

Permutation Routing in the Hypercube and Grid Topologies

Timothy Alan Carnes

Ran Libeskind-Hadas, Advisor

Francis Edward Su, Reader

May 6, 2005

HARVEY MUDD
COLLEGE

Department of Mathematics

Abstract

The problem of edge-disjoint path routing arises from applications in distributed memory parallel computing. We examine this problem in both the directed hypercube and two-dimensional grid topologies. Complexity results are obtained for these problems where the routing must consist entirely of shortest-length paths. Additionally, approximation algorithms are presented for the case when the routing request is of a special form known as a permutation. Permutations simply require that no vertex in the graph may be used more than once as either a source or target for a routing request. Szymanski conjectured that permutations are always routable in the directed hypercube, and this remains an open problem.

Contents

Abstract	iii
1 Introduction	1
1.1 Problem Statement	1
1.2 Known Results	3
2 NP-Completeness	5
2.1 The Class P	5
2.2 The Class NP	6
2.3 NP-completeness	7
3 The Complexity of Hypercube Routing	9
3.1 L3-SAT	9
3.2 Reduction to Hypercube	10
3.3 Convey Apparatus	12
3.4 Embedding	12
3.5 Extension of Reduction	16
4 The Complexity of Grid Routing	19
4.1 The Graph	19
4.2 The Reduction	19
5 Approximation Algorithms	23
5.1 Binary Representation	23
5.2 Multistage Interconnection Network	24
5.3 Routing Two Permutations	26
5.4 Routing One Permutation	27
5.5 A 2-Approximation Algorithm	29
6 Future Work	31

Bibliography

33

List of Figures

- 2.1 Abstract sketch of NP-completeness reduction 7
- 3.1 The variable-setting gadget for the hypercube reduction . . . 11
- 3.2 The clause-checker gadget for the hypercube reduction . . . 12
- 3.3 The basic convey apparatus construction 13
- 3.4 Convey apparatus linking variable-setter to clause-checker . 14
- 3.5 How paths may be swapped so that unit length requests can
block edges 16
- 4.1 Setup for blocked edges in the grid reduction 21
- 4.2 Attempted routings for unsatisfiable instance of grid problem 21
- 5.1 The MIN induced from the three-dimensional hypercube. . . 25
- 5.2 Edges used in the created MIN to route a single permutation 28

Chapter 1

Introduction

Distributed memory parallel computers comprise a collection of processors which are interconnected via some network topology. Processors frequently need to send messages to each other through this network. Although a complete graph is a theoretically ideal topology, since it permits direct communication between all nodes, the construction of such a topology is impractical due to physical constraints on the degree of each node and the number of connections (edges). Consequently other topologies are generally used and messages must now be routed from one node to another via a path.

One method of passing information is to divide the messages up into small packets, which then travel through the processor network. Sometimes a packet will get stalled at an intermediary processor while waiting for a particular connection link to become free. This approach does not work so well, however, when there is a need to transmit great quantities of information between processors. At this point it becomes necessary to actually establish a communication path between two processors, and gives rise to the edge-disjoint routing problem.

1.1 Problem Statement

The edge-disjoint routing problem is given a graph $G = (V, E)$, and p request pairs with elements in V , find disjoint paths connecting each pair such that no edge is used by more than one path. For a given request pair, we generally refer to the first element as the source, and the second element as the target. Note that this problem has many related variants, which arise from constraining the parameters of the problem more tightly. In partic-

ular we can specify more precisely what type of graph we are working on, whether it is directed or undirected, what types of request pairs are allowed, and what kinds of paths we can use to connect them.

1.1.1 Type of Graph

One particular type of graph that is studied in this problem is the hypercube, which is motivated by the practical applications. The hypercube is often used as the topology of a parallel computer due to many nice properties that it possesses. For example, the maximum distance between two vertices is fairly small as is the degree of each vertex, both being equal to the dimension of the cube. Another topology that can be considered is the two-dimensional grid, which has a constant maximum degree of four. This type of graph is also useful when considering VLSI routing problems.

1.1.2 Directed vs. Undirected Graphs

We can consider both directed and undirected graphs. Note that we can easily transform any undirected graph into a similar directed one, by simply replacing each undirected edge with two oppositely-oriented directed edges. In particular, this is what will be done when considering the directed hypercube.

1.1.3 Specifying Request Pairs

We may also specify more precisely what kinds of request pairs are allowed. A partial permutation is a set of request pairs where a vertex appears at most once as a source and at most once as a target. When each vertex appears exactly once as source and once as target then we have a permutation. Note that for a permutation there must be exactly $p = |V|$ request pairs. More generally, when each vertex appears at most h times as source and at most k times as target, we refer to the request pairs as an h - k request. In addition we can constrain the request pairs to be within some maximum distance in the graph.

1.1.4 Types of Paths Allowed

One final possibility is to only allow certain types of paths to connect the request pairs. We may define the length of a path as the number of edges that it contains, and we consider the set of all paths connecting a given request

pair. In this set we can find the smallest length of any path, say m . Then in the solution to the problem we may specify that the path connecting this request pair must have length m . By doing this for all the request pairs we can enforce the minimality of all the paths. This constraint can be relaxed so that each path is allowed to be within some constant factor of minimal. More specifically, for some constant c a given path will be allowed to be of length cm or less.

1.2 Known Results

Szymanski conjectured that there is always a routing for any permutation on a directed hypercube, where each path is of minimal length. This was shown to be false in the 4-dimensional hypercube when a counterexample was found by Lubiw (1990). However, the weaker conjecture, being that any permutation can be routed in a hypercube where the path lengths are not constrained, remains open. It was shown by Gu and Tamaki (1997) that permutations can always be routed if each directed edge is duplicated. More specifically, if we allowed each edge to be used by up to two paths, instead of just one, then we could always route permutations.

Due to the practical origins of these problems, it is important to consider the speed of finding edge-disjoint paths, not just considering if they exist or not. Several NP-completeness proofs have been established to this end. On the directed hypercube the problem of finding minimal edge-disjoint paths is NP-complete, even when all the pairs are constrained to be at most distance 4 from each other. The undirected case is also NP-complete, even when the pairs are made to be at most distance 3 from each other Gonzalez and Serena (2002). In both the undirected and directed cases if we relax the minimal path constraint the problems remain open. However, if the distance between each pair of vertices is at most 2, then the problem can be solved in polynomial time if every path is forced to be minimal. If the paths are not forced to be minimal, then even this problem is open.

Chapter 2

NP-Completeness

Before we can discuss proving the complexity of the problems we are concerned with, we must first introduce the concept of NP-completeness. In this chapter we will give a brief explanation of several complexity classes of problems, leading up to NP-completeness. Readers interested in learning more may look at Garey and Johnson (1990) or Sipser (1996).

2.1 The Class P

There are many ways in which one can categorize the difficulty of a problem. One natural approach is to classify a problem by how much time it takes to solve. Of course, we must also take into account the size of the problem instance. Take for example the problem PATH, in which we are given a directed graph as well as two vertices in that graph, s and t . The objective is to decide whether or not there is a path from s to t . Naturally when determining how much time it takes to solve, we must consider that a graph with three vertices will take much less time to solve than a graph with 100 vertices.

