1\overline{x}'}|$ would need e_{i-1} or E_{i-1} leading into two different sums. Then we are choosing from 3 options:

$$|S_x| = 1 + min\{|S_{x'}|, 1 + |S_{\overline{x}'}|, 1 + |S_{x'}|\}.$$

Hence, by lemma 3 we know that $|S_{x'}| \leq 1 + |S_{\overline{x}'}|$. The equation shows that using e_i is always leading to the minimal expansion. Using E_i would lead to the minimal expansion only in some cases. The algorithm is correctly adding e_i to the minimal expansion, skipping following 0 and recursively finds the rest of minimal expansion.

The last possible case is $x = 11x'$. Again, both E_i or e_i is possible to use. Similarly as in the previous case, we calculate the weight of minimal expansion of x:

$$|S_x| = min\{1 + |S_{1x'}|, 1 + |S_{0\overline{x}'}|\}.$$

And we expand the first part:

$$|S_x| = 1 + min\{1 + |S_{x'}|, 1 + |S_{\overline{x}'}|, |S_{\overline{x}'}|\}.$$

Since x' is a complement of $\overline{x}'$, we know that by lemma 3, $|S_{\overline{x}'}| \leq 1 + |S_{x'}|$. The equation shows that using E_i always leads to the minimal expansion. Using

e_i would lead to the minimal expansion only in some cases. The algorithm is correctly adding E_i to the minimal expansion, skipping following 0 and recursively finds the rest of the minimal expansion.

□

ME(0000) = {}	ME(0100)=e_2	ME(1000)=e_3	ME(1100)=E_3E_1
ME(0001)= E_0	ME(0101)=e_2E_0	ME(1001)=e_3E_0	ME(1101)=E_3e_1
ME(0010)= e_1	ME(0110)=E_2E_0	ME(1010)=e_3e_1	ME(1110)=E_3E_0
ME(0011)= E_1	ME(0111)=E_2	ME(1011)=e_3E_1	ME(1111)=E_3

Figure 2.2: Example of results of algorithm ME for nodes of MC_4.

Now we have defined a tool to create a minimal expansion for a node. In figure are examples of minimal expansions of different nodes. We can see that every node of MC_4 has no more than 2 edges in its minimal expansion.

The operation of finding minimal expansion is important not only when input is one node x but also in the combination of two: $x + y$. Note that if $x, y \in MC_n$, then $x + y \in MC_n$. So any combination of two nodes will have same minimal expansion like one of the nodes in the same subcube. If we look closely at the operation $x + y$, it shows which bits are different in x than in y. In the hypercube it would show which dimensions are needed to be routed at, to reach the other node. In Möbius cubes the distance between nodes are smaller, which that is exactly what ME($x + y$) says. If there is a substring consisting only of 1 in $x+y$, it is useful to take a "twisted" edge to solve them all.

The same rules for generating ME($x + y$) as for ME(x) apply, so the maximum number of edges gives us the lower bound on the diameter of MC_n, $\varnothing(MC_n) = \Omega(\lceil \frac{n}{2} \rceil)$.

Just from looking at the graphs of the Möbius cubes of small dimensions, we can tell that for some of them, this lower bound is unreachable. For most of the nodes (and the combination of two nodes) there exists more than 1 minimal expansion and the ME algorithm does not control for it if the generated edges are present in the MC_n. For example, if there is E_3 in minimal expansion for routing in 0-MC_4, there is no such edge in the whole graph. The minimal expansion is not generating a route between two edges, it is just a tool to generate an approximate direction.

2.2 Approximate shortest path routing algorithm

The shortest path routing algorithm will use the minimal expansion as input to generate routing between two nodes. First we will describe the approximate algorithm, which is easier to understand but is missing one operation to generate a correct route every time.

The algorithm, when routing from x to y already received the minimal expansion of $x+y$ but we now that this is the exact routing in only few cases. The algorithm has to determine if the edge proposed by ME$(x+y)$ really exists and if it does not, the algorithm has to substitute it. The approximate routing algorithm is in Figure 2.3.

Before we describe the algorithm, let us define some functions and labels used in the algorithm. Let's assume that S is a set of edges generated by ME$(x+y)$ and it is always sorted from the highest to the lowest index. Then, S.top returns its highest indexed edge, S.next returns the second highest edge, S.index returns an index of its highest indexed edge, S.remove(e_i) removes the edge from the set and S.add(e_i) adds the edge. The function isEdge(node x, edge e_i) returns $true$ if edge e_i is incident with node x and e_i is actual edge existing in context of considered MC$_n$, otherwise it returns $false$. This function depends on bit $i+1$ in address of x as it is decribed in definition of neighbors in the Möbius cubes. This routing algorithm works for both 0-MC$_n$ and 1-MC$_n$, which contains different types of edges in the highest dimensions. We assume that the algorithm and all other functions knows which MC$_n$ is currently being used. In case 2 of algorithm, t_j denotes any e_j or E_j with index smaller than i.

There are 4 cases in the algorithm, some divided into two parts. We keep the same order for labeling the cases as in [1]. But while describing them one by one, we will begin from case 4 to case 1 because this order is going from the easiest to the most difficult part to understand.

Case 4 This case happens if the edge with highest index i is E_i and such edge incident on node x does not exist. Therefore we need to find substitution for this edge:

1. Remove E_i from set S.
2. Add edges e_i and E_{i-1} to set S.
3. Apply the algorithm recursively on new set S.

There is an edge incident on x in every dimension so if E_i does not exist, there must be an existing e_i edge instead. So it is possible to add e_i to S and because we wanted to route to where E_i would have led and we know that $E_i - e_i = E_i + e_i = E_{i-1}$, we also add E_{i-1} into S. If this would have happened during generation of minimal expansion, the generated set would

```
approxSpr(x, y, S)
{
  int i = S.index;
  case 0: (x == y)
      break;
  case 1: (isEdge(x, S.top) && S.top == e_i)
    if(S.next.index == i+1 && isEdge(x,S.next) )
    {
      S.remove(e_i);
      approxSpr(x,getNeighbor(y, e_i), S);
      print(y);
    }else
    {
      x = getNeighbor(x, e_i);
      print(x);
      S.remove(e_i);
      approxSpr(x, y, S);
    }
  case 2: (isEdge(x, S.top) && S.top == E_i)
     if(∃ t_j ∈ S && isEdge(x, t_j) )
     {
       x = getNeighbor(x, t_j);
       print(x);
       S.remove(t_j);
       approxSpr(x, y, S);
     }else
     {
       x = getNeighbor(x, E_i);
       print(x);
       S.remove(E_i);
       approxSpr(x, y, S);
     }
  case 3: (!isEdge(x, S.top) && S.top == e_i)
    S.remove(e_i);
    S.add(E_i, E_{i-1});
    approxSpr(x, y, S);
  case 4: (!isEdge(x, S.top) && S.top == E_i)
    S.remove(E_i);
    S.add(e_i, E_{i-1});
    approxSpr(x, y, S);
}
```

Figure 2.3: The approximate shortest path routing algorithm.

have been different just at position i, which would have change from E to e, and at position $i-1$, which was vacant before because the ME generates an edge for at most every two bits, the rest of set S would have been unchanged. Then, $|S|$ would have been larger by 1. In the same way, if the algorithm uses case 4, the final path will be 1 node longer than was the size of original input set S.

Case 3 Another case is used when the edge with highest index i is e_i and it is not existing edge incident on node x. Therefore we need to find a substitution for this edge. This case is the same as case 4 just with switched e_i and E_i:

1. Remove e_i from set S.
2. Add edges E_i and E_{i-1} to set S.
3. Apply the algorithm recursively on new set S.

There is an edge incident on x in every dimension so if e_i does not exist, there must be an existing E_i edge instead. So it is possible to add E_i to S and because we wanted to route to where E_i would have lead and we know that $e_i - E_i = e_i + E_i = E_{i-1}$, we also add E_{i-1} into S. If this would have happen during generating minimal expansion, the generated set would have been different just at positions i - which would have changed from e to E, and at position $i - 1$ which was vacant before because the ME generates an edge for at most every two bits, the rest of set S would have been unchanged. Then, $|S|$ would have been larger by 1. In the same way, if the algorithm uses case 3, the final path will be 1 node longer than was the size of original input set S.

Case 2 This case is used only if the edge with the highest index i is an existing "twisted" edge from node x to node u where u can (but does not have to be) y. The algorithm is using a "twisted" edge so $u = neg_i(neg_{i-1}(...neg_0(x)))$. Case two is divided into two parts depending on set S.

Case 2 part 1 The first part of case two is used when any e_j or E_j exists in set S (denoted as t_j, $j < i$), which is an existing edge incident on x. The algorithm proceeds by following steps:

1. Find the neighbor of node x that is connected with x by edge t_j.
2. Rewrite x by this new node.
3. Add new x into resulting routing path.
4. Remove edge t_j from set S.
5. Apply the algorithm recursively on new set S and new node x.

The first part is the most time consuming part of the algorithm. The algorithm needs to check if there exists any e_j or E_j in set S, in algorithm denoted as t_j, $j < i$, which is existing edge incident on x. If it does exists, algorithm will first route along that edge and then recursively call a routing algorithm again (which will fall into case 2 with same E_i again). The reason why the algorithm CAN use t_j instead of E_i is that using t_j does not affect the bits above j. Using t_j and then E_i is same as first using E_i and then t_j: $E_i + E_j = E_j + E_i \wedge E_i + e_j = e_j + E_i; \forall i, j; i > j$. The reason why algorithm DOES use t_j instead of E_i first is that by using E_i it would leave the subcube MC_i and by using any t_j, $j < i$ it stays at the same subcube MC_i. It does not matter if x is at 0-MC_i or 1-MC_i but by taking E_i, the algorithm will route to the node u on position which is opposite (all bits reversed) to the position of x in the first subcube. Because of that, for any j, $0 < j < i$ if there is e_j incident on x, there is E_j incident on u and otherwise if there is E_j incident on x there is e_j incident on u with the exception of dimension 0 where $E_0 = e_0$. Therefore

by using the E_i first, the algorithm would have to substitute t_j later (by case 3 or 4) for 2 different edges and that would add unnecessary nodes to route.

Case 2 part 2 The second part of case two is used when part 1 con not be used. Only E_i is existing an edge incident on x. The algorithm proceeds similarly as in part 1:

1. Find neighbor of node x that is connected with x by edge E_i.
2. Rewrite x by this new node.
3. Add new x into resulting routing path.
4. Remove edge E_i from set S.
5. Apply the algorithm recursively on new set S and new node x.

In the second part of case 2, there is only one existing edge in set S that is incident on x. So the algorithm will simply use it. By using just case 2 every time, the algorithm would take the same number of steps as is size of input set S.

Case 1 Case 1 happens only if the edge with highest index i is existing "hypercubic" edge from node x to node u where u can but does not have to be y. We are using "hypercubic" edge so $u = neg_i(x)$. This case has two separate parts:

Case 1 part 1 In the first part, there is e_{i-1} or E_{i-1} in set S and this edge is an existing edge incident on x. Note that in such a scenario it is not possible to have been created by algorithm ME which adds only edges with index distant at least by 2. The algorithm proceeds in these steps:

1. Find neighbor of node y that is connected with y by edge e_i.
2. Rewrite y by this new node.
3. Remove edge e_i from set S.
4. Apply the algorithm recursively on new set S and new node y.
5. Add original y into resulting routing path.

