ALGORITHMS FOR EMBEDDING BINARY TREES INTO HYPERCUBES

By

GARTH PETER SMEDLEY

B.Sc., University of British Columbia, Canada, 1989

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES
(DEPARTMENT OF COMPUTER SCIENCE)

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

September 1989

© Garth Peter Smedley, 1989
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science
The University of British Columbia
Vancouver, Canada

Date Oct 11, 1989
Abstract

The problem of embedding a guest graph $G$ into a host graph $H$ arises in the process of mapping a parallel program to a multicomputer. The communication structure of the program is represented by $G$, the interconnection network of the multicomputer is represented by $H$ and the goal is to embed $G$ into $H$ such that some measure of the communication cost is minimized. In general, this problem is $NP$-complete and several types of approximation algorithms have been proposed. We evaluate these algorithms empirically using hypercube host graphs and binary tree guest graphs. These families of graphs are interesting because of the existence of both heuristic techniques and theoretical algorithms for this problem. Although for general trees the problem is $NP$-complete, for binary trees the complexity remains open. We have implemented and experimented with several different algorithms and discovered variations of a greedy approach which produce close to optimal solutions in a reasonable amount of time.
Contents

Abstract ii

List of Tables v

List of Figures vi

List of Algorithms vii

Acknowledgement viii

1 Introduction 1
  1.1 Definitions and Terminology .................................... 3
  1.2 The Mapping Problem ........................................... 6
  1.3 Previous Work ................................................ 7
  1.4 Random Binary Trees .......................................... 11

2 General Heuristics 15
  2.1 Simulated Annealing ......................................... 15
    2.1.1 Overview ................................................. 15
    2.1.2 Simulated Annealing and the Mapping Problem .......... 16
  2.2 Recursive Mincut Bipartioning ............................... 19
  2.3 Mincut-based Search ......................................... 22
  2.4 Results ...................................................... 25

3 A Greedy Approach 31
  3.1 Overview .................................................... 31
  3.2 Description of the Algorithm .................................. 32
  3.3 Modifications to the Algorithm ................................ 35
  3.4 Results ...................................................... 38
# A Graph Separator Based Approach

4.1 Outline of the Algorithm .............................................. 43
4.2 Description of the Algorithm ........................................... 46
  4.2.1 Binary Tree Decomposition ....................................... 47
  4.2.2 Colour Separators .................................................. 48
  4.2.3 Embedding the Thistle Tree ....................................... 49
  4.2.4 Node Rippling ...................................................... 50
4.3 Results ................................................................. 52

# Tree Folding

5.1 Outline of the Folding Algorithm ..................................... 55
5.2 Folding Strongly Balanced Binary Trees .............................. 58
5.3 Folding Arbitrary Binary Trees ....................................... 64
5.4 Results ................................................................. 65

# Results and Conclusions

6.1 Summary ............................................................... 69
6.2 Results ................................................................. 71
6.3 Conclusions ............................................................ 73

Bibliography 81
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Total Change in the Number of Degree Two Nodes</td>
<td>12</td>
</tr>
<tr>
<td>1.2</td>
<td>Optimal Embeddings — Averages Over 2,000 Trees</td>
<td>14</td>
</tr>
<tr>
<td>2.1</td>
<td>Description of General Heuristics</td>
<td>27</td>
</tr>
<tr>
<td>2.2</td>
<td>General Heuristics — Average Extra Dilation</td>
<td>27</td>
</tr>
<tr>
<td>2.3</td>
<td>General Heuristics — Average Dilation Ratio</td>
<td>28</td>
</tr>
<tr>
<td>2.4</td>
<td>General Heuristics — Percentage Optimal Solutions</td>
<td>28</td>
</tr>
<tr>
<td>2.5</td>
<td>General Heuristics — Average Time (milliseconds)</td>
<td>28</td>
</tr>
<tr>
<td>3.1</td>
<td>Description of Greedy Algorithms</td>
<td>38</td>
</tr>
<tr>
<td>3.2</td>
<td>Greedy Algorithms - Extra Dilation</td>
<td>39</td>
</tr>
<tr>
<td>3.3</td>
<td>Greedy Algorithms - Dilation Ratio</td>
<td>39</td>
</tr>
<tr>
<td>3.4</td>
<td>Greedy Algorithms - Percentage Optimal Solutions</td>
<td>41</td>
</tr>
<tr>
<td>3.5</td>
<td>Greedy Algorithms - Average Time (milliseconds)</td>
<td>41</td>
</tr>
<tr>
<td>4.1</td>
<td>BCLR Separator Algorithm - Averages over 2,000 Trees</td>
<td>53</td>
</tr>
<tr>
<td>4.2</td>
<td>BCLR Separator Algorithm - Worst Case over 2,000 Trees</td>
<td>53</td>
</tr>
<tr>
<td>4.3</td>
<td>Random Embedding - Averages over 2,000 Trees</td>
<td>53</td>
</tr>
<tr>
<td>4.4</td>
<td>Random Embedding - Worst Case over 2,000 Trees</td>
<td>53</td>
</tr>
<tr>
<td>5.1</td>
<td>Tree Folding Algorithm - Averages over 2,000 Trees</td>
<td>68</td>
</tr>
<tr>
<td>5.2</td>
<td>Tree Folding Algorithm - Worst Case over 2,000 Trees</td>
<td>68</td>
</tr>
<tr>
<td>6.1</td>
<td>Lowest Dilation Ratio by Direct Comparison (Greed/MBS)</td>
<td>75</td>
</tr>
<tr>
<td>6.2</td>
<td>Lowest Dilation Ratio by Direct Comparison (Greed-5)</td>
<td>75</td>
</tr>
<tr>
<td>6.3</td>
<td>Dilation — Worst Case Over 2,000 Trees of Each Size</td>
<td>76</td>
</tr>
<tr>
<td>6.4</td>
<td>Dilation Ratio — Average Over 2,000 Trees of Each Size</td>
<td>77</td>
</tr>
<tr>
<td>6.5</td>
<td>Dilation Ratio — Worst Case Over 2,000 Trees of Each Size</td>
<td>78</td>
</tr>
<tr>
<td>6.6</td>
<td>Dilation — Average Over 2,000 Trees of Each Size</td>
<td>79</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Change in the Number of Nodes of each Degree .................. 13
1.2 Sixteen Node Start Tree for Random Binary Tree Generation ........ 14

2.1 Dilation Ratio — Averages for General Heuristics ................ 29
2.2 Time — Averages for General Heuristics ......................... 30

3.1 Dilation Ratio — Averages for Greedy Algorithms ............... 42

4.1 Fifteen Node Thistle Tree ........................................ 44
4.2 Inorder Embedding of the Complete Binary Tree .................. 45
4.3 A Collinear Layout of a Binary Tree ............................. 48
4.4 Dilation — Averages for Separator and Random Algorithms ....... 54

5.1 Folding a Four Node Graph ..................................... 56
5.2 Path Fold of a Four Node Path ................................ 61
5.3 Subtree Fold of Two Subtrees ................................... 62
5.4 Converting a Binary Tree to be Strongly Balanced ............... 65
5.5 Total Dilation — Averages for 512 Node Trees ................... 67