To this end, we treat the amount of time it takes to solve a problem as a function of the input size. This notion is a little ambiguous as different computers have different processing speeds, which is why computer scientists use the concept of a *Turing machine*. A Turing machine is a very simplistic computer that operates by reading and writing symbols off of a single tape, that nevertheless is capable of performing any computable task. Time is more precisely defined as the number of steps a Turing machine will take to solve a given problem instance. However, regarding the amount of time an algorithm takes as the number of steps it has ends up being equivalent

to this definition. The class P is simply all problems that have algorithms which will solve them, such that the running time is bounded by some polynomial of the input size.

Let us go back to our example of the PATH problem. We can solve this with a breadth-first search algorithm that works as follows. First we mark the vertex s . Now on each subsequent step we examine all edges leaving marked vertices, and if any of them connect to unmarked vertices we will mark them. We keep doing this until we can no longer mark any additional vertices. Once we stop, we simply check to see whether or not t is marked. The number of times this algorithm scans the edges is bounded by the number of vertices, since each time one vertex is marked at least. The time it takes to scan the edges is bounded by the number of edges, so this problem takes a polynomial amount of time to solve. Thus $\text{PATH} \in P$. The reason that the class P is important is because the problems that it contains approximately corresponds with the problems that can be efficiently solved on a computer.

2.2 The Class NP

There are many interesting problems for which polynomial-time algorithms have not been discovered. This could be because the solution is still waiting to be found, or it could be because these problems just are that difficult. The class NP contains a multitude of these interesting problems. The name stands for nondeterministic polynomial time, which is because all problems in this class can be solved by a nondeterministic Turing machine in polynomial time. The concept of nondeterminism is subtle, and takes a fair amount of space to adequately explain. For the sake of conciseness, we will instead use a simpler, equivalent definition. The problems in NP are those which can be verified in polynomial time. That is, given a solution “certificate” there is an algorithm that can check its validity in polynomial time. Note that here we are still measuring polynomial time with respect to the input size, not the certificate size.

As all the problems in P can be solved in polynomial time, they can clearly be verified as well, so we know $P \subseteq NP$. However, it is not known whether or not $P = NP$, and this is actually one of the most important open questions in computer science to this day. An example of a problem in NP is 3SAT. For this problem we are given a boolean formula in conjunctive normal form where each clause has exactly three literals. Recall that a literal is just a boolean variable or its negation. A clause is a group of literals

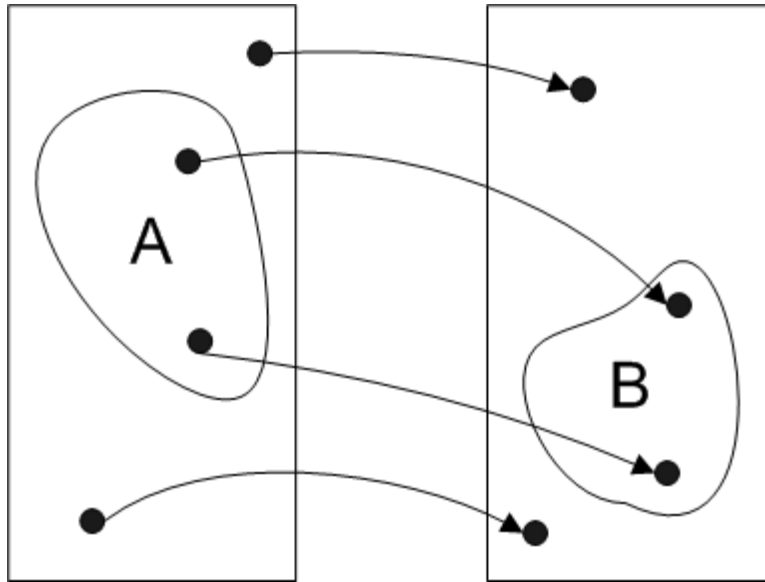


Figure 2.1: Reducing problem A to problem B

connected with ORs. Conjunctive normal form refers to a group of clauses connected with ANDs. The objective of this problem is simply to see if there is a truth assignment to the variables such that the formula is satisfied. We could simply have a solution certificate that gives a truth assignment to the variables. Then in polynomial time we can easily check to see that each clause has been satisfied. Thus $3SAT \in NP$.

2.3 NP-completeness

We are nearly ready to define what it means for a problem to be NP-complete, but before we do we must discuss the notion of reducibility. Suppose we have two decision problems A and B , which are both contained in NP. By decision problem, we simply mean that the answer to every problem instance is either “yes” or “no”. Now, we can reduce problem A to problem B if we can transform every problem instance of A to a problem instance of B , such that the transformation remains faithful. That is to say the answer to the instance of A will match the transformed instance of B , as in Figure 2.1. Moreover, we would like to be able to reduce problems in a polynomial amount of time. Notice that if we can do this, then we have

shown that problem B is at least as hard as problem A . This is because if we obtain a polynomial-time solution to B , then we can also solve A in polynomial-time by first reducing to B and then solving.

A problem is NP-complete if it satisfies two conditions. First, it must be contained in NP. Secondly, every single problem in NP must be polynomial time reducible to it. While at first it may seem impossible to prove that any problem satisfies this definition, a breakthrough was made when 3SAT was proven to be NP-complete by Cook (1971). After this thousands of problems were found to be NP-complete every year. Notice that once we have a problem that is NP-complete, it becomes much simpler to show a given problem, X , is also NP-complete. Now we just need to show a polynomial-time reduction from an NP-complete problem to X . Once we do this, we can reduce every problem in NP to X by first reducing it to the NP-complete problem, which we know we can do. Once we do this, we just use the reduction we found between that problem in X , and we're done. Thus Cook's proof laid the foundation work for all future NP-completeness results.

Observe now that if we could find a polynomial-time solution to even one NP-complete problem, then that would show that every problem in NP has a polynomial-time solution. Thus showing a given problem is NP-complete is nearly equivalent to showing that there is no fast way to solve it. By relating it to an entire class of problems, we can demonstrate that this problem is at least as hard as a great number of problems that very smart people have been unable to solve for centuries.

Chapter 3

The Complexity of Hypercube Routing

In this chapter, we will demonstrate that the problem of edge-disjoint permutation routing on a hypercube where all paths are of minimal length is NP-complete. To do this, we will show a reduction from L3-SAT, a variant of 3SAT, which was defined in Chapter 2. In addition, we will also show how to generalize this reduction so that it will work for the problem where the paths can be of arbitrary length. One drawback to this generalization is that it no longer allows us to restrict our routing requests to be permutations.

3.1 L3-SAT

We begin by defining L3-SAT. For a given instance of L3-SAT, we are given a boolean formula in conjunctive normal form, just as in 3SAT. Here however, each clause may consist of either two or three literals. Furthermore, each literal can only appear twice in the entire expression. Note, this means that each variable can appear up to four times: twice in its original form, and twice in its negated form.

We will now quickly demonstrate that L3-SAT is equivalent to 3SAT, namely that it is also NP-complete. First, it is clearly in NP, since we can request a solution certificate that is simply the valuation of the boolean variables, and check to see that it satisfies all of the clauses in polynomial time.