In this part the algorithm does not use the edge with he lower index like similar part 1 of case 2, instead it routes from y, which is the end node, to node v along edge e_i so it takes parallel edge to edge e_i incident on x. We know that this edge e_i incident on y exists because x and y must be in different subcube MC_i which are connected only by edges e_i (otherwise case 1 wouldn't

be happening). This routing is written into the resulting routing after the recursive step is called so the edge e_i is taken last. That is not a problem because e_i is reverting only bit i which cannot. be reverted by any further edges (they have lower index). The reason why the algorithm does not do an operation similar to case 2 is that if we first route in dimension $i-1$ and call case 1 on e_i again, we are sure that first part of the algorithm will never be done again because edge $i-1$ will be already in use. But in some cases we need to first use more than one edges before e_i to reach correct route. This cannot be done by the same way as in case 2 because edge e_i leads from x to u and x and u are both in different subcube MC_i but the types of edges they incident with in the same dimension j, $0 \leq j < i-1$ are both "twisted" or both "hypecubic". Only in dimension $i-1$, x and u are incidents with different edge type. This is because we are in 0-MC_{i+1}(otherwise there would not be edge e_i) that is created by joining 0-MC_i and 1-MC_i. To use more edges before e_i it is needed to call the recursive algorithm again for cases 1 or 2 to decide which edge to prefer. And then we are sure to use e_i in the end.

Case 1 part 2 If there is no edge e_{i-1} or E_{i-1} in set S that would be an existing edge incident on x. The algorithm takes the following steps:

1. Find neighbor of node x that is connected with x by edge e_i.
2. Rewrite x by this new node.
3. Add new x into resulting routing path.
4. Remove edge e_i from set S.
5. Apply the algorithm recursively on new set S and new node x.

The second part of case 1 is that there is no e_{i-1} or E_{i-1} existing edge incident on x and the rest of edges of S are incident on both x and u, so the algorithm uses the edge with highest index which is e_i. By taking just case 1 every time, the algorithm would take same number of steps as is size of input set S.

Case 0 If $x = y$, the algorithm will end recursion and start returning on the recursive tree.

2.2.1 Examples of routing

In this section, we provide examples of routing for each case of the algorithm. Showing how the set S and nodes x and y are changing. The approximate algorithm does not always generate the shortest possible routing. In the last example, the reason why that is, can be seen.

Example 1 Example 1 can be found in Figure 2.4.

approxSpr on 0-MC$_4$:
$x = 1110$
$y = 0001$
ME(1111) = $\{E_3\}$

1. **case 4**: $S = \{e_3, E_2\}$
2. **case 1**: $y = 1001$, $S = \{E_2\}$
3. **case 1**: $x = 1001$, print(1001), $S = \{\}$
4. **case 0**: $x = y$
5. **finalize step 2**: print(0001)

Result: $1110 \rightarrow 1001 \rightarrow 0001$

In this example, the minimal expansion is only one edge. Note that if it would have been routing in 1-MC$_4$, this edge would be sufficient. But in 0-MC$_0$ it does not exists so in the first step of the algorithm case 4 is used to substitute E_3 for e_3 and E_2. In the second step algorithm can take both e_3 or E_2. Note that if it takes e_3 into 0110, there would be no use for E_2. The algorithm leaves e_3 for the last step and uses E_2 into 1001. The reason why it is important to leave the edge e_i for the very last step is made clear in example 3.

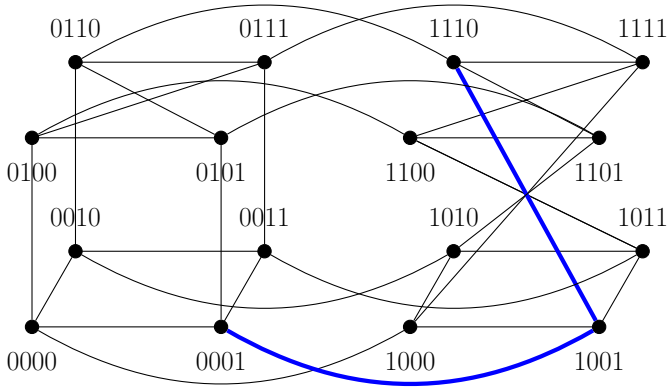


Figure 2.4: Routing from example 1 from 1110 to 0001.

Example 2 Example 2 can be found in Figure 2.5.

approxSpr on 1-MC$_4$:
$x = 0010$
$y = 1111$
ME(1101) = $\{E_3, e_1\}$

1. **case 2**: $x = 0000$, print(0000), $S = \{E_3\}$
2. **case 2**: $x = 1111$, print(1111), $S = \{\}$
3. **case 0**: $x = y$

Result: $0010 \rightarrow 0000 \rightarrow 1111$

In example 2, the input from ME would be correct if the routing would have been from y to x. In our case, the algorithm in step 1 must decide to use e_1 to 0000, according to first part of case 2. Otherwise it would end up in 1101, where is no straight connection to 1111. From 0000 the algorithm routes to 1111 again by case 2, this time its second part.

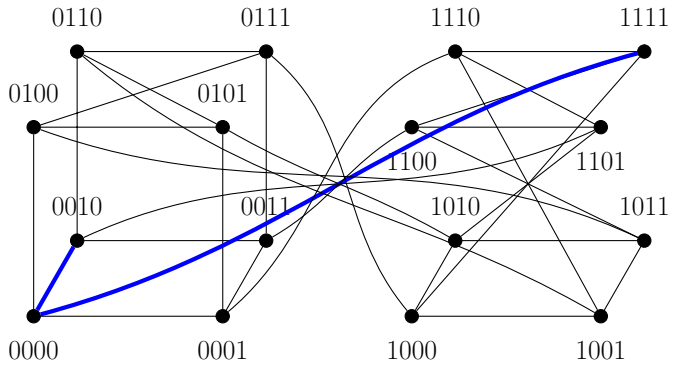


Figure 2.5: Routing from example 2 from 0010 to 1111.

Example 3 Example 3 can be found in Figure 2.6.

approxSpr on 0-MC$_4$:
$x = 1000$
$y = 0101$
ME(1101) = $\{E_3, e_1\}$

1. **case 4**: $S = \{e_3, E_2, e_1\}$
2. **case 1**: $y = 1101$, $S = \{E_2, e_1\}$
3. **case 2**: $x = 1010$, print(1010), $S = \{E_2\}$

4. **case 2**: $x = 1101$, print(1101), $S = \{\}$
5. **case 0**: $x = y$
6. **finalize step 2**: print(0101)

Result: $1000 \to 1010 \to 1101 \to 0101$

This is example of the most complicated routing in 0-MC$_4$, it is a showcase of the difference between first the part of case 1 and the first part of case 2. Step 2 where the algorithm leaves edge e_3, is important for the last step and then step 3 can be called to decide if it is necessary to take E_2 or if e_1 and e_1 is selected. If case 1 was programmed the same way as case 2 (if there is edge $i-1$ take this edge first and then call recursive step for e_i again), it would take E_2 into 1111 first, then e_3 to 0111 and then 0100 and 0101 which is one node longer than the correct routing.

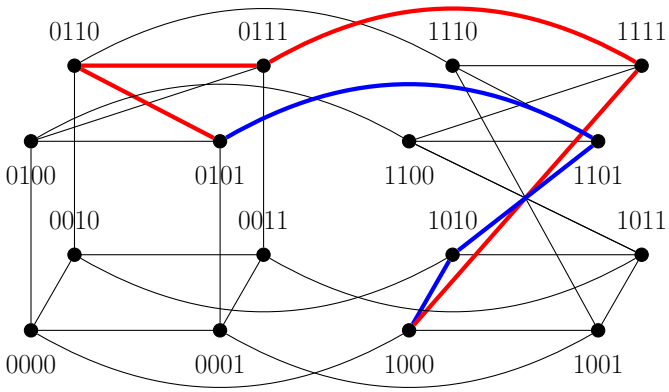


Figure 2.6: Routing from example 3 from 1000 to 0101. Blue is the correct routing, red is the wrong routing - for example if case 1 is programmed same way as case 2.

Example 4 Example 4 can be found in Figure 2.7.

approxSpr on 0-MC$_4$:
$x = 0000$
$y = 1100$
ME(1100) $= \{E_3, E_1\}$

1. **case 4:** $S = \{e_3, E_2, E_1\}$
2. **case 1:** $x = 1000$, print(1000), $S = \{E_2, E_1\}$
3. **case 2:** $x = 1111$, print(1111), $S = \{E_1\}$

4. **case 2:** $x = 1100$, print(1100), $S = \{\}$
5. **case 0:** $x = y$

Result: $0000 \rightarrow 1000 \rightarrow 1111 \rightarrow 1100$

This is example of routing where approximate algorithm does not produce the shortest possible routing. Instead it produces routing that is one node longer, but it is still routing between node 0000 and 1100. The mistake that is done is in step 1. Here instead of e_3, E_2, E_1 we could use e_3, e_2, which would route to 0100 and then to 1100. This showcases the only case (together with same problem in case 3) where approximate routing does not produce the shortest path. So if we fix this, we will get the correct shortest path routing.

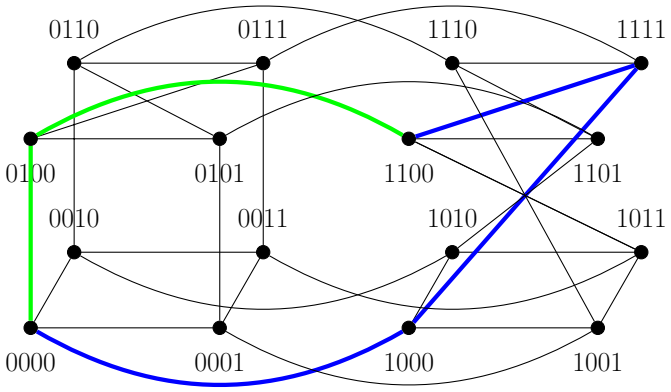


Figure 2.7: Routing from examples 4 and 5 from 0000 to 1100. Blue is routing of approximate algorithm and green is routing of *spr* algorithm.

2.3 Shortest path routing algorithm

To improve the approximate algorithm into the shortest path routing algorithm we need to discover only one thing - how and when it is possible to substitute minimal expansion with another equal minimal expansion. We will always want to substitute the edge with highest index i. First, we will begin with changing E_i into e_i.

We assume that we have minimal expansion created by ME algorithm so there is no space between indexes of edges less than 2. Then if we want to change E_i we have to keep changing all e edges behind until we find E edge in distance $i - 2k$ where k is any constant.

Lemma 4. *If a minimal expansion of string x produced by the ME algorithm has a highest indexed term E_i, then another minimal expansion with highest*

indexed term e_i, exists iff:

$$S_x = E_i\, e_{i-2} \ldots e_{i-2k+2}\, E_{i-2k} + S_{x'}$$

Where $k > 0$ and $S_{x'}$ has no edge indexed higher than $i - 2k - 4$. Then the substitution is :

$$E_i\, e_{i-2} \ldots e_{i-2k+2}\, E_{i-2k} = e_i\, e_{i-1}\, e_{i-3} \ldots e_{i-2k+1}.$$

Proof. Since the routing algorithms are working primarily with edges with the highest index, we need to always change the highest indexed edge into the edge of the other type (here it is "twisted" into "hypercubic"). Therefore we assume that any considered substitution must start with e_i.