6.1 Dilation Ratio — Averages ..................................... 77
6.2 Dilation Ratio — Worst Cases ................................... 78
6.3 Maximum Dilation — Averages ................................. 79
6.4 Time — Averages ............................................. 80
List of Algorithms

2.1 Simulated Annealing ........................................... 18
2.2 Recursive MinCut Bipartitioning ............................ 20
2.3 MinCut Bipartitioning Procedure ......................... 21
2.4 MinCut-based Search ........................................ 23
3.1 Greedy Algorithm ........................................... 33
4.1 BCLR Separator-based Algorithm ......................... 46
4.2 Node Rippling Procedure .................................. 51
5.1 General Folding Algorithm ................................ 57
5.2 Fold a Strongly Balanced Binary Tree .................... 60
5.3 Valid Path-Fold Procedure ................................ 62
5.4 Valid Subtree-Fold Procedure .............................. 63
Acknowledgements

I would like to thank my supervisor, Alan Wagner, for his time, patience and financial support. Thanks also to Louise and Liam for many helpful late night conversations.
Chapter 1

Introduction

Multicomputers are parallel computing systems consisting of a set of processors with private local memory which share data by interchanging messages. In order to effectively execute parallel programs on a multicomputer the subtasks of the program must be mapped to the processors of the multicomputer in such a way as to take full advantage of the available processing power. The mapping problem has traditionally been divided into two subproblems: the load balancing problem and the processor allocation problem. The load balancing problem is grouping tasks on processors such that processor utilization is maximized. The processor allocation problem is assigning the task groups to specific processors such that communication overhead is minimized. If the communication structure of the algorithm and the interconnection structure of the multicomputer are represented with graphs, then the processor allocation problem can be expressed as finding an embedding of the communication graph into the interconnection graph which minimizes some measure of communication cost.

One powerful and popular type of interconnection network for multicomputers is the hypercube. Several hypercube multicomputers have been designed including the INTEL iPSC, NCUBE/Ten, the Cosmic Cube and the Ametek/System 14. The hypercube has several advantages which make it popular as an interconnection structure: large
bandwidth, logarithmic diameter, a regular recursive structure and efficient routing algorithms. A number of theoretical results have been established concerning the mapping of communication graphs into hypercubes. Also, several algorithms have been developed for mapping different classes of communication graphs into hypercubes.

The focus of this thesis will be the comparison of algorithms for embedding communication graphs into hypercube interconnection graphs. Several such algorithms have been proposed and used but so far no comprehensive evaluation of them has been attempted. Our goal will be to rate the different types of algorithms empirically by testing them on a substantial set of guest graphs.

Binary trees were chosen as guest graphs for our evaluation of embedding algorithms for several reasons. Firstly because binary trees form an important class of communication structure. They appear in divide-and-conquer, branch-and-bound, and functional expression evaluation algorithms among others. Binary trees are interesting also because of the complexity of the binary trees to hypercube embedding problem. The problem of embedding general trees into hypercubes has been proven to be \( NP \)-complete [37], but the complexity of mapping binary tree to hypercubes remains open. Thus binary trees are on the boundary between graphs which are easily embedded and those which are provably difficult to embed. Others have approached the binary tree to hypercube embedding problem and have derived approximation algorithms with lower bounds. These theoretical results form a contrast with the other more heuristic embedding algorithms.

There are two methods for dealing with problems for which no polynomially bounded algorithm is known [17]. One is to design an algorithm which produces optimal solutions in possibly exponential time but which performs well in practice. These algorithms are usually improvements on straightforward exhaustive search through such techniques as branch-and-bound and dynamic programming. These techniques work by creating partial solutions and extending them to a complete solution. Entire branches of the search space
can be trimmed by recognizing and rejecting impossible partial solutions.

The second approach is to use an approximation algorithm which produces a less than optimal solution in less than exponential time. Approximation algorithms for combinatorial optimization problems, such as the mapping problem, fall into two classes, tailored algorithms and general algorithms. General algorithms have the advantage of being widely applicable to many different types of problems. Tailored algorithms have a limited applicability but usually produce better solutions in less time than general techniques. General algorithms include such things as simulated annealing and neighbourhood search. Tailored algorithms vary greatly from problem to problem.

Several approximation algorithms for mapping communication graphs to hypercubes have been proposed in the literature. These include theoretical results which provide upper bounds on the deviation from optimal solutions and more practical, difficult to analyze mapping algorithms. Some of these algorithms are specific to trees and hypercubes while others are more general in nature.

In the second chapter we examine the performance of three general algorithms: simulated annealing and two adaptations of the Lin-Kernighan minimum cut heuristic. Chapter three is a discussion of the greedy approach, which is another general algorithm. Chapters four and five deal with algorithms which are tailored specifically for the binary tree to hypercube mapping problem: the first is based on graph separators and second on a kind of graph contraction operation known as folding. In the final chapter the different algorithms are compared.

1.1 Definitions and Terminology

We consider only simple undirected graphs $G = (V, E)$ where $V$ is a set of vertices and $E$ is a set of edges. If there is any possibility of confusion $G(V)$ and $G(E)$ will be used to refer to the vertex and edge sets, respectively, of $G$. The degree of a node $v$ is denoted
CHAPTER 1. INTRODUCTION

by $\text{deg} v$, and the degree of a graph by $\Delta(G)$.

The $n$-dimensional hypercube is a graph whose vertex set is the set $\{0, 1\}^n$ of length $n$ binary strings and whose edges connect just those pairs of vertices which differ in exactly one bit. An edge is said to be in dimension $d$ if it connects two nodes which differ only in bit $d$. A hypercube of dimension $n$, known as the $n$-cube, will be denoted $Q_n$. One useful property of hypercubes is that they are colour-balanced. A graph is colour-balanced if the cardinalities of the two vertex sets of a bipartition of the graph are equal.

Some algorithms take advantage of the recursive structure of the hypercube. An $n$-dimensional hypercube is split into two $2^{n-1}$-node hypercubes by cutting all of the $2^{n-1}$ edges in any one of the $n$ dimensions. Each edge in dimension $d$ connects two nodes which differ only in bit $d$. All of the nodes with bit $d$ equal to 1 reside in one subcube, known as the upper subcube, and all of the nodes with bit $d$ equal to 0 reside in the other subcube, known as the lower subcube. The upper and lower subcubes of $Q$ will be denoted $Q_U$ and $Q_L$ respectively.

A binary tree $T$ is an acyclic graph such that $\Delta(T) \leq 3$. Nodes of degree one are leaf nodes while the remaining nodes of degree 2 and 3 are interior nodes. The weight of a node $v$, denoted $W(v)$, in a rooted binary tree is equal to 1 plus the number of descendants of $v$. The sparseness of a binary tree is measured by the number of degree 2 nodes in the tree. The larger the number of degree 2 nodes the more sparse a tree said to be.