Now, for a given instance of 3SAT, we check to see if there are any literals that appear more than twice in the expression. If there are, we pick one,

and call the corresponding variable u . We then take the maximum of the number of times that u appears, and the number of times that \bar{u} appears, and call that ℓ , where $\ell \geq 3$. Next, we introduce ℓ new variables, which we will denote a_1, a_2, \dots, a_ℓ and add in the ℓ additional clauses

$$\{a_1, \bar{a}_2\}, \{a_2, \bar{a}_3\}, \dots, \{a_{\ell-1}, \bar{a}_\ell\}, \{a_\ell, \bar{a}_1\}.$$

Notice here that if a_1 is set to true, then so must all of the other new variables in order to satisfy the clauses above. Similarly, if a_1 is set to false, then all of the a_i must be set to false. Also observe that each literal in the above set of clauses appears exactly once. Now we can replace each occurrence of the variable u with one of the a_i . The first time that u appears we replace it with a_1 , or with \bar{a}_1 if it is \bar{u} . In the same way we replace all the other occurrences of u with the variables a_2 through a_ℓ .

After repeating this process for all literals that appear in three or more clauses, we end up with an instance of L3-SAT. This instance is satisfiable if and only if the original 3SAT instance was satisfiable, since we only replace a variable with a set of variables that all must evaluate to the same value. Therefore L3-SAT is NP-complete.

3.2 Reduction to Hypercube

Having established the NP-completeness of L3-SAT, we are now ready to show the reduction found by Gonzalez and Serena (2002) for the problem of permutation routing in the hypercube with minimal-length paths. This reduction uses three principle constructions, or gadgets. These are the *variable-setter*, *clause-checker* and *convey apparatus*. Each of these gadgets will be discussed independently, and then we will show how they are connected and embedded within the hypercube.

3.2.1 Variable-Setter

For each variable in the L3-SAT instance, we will create a variable-setter gadget as shown in Figure 3.1. We see that the gadget simply consists of a request pair, (s, t) that has distance two. Notice that there are only two possible shortest paths for this request pair. On each possible shortest path from s to t , we will associate the two edges with one valuation of the variable. Thus, if one path is chosen, we can consider the variable set to true, and if the other is chosen we will consider it set to false.

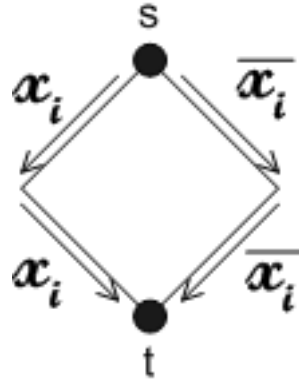


Figure 3.1: A variable-setting gadget with only two possible shortest paths from s to t .

3.2.2 Clause-Checker

The clause checker is as shown in Figure 3.2, and consists solely of one request pair of either distance two or three, depending on whether there are two or three literals in the clause. Each of the edges, (b_i, t) , where $i \in \{1, 2, 3\}$, will be associated with one of the literals in that clause. For the literal corresponding to (b_i, t) , we will find the edge in the variable-setting gadget that corresponds to the negation of that literal. This edge will then be linked to (b_i, t) by the convey apparatus, which will be discussed in the following section. The convey apparatus simply ensures that if the edge in the variable-setting gadget is used, which means that this literal will not satisfy the clause, then the edge (b_i, t) will also be used. This means that to connect s to t in the clause checker gadget, we will no longer be able to use the (b_i, t) edge. Notice then that if all the (b_i, t) edges are used up, there is no way to route s to t anymore using a shortest-length path. Thus the clause checker is only routable if at least one literal in the clause is satisfied. Observe that the two edges per literal in the variable-setting gadget suffice, since we are working with L3-SAT, which does not permit any literal to appear more than twice.

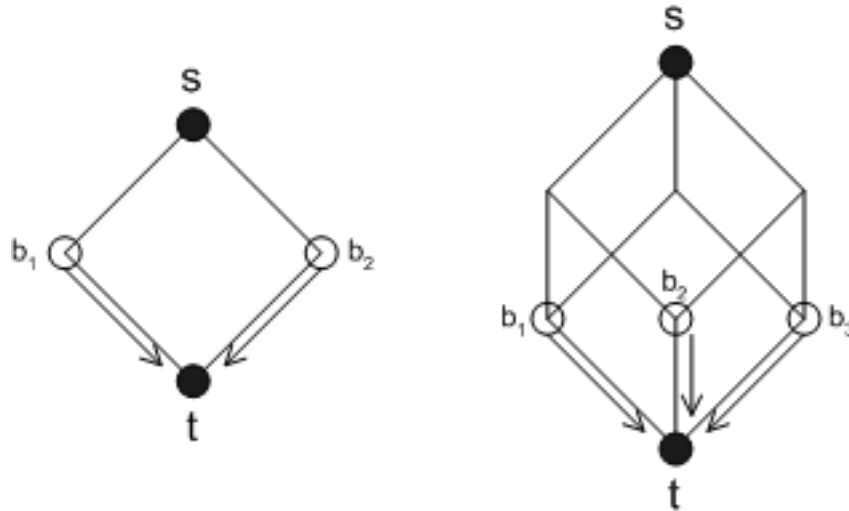


Figure 3.2: The clause-checker gadget consisting only of the request pair (s, t) .

3.3 Convey Apparatus

The convey apparatus consists simply of a chain of requests that each have distance two. These are designed to link an edge A to an edge B , in such a way as to force B to be used if A is used. The basic construction can be seen in Figure 3.3. As can be seen in the figure, there are two possible shortest paths from s_1 to t_1 . However, if A is already being used, then this path must go through t_2 instead. Likewise the path from s_2 to t_2 must go through t_3 . This continues even through the bends until eventually the path from s_k to t_k is forced to use edge B .

3.4 Embedding

The only thing that remains to be done is to somehow embed all of these gadgets in the hypercube. In order to make this simpler, we will be referring to the binary representation of the hypercube. That is, we will think of each vertex in the n -dimensional hypercube as a binary number of length n . The vertex set will simply be all possibly binary numbers of length n . Finally, two vertices will have two oppositely-oriented directed edges be-

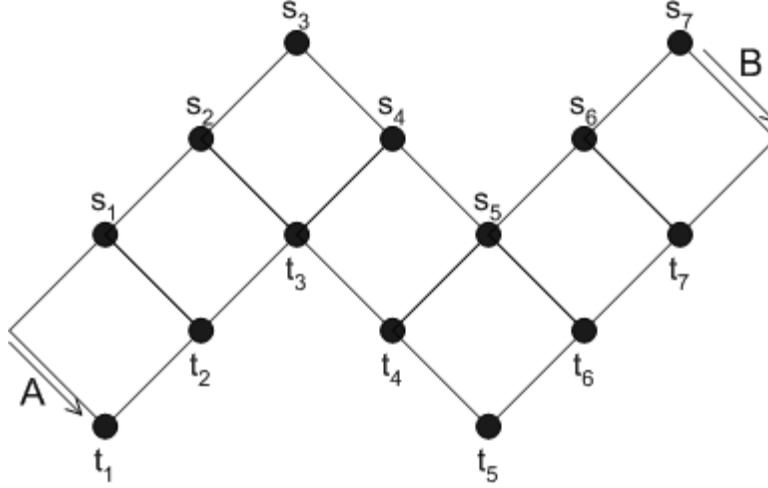


Figure 3.3: The basic convey apparatus construction. If edge A is being used, then the series of requests will force edge B to be used as well.

tween them if and only if their two binary representations only differ in one place.