The greedy minimal expansion algorithm ME produces "twisted" edge in cases there is 11 string in the beginning of input string x. Then, if we have changed E_i into e_i, we need to deal with 1 on position $i - 1$. There are two options: E_{i-1} or e_{i-1}.

We also know there was e_{i-2} in the original minimal expansion, therefore if we use E_{i-1}, we would have to use e_{i-2} as well.

$$E_i\, e_{i-2} = e_i\, E_{i-1}\, e_{i-2}.$$

Clearly the rest of the minimal expansion would remain the same as in the original outcome. Therefore expansion with E_{i-1} would have weight $|S_x| + 1$.

If we use e_{i-1} instead, the outcome is different. Let us demonstrate it on the string x where $x + S_x = 00...00$. Then the string for the original minimal expansion must be $x = 11(01)^*00$. Then $x + E_i = 00(10)^*11$ and $x + e_i + e_{i+1} = 00(01)^*00$. We then only need $e_{i-3} + ... + e_{i-2k+1}$ to get string 00..00. It does not matter what bits are behind position $i - 2k$ since the original expansion used two "twisted" edges and the new one does not use any "twisted" edges. The weight of S_x remains unchanged. □

11010100101	$E_{10}\, e_8\, e_6\, E_4\, e_2\, E_0 \rightarrow e_{10}\, e_9\, e_7\, e_5\, e_2\, E_0$
10101011001	$e_{10}\, e_8\, e_6\, E_4\, E_2\, E_0 \rightarrow E_{10}\, e_9\, e_7\, e_5\, E_2\, E_0$

Figure 2.8: Two examples of substitution of E_i into e_i and otherwise while keeping the minimal number of edges.

Same way edge e_i can be transformed into E_i.

Lemma 5. *If a minimal expansion of string x produced by the ME algorithm has a highest indexed term e_i, then another minimal expansion with highest indexed term E_i, exists iff:*

$$S_x = e_i e_{i-2} \dots e_{i-2k+2} E_{i-2k} S_{x'}$$

Where $k > 0$ and $S_{x'}$ has no edge indexed higher than $i - 2k - 4$. Then the substitution is :

$$e_i\, e_{i-2} \dots e_{i-2k+2}\, E_{i-2k} = E_i\, e_{i-1}\, e_{i-3} \dots e_{i-2k+1}.$$

Proof. The proof is very similar to the previous lemma. We want to change the highest indexed edge e_i into E_i.

The greedy minimal expansion algorithm ME produces "hypercubic" edge in cases there is 10 string in the beginning of input string x. Then, if we have changed e_i into E_i, we need to deal with 1 on position $i - 1$ ($10 + E_i = 01$). There are two options: E_{i-1} or e_{i-1}.

We also know there was e_{i-2} in the original minimal expansion, therefore if we use E_{i-1}, we would have to use e_{i-2} as well.

$$e_i\, e_{i-2} = E_i\, E_{i-1}\, e_{i-2}.$$

Clearly the rest of the minimal expansion would remain the same as in the original outcome. Therefore expansion with E_{i-1} would have weight $|S_x| + 1$.

If we use e_{i-1} instead, the outcome is different. Let us demonstrate it on the string x where $x + S_x = 00...00$. Then the string for the original minimal expansion must be $x = 10(10)^*11$. Then $x + e_i = 00(10)^*11$ and $x + E_i + e_{i+1} = 00(01)^*00$. We then only need $e_{i-3} + ... + e_{i-2k+1}$ to get string 00..00. It does not matter what bits are behind position $i - 2k$ since the original expansion used one "twisted" edge and the new one also uses one "twisted" edge. The weight of S_x remains unchanged. □

When we have defined the operation of substitution of minimal expansion, we can use it in the shortest path routing. In example 4 in the previous section we saw, that this substitution is needed in cases 3 and 4 of the routing algorithm. In fact, it is important to do this substitution every time such substitution is possible. In the approximate algorithm, cases 3 and 4 were substitutions of one edge by two. Now if it is possible to make substitutions that do not add unnecessary edges, it will be done. The new shortest path algorithm is described in figure 2.9. The only change from the approximate algorithm is in cases 3 and 4.

There are 4 cases in the algorithm, with every part being divided into two parts. We keep the same order of labeling the cases as in [1]. The same operations as in the approximate shortest path routing algorithm are needed.

Case 0, Case 1 and Case 2 remains unchanged from the approximate shortest path routing algorithm from section 2.2.

Case 3 This case is used when the edge with highest index i is e_i and it is not an existing edge incident on node x. Therefore we need to find a substitution for this edge. There are two parts of this case, depending on if it is possible to substitute it with an equal minimal dimension:

Case 3 part 1 If there are edges $e_i, e_{i-2}, ..., e_{i-2k+2}, E_{i-2k}$ in set S, we substitute this part of S by lemma 5.

1. Remove $e_i, e_{i-2}, ..., e_{i-2k+2}, E_{i-2k}$ from set S.
2. Add edges $E_i, e_{i-1}, e_{i-3}..., e_{i-2k+1}$ to set S.
3. Apply the algorithm recursively on new set S.

The substituted sequence of edges can have any possible weight. The smallest possible substitution is $e_i, E_{i-2} \rightarrow E_i, e_{i-1}$.

Case 3 part 2 Otherwise, a substitution which adds one edge is applied.

1. Remove e_i from set S.
2. Add edges E_i and E_{i-1} to set S.
3. Apply the algorithm recursively on new set S.

There is an edge incident on x in every dimension so if e_i does not exist, there must be existing E_i edge instead. So it is possible to add E_i to S and because we wanted to route to where E_i would have lead and we know that $e_i - E_i = e_i + E_i = E_{i-1}$, we also add E_{i-1} into S. If this would have happen during the generation of minimal expansion, the generated set would have been different just at positions i - which would have changed from e to E, and at position $i - 1$ which was vacant before because the ME generates edge for at most every two bits, the rest of set S would have been unchanged. Then, $|S|$ would have been larger by 1.

Case 4 This case is similar to case 3. It happens if the edge with highest index i is E_i and it is not existing edge incident on node x. Therefore we need to find substitution for this edge. There are two parts of this case, depending if it is possible to substitute with equal minimal dimension:

Case 4 part 1 If there are edges $E_i, e_{i-2}, ..., e_{i-2k+2}, E_{i-2k}$ in set S, we substitute this part of S by lemma 4.

1. Remove $E_i, e_{i-2}, ..., e_{i-2k+2}, E_{i-2k}$ from set S.
2. Add edges $e_i, e_{i-1}, e_{i-3}..., e_{i-2k+1}$ to set S.
3. Apply the algorithm recursively on new set S.

The substituted sequence of edges can have any possible weight. The smallest possible substitution is $E_i, E_{i-2} \rightarrow e_i, e_{i-1}$.

Case 4 part 2 Otherwise, substitution which adds one edge is applied.

1. Remove E_i from set S.
2. Add edges e_i and E_{i-1} to set S.
3. Apply the algorithm recursively on new set S.

There is an edge incident on x in every dimension so if E_i is not existing, there must exist e_i edge instead. So it is possible to add e_i to S and because we wanted to route to where E_i would have led and we know that $E_i - e_i = E_i + e_i = E_{i-1}$, we also add E_{i-1} into S. If this would have happened during generating minimal expansion, the generated set would have been different just at position i, which would have changed from E to e, and at position $i - 1$, which was vacant before because the ME generates edge for at most every two bits, the rest of set S would have been unchanged. Then, $|S|$ would have been larger by 1.

Theorem 2. *The shortest path routing algorithm correctly routes in a minimal number of steps from source node x to destination node y, given input set S containing "greedy" minimal expansion of $x + y$.*

Proof. We prove that the number of steps used is necessary and sufficient by inductive proof.
Base step. $|S| = 0$. Zero routing steps are necessary and sufficient, because $x = y$, and the algorithm correctly terminates after doing nothing.

Inductive step. $|S| = 0$. Assume that the theorem is true for $|S| = 1, ..., k-1$. Consider the lowest indexed edge in S. Let this edge have index i. Routing along any edge with index j, $j < i$ doesn't affect bit i in the address of X. Routing along any edge j, $j > i$ doesn't lead to a minimal path from x to y, by Lemma 2. So the algorithm must eventually route along only one of the edges $z + e_i$ or $z + E_i$ for some node z on the path between x and y.

Each of the four cases in the algorithm examines a possible condition for the edge at position i:

```
spr(x, y, S)
{
  int i = S.index;
  case 0: (x == y)
      break;
  case 1: (isEdge(x, S.top) && S.top == e_i)
    if(S.next.index == i+1 && isEdge(x,S.next) )
    {
      S.remove(e_i);
      approxSpr(x,getNeighbor(y, e_i), S);
      print(y);
    }else
    {
      x = getNeighbor(x,e_i);
      print(x);
      S.remove(e_i);
      approxSpr(x,y,S);
    }
  case 2: (isEdge(x,S.top) && S.top == E_i)
     if(∃ t_j ∈ S && isEdge(x,t_j) )
     {
       x = getNeighbor(x,t_j);
       print(x);
       S.remove(t_j);
       approxSpr(x,y,S);
     }else
     {
       x = getNeighbor(x,E_i);
       print(x);
       S.remove(E_i);
       approxSpr(x,y,S);
     }
  case 3: (!isEdge(x,S.top) && S.top == e_i)
     if(e_i, e_i-2 ,..., e_i-2k+2, E_i-2k ∈ S){
        S.remove(e_i, e_i-2 ,..., e_i-2k+2, E_i-2k);
        S.add(E_i, e_i-1 ,..., e_i-2k+1));
     }else{
       S.remove(e_i);
       S.add(E_i, E_i-1);
     }
     spr(x,y,S);
 case 4: (!isEdge(x,S.top) && S.top == E_i)
     if(E_i, e_i-2 ,..., e_i-2k+2, E_i-2k ∈ S){
        S.remove(E_i, e_i-2 ,..., e_i-2k+2, E_i-2k);
        S.add(e_i, e_i-1 ,..., e_i-2k+1));
     }else{
        S.remove(E_i);
        S.add(e_i, E_i-1);
     }
     spr(x,y,S);
}
```

Figure 2.9: The shortest path routing algorithm.

Case 1. The highest indexed edge in S is e_i that is adjacent to node x.
In the first part of this case, the algorithm routes recursively from x to $y - e_i$, then along edge e_i to y in one necessary and sufficient step. If all edges in S are correct edges with respect to route between x $y - e_i$, by the assumption the routing from x to $y - e_i$ takes $|S| - 1$ steps, so the routing from x to y takes $|S|$ steps. Otherwise, by the assumption the routing from x to $y - e_i$ takes $|S| - 1$ or $|S|$ steps, thus the routing from x to y takes $|S|$ or $|S| + 1$ steps.

In the second part of this case, the algorithm routes along edge from x to

$x + e_i$ in one necessary and sufficient step, then recursively from $x + e_i$ to y. By the same arguments as above, the hypothesis holds.