An embedding of a graph $G$ into a graph $H$ is a one-to-one mapping $\varphi : V(G) \rightarrow V(H)$. The graphs $G$ and $H$, are known as the guest and host graphs respectively. The dilation of an edge $(u, v) \in G(E)$ in an embedding $\varphi$ is defined as the distance between $\varphi(u)$ and $\varphi(v)$ in $H$. The dilation of an embedding is the maximum edge dilation. The total dilation of an embedding is the sum of the dilations of all guest graph edges. Average dilation is total dilation divided by the number of edges. Extra dilation
CHAPTER 1. INTRODUCTION

is the sum of the dilations of all dilated edges — edges with dilation equal to one are considered to be undilated. Dilation, total dilation, average dilation and extra dilation will be denoted \(d\), \(d_t\), \(d_a\) and \(d_e\) respectively. The expansion of an embedding is the ratio of the number of vertices of the host graph to the number of vertices of the guest graph.

The dilation ratio, denoted \(d_r\), of an embedding is the total dilation divided by the optimal dilation. This requires some explanation since the optimal dilation is not necessarily known. However, if we assume that trees which are colour balanced can be embedded into the hypercube of the nearest larger size with no dilated edges, (Havel's conjecture [20]) then we can easily calculate the optimal dilation. Trees which are not colour balanced require at least one dilation two edge to balance the colours. Adding a dilation two edge will flip the colours of all the nodes on one side of that edge. The exact number of dilation two edges required to balance the colours can be determined in polynomial time using dynamic programming. Thus a lower bound on the total dilation is the total number of edges in the tree plus the number of dilation two edges required to colour balance the tree. The dilation ratio is the actual total dilation divided by the optimal total dilation.

An edge mapping of \(G\) onto \(H\) maps each edge of \(G\) onto a path in \(H\). Under an embedding of \(G\) into \(H\), each undilated edge of \(G\) is mapped onto a single edge of \(H\) while dilated edges are mapped onto paths of length greater than 1. The congestion of an edge \(e\) in \(H\) is defined to be the number of paths which cross \(e\). Similarly, the node congestion of a node \(n\) is the number of paths which include \(n\). The edge (node) congestion of an embedding is the maximum edge (node) congestion in the embedding. So a dilation 1 embedding of \(G\) into \(H\) has edge and node congestion equal to 1. If the dilation is greater than 1 then the total edge congestion is equal to the total dilation. The congestion, however, depends on how the dilated edges of \(G\) are mapped to paths
in $H$.

### 1.2 The Mapping Problem

The mapping problem can be loosely defined as the implementation of algorithms on parallel systems. The term "mapping problem" has been used to refer to several different aspects of this problem including process to processor assignment, compilation of sequential programs into parallel forms, and scheduling on systolic arrays [2]. Our main concern is with process to processor embeddings and we formally define the mapping problem to be the assignment of the processes of a parallel algorithm to the processors of a multicomputer such that some measure of the execution time is minimized.

A common approach to this problem has been to divide it into two subproblems: partitioning tasks so that the number of task partitions is equal to the number of processors then assigning these partitions to individual processors [3]. If the communication structure of the partitioned tasks and the interconnection structure of the multicomputer are represented as graphs then the mapping problem can be expressed as finding the best embedding of the communication graph into the interconnection graph.

Since the goal is to find optimal mappings from communication graphs to interconnection graphs some measure of the quality of a mapping must be defined. In practical terms the best mapping is the one which provides for the most efficient execution of a parallel program. Efficiency of parallel programs can be measured by the speedup over sequential versions of the same program. Speedup is the parallel execution time divided by the sequential execution time, so for a system with $p$ processors the optimal speedup is $p$. Optimal speedup is seldom achieved due to communication overhead and load imbalance. Load imbalance occurs when some processors are idle while others remain active. Communication overhead includes the time spent initiating, transferring, and receiving messages, to the extent that these operations interfere with useful processing.
Of the two mapping subproblems, task partitioning and processor assignment, we shall be exclusively concerned with the latter. We assume that the number of task partitions is equal to the number of host processors. In the partitioning step both load balance and communication overhead must be addressed, while in the processor assignment phase the only issue is communication overhead. Our main focus, therefore, will be on finding embeddings which minimize some measure of communication overhead.

1.3 Previous Work

The graph embedding problem is an example of the partial subgraph isomorphism problem which is known to be NP-complete for arbitrary host and guest graphs. More specifically, it has been proven that for a given graph $G$ and an integer $k$, the problem of deciding whether $G$ is a subgraph of the $k$-dimensional hypercube is $\text{NP}$-complete [25]. Even when the guest graphs are restricted to trees, the problem is still $\text{NP}$-complete [39]. However, the complexity of the more restricted problem of embedding binary trees into hypercubes remains open.

Any graph which embeds perfectly into a hypercube must be colour balanced. Havel [20] has conjectured that for binary trees colour balance is also a sufficient condition. So far this conjecture has been difficult to prove. Bhatt and Ipsen [7] have proposed two weaker versions of Havel's conjecture: all binary trees are embeddable with expansion two and all binary trees are embeddable with expansion one and dilation two.

Several results exist for certain specific families of trees. Linear arrays can be embedded into their optimal hypercube since hypercubes have Hamiltonian paths [32]. More complex analysis shows that colour-balanced quasistars [28], binary trees with a single vertex of degree three, and colour-balanced one-legged caterpillars [21], binary trees with all degree three vertices along a common path, also embed into their optimal hypercubes.

The complete binary tree of size $n$ cannot be embedded into the hypercube of the
same size simply because such trees are not colour-balanced. Nebeský [27] has shown that every $n$-level complete binary tree can be embedded within the $n$-dimensional hypercube with one edge of dilation two and every other edge of dilation one. This follows from the result that $n$-vertex two-rooted trees embed perfectly into $n$-vertex hypercubes. An $N$-node two-rooted binary tree is formed by connecting the roots of two $N/2$-node complete binary trees. Bhatt and Ipsen [7] also showed that arbitrary $N$-node binary trees can be embedded with unit dilation in a hypercube of $O(N^{1.71})$ nodes. Wagner [37] has shown that every $N$-node binary tree is a subgraph of the hypercube with $O(N \log N)$ nodes.

Bhatt and Cai [5] considered the problem of mapping a dynamically growing and shrinking binary tree to a hypercube. They use a simple randomized embedding technique which guarantees dilation of $O(\log \log N)$ and with high probability assigns no more than $O(1 + M/N)$ tree nodes per hypercube node — $M$ is the number of nodes in the tree and $N$ is the number of nodes in the hypercube.

Monien and Sudborough [34] show how to embed any $N$-node binary tree with expansion one and dilation five or with dilation three and expansion $O(1)$. Bhatt et al. [6] have designed an algorithm which maps $N$-node binary trees to hypercubes with expansion one, dilation $O(1)$, and congestion $O(1)$.

In multiprocessing environments communication overhead is one of the most important factors contributing to the time required to execute a program. The common approach is to attempt to minimize communication costs by defining some objective function of the mapping which is related to communication and minimizing (or maximizing) that function. Several different objective functions have been proposed based on different assumptions about the guest communication graph and about the host multiprocessor system.

One of the first measures proposed is by Bokari [10]. Bokari assumes a very simple model of parallel computation in which the cardinality of a mapping determines its
quality. Cardinality is defined as the number of guest edges which fall on host edges, or equivalently, the number of dilated edges.