For an L3-SAT instance with v variables and k clauses, we will work in a hypercube of dimension $\lceil \log_2(v+1) \rceil + \lceil \log_2(k+1) \rceil + 12$. Notice that the number of vertices in this hypercube is of order $O(vk)$ so this remains a polynomial transformation. Now we will label the bits of the binary representations for the vertices as follows

$$D_0 D_1 \dots D_{\lceil \log_2(v+1) \rceil} E_0 E_1 \dots E_{\lceil \log_2(k+1) \rceil} FG \alpha_0 \alpha_1 \alpha_2 \alpha_3 IJK \beta_0 \beta_1 \beta_2$$

The bits $D_0 D_1 \dots D_{\lceil \log_2(v+1) \rceil}$ will represent the binary value of the variable index, counting from one instead of zero. The bits FG will be used to indicate the position within a variable gadget. The α_i bits will be used to indicate which edge in the variable-setting gadget we are linking. Similarly, the bits $E_0 E_1 \dots E_{\lceil \log_2(k+1) \rceil}$ will denote the binary value of the clause index, counting from one. The bits IJK are used to represent the position within the clause-checker gadget, which may be either two- or three-dimensional. Finally, the β_i bits will be used to indicate which edge in the clause-checker we are linking to.

The main consideration in the construction of the convey apparatus is ensuring that no two convey apparatuses intersect. To accomplish this, we

ensure that every vertex in the apparatus contains a unique identification of either the edge in the variable-setting gadget being linked, or the edge in the clause-checker gadget being linked. Since neither of these two edges will be linked by any other convey apparatus, this will suffice. To see how this works, we will show how an edge in the variable-setting gadget for the i th variable will be connected to an edge in the clause-checker gadget for the j th clause.

An illustration of how the convey apparatus links the variable-setter to the clause-checker can be seen in Figure 3.4. Note that by labelling the

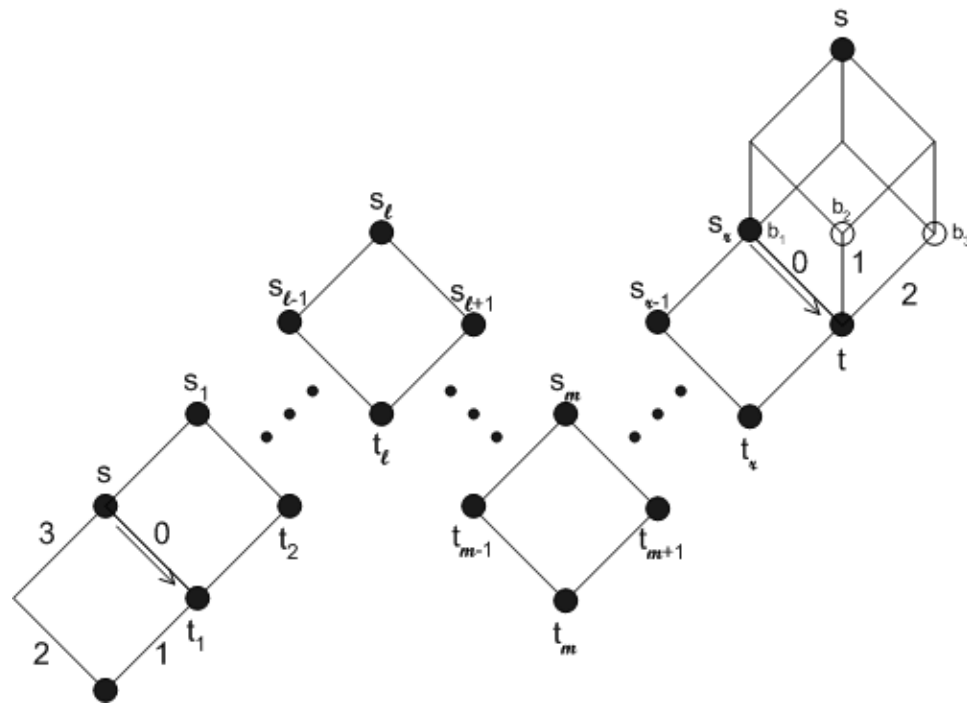


Figure 3.4: The convey apparatus linking edge 0 in the i th variable-setting gadget to edge 0 in the j th clause-checker gadget.

edges in the variable-setter, we can use the α_i bits to indicate what edge we are linking by simply changing α_i to a one, where i is the number of the edge in question. Thus our path begins with the following binary representations of vertices. Note that by $bitrep(i)$ we are referring to the binary representation of the integer i , and by $bitrep(v)$ we are referring to the bi-

nary representation of vertex v within its respective gadget. Finally, by (B) , we simply mean that the bit B is a one, and any bits that are not explicitly mentioned are assumed to be set to zero. Using this notation we have

$$\begin{aligned} s &= \text{bitrep}(i) \dots \text{bitrep}(s) \dots \\ t_1 &= \text{bitrep}(i) \dots \text{bitrep}(t_1) \dots \\ s_1 &= \text{bitrep}(i) \dots \text{bitrep}(s)(\alpha_0) \dots \\ t_2 &= \text{bitrep}(i) \dots \text{bitrep}(t_1)(\alpha_0) \dots \end{aligned}$$

As this beginning allows for unique identification of the edge we are linking in the variable-setter, we may now begin adding ones as needed until the vertices also uniquely identify the edge in the clause-checker. This happens when we reach the vertices

$$\begin{aligned} s_{\ell-1} &= \text{bitrep}(i)\text{bitrep}(j)\text{bitrep}(s)(\alpha_0) \dots \text{bitrep}(b_1)(\beta_0) \dots \\ t_{\ell} &= \text{bitrep}(i)\text{bitrep}(j)\text{bitrep}(t_1)(\alpha_0) \dots \text{bitrep}(b_1)(\beta_0) \dots \\ s_{\ell} &= \text{bitrep}(i)\text{bitrep}(j)\text{bitrep}(s) \dots \text{bitrep}(b_1)(\beta_0) \dots \\ s_{\ell+1} &= \text{bitrep}(i)\text{bitrep}(j)\text{bitrep}(t_1) \dots \text{bitrep}(b_1)(\beta_0) \dots \end{aligned}$$