Case 2. The highest indexed edge in S is E_i that is adjacent to node x.
In the first part of this case, the algorithm routes along an edge $x + t_j$, $j < i$. Because $S - t_j$ then contains edge E_i adjacent to $x + t_j$ (as it is explained in description of algorithm), by induction, the sufficient and necessary number of steps is $|S| - t_j + 1 = |S|$.

In the second part of this case, the algorithm routes along the edge from x to $x + E_i$ in one necessary and sufficient step. Because $S - E_i$ then contains only correct edges with respect from $x + E_i$ to y, by induction, the necessary and sufficient number of steps is $|S| - E_i + 1 = |S|$.

Case 3. Highest indexed edge in S is e_i that is NOT adjacent to node x.
Because e_i is not existing edge in MC_n with respect to x, E_i is adjacent to x, and the algorithm must eventually route along some edge from z to $z + E_i$.

If by Lemma 5 there is an alternate minimal expansion for S that begins with E_i, then the algorithm can replace S with that expansion. Then by Case 2 above, $|S|$ steps are necessary and sufficient.

If no alternate expansion exists for S that begins with E_i, then $|S| + 1$ steps are necessary. The algorithm replaces ei in S with E_i and E_{i+1}, so by Case 2 above, $|S| + 1$ steps are necessary and sufficient.

Case 4. Highest indexed edge in S is E_i that is NOT adjacent to node x.
Because E_i is not existing edge in MC_n with respect to x, e_i is adjacent to x, and the algorithm must eventually route along some edge from z to $z + e_i$.

If by Lemma 4 there is an alternate minimal expansion for S that begins with e_i, then the algorithm can replace S with that expansion. Then by Case 1 above, $|S|$ or $|S| + 1$ steps are necessary and sufficient.

If no alternate expansion exists for S that begins with e_i, then $|S|+1$ steps are necessary. The algorithm replaces ei in S with e_i and E_{i+1}, so by Case 2 above, $|S| + 1$ steps are necessary and sufficient.

If $|S|$ steps are used, then the algorithm is clearly minimal. Since $|S| + 1$ steps are used only when $|S|$ steps are not sufficient, the routing algorithm is minimal. □

Now, on examples we will show, that the routing is correct where the approximate routing was wrong and it remains same in cases where it was correct.

Example 5 Example 5 can be found in Figure 2.7 on page 30.

spr on 0-MC$_4$:
$x = 0000$
$y = 1100$
ME(1100) = $\{E_3, E_1\}$

1. **case 4:** $S = \{e_3, e_2\}$
2. **case 1:** $y = 0100$, $S = \{e_2\}$
3. **case 1:** $x = 0100$, print(0100), $S = \{\}$
4. **case 0:** $x = y$
5. **finalize step 2:** print(0100)

Result: $0000 \rightarrow 0100 \rightarrow 1100$

This is the same example as example 4 where the approximate algorithm made mistake in step 1. Now according to the rule: $E_i e_{i-2} ... e_{i-2k+2} E_{i-2k}$ $= e_i e_{i-1} e_{i-3} \ ... e_{i-2k+1}$ there are the E_3 and E_1 edges in S, which can be interpreted as E_i and E_{i-2k} where $k = 1$, thus the substitution can be done. E_3 is substituted to e_3 and e_2 is added and E_1 is omitted. Then the algorithm will find the shortest possible routing.

Example 6 Example 6 can be found in Figure 2.10.

spr on 0-MC$_4$:
$x = 1000$
$y = 0101$
ME(1100) = $\{E_3, e_1\}$

1. **case 4:** $S = \{e_3, E_2, e_1\}$
2. **case 1:** $y = 1101$, $S = \{E_2, e_1\}$
3. **case 2:** $x = 1010$, print(1010), $S = \{E_2\}$
4. **case 2:** $x = 1101$, print(1101), $S = \{\}$
5. **case 0:** $x = y$
6. **finalize step 2:** print(0101)

Result: $1000 \rightarrow 1010 \rightarrow 1101 \rightarrow 0101$

This is the same example as example 3. It is showing that when the substitution of the same number of edges is not possible, the routing will remain same as in the approximate algorithm. Also examples 1 and 2 would remain

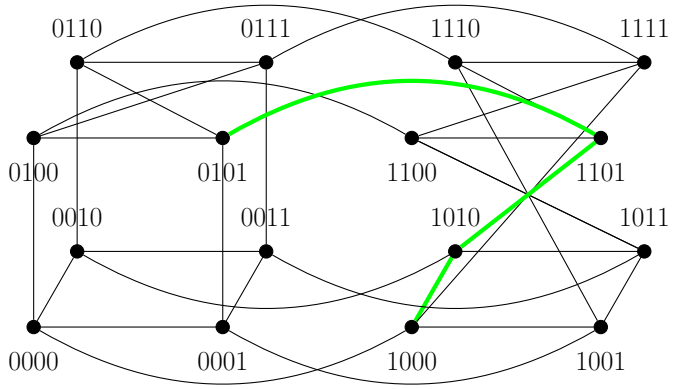


Figure 2.10: Routing from example 6 from 1000 to 0101.

unchanged.

We will show the diameter and average distance of 0-MC$_n$ and 1-MC$_n$ in the next section, the runtime of the *spr* algorithm as it is introduced in figure 2.9 is $O(n^2)$. That is caused by the sections 2, 3 and 4. But if we implement the algorithm correctly the computation time can be lowered to $O(n)$. More of the implementation is in chapter 5.

2.4 Diameter and average distance

Because the *spr* algorithm is generating the shortest possible routings, the average distance and the diameter are same as the values of the Möbius cubes. Although the algorithm is identical for both topologies, it uses different cases while routing from the same nodes in the different cube types. Therefore the diameter and the average distance will be different too.

2.4.1 Diameter

Theorem 3. *The diameter of 0-MC$_n$:*

$$\varnothing(\textit{0-MC}_n) = \left\lceil \frac{n+2}{2} \right\rceil, n \geq 4.$$

Proof. The minimal expansion generates edges with the distance between indeces at least 2. The maximal number of edges in the minimal expansion is $\lceil \frac{n}{2} \rceil$. That is the minimal number of steps the *spr* algorithm will take but by cases 3 and 4, the number can rise. We know that the diameter of *spr* is same as for the topology. We are looking for a string s which minimal expansion leads to as long routing as possible. Such string is $s = 11(01)^{(n/2)-1}$. This

string has E_{n-1} in its minimal expansion, which in 0-MC$_n$ does not exist. In cubes with even n, minimal expansion of s will be $E_{n-1}\, e_{n-3}...e_1$, so E_{n-1} can be subtitued only by $e_{n-1}\, E_{n-2}$ that is adding one edge to the routing. In cubes with odd n we also get E_0 in the minimal expansion so it is possible to substitute the minimal expansion of s into $e_{n-1}\, e_{n-2}e_{n-4}...e_1$ which does not raise the total number of edges. Note that the routing is consisting only of "hypercubic" edges and there is 2^n different pairs (one routing from each nodes) with this minimal expansion. Some of them needs to route through a "twisted" edges using case 3 and adding one more edge. The original number of the edges in the minimal expansion was $\lceil \frac{n}{2} \rceil$, if we add one edge it becomes $\lceil \frac{n}{2} + 1 \rceil = \left\lceil \frac{n+2}{2} \right\rceil$. □

Theorem 4. *The diameter of 1-MC$_n$:*

$$\varnothing(\textit{1-MC}_n) = \left\lceil \frac{n+1}{2} \right\rceil, n \geq 1.$$

Proof. The proof is similar as for the 0-MC$_n$. We know that the minimal number of steps is $\lceil \frac{n}{2} \rceil$. Again we are looking for the worst possible scenario. For even n, there is a string $(10)^{n/2}$ which will cause only "hypercubic" edges to be in the minimal expansion anf one of them will be e_{n-1} which is surely a "twisted" edge in 1-MC$_n$. Therefore the substitution will happen and there is no E_i so the number of steps will be raised by 1. On the other hand, with odd n, there is no way to generate the minimal expansion of size $\lceil \frac{n}{2} \rceil$ without having E_0 in address. Therefore, every substitution will not change the number of steps. The number of steps needed to reach any node in even dimensioned cube is $\frac{n}{2}+1$ and in odd dimensioned it is $\lceil \frac{n}{2} \rceil$. Combined together it is $\left\lfloor \frac{n+2}{2} \right\rfloor = \left\lceil \frac{n-1}{2} \right\rceil$. □

2.4.2 Average distance

Following text is taken from [1] with changes: average distance for expected distance and β^n for Z_2^n.
The asymmetries of the Möbius cubes makes their average distance difficult to calculate exactly. However, it is possible to bound them within values significantly below that of the hypercube's average distance. We show the bounds by computing $D(n)$, the expected number of terms in the minimal expansion of any two strings. If we choose a uniform distribution of source and destination address strings, the sums of the source and destination will be uniformly distributed over β^n. Using the "greedy" minimum expansion algorithm the average weight of all elements of β^n is:

$$D(n) = \frac{1}{2}D(n-1) + \frac{1}{2}(1 + D(n-2)), D(1) = \frac{1}{2}, D(2) = \frac{3}{4}.$$

The solution of this recurrence relation is:

$$D(n) = \frac{n}{3} + \frac{1}{9}\left[1 - \left(-\frac{1}{2}\right)^n\right].$$

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, and the *spr* routing algorithm uses at most one more step than the number of terms in the expansion, we can bound the average distance:

$$\frac{n}{3} + \frac{1}{9}\left[1 - \left(-\frac{1}{2}\right)^n\right] \leq \overline{dist}(MC_n) \leq \frac{n}{3} + \frac{1}{9}\left[1 - \left(-\frac{1}{2}\right)^n\right] + 1.$$

Although the Mobius cubes have a diameter approximately half that of the hypercube, its average distance is approximately two thirds the hypercube's average distance.