Ercal et al. [14] use the summed-cost model of parallel computation. This model is based on deviation from an ideal mapping in which computational load is uniformly distributed and no communication costs are incurred at all. The communication load of an edge is defined as the weight of the edge multiplied by its dilation, where the weight of an edge represents the relative amount of communication along that edge. Under this model total weighted dilation is to be minimized.

Lee and Aggarwal [26] consider a model of parallel computation which takes into account precedence relationships among subtasks. In their model program tasks are divided into computation stages such that the tasks in one stage may not begin until all tasks in the previous stage have completed. Communication edges within different stages may be treated independently since they are used at different times. Four objective functions are defined in terms of the communication overhead of individual edges.

Lee and Aggarwal take node and edge congestion into account in their definition of communication overhead. The communication overhead of an edge is defined as the communication weight of the edge times its actual distance in the mapping. Actual distance is determined by a combination of edge dilation and congestion. The congestion of a mapping is measured by running a simple simulation of the network. Each node sends a number of packets along its edges proportional to the communication weight of the edge. In each step of the simulation packets are moved toward their destinations such that no more than one packet at a time is carried by each edge. The communication overhead of an edge is the number of elapsed simulation steps required to transfer all of its packets. In this scheme dilated edges are routed according to the fixed routing rules of the given system. If the routing scheme can be controlled by the user than there is further room for optimization. The simulation only makes sense when the host system
uses synchronous communication. If communication is asynchronous, no attempt is made to incorporate congestion so the communication overhead of an edge is defined as dilation times communication weight.

Given this definition of communication overhead for the edges of a problem graph, Lee and Aggarwal describe four objective functions which measure the total communication overhead of an embedding. Each of the functions applies to a different pattern of communication between the computation stages. First, it is assumed that no two communication edges are needed at the same time. This communication pattern describes a form of sequential processing. The appropriate objective function in this case is the sum of the communication overheads of all the edges. Second, if all the communication edges are required at the same time then the edge with the maximum communication overhead determines the total communication overhead. There is only one computation stage and all communication takes place simultaneously within that stage. The third is a combination of the first two in which there are several computation stages and some problem edges are needed in the same stage. In this case the objective function is the sum of the maximum communication edges in each stage. The fourth objective function measures communication overhead when pipelining is allowed between the computation stages. In pipelined systems several problems of the same kind are executed in parallel and the throughput of the system is an important measure of its performance. If each stage corresponds to a pipelined process then the throughput is determined by the slowest stage and the objective function which corresponds to this is the maximum over the stages of the maximum communication edge in each stage.

Many authors [13, 6, 1, 3, 5, 15] use a model in which the guest graph is an undirected graph whose edges represent the underlying pattern of communication without specifying communication dependencies. Under this model either dilation or total dilation has been used to measure communication cost.
CHAPTER 1. INTRODUCTION

From the above it can be seen that edge dilation is an important measure of communication overhead. Under different circumstances one of dilation, average dilation or weighted dilation will be the most appropriate measure. We shall assume that communication is evenly distributed, that is, the communication weight of every edge is equal to one. This approach gives us a basis for comparing different embedding algorithms. In the following chapters we will use maximum and average dilation as measures of the quality of graph embeddings.

1.4 Random Binary Trees

In order to compare the various embedding algorithms each was run on the same set of randomly generated binary trees. The embedding algorithms were run on binary trees of sizes ranging from 16 to 1,024 nodes. Only trees with the same number of nodes as the hypercube were used in the tests, since these are the most difficult to embed. Most algorithms were tested on 2,000 trees of each size from 16 to 512 nodes. Some of the slower algorithms were not run on 512 node trees and some of the faster algorithms were tested on 1,024 node trees.

In order to test the various embedding algorithms a set of binary trees was randomly generated. The ideal would have been to produce a uniform distribution of non-isomorphic binary trees. Unfortunately the problem of generating such a distribution has not been addressed in the literature. Instead a method was used with allows some control of the sparseness of the trees in the distribution. Controlling the sparseness of the trees is desirable since sparser trees appear to be easier to embed.

Several authors [24, 30, 29, 33, 41] have examined the problem of randomly generating ordered binary trees using various forms of ranking and unranking algorithms. These algorithms were not suited to our purposes because of the sparseness of ordered trees and also because the time required to generate each tree is $O(N \log N)$ for $N$-node trees.
CHAPTER 1. INTRODUCTION

We chose an algorithm, based on an Broder [12], which performs a random walk on the binary spanning of the complete $N$-node graph. This algorithm is very efficient, the next tree in a walk is generated in constant time. It also has the advantage that the sparseness of the trees can be varied.

Broder's algorithm performs a random walk on the spanning $k$-ary trees of a $k$-degree graph such that the trees converge to the uniform distribution. It starts with a spanning tree, $T$, of an undirected graph, $G$, and modifies $T$ by adding one edge and removing another at random. First an edge is picked, uniformly at random, from $G - T$ and added to $T$, forming a cycle. Then one of the edges on the cycle is chosen, again uniformly at random, to be deleted. The series of trees generated by a sequence of these perturbations forms a random walk over the spanning trees of $G$.

Broder's algorithm was modified to generate binary trees instead of $k$-ary trees. This is done by starting with a binary tree and disallowing any changes which increase the degree. When an edge is picked to be inserted only those edges which are not incident on degree three nodes of the tree are considered for addition. Edges are deleted, as before, by choosing one at random from the cycle created by the new edge. The complete graph of size $N$ can be used as the underlying graph to generate any $N$-node binary tree.

In order to change the sparseness of the distribution, the way in which edges are added can be altered. Figure 1.1 shows all the ways in which edges can be added and deleted. The figure also shows the effect of the different types of additions and deletions.
on the number of nodes of each degree. The addition of an edge is not allowed to increase the degree so every new edge will connect: 2 degree one nodes, 2 degree two nodes, or a degree two node and a degree one node. Deleted edges are always removed from a cycle so they will connect: 2 degree three nodes, 2 degree two nodes or a degree three node and a degree two node. Table 1.1 shows the nine ways in which the three types of edge deletions and the three types of edge additions can be combined. The table also shows the effect of each combination on the number of degree two nodes in the tree. The first row shows that if the added edge connects two leaves then the number of degree two nodes will rise regardless of which type of edge is deleted. The sparseness of the distribution can be adjusted by changing the probability with which the added edge is allowed to connect two leaves. Increasing this probability increases the sparseness of the trees.

A binary tree with no degree two nodes, Figure 1.2, was used as the start tree for all samples since this is the least sparse binary tree. The probability of connecting two leaves was set as low as possible, that is, two leaves are never connected unless there is no other choice. The sets of trees produced by this method were used to test the various embedding algorithms described in the following chapters. Table 1.2 shows averages of the dilation, total dilation, extra dilation and average dilation for optimal embeddings of
Figure 1.2: Sixteen Node Start Tree for Random Binary Tree Generation

<table>
<thead>
<tr>
<th>Measure</th>
<th>Hypercube Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td>Dilation</td>
<td>2.00</td>
</tr>
<tr>
<td>Total Dilation</td>
<td>15.6</td>
</tr>
<tr>
<td>Extra Dilation</td>
<td>1.1960</td>
</tr>
<tr>
<td>Average Dilation</td>
<td>1.0399</td>
</tr>
</tbody>
</table>

Table 1.2: Optimal Embeddings — Averages Over 2,000 Trees

each set of trees.