Note that we are now able to remove the α_0 bit because the vertices are now referencing the edge in the clause-checker. Now we would like to begin changing the bits we no longer want to zero, but there is a slight complication. We know $\text{bitrep}(s)$ and $\text{bitrep}(t_1)$ are going to have different numbers of ones, so we cannot reduce them both to zero at the same time. However, by changing the direction of the convey apparatus we can effectively change the discrepancy from the representations of the vertices in the variable-setter to the representations of the vertices in the clause-checker. Thus we can arrive at the vertices

$$\begin{aligned} t_{m-1} &= \text{bitrep}(i)\text{bitrep}(j) \dots (\alpha_0) \dots \text{bitrep}(b_1)(\beta_0) \dots \\ s_m &= \text{bitrep}(i)\text{bitrep}(j) \dots \text{bitrep}(b_1)(\beta_0) \dots \\ t_m &= \text{bitrep}(i)\text{bitrep}(j) \dots (\alpha_0) \dots \text{bitrep}(t)(\beta_0) \dots \\ t_{m+1} &= \text{bitrep}(i)\text{bitrep}(j) \dots \text{bitrep}(t)(\beta_0) \dots \end{aligned}$$

Now we can change direction once again, and all that remains is to remove all the remaining ones that are not part of the binary representation of b_1 or t . The only consideration we must make is to remove the β_0 bit last, as we need this to uniquely identify the edge in the clause-checker. This brings

us to the end with the vertices

$$\begin{aligned} s_{r-1} &= \text{bitrep}(j) \dots \text{bitrep}(b_1)(\beta_0) \dots \\ t_r &= \text{bitrep}(j) \dots \text{bitrep}(t)(\beta_0) \dots \\ b_1 = s_r &= \text{bitrep}(j) \dots \text{bitrep}(b_1) \dots \end{aligned}$$

Thus we have linked edge (s, t_1) to edge (b_1, t) in the manner desired, and every vertex in the convey apparatus identifies at least one of these edges, and so is guaranteed to be unique.

3.5 Extension of Reduction

This reduction can be extended to the problem that allows paths to be of arbitrary length quite simply. The main observation to be made is that for a routing request, (s, t) , of distance one, we can assume that the routing satisfies this request using the single edge connecting source to sink. If there is a routing where this is not the case, then we can show how we can transform it into a routing in which it is. If the edge connecting the source to the target is not being used at all, then we can simply remove whatever path is currently connecting (s, t) and replace it with that edge. The only remaining possibility is that some other path is making use of this edge, which is forcing the path p connecting (s, t) to go around some other way. In this case we may simply swap the paths, as is demonstrated in Figure 3.5. Whatever path was previously using the edge, (s, t) , will now

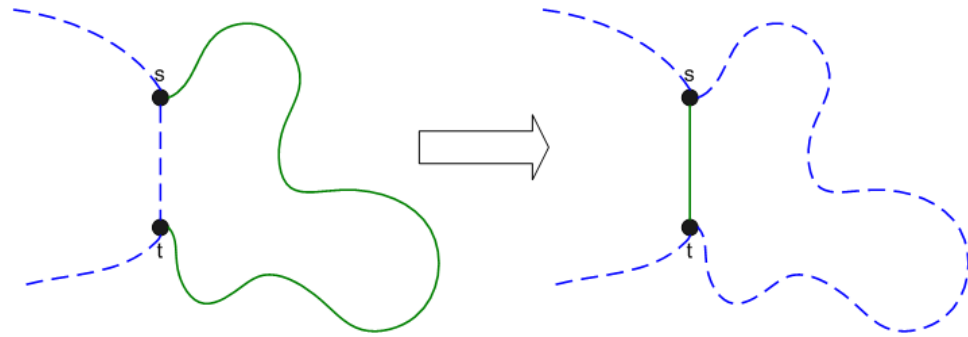


Figure 3.5: Paths may be swapped so that routing requests of distance one are satisfied by paths of length one.

when reaching s , follow the path p to get to t , and then continue on as

before. This frees up the edge, (s, t) , so that it may be used to satisfy the unit distance routing request.

This idea allows one to effectively block edges by making routing requests over the edges that are desired to be blocked. So to reduce L3-SAT to routing on the hypercube such that arbitrary length paths are allowed, we can use the same reduction as above. Now we simply block the necessary edges such that all of the paths are forced to be shortest length. The downside to this approach is that it requires us to make vertices be the sources and targets of more than one request, and so we no longer retain the permutation constraint.

Chapter 4

The Complexity of Grid Routing

As was mentioned in Chapter 1, another important topology to consider for these routing problems is the two-dimensional grid. The grid can be thought of simply as a rectangular lattice over the cartesian coordinate plane. This graph has the nice property of having a constant maximum degree, namely four, while maintaining a fairly small diameter.

We will show that the edge-disjoint routing problem with shortest paths is NP-complete on the grid using the reduction from Gonzalez and Serena (2002). It is also true that the problem when relaxed to arbitrary-length paths is also NP-complete, as can be seen from Kramer and van Leeuwen (1984), though a few slight modifications to their logic are needed.

4.1 The Graph

First we need a more formal definition of the grid, in order to describe the reduction more easily. In general, a grid with dimensions n_x by n_y is a graph, $G = (V, E)$ where each vertex is labelled by a pair of integers, and $(x, y) \in V$ if $0 \leq x \leq n_x$ and $0 \leq y \leq n_y$. The grid is undirected, and there is an edge $\{(x_1, y_1), (x_2, y_2)\} \in E$ provided $|x_1 - x_2| + |y_1 - y_2| = 1$.

4.2 The Reduction

We will be reducing from SAT, which was described in Chapter 2. Given an instance of SAT with v variables, x_1, x_2, \dots, x_v and k clauses, we will create

a n_x by n_y grid, where $n_x = 4v$ and $n_y = 3k + 1$. For each variable, x_i , we will create a request pair from vertex $(4i - 3, 0)$ to vertex $(4i - 1, n_y - 1)$. For the j th clause, where $1 \leq j \leq k$, we will create two request pairs. One from vertex $(0, 3j - 2)$ to vertex $(v_x - 1, 3j)$ and another from vertex $(0, 3j)$ to vertex $(v_x - 1, 3j - 2)$.

In addition to these requests we will also block out certain edges within the grid. We can do so by adding adjacent pair requests, just as we did in Chapter 3, Section 3.5. Note that now our graph is undirected, but the same logic still applies.

The main idea in this reduction is that the paths connecting the two request pairs for each clause need to cross once. Edges will be blocked so that the only place where these paths may cross will correspond to a variable that satisfies the clause. This is accomplished by blocking sufficient edges such that there are only two possible shortest paths for each variable pair request. Either it moves all the way down the graph and then over, which will correspond to setting it false, or it moves to the right and all the way down, which will correspond to setting it true. Thus the actual edges that are blocked will depend on how each variable appears in each clause, and there are three possibilities. Either the variable appears uncomplemented (if it is set to true the clause will be satisfied), complemented (if it is set to false the clause will be satisfied), or else it does not appear at all (the valuation of this variable will not affect the satisfiability of the clause). An example showing both the request pairs as well as the blocked edges can be seen in Figure 4.1.

In Figure 4.2, a specific grid routing instance is created for the L3-SAT instance $\{x_1, x_2\}, \{x_1, \bar{x}_2\}, \{\bar{x}_1, x_2\}$, and $\{\bar{x}_1, \bar{x}_2\}$. As this instance of L3-SAT is not actually satisfiable, there should be no routing possible if this reduction is faithful. Indeed, no matter how the variable paths are connected, there remains at least one clause whose pairs are unable to cross.