2.5 Routing algorithm variants

One of the goals of this thesis was to introduce a different routing algorithm in MC_n. There are many possibilities how to route and keep distances at the minimum. Because (as it is proven in next Chapter) no shortest possible routing is deadlock-free, therefore none other routing was finished. In the following figures, there is a routing table of *spr* algorithm and of another different yet also minimal routing.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0		1	2	2	4	4	2	4	8	8	8	8	4	8	8	8
1	0		3	3	5	5	5	3	9	9	9	9	9	5	9	9
2	0	3		3	0	6	6	6	10	10	10	10	10	10	6	10
3	2	1	2		7	1	7	7	11	11	11	11	11	11	11	7
4	0	0	5	7		5	7	7	0	12	12	12	12	12	12	12
5	1	1	6	1	4		6	6	13	1	13	13	13	13	13	13
6	2	5	2	2	7	5		7	14	14	2	14	14	14	14	14
7	4	3	3	3	4	6	6		15	15	15	3	15	15	15	15
8	0	0	0	0	0	10	15	15		9	10	10	15	10	15	15
9	1	1	1	1	11	1	14	14	8		11	11	11	14	14	14
10	2	2	2	2	11	13	2	8	8	11		11	11	13	13	8
11	3	3	3	3	12	10	9	3	10	9	10		12	10	9	12
12	4	11	11	11	4	4	4	4	15	11	11	11		13	15	15
13	10	5	10	10	5	5	5	5	10	14	10	10	12		14	14
14	15	9	6	9	6	6	6	6	15	9	13	9	15	13		15
15	8	14	8	7	7	7	7	7	8	14	8	12	12	14	14	

Figure 2.11: Routing table for the *spr* algorithm in 0-MC_4. Address of the node is changed into decimal value. Table says which node to route from left column to top row. The colours determines dimensions of the edge.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0		1	2	2	4	4	2	4	15	15	2	4	15	2	15	15
1	0		3	3	5	5	5	3	14	14	5	3	3	14	14	14
2	0	0		3	0	6	6	6	0	6	13	3	3	13	13	0
3	1	1	2		7	1	7	7	7	1	2	12	12	2	1	12
4	0	0	5	7		5	7	7	7	11	11	11	11	11	11	0
5	1	1	6	1	4		6	6	10	6	10	10	10	10	1	10
6	2	5	2	2	5	5		7	7	9	5	9	9	2	9	7
7	4	3	3	3	4	4	6		8	6	8	4	3	8	6	8
8	15	15	15	7	7	10	7	7		9	10	10	15	10	15	15
9	14	14	6	14	11	6	6	6	8		11	11	11	14	14	14
10	13	5	13	13	11	5	5	8	8	11		11	11	13	13	8
11	4	12	12	12	4	10	9	4	10	9	10		12	10	9	12
12	15	3	3	3	11	11	11	3	15	11	11	11		13	15	15
13	2	14	2	2	10	10	2	10	10	14	10	10	12		14	14
14	15	1	13	1	9	1	9	9	15	9	13	9	15	13		15
15	0	14	0	12	0	8	8	8	8	14	8	12	12	14	14	

Figure 2.12: Routing table for the *spr* algorithm in the 1-MC_4.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0		1	2	2	4	4	2	4	8	8	8	8	4	8	8	8
1	0		3	3	5	5	5	3	9	9	9	9	9	5	9	9
2	0	0		3	0	6	6	6	10	10	10	10	10	10	6	10
3	1	1	2		7	1	7	7	11	11	11	11	11	11	11	7
4	0	0	0	7		5	7	7	0	12	12	12	12	12	12	12
5	1	1	6	1	4		6	6	13	1	13	13	13	13	13	13
6	2	5	2	2	5	5		7	14	14	2	14	14	14	14	14
7	4	3	3	3	4	4	6		15	15	15	3	15	15	15	15
8	0	0	0	0	0	0	0	15		9	10	10	15	10	15	15
9	1	1	1	1	1	1	14	1	8		11	11	11	14	14	14
10	2	2	2	2	2	13	2	2	8	8		11	13	13	13	8
11	3	3	3	3	12	2	3	3	9	9	10		12	12	9	12
12	4	4	4	11	4	4	4	4	15	11	11	11		13	15	15
13	5	5	10	5	5	5	5	5	10	14	10	10	12		14	14
14	6	9	6	6	6	6	6	6	9	9	13	9	13	13		15
15	8	7	7	7	7	7	7	7	8	8	8	12	12	12	14	

Figure 2.13: Another possible routing table in the 0-MC_4. It is visibly more regular than the *spr* algorithm.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0		1	2	2	4	4	2	4	15	15	15	4	15	2	15	15
1	0		3	3	5	5	5	3	14	14	5	14	3	14	14	14
2	0	0		3	0	6	6	6	13	6	13	13	13	13	13	0
3	1	1	2		7	1	7	7	7	12	12	12	12	12	1	12
4	0	0	0	7		5	7	7	7	11	11	11	11	11	11	0
5	1	1	6	6	4		6	6	10	6	10	10	10	10	1	10
6	2	5	2	2	5	5		7	9	9	5	9	9	2	9	9
7	4	3	3	3	4	4	6		8	8	8	4	3	8	8	8
8	15	7	7	7	7	10	7	7		9	10	10	15	10	15	15
9	6	14	6	6	11	6	6	6	8		11	11	11	14	14	14
10	5	5	13	5	5	5	5	8	8	8		11	13	13	13	8
11	4	4	4	12	4	4	9	4	9	9	10		12	12	9	12
12	15	3	3	3	11	3	3	3	15	11	11	11		13	15	15
13	2	14	2	2	10	10	2	2	10	14	10	10	12		14	14
14	1	1	13	1	1	1	9	1	9	9	13	9	12	13		15
15	0	0	0	12	0	0	0	8	8	8	8	12	12	12	14	

Figure 2.14: Another possible routing table in the 1-MC_4. It is visibly more regular than the *spr* algorithm.

Chapter 3

Deadlock freeness of the Möbius cubes

For any multiprocessor network it is necessary to have a routing algorithm that is deadlock-free. Otherwise we cannot establish a communication between a multiple nodes. We have already explained how the shortest path routing algorithm works. Now it is time to evaluate its deadlock freeness and eventually introduce another deadlock-free algorithm.

3.1 Deadlock on spr algorithm

If we compare the shortest path algorithm on MC_n with the *e-cube* routing on Q_n we can see that the *spr* does not route on the edges in ordered dimensions. The *spr* usually goes from the most significant bit to the least significant bit, but for example if the algorithm uses the first part of case 1 or the first part of case 2, it skips a higher dimension for lower. If such pattern is found, it is the first sign that it is possible for the routing to contain cycles.

The problematic part, where the routing uses unsorted dimensions, is made by the edges that does not have the parallel edges in opposite subcube. That means that there is only one routing with the minimal length for any route that goes through this edge. Then if we check a set of routings that goes for every pair, source and destination node, from one subcube to another, using one edge that has no parallel edge in the other subcube, we will find that there is a deadlock cycle. Examples of such a routing for 0-MC_3 and 1-MC_3 are shown in the examples 1 and 2.

Example 1 Routing from the example 1 is described in Figure 3.1. This example shows 8 routings, each of length 2, where each route goes through the node where previous routing ends and next one starts. Together when started

at one time, they will all route to the next node where they will be waiting for the next edge to become vacant, which will never happen because they fell into deadlock cycle. In this example, every second row is using different dimension ordering.

0-MC_3:
FROM 000 TO 111: $000 \to 100 \to 111$
FROM 100 TO 011: $100 \to 111 \to 011$
FROM 100 TO 001: $111 \to 011 \to 001$
FROM 011 TO 101: $011 \to 001 \to 101$
FROM 001 TO 110: $001 \to 101 \to 110$
FROM 101 TO 010: $101 \to 110 \to 010$
FROM 110 TO 000: $110 \to 010 \to 000$
FROM 010 TO 100: $010 \to 000 \to 100$

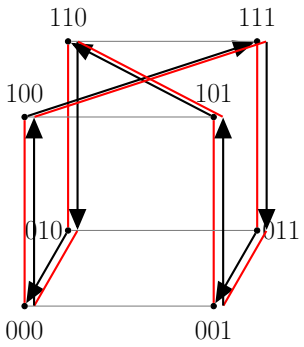


Figure 3.1: Example of deadlock in 0-MC_3. Red lines are edges with waiting communication.

Example 2 This is very similar example to routing in the example 1, but in this case, the routing is in 1-MC_3. Thus, the *spr* routing is falling into deadlock, no matter if 0-MC_n or 1-MC_n is used. The routing is described in the figure 3.2.

In these two routings examples we have shown the smallest possible deadlock while using the *spr* algorithm. Same deadlocks can be found in any MC_n, $n > 2$. This deadlock is not possible to remove thanks to its occurrence on the routings that have only one possible path to be shortest possible. In the examples above, there is no routing pair, where routing can be altered with a path of same length. Therefore, we need to search for an algorithm, that will be deadlock-free and will be using not the shortest existing paths but the shortest possible paths that does not cause deadlock. The other approach how to solve this problem would be adding virtual channels to some of the edges

1-MC_3:
FROM 000 TO 100: 000 → 111 → 100
FROM 111 TO 011: 111 → 100 → 011
FROM 100 TO 001: 100 → 011 → 001
FROM 011 TO 110: 011 → 001 → 110
FROM 001 TO 101: 001 → 110 → 101
FROM 011 TO 010: 110 → 101 → 010
FROM 101 TO 000: 101 → 010 → 000
FROM 010 TO 111: 010 → 000 → 111

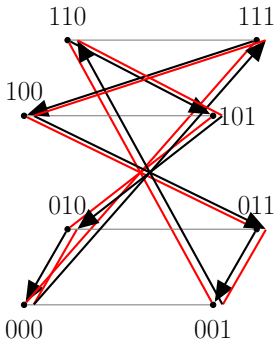


Figure 3.2: Example of deadlock in 1-MC_3. Red lines are edges with waiting communication.

so no existing cycle can fall into the deadlock anymore. In this text, we will focus on the first choice.

3.2 Deadlock-free routing algorithm

The theory of Mobius cube is coming from the Hypercube which has the ability of choosing any fixed order of dimensions that give us both the shortest path and the deadlock free routing. Such routing algorithm is called *e-cube* routing. As we can see from the shortest path routing algorithm, in many cases it is the fastest to use the most significant dimension first. We also know that the changing of the order of dimensions would cause the shortest path routing algorithm to fail. Therefore it is not possible to use any *e-cube* routing but only the one that goes strictly from the most significant bit to the least significant. Such routing will not be the shortest path possible in many cases, but will give correct routing.

Theorem 5. *The dfr algorithm is correct routing in MC_n.*

Proof. In each step of this algorithm, one bit of same dimension i in each

```
dfr(x, y, dim)
{
  if(x != y)
  {
    if(x + y & 2^dim)
    {
      x = neg_dim(x);
      print(x);
    }
    dfr(x, y, dim − 1);
  }
}
```

Figure 3.3: Routing algorithm in MC_n from the MSB to the LSB. The variable dim is equal to n on start. Note this is the same algorithm as *e-cube* in Q_n.

address is compared. If the result is 1, it means that nodes x and y resides in different subcube MC_i. It does not matters if MC_{i+1} is 0-MC_{i+1} or 1-MC_{i+1} because both the e_i and the E_i will change bit i in the address of x so the the routing will get to the same subcube as is y. Taking edge in the dimension i has no effect on bits in the higher dimensions, so by every routing the algorithm is getting into the smaller subcube. After the maximum of n steps the subcube will be MC_0 and routing will be successfully finnised. □

Also by taking any other ordering of the dimensions, the routing will never be successful because if going up in the dimension order and the E_i edge is used, it will change all the bits bellow, changing the bits that were already in the correct state.

Theorem 6. *The dfr algorithm is deadlock-free.*

Proof. Proof by induction.