The various algorithms were run on a dedicated INMOS T800 Transputer. Timings for the various algorithms are accurate measures of running times on this hardware.
Chapter 2
General Heuristics

2.1 Simulated Annealing

2.1.1 Overview

Simulated annealing is a probabilistic search technique [23, 35] which has been used to provide good solutions to many difficult combinatorial optimization problems. The basic idea is to start at a random place in the search space then apply local changes which bring the search object closer to the goal, occasionally allowing changes which move away from the goal to avoid local minima. Several authors have applied this technique to various aspects of the mapping problem.

In order to apply the simulated annealing technique to a problem it is first necessary to represent the problem in terms of a combinatorial search. This requires the specification of three elements: a search space consisting of solution instances, a measure of the quality of the solution, and an operator which incrementally permutes a solution. There are many different ways to express any given problem in terms of a combinatorial search simply by varying any of these three elements.

The simulated annealing method starts with a random solution from the search space. A new solution is generated using the permutation operator and the difference in the qual-
ity of the two solutions is determined. If the new solution is an improvement over the 
old one or if the two are equivalent then the new solution is accepted unconditionally. If 
however, the new solution is worse than the old one, then the new solution is accepted 
with probability $e^{-c/T}$ where $c$ is the difference in cost between the two solutions. $T$ is a 
parameter, called the temperature, which controls the algorithm’s ability to jump out of 
local minima. Higher temperatures increase the likelihood of accepting moves which lead 
away from the goal. The temperature starts high and is slowly lowered until a prede-
termined freezing point is reached. At each temperature a fixed number of solutions are 
generated. This is known as the inner-loop criteria and is usually specified by restricting 
either the total number of permutations or the number of accepted permutations. After 
the inner-loop criteria has been met, the temperature is lowered according to an update 
function. Together the starting temperature, freezing point and the temperature up-
date function are known as the cooling or annealing schedule. In order to use simulated 
annealing it is necessary to specify an inner-loop criteria and an annealing schedule.

2.1.2 Simulated Annealing and the Mapping Problem

Bollinger and Midkiff [11] break the mapping problem into two subproblems and use 
simulated annealing to solve each subproblem in turn. First processes are assigned to 
processors then dilated guest edges are assigned specific routing paths in the host network. 
The measure used for the first subproblem is a combination of the maximum dilation and 
the average dilation. Maximum dilation is ignored during high temperature annealing, 
but is not allowed to increase during the later stages of annealing. Solutions are permuted 
by pairwise exchanges of processes. Unfortunately Bollinger and Midkiff do not discuss 
any of the details of their inner-loop criteria or their annealing schedule.

Yu and Yu [40] combine task partitioning and processor assignment into one problem 
and use simulated annealing to solve it. The cost function to be minimized is the sum of
a communication factor and a load balance factor, weighted by proportionality constants. Total weighted dilation is used as a measure of communication cost. The permutation operator used here is quite different from the one used in the previous approach. Host graphs are limited to hypercubes and annealing is done one dimension at a time. The edges in each dimension of a hypercube bisect the hypercube, so each annealing process splits the nodes of the guest graph into two sets of equal size. The permutation operator randomly chooses a node from each side of the current partition and swaps them. Once a dimension has been annealed it is fixed for the remainder of the algorithm. An annealing schedule with a standard exponential temperature update function is used.

Ercal et al. [15, 31, 14] present a more detailed study of the use of simulated annealing to simultaneously solve both the task assignment and the load balancing parts of the mapping problem. The cost function to be minimized is the total weighted dilation subject to the constraint that the load balance remain within a specified tolerance. Ercal [15] found that keeping load balance always within the tolerance resulted in poor quality solutions due to the fact that the annealing would get trapped in a configuration with perfect load balance and high communication cost. This can be solved by accepting some solutions which are not within the tolerance. The allowed deviance from the tolerance is derived dynamically by performing a set of trial runs with varying allowed deviations. Empirical evidence shows that the allowable deviance can be determined in a reasonable amount of time by using a low inner-loop criteria for the trial runs.

Ercal et al. provide a detailed description of their annealing schedule. The starting temperature is set so as to give a probability of 0.9 of accepting a mean decrease in the cost function. This mean decrease is calculated by averaging the cost decreases of all perturbations of the initial configuration. The freezing point is set so that the probability of accepting a move which produces a unit increase in the cost function is $2^{-31}$. The temperature update function used is $T_{\text{new}} = 0.95 \times T$. Two parameters, $M$
Algorithm: Simulated Annealing
(Constructs a mapping $\varphi$ of a guest graph $G$ onto a host graph $H$)

Randomly map $G$ to $H$
Calculate the starting temperature ($T$)
Repeat until $T \leq$ freezing point
{ 
    Repeat for $Mt \times N$ attempted moves or $Ma \times N$ accepted moves 
    { 
        Choose a pair of guest nodes $(a, b)$ randomly 
        Temporarily swap $\varphi(a)$ and $\varphi(b)$ 
        Calculate the increase in total dilation, $c$, caused by the swap 
        If $c \leq 0$ Then 
            Accept the swap 
        Else 
            Accept the swap with probability $e^{-c/T}$ 
        
    } 
    Update the temperature ($T = 0.95 \times T$) 
}

Algorithm 2.1: Simulated Annealing

and $Ma$, control the inner-loop criterion. The inner loop terminates when either $Mt \times N$ moves have been attempted or when $Ma \times N$ moves have been accepted, where $N$ is the number of possible moves.

Ercal et al. evaluated two permutation operators under their annealing scheme: process migration from one processor to another and process swapping. Process swapping was found to work better than process migration when the load balance tolerance was strictly enforced. Process migration with dynamically scaled tolerance worked better than process swapping without dynamically scaled tolerance, but no results were presented for process swapping with dynamically scaled tolerance. These results were obtained from trial runs using a hypercube host graph and both regular and irregular mesh guest graphs.

Since we are concerned with communication overhead and not load balancing, we were able to avoid the problems associated with trading off these two optimization factors. Also, node migration does not make sense when the number of guest nodes is equal to
CHAPTER 2. GENERAL HEURISTICS

the number of host nodes so node swapping was used as the permutation operator. We used the same cooling schedule as Ercal and found that setting $Mt = Ma = 0.1$ produced good solutions in a fairly reasonable amount of time. Algorithm 2.1 is an outline of the simulated annealing algorithm.

This version of simulated annealing is $O(N^2 \log \log N)$. The inner loop is executed $0.1 \times (N^2 - N)$ times for each temperature since $Mt = Ma = 0.1$ and the number of possible swaps is $N^2 - N$. The outer loop is executed until the temperature is lowered to the freezing point. In this case the freezing point is fixed, so the number of outer loop iterations grows logarithmically with the starting temperature. However, the starting temperature is calculated from and is directly proportional to the mean decrease in the total dilation which, in turn, is dependant upon two factors; the degree of the guest graph and the diameter of the host graph. For binary tree guest graphs the degree is constant and for hypercube host graphs the diameter is the log of the number of nodes; so the starting temperature grows logarithmically with the number of nodes. To sum up, the number of outer loop iterations is $O(\log T)$ and $T$ is $O(\log N)$ so the outer loop is $O(\log \log N)$ and the entire algorithm is $O(N^2 \log \log N)$.