By this construction, the paths for all of the variable pair requests determine the valuation of the variables. If a variable is set to satisfy a particular clause, then the two pair requests for that clause may cross at that point, and thus be satisfied. Therefore if the original SAT instance was satisfiable, then the grid routing instance will be satisfiable. This is because we can set all the variable pair request paths to correspond to a satisfying valuation, and then each clause may have its two pair request cross where one of its literals is satisfied.

Alternatively, if the grid routing instance is satisfiable, then we can use the valuation of the variables corresponding to the variable pair request paths. Every clause must have the paths for its two pair requests cross at

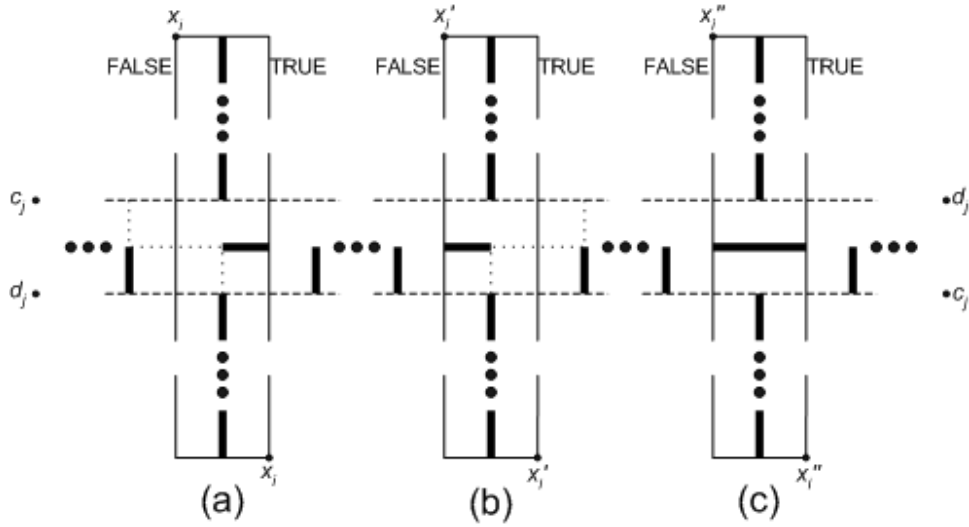


Figure 4.1: The blocked edges and pair requests in the grid routing instance, where in the SAT instance: (a) variable x_i appears uncomplemented in the j th clause; (b) variable x_i' appears complemented in the j th clause; and (c) variable x_i'' does not appear at all in the j th clause.

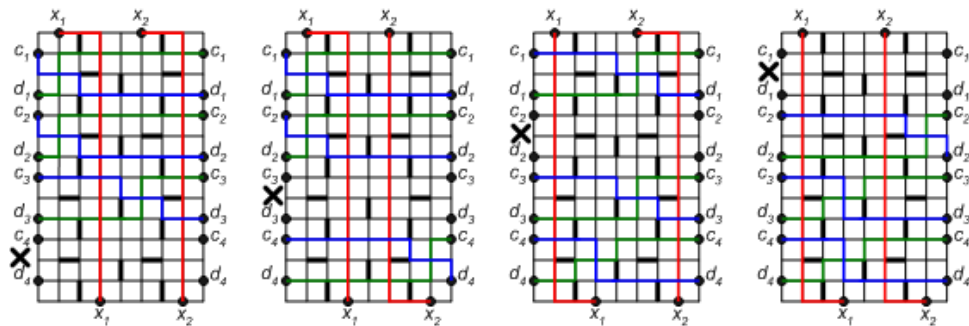


Figure 4.2: Grid routing instance created from $\{x_1, x_2\}$, $\{x_1, \bar{x}_2\}$, $\{\bar{x}_1, x_2\}$, and $\{\bar{x}_1, \bar{x}_2\}$. All four possible truth assignments to x_1 and x_2 are shown, but since this instance is not satisfiable in each case there is one clause that cannot be routed, which is indicated by a 'X'.

some point, which implies that each clause has at least one literal satisfied. Therefore this valuation will be satisfying for the original SAT instance.

Now we verify that shortest-path permutation routing in the grid is NP-complete. This is clearly true, since given a solution we can in polynomial time verify that the paths connect every source to its target, and that no edge is used twice. Thus, this problem is NP-complete.

Chapter 5

Approximation Algorithms

In this chapter we will present several approximation algorithms for the edge-disjoint permutation routing problem on the directed hypercube. The algorithms are for the unrestricted path length version of the problem, which has not yet been shown to be NP-complete. This means that it is still possible that there does exist a polynomial-time algorithm which will solve this problem exactly, but here we will show how to obtain solutions that are close to optimal in polynomial-time. The results presented below are primarily the work of Choi and Somani (1993) and Gu and Tamaki (1997).

It is useful in these sections to refer to the binary representation of the hypercube, which will be explained in the following section. In addition, we will demonstrate how routing in the hypercube can be seen as switching messages in what is known as a Multistage Interconnection Network. Using this model, the first algorithm presented will show that we can do twice as much if we have twice as many resources. That is, if we double every directed edge, then we can route two permutations, which is simply a 2-2 routing request.

5.1 Binary Representation

Each vertex will correspond to a binary number, and the vertex set of the n -dimensional hypercube will simply consist of all 2^n n -bit binary numbers. Vertices will be connected if their binary representations only differ in one bit. Edges connecting vertices differing in the k th bit will be referred to as k -dimensional edges. Also, the 0-subcube will be used to refer to all vertices with a 0 as the first bit, as well as the edges connecting them. We similarly define the 1-subcube. Note that both the 0-subcube and the 1-subcube are

isomorphic to the $(n - 1)$ -dimensional hypercube. Finally, for a vertex v , the mapping $\pi_i(v)$ is the vertex that is identical to v , except the first bit is now i , where $i \in \{0, 1\}$. Note that if the first bit of v is i , then $\pi_i(v) = v$.

5.2 Multistage Interconnection Network

We now show how we can view routing in the hypercube as passing messages in a dynamic network. Here we will view each request pair as beginning at the source node, and then being transferred to other nodes over a series of “stages”. At each stage, each pair request will have the option of staying at the same node or being transferred to one other specific node. This dynamic network is known as a Multistage Interconnection Network, or MIN. An example of the MIN constructed from the three-dimensional hypercube is shown in Figure 5.1. Note that each stage consists of all of the vertices, and there are just two edges leading out of every node.

In Choi and Somani (1993), the following transformation is given from the hypercube to the MIN. For an n -dimensional hypercube, we simply create $2n + 1$ copies of the vertex set, V_0, V_1, \dots, V_{2n} . We will have V_0 and V_1 comprise a directed bipartite graph, where all of the edges leading from V_0 are the 1-dimensional edges. Similarly, vertex sets V_{i-1} and V_i will comprise a directed bipartite graph where all of the edges leading from V_{i-1} are the i -dimensional edges, where $1 \leq i \leq n$. For the remaining vertex sets, we will also treat V_j and V_{j+1} as a directed bipartite graph, where we use every $(2n - j)$ -dimensional edges, where $n \leq j \leq 2n - 1$. The entire graph will be symmetric about V_n . Finally, we add an edge from every vertex in V_k to its corresponding copy in V_{k+1} , where $0 \leq k \leq 2n - 1$.