Base step. The 0-MC_2 is the same topology as Q_2, therefore the *dfr* algorithm produces the same routing as the *e-cube*, which is deadlock-free. The 1-MC_2 topology is also deadlock-free. It can be confirmed by seeing all possible routings (trivial cases where the nodes are neighbors are ommited):

FROM 00 TO 10: $00 \rightarrow 11 \rightarrow 10$
FROM 01 TO 11: $01 \rightarrow 10 \rightarrow 11$
FROM 10 TO 00: $10 \rightarrow 01 \rightarrow 00$
FROM 11 TO 01: $11 \rightarrow 00 \rightarrow 01$

Induction step. Every MC_n is 0-MC_{n-1} connected with 1-MC_{n-1} by "twisted" or "hypercubic" edges. In both cases, any routing (concider only the routings

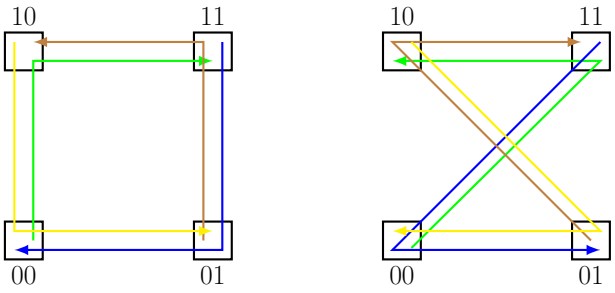


Figure 3.4: The *dfr* algorithm on 0-MC_2 and 1-MC_2.

that goes to the other MC_{n-1} subcube) will first use one of these highest indexed edges and than continue as a routing in MC_{n-1}. Then, if there was no deadlocking set of the routings in MC_{n-1}, there will not be any deadlocking set in MC_n. □

Unfortunately usage of the *dfr* algorithm means to lose the main advantage of the Möbius cubes - very good diameter.

Theorem 7. *Routing by the dfr algorithm from one node to every other node of MC_n creates the binomial tree which is the subgraph of Q_n. Therefore the dfr routing has the same diameter and the average distance as the hypercube.*

Proof. The equality of the *e-cube* and the *dfr* routing can be seen from the figure 3.5 where the binomial trees of routing from the node 0 are compared. Trees rooted in any other node will have the same shape.

The *dfr* routing routes to the subcube MC_{n-1} by only one edge so the nuber of nodes connected to the source node through the edge in dimension i is 2^i. Even by using the "twisted" edges it can never occur that two same nodes are in the two different subtrees because every edge have effect only on the bits below and routing goes strictly from the MSB to the LSB. There will be always one node in distance n, the one where the algorithm has to change dimension every single time. The number of the nodes in distance k will be $\binom{n}{k}$. This number comes from the the number of nodes connected to the root by one edge. Each such edge connects the whole subcube of dimension i. The source node has n neigbors so each this neighbor has in its subcube i neighbors. Therefore, the source node will have $n = \binom{n}{1}$ nodes in distance 1, $n-1+n-2+...1+0 = \binom{n}{2}$ neighbors in distance 2, $n-1-1+n-1-2+...1+0+n-2-1+n-2-2+...1+0...+1-1 = \binom{n}{3}$ in distance 3 and so on. These numbers are identical to the *e-cube* routing. □

The *dfr* algorithm is not the shortest possible deadlock-free routing. It would be possible to improve the *dfr* by a few routings from *spr*. But it would have to be controlled in every detail for every routing that any deadlock

situation cannot occur and the resulting impact on the diameter would be minimal. Plus the algorithm would be changed into set of exceptions and lost its generality.

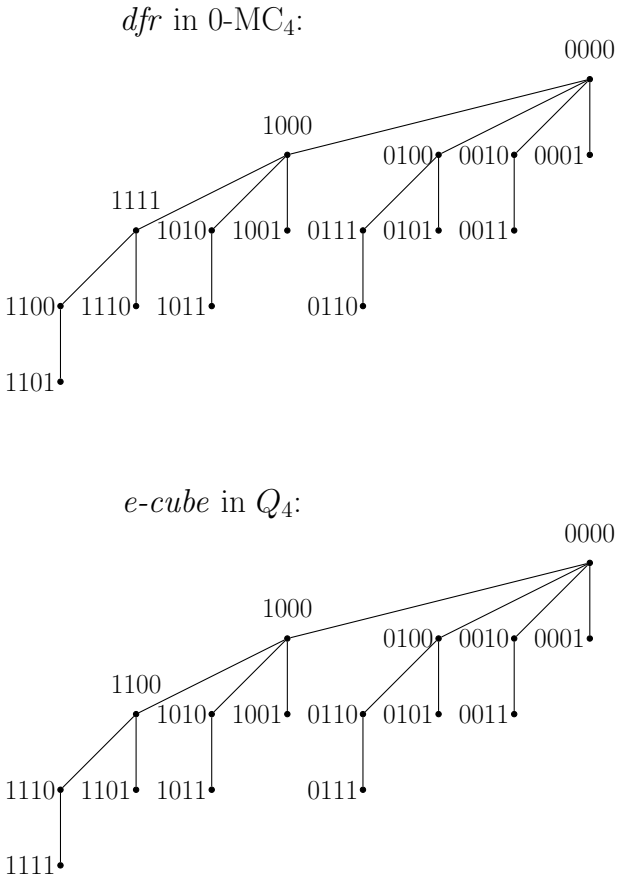


Figure 3.5: Routing trees of routing from node 0000 in 0-MC$_4$ using *dfr* algorithm and in Q_4 using *e-cube* routing.

3.3 The diameter and average distance of dfr algorithm

The *dfr* algorithm is using subgraph of the hypercube so the diameter and average distance are the same as on hypercube. For both 0-MC$_n$ and 1-Mc$_n$, the diameter and the average distance are identical. These two topologies have different edges in the highest dimension but the *dfr* algorithm use them for

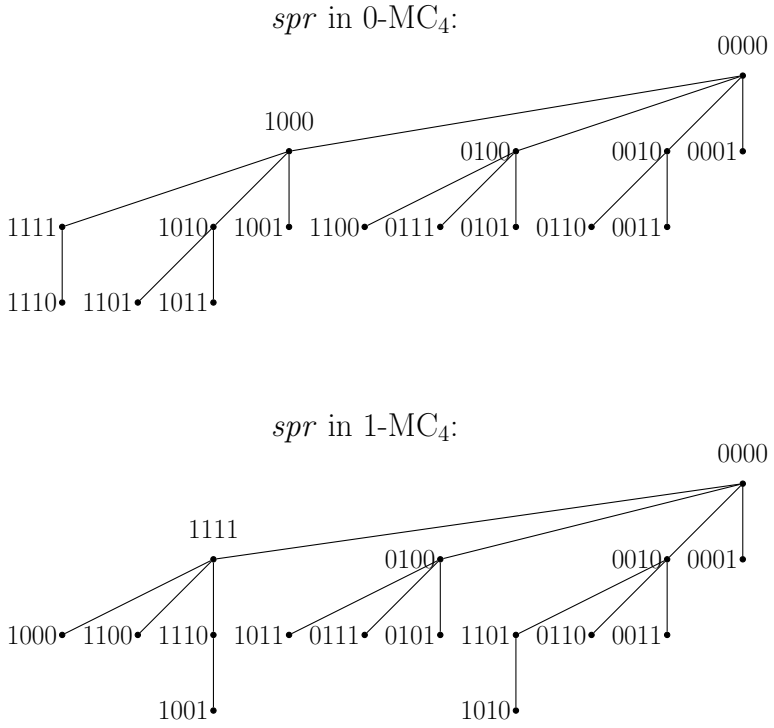


Figure 3.6: routing trees of routing from node 0000 in 0-MC_4 and in 1-MC_4 using *spr* algorithm.

the same purpose at same time - just to change the highest bit. The diameter of *dfr* algorithm is n and the average distance is $\overline{dist}(dfr) \doteq \lceil \frac{n}{2} \rceil$.

The comparament between the *spr* and the hypercube is in previous chapter. Because the data of the hypercube and the *dfr* algorithm are identical, there is no need to compare the diameters and the average distances of the *spr* and the *dfr*.

Chapter 4

Multicast algorithm

In this chapter, we will desrcibe a multicast algorithm for 1-port WH hypercubes and proof that the same algorithm is working on the Möbius cubes and it is a round-optimal algorithm.

4.1 ADOC - based MC algorithm on 1-port WH hypercubes

Definition 4. *Consider a WH hypercube Q_n with the e-cube routing based on checking the bits from the MSB to the LSB. Then an ascending dimension-ordered chain (ADOC) is any sequence of nodes $u_1, ..., u_k$ in Q_n such that $u_i < u_{i+1}$ for all $i = 1, ...k-1$, where $<$ is a standard lexicographical ordering of binary strings.*

For example, sequence 0100, 0101, 1000, 1011 is an ADOC in Q_4.

Lemma 6. *Let u, v, w, z be an ADOC in Q_n Then (see figure)*
1. paths $u \to v$ and $w \to z$ are link-disjoint,
2. paths $v \to u$ and $z \to w$ are link-disjoint,
3. paths $v \to u$ and $w \to z$ are link-disjoint,
4. paths $u \to v$ and $u \to z$ are not link-disjoint, $u \to z$ has priority,
5. paths $u \to z$ and $v \to w$ are not link-disjoint, $u \to z$ has priority.

Proof. We will proof every part of lemma 6 by showing every possible situation that can happen in ADOC routing. When talking about the subcubes, we will consider only subcubes how they are visited by the MSB to LSB routing. We consider only 2 subcubes of cube defined by the highest bit, 4 subcubes defined by the 2 highest bits and so on. The node u' is a neighbor of u in some subcube. First we will describe all the situations:
a) u and v are in different subcube than w and z.
b) u, v and w are in the same subcube while z is in different one.

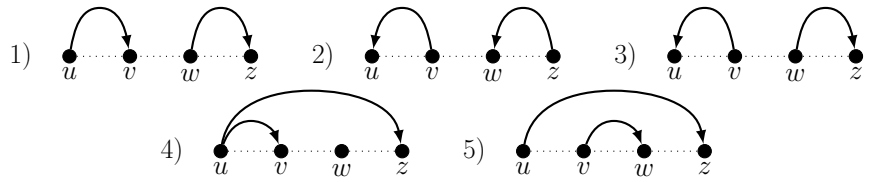


Figure 4.1: Five cases of paths joining lexicographically ordered hypercube nodes.

c) u is in different subcube than other nodes but u' and v are in different smaller subcube(defined by more bits) than w and z.
d) u is in different subcube than other nodes and u' is on the routing $w \to z$.
e) u is in different subcube than other nodes and $u' = w$.
f) v and w are on the route $u \to z$.

No other possibilities are possible because $u < v < w < z$. Now we will discuss each situation in each part of lemma 6.
1) $u \to v$ $w \to z$
1-a) The paths are node and edge disjoint as no routing goes to the other subcube if its destination node does not belongs to such subcube.
1-b) The paths are node and edge disjoint. The routing $u \to v$ does not include w because $w > v$ and the routing from w goes directly to other subcube where z is.
1-c) This situation is same as 1-a.
1-d) The paths are edge disjoint. In the worst case, the routing $u' \to v$ will route one edge to w and then to v while the routing $w \to z$ uses the same edge but in the opposite direction, which is acceptable. This is proven by $v < w < z$, therefore routing from w have to lead different direction than $u' \to v$.
1-e) The paths are edge disjoint. The nodes v, $u' = w$ and z are in the same subcube but $v < z$ so for any smaller subcubes they are in a different subcube so the routing from w will use a different edge for both nodes.
1-f) This is just special case of 1-c and therefore of 1-a as well.

2) $u \leftarrow v$ $w \leftarrow z$
2-a) Same as 1-a.
2-b) The paths are edge disjoint. In the worst case, the routing $z' \to w$ will route one edge to v and then to w while the routing $v \to u$ uses the same edge but in the opposite direction, which is acceptable. This is proven by $u < v < w$, therefore routing from v have to lead different direction than $z' \to w$.
2-c) Same as 2-a.
2-d) The paths are edge disjoint. $v \to u$ will first use edge to different sub-

cube, leaving all edges of original subcube for $z \to w$.
2-e) Same as 2-d.
2-f) Same as 2-a.