2.2 Recursive Mincut Bipartitioning

Ercal, Ramujam and Sadayappan [15, 14] describe a task allocation scheme for hypercubes which is based on a variation of the Kernighan and Lin [22] mincut heuristic proposed by Fiduccia and Mattheyas [16]. The optimality criterion used is the total weighted dilation subject to the constraint that the computational loads are balanced to within a specific tolerance. The quality of a mapping is determined by its deviation from an ideal where the ideal is to have the computational load uniformly distributed among all processors while incurring no communication costs at all.

The mincut procedure divides the guest nodes into two groups. The nodes are initially
Algorithm: Recursive Mincut Bipartitioning
(Recursively constructs a mapping \( \varphi \) of a graph \( G \) onto a hypercube \( Q \))
\((Q_U \text{ and } Q_L \text{ are the upper and lower subcubes of } Q)\)

If \( G \) and \( Q \) both have exactly one node
Then RETURN

Divide \( V(G) \) into two sets \( U \) and \( L \) of equal size using Algorithm 2.3
Map \( U \) to \( Q_U \) and \( L \) to \( Q_L \) as follows
\[
\{ \\
\quad \text{Let } d \text{ be the dimension that } Q_U \text{ and } Q_L \text{ differ in} \\
\quad \text{Set the } d\text{th bit of } \varphi(l) \text{ for } l \in L \text{ to } 0 \\
\quad \text{Set the } d\text{th bit of } \varphi(u) \text{ for } u \in U \text{ to } 1 \\
\}
\]
Recursively map \( U \) onto \( Q_U \)
Recursively map \( L \) onto \( Q_L \)

Algorithm 2.2: Recursive Mincut Bipartitioning

split into two partitions then the split is improved by a sequence of node transfers from
one partition to the other. This procedure is applied recursively to produce a \( K \)-way
partition of a graph where \( K \) is a power of two. The assignment of guest nodes to
host nodes is made incrementally during this recursive partitioning. Each guest node is
assigned a \( \log N \) bit label which corresponds to a hypercube node. At the \( i \)th level of
recursion the \( i \)th bit of each label is set — bit \( i \) is set to zero for all nodes in one partition,
and set to one for all nodes in the other partition. After \( \log N \) recursive levels each guest
node will have been assigned a unique \( \log N \) bit label. Algorithm 2.2 is an outline of the
mincut algorithm.

At each recursive level the mincut procedure is performed twice. The first time nodes
are transferred when the transfer will decrease communication costs. The second time
load balancing is also taken into consideration when calculating the desirability of a
transfer. If after these two steps the load balance is not within a desired tolerance then
nodes are moved from the large partition to the small partition. So there are three

\(^{1}\text{Unless otherwise stated, all log's are base 2}\)
CHAPTER 2. GENERAL HEURISTICS

Procedure: MinCut Bipartitioning
(Bipartitions the nodes of a graph $G$)
($\varphi$ is a partial mapping of $G$ to $H$)

Randomly divide $V(G)$ into two subsets $U$ and $L$ of equal size
Temporarily extend $\varphi$ by mapping $U$ to $Q_U$ and $L$ to $Q_L$
For each node $g \in V(G)$ calculate $gain(g)$ as follows
{
  Temporarily transfer $g$ to the other subset ($U$ or $L$)
  Calculate the decrease in total dilation, $c$, caused by the transfer
  Set $gain(g)$ to $c$
}
Repeat until no improvement in total dilation can be made
{
  Unmark all nodes
  Repeat until all nodes have been marked
  {
    Chose the unmarked node, $g$, which has the highest $gain$
    Temporarily transfer $g$ to the other subset
    Update the $gain$'s of all neighbours of $g$
    Mark $g$
  }
  For each transfer $t$ (in the order performed) calculate the prefix sum ($\sum_{i} gain[i]$)
  Permanently perform all transfers up to and including the one with the largest prefix sum
}

Algorithm 2.3: MinCut Bipartitioning Procedure

steps: first the mincut procedure is used to reduce inter-task communication, then the procedure is repeated to improve load balancing, finally the load balance is brought to within the specified tolerance.

The mincut procedure itself consists of two phases. It begins, like simulated annealing, with a random partition which is reasonably balanced. In the first phase a sequence of trial moves is generated, in the second phase the best first $k$ moves of this sequence are actually applied. Each trial move is generated by finding the best move from among the nodes which have not yet been selected for a trial move. This continues until all nodes have been selected, then the second phase is performed. The entire two-phase procedure
CHAPTER 2. GENERAL HEURISTICS

is repeated until no improvement can be made. Algorithm 2.3 is an outline of the mincut procedure.

In our implementation of this algorithm we considered the case in which all the processes have the same computational load, so the best load balance can be achieved by assigning one task to each processor. Our implementation leaves out the second and third steps of the algorithm, the ones that deal with load balancing. We require that there be no more than one guest node mapped to each host node. In order to ensure this, each bipartitioning step must divide the nodes exactly in half. The nodes are split randomly into two equal size groups then transfers are made in pairs such that after an even number of transfers the partitions are of equal size.

The time complexity of Recursive Mincut Bipartitioning is $O(N \log N)$ because there are $\log N$ levels of recursion and each level has $O(N)$ work.

2.3 Mincut-based Search

Ercal [15] proposes a local search procedure which has a form similar to the mincut heuristic. This procedure is designed to produce solutions to the embedding problem when the number of guest nodes is equal to the number of host nodes. The cost measure used is the total weighted dilation. Load balance is not considered, so the procedure always produces one-to-one embeddings.

The procedure searches for an assignment of guest to host nodes such that the total weighted dilation is minimized. It starts with a random assignment, then swaps nodes so as to improve the embedding. Rather than simple local search, a method similar to the mincut heuristic is used to select nodes for swapping. A local search procedure would chose the pair which gave the largest improvement, swap them then repeat until no further improvement could be made. This leads quickly to a local minima which may be quite far removed from the global minima. The mincut-like heuristic is able has some
Algorithm: Mincut-based Search
(Constructs a mapping \( \varphi \) of a guest graph \( G \) onto a host graph \( H \))

Randomly map \( G \) to \( H \)
For every pair of guest nodes \( (a, b) \)
{  
  Temporarily swap \( \varphi(a) \) and \( \varphi(b) \)
  Calculate the decrease in total dilation, \( c \), caused by the swap
  Set \( \text{gain}((a, b)) \) to \( c \)
}
Repeat until no improvement in total dilation can be made
{  
  Unmark all nodes
  Repeat until all nodes have been marked
  {    
    Chose the pair of unmarked nodes, \( (a, b) \), which has the highest \( \text{gain} \)
    Temporarily swap \( \varphi(a) \) and \( \varphi(b) \)
    Update the \( \text{gain} \)'s of all neighbours of \( a \) and \( b \)
    Mark \( a \) and \( b \)
  }
  For each swap \( s \) (in the order performed) calculate the prefix sum (\( \sum_{1}^{s} \text{gain}[s] \))
  Permanently perform all swaps up to and including the one with the largest prefix sum
}

Algorithm 2.4: Mincut-based Search

ability to escape from local minima. Sequences of swaps are applied such that there is an overall improvement, however, some of the individual swaps may temporarily degrade the quality of the solution. This enables the procedure to move through solutions of equal or poorer quality on the way to a better solution. This happens, for example, when there is a short sequence of swaps which does not change the quality of the solution, but which allows a further swap to create some improvement. Of course not all such sequences are found by the algorithm, only sequences of length \( \frac{1}{2} | V(H) | \) are examined, and even when such a sequence does exist, it is not a certainty that that the algorithm will find it.