The MINs that are created from hypercubes in this fashion are part of a special class of MINs known as Beneš networks. The advantage to converting a hypercube into a MIN, is that MINs have been studied extensively and we can apply the known results. Notice that in our construction, we actually use every edge of the hypercube exactly twice. Once in the first half, and once in the second half. Note also that routings in the MIN directly correspond to routings in the hypercube. We may simply use the same edges, and just ignore any of the edges used that connect a vertex to itself.

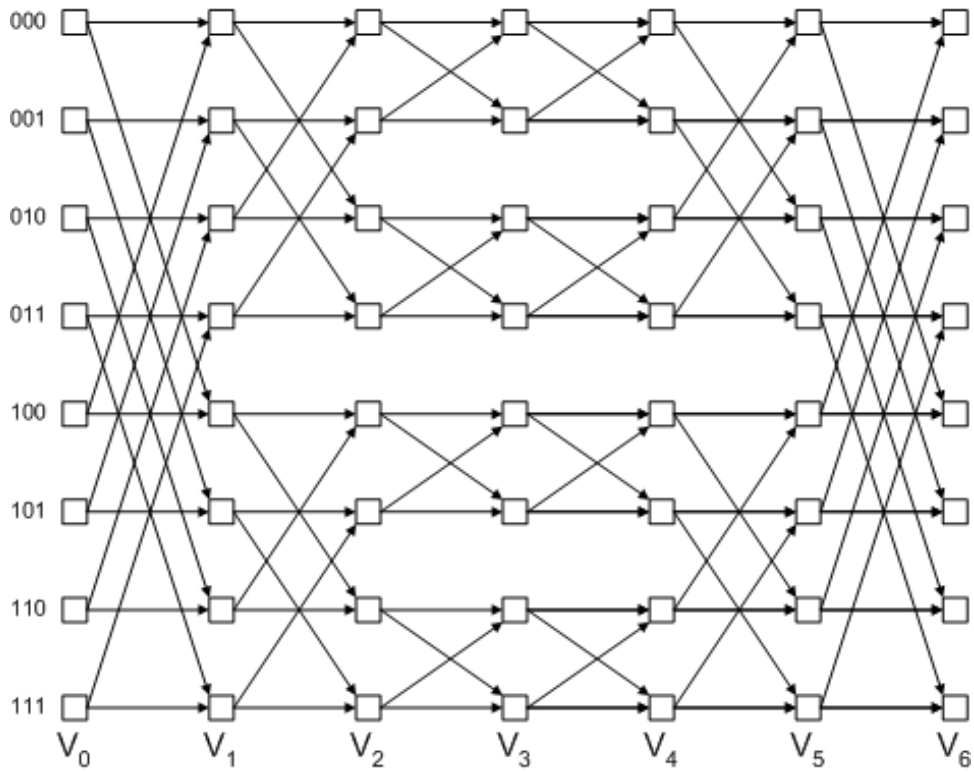


Figure 5.1: The MIN induced from the three-dimensional hypercube.

5.3 Routing Two Permutations

An algorithm to route two permutations given twice as many edges on the directed hypercube (or Beneš network) is obtained from Beneš (1965). We will explain how this routing algorithm works here in an inductive fashion. First, suppose we have a 0-dimensional hypercube, which is simply one point. The only possible request we can have is for two requests from that node to that node. These can be satisfied quite easily with two paths of zero length.

Now, suppose that we are able to route any two permutations on an n -dimensional hypercube. We will now demonstrate that we can in fact route any two permutations on an $(n+1)$ -dimensional hypercube. The main idea behind this algorithm relies on splitting the request into two separate permutations. Note that we can do this even for a general 2-2 routing request. This can be seen by representing the request as a bipartite graph, where both the left and the right node sets are copies of the vertex set of the hypercube. For each source-target pair, we create an edge from the source on the left to the target on the right. We can find one permutation simply by finding a perfect matching on this bipartite graph, which can be done using a network flow algorithm. After removing this permutation, what is left must be another permutation. So we have demonstrated that a 2-2 routing request really is equivalent to two permutations.

Now, we can take one permutation, and map it down to the 0-subcube as follows. For each source-target pair, (s, t) , we simply map it to the source-target pair, $(\pi_0(s), \pi_0(t))$. Note that since every vertex was used as a source once and as a target once, every vertex in the 0-subcube is now used as a source twice and a vertex twice. This means we now have a 2-2 routing request in the 0-subcube, which is a n -dimensional hypercube, so by the induction hypothesis we know how to route it. Thus, the path for a given source-target pair, (s, t) , within the permutation is found as follows. If $s \neq \pi_0(s)$, then we begin with the edge from s to $\pi_0(s)$. We then use the edge-disjoint path that we know must exist between $\pi_0(s)$ and $\pi_0(t)$. Finally, if $\pi_0(t) \neq t$ then we finish with the edge from $\pi_0(t)$ to t . In this manner we route the first permutation, and we've used all of our 1-dimensional edges exactly once. Next, we route the other permutation in the same manner, except we map it down to the 1-subcube. By following the same procedure, we know use each 1-dimensional edge one more time, and so each edge is used at most twice. We know that all of the other dimensional edges are used no more than twice by the inductive hypothesis, so we have now found a routing algorithm for the $(n+1)$ -dimensional

hypercube.

5.4 Routing One Permutation

To route one permutation, we may simply follow the same procedure as described in the previous section. Note that this will simply map the permutation into the 0-subcube using the 1-dimensional edges, while all of the edges in the 1-subcube remain unused. Once we obtain this routing, we can make a modification discovered by Choi and Somani (1993), that will take advantage of the unused edges in order to reduce the number of edges we must double. To do this we notice when the paths reach the V_n stage each vertex has two requests, and no edge has been used more than once. We will transfer these requests to the portion of the V_n stage in the 1-subcube using a “bridge” of the 1-dimensional edges. Now we will complete the routing just as we did in the 0-dimensional subcube, except now we will not be reusing the same edges, because none of the edges within the 1-subcube have been used yet. See Figure 5.2 for an illustration of which edges will be used in this routing scheme. Using this routing method, we do not have to double any of the edges, except for the 1-dimensional ones, which are each used exactly twice, at the beginning and end of the paths, as well as at the bridge.

So we now have shown that by simply doubling the edges in a single dimension, we may route an arbitrary permutation. While this result is great, it does not fit into the context of the motivation for this problem. That is, within the parallel computing framework, we cannot simply add edges into our network when we want to perform a routing. So we could either change the original network structure, or we could break up our routing request. That is, we could partition our routing request into several pieces such that each piece is routable on the directed hypercube. In this way routing is performed in several rounds; if messages cannot all be passed at the same time, we will at least pass them within a small number of communication rounds.