3) $u \leftarrow v\ w \to z$
3-a) Same as 1-a.
3-b) The paths are node and edge disjoint. $w \to z$ will first use edge to the different subcube, leaving all edges of the original subcube for $v \to u$ which will never route through w because $u < v < w$.
3-c) Same as 1-a.
3-d) The paths are node and edge disjoint. $v \to u$ will first use edge to the different subcube, leaving all edges of the original subcube for $w \to z$ which will never route through v because $v < w < z$.
3-e) Same as 3-d.
3-f) Same as 1-a.
Note that all routings in part 3 are both node and edge disjoint.

Part 4 will be contain a collision in each routing where v and z are in the same subcube. Part 5 will contain a collision in 4-f, clearly from the description of the situation.

□

Since Q_n is vertex-symmetric, we can assume that the multicast always starts from source $s = 0^n$. If not, we can apply a translation. The success of the optional 1-port multicast is based on lemma 6. Simply, we take $M \cup s$ and transform it into an ADOC. This allows us to interpret it as 1-D mesh and we can apply the recursive doubling for 1-port WH 1-D meshes. Hence,

$$\text{MC}(Q_n, M, s) = \text{1-DMeshRDOAB}(ADOC(M \cup \{s\}), s).$$

This is clearly an optimal algorithm, since

$$r_{MC}(Q_n, M, s) = \lceil \log(|M| + 1) \rceil = \rho(Q_n, M, s).$$

4.2 ADOC algorithm on 1-port WH Möbius cubes

We will discuss using both the *spr* and the *dfr* routing algorithms with ADOC algorithm on 1-port WH Möbius cubes.

4.2.1 Using spr algorithm with ADOC

The shortest path routing algorithm is generating the shortest possible routings but it is not deadlock free routing while using the wormhole switching. That does not necessary means that it will be also blocking on ADOC. ADOC

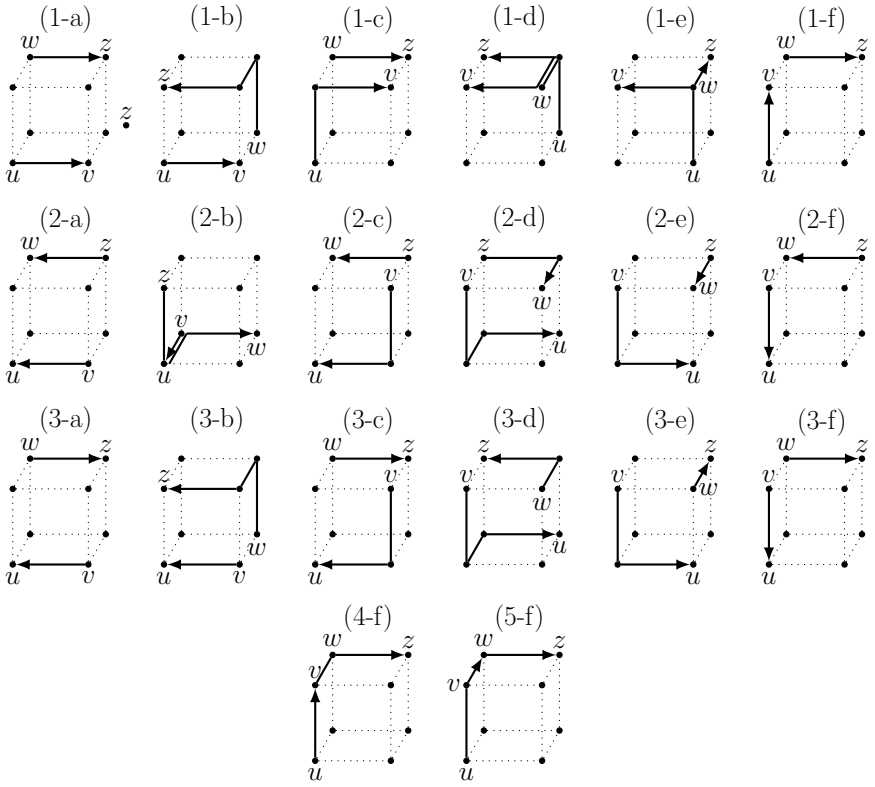


Figure 4.2: The examples of cases of the proof of lemma 6.

1-DMeshRDOAB$(M(z), s)$:

Phase 1.1:	Source s splits the mesh to 2 halves: if z is odd, s keeps the smaller part.
Phase 1.2:	Source s sends the packet to first node in the other half.
Phase 2:	Repeat phase 1 recursively in both halves simultaneously.

Figure 4.3: 1-D mesh recursive doubling one-to-all broadcast algorithm on M(z).

algorithm is based on using the MSB to LSB routing, but the *spr* contains exceptions that allows changing the order of the dimensions. For example the first part of case 2 of the *spr* algorithm in 1-MC$_n$ can cause that the routing will use same edge as routing in the other half. We can describe this situation with sorted ordered nodes u, v, w and x, where u and w has the message and when w is routing to x, it is possible that it will use the edge to v. In some case this can be the edge which u needs to route with. Such situation would cause one of the routings to be blocked and wait for the other one.

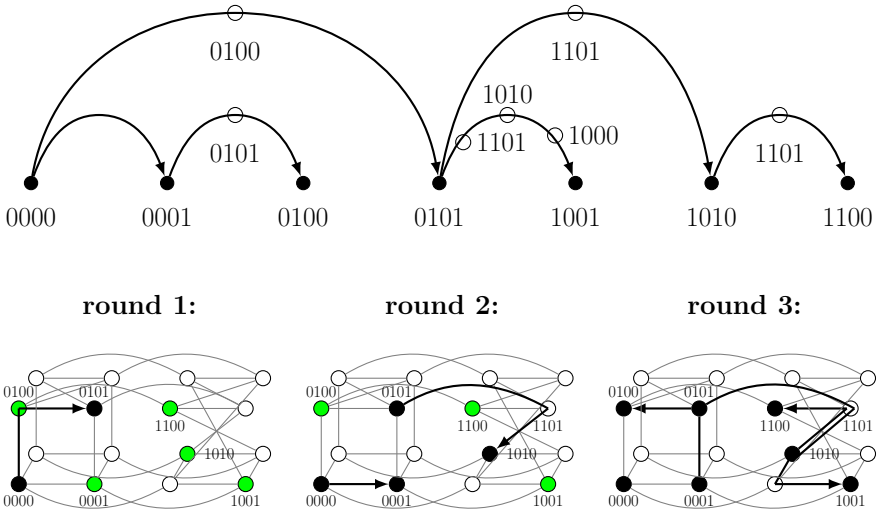


Figure 4.4: An example of time-optimal multicast in 0-MC_4

4.2.2 Using dfr algorithm with ADOC

The *dfr* algorithm is the same routing algorithm as the *e-cube* for the hypercube. And the result of *dfr* routing will always be a subgraph of hypercube but this subgraph is different from the hypercube by the labels of its nodes. Ee need to check that this routing will be truly non-blocking.

Theorem 8. *The ADOC algorithm with dfr algorithm is non-blocking routing on the Möbius cubes. The same 5 rules from lemma 6 aplies.*

Proof. We know that the ADOC with The *e-cube* routing is non-blocking routing on the hypercube. In the Möbius cubes, around half of the edges is "hypercubic", which means that there is same edge between u and v in both MC_n and Q_n. For these "hypercubic" edges is the algorithm same as in the hypercube. Hence, it is non-blocking.

For the "twisted" edges, we will proof every part of the lemma 6 by showing every possible situation that can happen in the ADOC routing. When talking about the subcubes, we will consider only subcubes how they are visited by the MSB to LSB routing. So only 2 subcubes of cube defined of the highest bit, 4 subcubes defined by the 2 highest bits and so on. The node u' is a neighbor of u in some subcube. First we will describe all the situations:
a) u and v are in a different subcube than w and z.
b) u, v and w are in the same subcube while z is in different one.
c) u is in different subcube than other nodes but u' and v are in a different smaller subcube(defined by more bits) than w and z.
d) u is in different subcube than other nodes and u' is on the routing $w \to z$.
e) u is in different subcube than other nodes and $u' = w$.

f) v and w are on the route $u \rightarrow z$.

No other possibilities are possible because $u < v < w < z$. Now we will discuss each situation in each part of lemma 6.
1) $u \rightarrow v$ $w \rightarrow z$
1-a) The paths are node and edge disjoint as no routing goes to the other subcube if its destination node does not belongs to such subcube.
1-b) The paths are edge disjoint. $w \rightarrow z$ will first use edge to the different subcube, leaving all edges of the original subcube for $u \rightarrow v$.
1-c) This situation is same as 1-a.
1-d) The paths are edge disjoint. In the worst case, the routing $u' \rightarrow v$ will route one edge to w and then to v while the routing $w \rightarrow z$ uses the same edge but opposite direction, which is acceptable. This is proven by $v < w < z$, therefore routing from w have to lead in the different direction than $u' \rightarrow v$.
1-e) The paths are edge disjoint. The nodes v, $u' = w$ and z are in the same subcube but $v < z$ so for any smaller subcubes they are in the different subcube so the routing from w will use different edge for both nodes.
1-f) This is just special case of 1-c and therefore of 1-a as well.

2) $u \leftarrow v$ $w \leftarrow z$
2-a) Same as 1-a.
2-b) The paths are edge disjoint. In the worst case, the routing $z' \rightarrow w$ will route one edge to v and then to w while the routing $v \rightarrow u$ uses same edge but in the opposite direction, which is acceptable. This is proven by $u < v < w$, therefore routing from v have to lead in the different direction than $z' \rightarrow w$.
2-c)The paths are edge disjoint. $v \rightarrow u$ will first use edge to the different subcube, leaving all edges of the original subcube for $z \rightarrow w$.
2-d) Same as 2-c.
2-e) Same as 2-c.
2-f) Same as 2-a.

3) $u \leftarrow v$ $w \rightarrow z$
3-a) Same as 1-a.
3-b) The paths are node and edge disjoint. $w \rightarrow z$ will first use edge to the different subcube, leaving all edges of the original subcube for $v \rightarrow u$ which will never route through w because $u < v < w$.
3-c) Same as 1-a.
3-d) The paths are node and edge disjoint. $v \rightarrow u$ will first use edge to the different subcube, leaving all edges of the original subcube for $w \rightarrow z$ which will never route through v because $v < w < z$.
3-e) Same as 3-d.
3-f) Same as 1-a.
Note that all routings in part 3 are both node and edge disjoint.

Part 4 will be contain collision in each routing where v and z are in the same subcube. Part 5 will contain collision in 4-f, clearly from the description of the situation.