The mincut-based search algorithm starts with a random embedding. The gain is calculated for each pair of nodes, where the gain of a pair is the improvement obtained
by swapping the two nodes. First, a sequence of swaps is generated by choosing the pair with the largest gain, temporarily swapping them, then choosing the best pair from the remaining nodes until all of the nodes have been chosen. Then the first $k$ pairs are swapped permanently, where $k$ is chosen to minimize the total dilation. The steps of creating a sequence and swapping the first $k$ pairs in the sequence are repeated until no further improvement can be made. Algorithm 2.4 is an outline of the mincut-based search algorithm.

Ercal compares node-swapping with two other permutation operators for mincut-based search, bit-toggling and bit-swapping. Bit-swapping works by running the MBS (mincut-based search) algorithm for each bit in the hypercube node label as long as there is some improvement in the gain. For each bit $i$, only nodes which differ exactly in bit $i$ are swapped. Bit-toggling works by splitting the hypercube into upper and lower subcubes and iteratively rotating the lower subcube. The node labels in the upper subcube all differ in exactly one bit $l$ from their respective neighbours in the lower subcube. The lower subcube is iteratively rotated by toggling, in turn, each bit $i$ ($i \neq l$), of every node label of the subcube. The MBS algorithm is run over each of the $\log N$ different ways to split the cube into two subcubes (ie. each dimension), and over each rotation of the lower subcube, as long as some improvement is possible. Node-swapping was found to be superior in terms of the quality of the solutions produced.

The only change made to the mincut-based search algorithm was to restrict the way in which guest nodes are mapped to host nodes. If the tree is colour balanced then all even tree nodes must map to even hypercube nodes (and odd tree nodes to odd hypercube nodes) for the mapping to be optimal. Which of the tree and hypercube nodes are even and which are odd can be determined before running the algorithm. The algorithm can be modified firstly, to correct the random embedding so that nodes of the same parity are mapped together, and secondly, to swap only nodes of equal parity.
CHAPTER 2. GENERAL HEURISTICS

This simple modification makes the algorithm run faster since fewer swaps are examined in each iteration. It is also possible to extend this algorithm to work on unbalanced trees. Unbalanced trees cannot be embedded with dilation 1 into hypercubes — at least one edge must be dilated. Dilating an edge with dilation two (or any even dilation) has the effect of reversing the parities of all nodes on one side of the edge. A dynamic programming technique can be used to find the smallest set of dilation two edges required to balance any unbalanced tree. Once this set of edges has been found, the parities can be corrected and the modified mincut-based search algorithm run on this new tree.

The time complexity of mincut-based search is $O(N^2 \log N)$. Observation of the algorithm shows that the outer loop is executed a constant number of times independent of the size of the problem. The inner loop is executed $O(N)$ times and finding the pair with largest gain takes $O(N^2)$. This can be reduced to constant time if the gains are kept sorted, in which case the gain calculation step requires an $O(\log N)$ insertion operation for each of its $O(N^2)$ gain calculations. The gain calculation step becomes the dominating part of the algorithm, taking $O(N^2 \log N)$ time.

2.4 Results

Ercal et al. [14] found that recursive mincut bipartitioning was more practical than simulated annealing for the combined processor allocation load-balancing problem. Their mincut algorithm produced solutions similar in quality to simulated annealing in much less time. These results are based on tests performed on seven mesh-like and random guest graphs ranging in size from 144 to 602 nodes, five of these are representative of graphs arising from finite element modelling applications.

Berman [1] compared simulated annealing with the recursive mincut bipartitioning algorithm for processor allocation. They used a grid host graph and an unspecified set of guest graphs, all with the number of guest nodes less than or equal to the number
of host nodes. A node swapping permutation operator was used for both algorithms. Berman examined these algorithms for use in a larger parallel programming system in which communication graphs are contracted, embedded and routed to run on a grid architecture. No quantitative results are presented but Berman decided to use the mincut algorithm because of the large amounts of computation required to achieve good results with simulated annealing.

Our results support these findings for the processor allocation problem. In total three different mincut-based algorithms were compared with simulated annealing (see Table 2.1). Mincut is recursive mincut bipartitioning, MBS-1 is mincut-based search and MBS-2 is a modified version of mincut-based search which matches the parities of the guest and host nodes. The results of running the four algorithms on 2,000 random binary trees of each size are shown in Tables 2.2 through 2.5.

Table 2.3 and the graph in Figure 2.1 shows the results for dilation ratio. Although simulated annealing performed well on sixteen node trees its performance quickly deteriorated as the size of the trees increased. In general simulated annealing is the worst of the four algorithms in terms of both time and average dilation. This was to be expected since simulated annealing is a general algorithm which does not take advantage of the specific nature of the problem.

Mincut-based search and recursive mincut bipartitioning produced very similar results. What is surprising is how much mincut-based search was improved for small graphs simply by matching guest and host parities. This also improved the running time of the algorithm since half the number of swaps need to be considered. The average running time of the algorithms on an INMOS T800 Transputer is shown in Table 2.5 and the graph in Figure 2.2. Time is shown on a logarithmic scale since the graphs grow exponentially. MBS-2 is a considerable improvement over MBS-1 for graphs of small size. However, the dilation ratio and running time of the two algorithms appears to converge as the
CHAPTER 2. GENERAL HEURISTICS

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mincut</td>
<td>Recursive mincut bipartitioning</td>
</tr>
<tr>
<td>MBS-1</td>
<td>Mincut-based search</td>
</tr>
<tr>
<td>MBS-2</td>
<td>Mincut-based search with parity checking</td>
</tr>
<tr>
<td>Anneal</td>
<td>Simulated Annealing</td>
</tr>
</tbody>
</table>

Table 2.1: Description of General Heuristics

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Number of Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
</tr>
<tr>
<td>Mincut</td>
<td>2.50</td>
</tr>
<tr>
<td>MBS-1</td>
<td>2.28</td>
</tr>
<tr>
<td>MBS-2</td>
<td>1.17</td>
</tr>
<tr>
<td>Annealing</td>
<td>2.12</td>
</tr>
</tbody>
</table>

Table 2.2: General Heuristics — Average Extra Dilation

size of the graphs is increased. The effect of parity matching is not as great at higher dimensions. This is because colour balance is a necessary but not a sufficient condition for a perfect embedding. An embedding with a badly skewed balance can still be close to optimal.