We will now show that the algorithm described above for routing a permutation by doubling the 1-dimensional edges transforms readily into a 4-approximation algorithm. That is, we can divide the permutation into four pieces such that each piece is routable by that algorithm using each edge of the directed hypercube once. To begin with, we can partition the permutation routing request into those request with sources in the 0-subcube which we will denote, R_0 , and those with sources in the 1-subcube, R_1 . Note that

by keeping R_0 in the 0-subcube, we avoid having to use any 1-dimensional edges at the start of the path. The problem though is that when we get to the bridge, there may be two requests at a node, so we will simply divide up R_0 so that we do not need to double up any edges on the bridge. We can do the same for R_1 by keeping it the 1-subcube and using a bridge to bring it into the 0-subcube. Since both R_0 and R_1 need to be divided into at most two pieces each, we have a 4-approximation.

5.5 A 2-Approximation Algorithm

As is shown by Gu and Tamaki (1997), we can actually achieve a 2-approximation using only a slight modification on Choi and Somani's algorithm. We will again split up the permutation into R_0 and R_1 , but this time we will be a little more careful in routing so that we do not have to subdivide them. The reason we had to subdivide those request groups before is that we could end up with more than one request per vertex at the V_n stage, which meant that our bridge would require doubled edges. Now we will show how we can avoid placing more than one request on a vertex.

Observe that when we map R_0 down to the 0-subcube, we end up with each vertex serving as only one source. It is possible though that a vertex will end up serving as more than one target, since there could be two targets connected by a 1-dimensional edge. This means that we are dealing with a 1-2 routing request, whereas our routing algorithm was treating it like a 2-2 routing request. We were splitting this request into two 1-1 routing requests, and then mapping one to the 0-subcube and the other to the 1-subcube. The problem is that when we map a 1-1 request down to a subcube, it can become a 2-2 request. What we would like to do is split the 1-2 request into two pieces, such that when each piece is mapped down to a subcube, it remains a 1-2 request.

To determine how to split up the 1-2 request, we will observe what happens when we map the sources down to a subcube, but leave the targets alone. Since each vertex is at most one source, we will end up with a 2-2 routing request. Now we can split this up into two 1-1 routing requests, just as before, and this will be the split we will use for the 1-2 routing request. Note that now when we map each piece down to a subcube, each vertex must be at most one source, and again each vertex may serve as up to two targets. Thus we obtain another 1-2 routing request, and this process may be repeated. So now when we reach the V_n stage, each vertex will comprise only one request, so our bridge will require no doubled edges. In

this manner we can route each of R_0 and R_1 without doubling any edges in the directed hypercube. Therefore, we have a 2-approximation.

Chapter 6

Future Work

Although some complexity results have been established for routing on the hypercube, Szymanski's conjecture still remains open. Determining whether or not permutations are always routable in the hypercube when the connecting paths may be of arbitrary length is still a valid question. Furthermore, we have only shown the shortest-path version of the problem to be NP-complete. In order to relax the path length restriction, we also had to remove the permutation constraint. Is routing a permutation on the hypercube with arbitrary length paths NP-complete, or does there exist a polynomial time algorithm for it?

One direction in trying to establish an NP-completeness result for this problem is to modify the reduction given by Gonzalez and Serena (2002). Perhaps there is some way to modify it so that it may apply to arbitrary length paths and still deal with permutations. It is also possible that a simpler NP-completeness proof can be found by reducing from a different NP-complete problem. Theoretically, if a problem is NP-complete then one can reduce any problem in NP to it, but some reductions are much more natural than others.

The best approximation algorithm presented was a 2-approximation, meaning that we could split any permutation into two pieces, such that each piece is routable on the directed hypercube. Can we find any better approximation than this? Alternatively, we could find a special subclass of the permutations which we could prove is always routable. As Gu and Tamaki (1997) suggest, another approach would be to try and find some upper bound f on the size of a 1-1 routing request such that any 1-1 routing request with f or fewer request pairs is provably routable.

Bibliography

- Barden, B., Libeskind-Hadas, R., Davis, J., and Williams, W. (1999). On edge-disjoint spanning trees in hypercubes. *Information Processing Letters*, 70:13–16.
- Baudon, O., Fertin, G., and Havel, I. M. (2001). Routing permutations and 2-1 routing requests in the hypercube. *Discrete Applied Mathematics*, 113(1):43–58.
- Benes, V. E. (1965). *Mathematical Theory of Connecting Networks and Telephone Traffic*. Academic Press.
- Boppana and Raghavendra (1990). Optimal self-routing of linear-complement permutations in hypercubes. In *DISTMEMCC: 5th Distributed Memory Computing Conference*. IEEE Computer Society Press.
- Chekuri, C. and Khanna, S. (2003). Edge disjoint paths revisited.
- Choi, S. B. and Somani, A. K. (1993). Rearrangeable circuit-switched hypercube architectures for routing permutations. *J. of Parallel and Distributed Computing*, 19:125–130.
- Cook, S. A. (1971). The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA. ACM Press.
- Frank, A. (1990). Packing paths, circuits, and cuts – a survey. In *Paths, Flows and VLSI-Layouts*, pages 47–100.
- Frieze and Zhao (1999). Optimal construction of edge-disjoint paths in random regular graphs. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*.

- Garey, M. R. and Johnson, D. S. (1990). *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA.
- Gonzalez, T. F. and Serena, F. D. (2002). Complexity of k-pairwise disjoint shortest paths in the hypercube and grid networks.
- Gu, Q.-P. and Tamaki, H. (1997). Routing a permutation in the hypercube by two sets of edge disjoint paths. *Journal of Parallel and Distributed Computing*, 44(2):147–152.
- Guruswami, V., Khanna, S., Rajaraman, R., Shepherd, B., and Yannakakis, M. (1999). Near-optimal hardness results and approximation algorithms for edge-disjoint paths and related problems. pages 19–28.
- Hwang, F. K., Yao, Y. C., and Dasgupta, B. (2002). Some permutation routing algorithms for low-dimensional hypercubes. *Theoretical Computer Science*, 270(1–2):111–124.
- Kleinberg, J. and Tardos, É. (1995a). Approximations for the disjoint paths problem in high-diameter planar networks. pages 26–35.
- Kleinberg, J. M. and Tardos, E. (1995b). Disjoint paths in densely embedded graphs. In *IEEE Symposium on Foundations of Computer Science*, pages 52–61.
- Kolliopoulos, S. G. and Stein, C. (1998). Approximating disjoint-path problems using greedy algorithms and packing integer programs. *Lecture Notes in Computer Science*, 1412:153–??
- Kramer, M. R. and van Leeuwen, J. (1984). The complexity of wirerouting and finding minimum area layouts for arbitrary vlsi circuits. *Advances in Computing Research*, 2:129–146.
- Lubiw, A. (1990). Counterexample to a conjecture of Szymanski on hypercube routing. *Inf. Process. Lett.*, 35(2):57–61.
- Sipser, M. (1996). *Introduction to the Theory of Computation*. International Thomson Publishing.
- Sprague, A. P. and Tamaki, H. (1994). Routings for involutions of a hypercube. *Discrete Applied Mathematics*, 48:175–186.

Varvarigos, E. A. and Bertsekas, D. P. (1994). Performance of hypercube routing schemes with or without buffering. *IEEE/ACM Transactions on Networking*, 2(3):299–311.