□

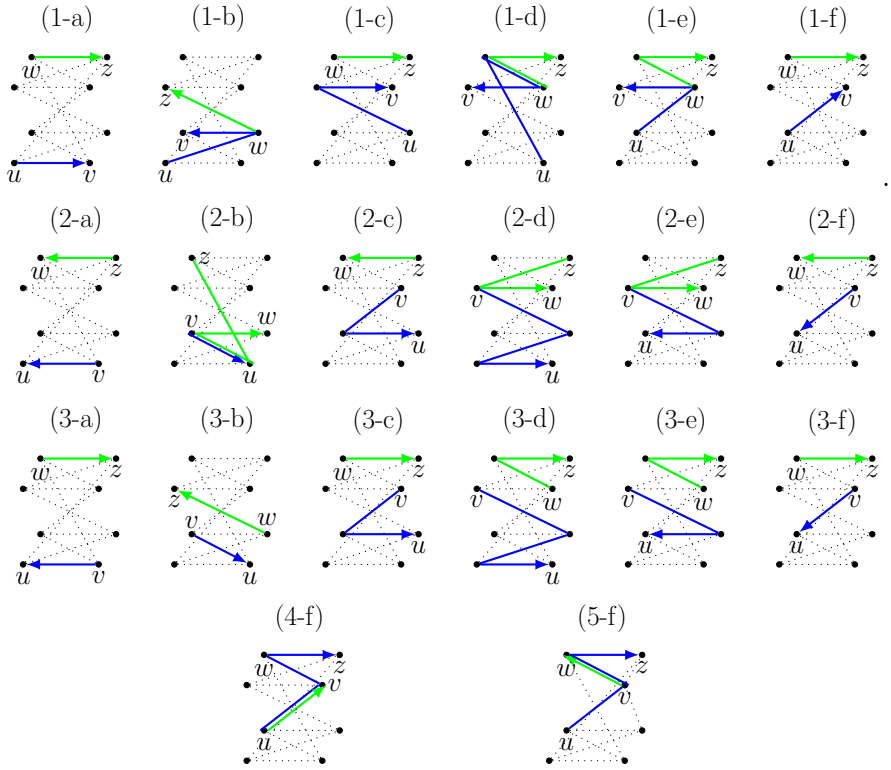


Figure 4.5: The examples of cases of proof of theorem 8. The examples are on cube with only "twisted" edges to show the most difficult scenarios.

Note that the algorithm for the hypercube can use the node symmetry to transform the source node into the node 0...0. That is not possible in MC_n but the algorithm is proven to send message to both directions - nodes with smaller value and also greater value than the source node. So the source node can be any node of MC_n.

4.2.3 optimality of ADOC algorithm on the Möbius cubes

The ADOC algorithm is same efficient on the hypercube as on the Möbius cubes. The nodes in both topologies are same denoted and same rules applies. The number of the rounds complexity of the algorithm is:

$$r_{MC}(MC_n, M, s) = \lceil \log(|M| + 1) \rceil.$$

Because every turn, the number of reached nodes is doubled. This is clearly an optimal algorithm, $r_{MC}(MC_n, M, s) = \rho(MC_n, M, s)$. Thanks to the wormhole routing, it is not interesting how long are the routings, only important is number of rounds to send message to all destination nodes.

Chapter 5

Implementation

This chapter covers details of the implementation of the *spr* algorithm, the *dfr* algorithm and the mctools library. The source code can be found on cd.

5.1 Used Tools

All programs were written in the C++ programming language. The programs itself does not need much of the C++ features and C language could have been used as well, the used features of C++ were just for the programmer comfort.

For the programing, a simple IDE Geany on the Debian operating system was used. The code was complied by GNU GCC.

5.2 Mctools library

The mctools library contains functions for confirming the correctness of the algorithm as well as the functions for routing algorithms to make the routing calculations easier. This library should be useful for implementation of various other problems on the Möbius cubes such as one-to-many communication algorithms. One part of the library are algorithms to confirm the correctness of the routing - if the routing is existing path in the topology and if it is the shortest possible routing.

To check the correctness of the routing is easy. We just need to check if any two nodes are neighbours. The library uses *isNeighbour()* function for this operation. The code is checking the differences in the addresses. It is enough to find the highest different bit and then check if all or none lower bits are reversed. So $O(n)$ operations is needed for the nodes notated by n bits.

To check if the routing is the shortest routing possible is more complicated operation as it need a algorithm that will count the minimal distance between two nodes and is general - not based on the specifics of the topology. Such

general algorithm is the depth-first searching. This search is done by the function *getDistance()*, which is called by the function *isShortest()*, which also checks if the routing is correct by the function *isRoute()*. The function *isShortest()* is sufficient tool to confirm that any routing is the shortest possible routing and if it does not contains wrong nodes.

For future work, it would be valuable if the library also contained functions for checking the deadlock freeness of algorithm. One of the variants would be to have algorithm that constructs the CDG graph.

5.3 Routing algorithms

There are three different routing algorithms in the source files. The first on is the *spr* algorithm. This is the original routing algorithm, programmed as it was introduces in [1]. In its source file, there is also the greedy minimal expansion algorithm, which is a necessary component of *spr*.

Second routing is the *dfr* algorithm. This routing algorithm is very trivial, its easiest variant could be written in just 10 lines.

The last included source file is the experimental *rsa* algorithm. This algorithm was meant to be a the new shortest possible routing algorithm, that would be easier than *spr*. Unfortunately, the correct mathematical explanation was not found and the algorithm grew into large set of exceptions and its development ended and is not likely to continue due to *rsa* algorithm being not deadlock-free. The routing table of *rsa* is in figure 2.13 and in figure 2.14.

Conclusion

In this thesis we have performed a survey of the Möbius cube theory. We have discussed the routing algorithm and introduced the new routing algorithm with respect to the deadlock-freeness. Lastly, we have discussed the problem of the round-optimal multicast and presented an optimal algorithm. Now we will summarize each of the goals and see the results.

Routing in the Möbius cube

There is a routing algorithm that we call the shortest path routing - *spr* algorithm and that was introduced together with the Möbius cubes in [1]. In Chapter 2, we have analyzed the algorithm and described every part of this difficult algorithm beginning with the input minimal expansion and following with all cases and subcases of the actual code. Using this shortest possible algorithm, the diameter of 0-MC$_n$ is:

$$\varnothing(\text{0-}MC_n) = \left\lceil \frac{n+2}{2} \right\rceil, n \geq 4.$$

And diameter of 1-MC$_n$ is:

$$\varnothing(\text{1-}MC_n) = \left\lceil \frac{n+1}{2} \right\rceil, n \geq 1.$$

Another important value is the average distance which, for any MC$_n$, we can bind:

$$\frac{n}{3} + \frac{1}{9}\left[1 - \left(-\frac{1}{2}\right)^n\right] \leq \overline{dist}(MC_n) \leq \frac{n}{3} + \frac{1}{9}\left[1 - \left(-\frac{1}{2}\right)^n\right] + 1.$$

All three values are excellent results of the Möbius cube theory, having the diameter approximately half of the diameter of Q_n and the average distance approximately two-thirds of the average distance of Q_n. In practical measures, for $n = 10$, the diameter is improved by 4 for both Möbius cubes and

the average distance is reduced from 5 under 4.

Another requirement for the routing algorithm is deadlock-freeness. In the chapter 3, we have proven that the *spr* algorithm is NOT deadlock-free and therefore its use is very limited. We have introduced another routing algorithm - *dfr*, which is deadlock-free but whose toll is very high. The diameter and the average distance is reduced to the values of the hypercube. Until a better deadlock-free algorithm is found, which is highly unexpected, the Möbius cubes loses its only advantage and becomes just a hypercube with a difficult edge ordering.

Multicast in the Möbius cube

While deadlock-free routing brings negative results, there are other new algorithms being developed that do not need to have the deadlock-free routing because the routing is restricted by other rules. Such an algorithm could be the multicast in 1-port wormhole network. This problem has very nice solution in the hypercube and we have proven that the exact same algorithm can be used in any Möbius cube. Also the number of rounds complexity of the algorithm is same as for Q_n:

$$r_{MC}(MC_n, M, s) = \lceil \log(|M| + 1) \rceil.$$

Summary

We have successfully performed research of the Möbius cubes and we have successfully introduced a deadlock-free routing algorithm, although the result casts a negative light on the Möbius cubes. Any shortest possible routing algorithm is deadlock-prone because it is not using fixed dimension ordering. To repair the deadlock-freeness it is inevitable to bring down the diameter back to the values of the hypercube. We have also successfully proven that the hypercube's multicast algorithm works efficiently in the Möbius cubes.

Bibliography

[1] Cull, P. and Larson, S., M: *The Mobius cubes*, in IEEE Transactions on Computers, vol. 44, no. 5, pp. 647-659, May 1995.

[2] Cull, P. and Larson, S., M: *Wormhole routing algorithms for twisted cube networks*, Parallel and Distributed Processing, 1994. Proceedings. Sixth IEEE Symposium on, Dallas, TX, 1994, pp. 696-703.

[3] Efe, D.: *A variation on the hypercube with lower diameter*, in IEEE Transactions on Computers, vol. 40, no. 11, pp. 1312-1316, Nov 1991.

[4] Fan, J.: *Hamilton-connectivity and cycle embedding of the möbius cubes*, Information Processing Letters, Volume 82, pp 113-117. 2002.

[5] Hsieh, S.-Y. and Chen, C.-H.: *Pancyclicity on möbius cubes with maximal edge faults*, Parallel Computing, Volume 30, pp 407-421. 2004.

[6] Kocik, D., Hirai, Y., and Kaneko, K.: *Analgorithm for node-to-node disjoint paths problem in a mobius cube*, Proceedings of the 2015 International Conference on Parallel and Distributed Processing Techniques and Applications, pp. 149-155. 2015.

[7] Li, Zhoujun; Sun, Yun and Wang, Deqiang: *Diameter bounds of cubelike recursive networks*, Parallel and Distributed Systems, 2007 International Conference on, Hsinchu, 2007, pp. 1-8.

[8] Tsai, C.-H.: *Embedding of meshes in möbius cubes*, Theoretical Computer Science, 401, pp 181-190. 2008.

[9] Tvrdik, Pavel: *Parallel algorithms and computing*, Czech Technical University in Prague, 2009. ISBN 978-80-01-04333-2.

[10] Tzeng, N., F. and Wei, S.: *Enhanced hypercubes*, in IEEE Transactions on Computers, vol. 40, no. 3, pp. 284-294, Mar 1991.

[11] Xu, J.-M.; Ma, M. and Lu, M.: *Paths in möbius cubes and crossed cubes*, Information Processing Letters, 97, pp 94-97. 2006.

[12] Yang, J.; Wu, M.; Chang, J. and Chang, Y.: *A fully parallelized scheme of constructing independent spanning trees on Möbius cubes*, The Journal of Supercomputing, Volume 71, Issue 3 , pp 952-965. 2015.

[13] Yang, X.; Megson, G., M. and Evans, D., J.: *Pancyclicity of möbius cubes with faulty nodes*, Microprocessors and Microsystems, Volume 30, pp 165-172. 2006.

[14] Zhou, W.; Fan, J.;Chen, B.; Han, Y. and Wang, Y.: *Broadcast in bijective connection networks with the restricted faulty node set*, Computer Science and Education (ICCSE), 2010 5th International Conference on, Hefei, 2010, pp. 230-235.

Appendix A

Contents of CD

- `readme.txt` ..... the file with CD contents description
- `src` ..... the directory of source codes
  - `mctools` ..... the directory of McTools library
    - `Makefile` ..... the makefile of McTools library (UNIX)
  - `thesis` ..... the directory of LaTeX source codes of the thesis
    - `thesis.tex` ..... the LaTeX source code files of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the Diploma thesis in PDF format
  - `thesis.ps` ..... the Diploma thesis in PS format