MBS-2 requires that the guest graph be acyclic in order to calculate the node parities. The guest graph must also be of small degree if the procedure which finds the dilation two edges required to correct the parity is to be efficient. The other algorithms are equally applicable to any type of guest graph. There is reason to believe mincut would perform better on regularly structured graphs such as grids, because of the recursive nature of the algorithm. Otherwise we would expect that the relative performance of the algorithms will carry over to other types of guest graphs.
## CHAPTER 2. GENERAL HEURISTICS

### Table 2.3: General Heuristics — Average Dilation Ratio

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mincut</td>
<td>1.1230</td>
<td>1.1751</td>
<td>1.2256</td>
<td>1.2776</td>
<td>1.3193</td>
<td>1.3670</td>
</tr>
<tr>
<td>MBS-1</td>
<td>1.1089</td>
<td>1.1814</td>
<td>1.2379</td>
<td>1.2860</td>
<td>1.3216</td>
<td>—</td>
</tr>
<tr>
<td>MBS-2</td>
<td>1.0365</td>
<td>1.1096</td>
<td>1.1778</td>
<td>1.2440</td>
<td>1.2940</td>
<td>—</td>
</tr>
<tr>
<td>Annealing</td>
<td>1.0985</td>
<td>1.1718</td>
<td>1.2440</td>
<td>1.2933</td>
<td>1.4011</td>
<td>1.5784</td>
</tr>
</tbody>
</table>

### Table 2.4: General Heuristics — Percentage Optimal Solutions

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mincut</td>
<td>0.04</td>
<td>0.19</td>
<td>0.86</td>
<td>4.00</td>
<td>18.91</td>
<td>89.05</td>
</tr>
<tr>
<td>MBS-1</td>
<td>0.08</td>
<td>0.53</td>
<td>3.16</td>
<td>20.38</td>
<td>141.26</td>
<td>—</td>
</tr>
<tr>
<td>MBS-2</td>
<td>0.06</td>
<td>0.41</td>
<td>2.56</td>
<td>16.58</td>
<td>115.52</td>
<td>—</td>
</tr>
<tr>
<td>Annealing</td>
<td>0.43</td>
<td>1.90</td>
<td>7.91</td>
<td>32.53</td>
<td>132.52</td>
<td>539.18</td>
</tr>
</tbody>
</table>

### Table 2.5: General Heuristics — Average Time (milliseconds)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mincut</td>
<td>0.04</td>
<td>0.19</td>
<td>0.86</td>
<td>4.00</td>
<td>18.91</td>
<td>89.05</td>
</tr>
<tr>
<td>MBS-1</td>
<td>0.08</td>
<td>0.53</td>
<td>3.16</td>
<td>20.38</td>
<td>141.26</td>
<td>—</td>
</tr>
<tr>
<td>MBS-2</td>
<td>0.06</td>
<td>0.41</td>
<td>2.56</td>
<td>16.58</td>
<td>115.52</td>
<td>—</td>
</tr>
<tr>
<td>Annealing</td>
<td>0.43</td>
<td>1.90</td>
<td>7.91</td>
<td>32.53</td>
<td>132.52</td>
<td>539.18</td>
</tr>
</tbody>
</table>
Figure 2.1: Dilation Ratio — Averages for General Heuristics
Figure 2.2: Time — Averages for General Heuristics
Chapter 3

A Greedy Approach

3.1 Overview

In this chapter we examine a greedy approach to the mapping problem. The greedy approach is a search strategy which emphasizes speed sometimes at the expense of solution quality. The strategy uses local optimization to chose between alternative paths in the search space and such decisions once made are irrevocable. The greedy approach is also known as neighbourhood search or hill-climbing.

For some problems there exists a greedy algorithm which is guaranteed to produce optimal solutions. This is not the case for greedy approaches to the mapping problem. We know of no greedy algorithm which always finds optimal solutions for the binary tree to hypercube embedding problem.

Chen and Gehringer [13] propose a graph-oriented greedy algorithm for the mapping problem. The algorithm operates on host and guest graphs of any type as long as the number of host nodes is greater than or equal to the number of guest nodes. Guest edges are assumed to be weighted with individual communication costs and the average weighted dilation is used to measure the quality of embeddings. We will assume that the communication costs are uniform among the edges of the guest graph so that average dilation can be used in place of average weighted dilation.
In the previous chapter we described algorithms which, like the greedy algorithm, perform some sort of search. The algorithm described here differs from these both in the strategy used to conduct the search and, more importantly, in the representation of the search space. Recall that simulated annealing and the mincut based search technique start with a random solution then repeatedly apply permutations in an attempt to improve upon the original solution. The search space consists of all possible mappings of the guest graph to the host graph. In contrast, Chen and Gehringer's algorithm starts with an "empty" solution then incrementally extends it until a complete solution is obtained. Here the search space consists of all possible partial mappings of the guest graph to the host graph. Note that it is possible to apply the greedy strategy to either of these representations. In fact, the mincut-based search technique is actually a slight variation on the greedy strategy. Search space representation is the factor that most clearly distinguishes Chen and Gehringer's algorithm from the others.

3.2 Description of the Algorithm

Chen and Gehringer's algorithm operates by mapping guest nodes to host nodes one at a time. The algorithm starts by mapping the node of the guest graph with the largest degree to an arbitrary host node. The first host node can be chosen arbitrarily since hypercubes are symmetric. The guest node with the largest degree is mapped first in order to maximize the chances of optimally mapping its incident edges. The algorithm proceeds by repeatedly choosing the best pair from among the unmapped nodes until all guest nodes are mapped. The best pair is chosen by calculating a quantity, called the gain, for each guest-host pair and then choosing the pair with the highest gain. The gain of mapping a guest node $g$ to a host node $h$ is defined as the sum, over all the edges between $g$ and mapped neighbours of $g$, of the diameter of the host graph minus the weighted dilation of the edge. Note that the dilation of an edge cannot be larger that
Algorithm: Greedy
(Constructs a node mapping $\varphi$ of a guest graph $G$ onto a host graph $H$)

Set $\varphi(g)$ to NULL for all nodes $g \in V(G)$
Find the guest node with the largest degree $g_1$
Chose an arbitrary host node $h_1$
Set $\varphi(g_1)$ to $h_1$
Repeat until $\varphi$ is complete
{   
    Find the pair $(g, h)$ with the maximum gain such that $g \in V(G)$ and $\varphi(g) = \text{NULL}$
    $h \in V(H)$ and $\varphi^{-1}(h) = \text{NULL}$
    Set $\varphi(g)$ to $h$
}

Procedure: Gain
(Calculates the gain of mapping a guest node $g$ to a host node $h$ under the partial mapping $\varphi$)

Set the gain to 0
For every neighbour, $g_n$, of $g$ such that $\varphi(g_n) \neq \text{NULL}$
{   
    Calculate the dilation, $d$, of the edge $(g, g_n)$ under $\varphi$
    Add $D - d$ to the gain — where $D$ is the diameter of the host graph
}

Algorithm 3.1: Greedy Algorithm

the diameter of the host graph. Therefore the gain is always a positive number and large gains correspond to low dilations. Algorithm 3.1 is an outline of Chen and Gehringer’s greedy algorithm and the gain procedure.

In order to choose the best pair it is not necessary to calculate the gain of mapping every unmapped guest node to every unmapped host node. Many of the potential mapping pairs can be eliminated in advance because of the behaviour of the gain function. Since the gain calculation is a sum over the edges between a guest node and its mapped neighbours, guest nodes with no mapped neighbours will have a gain of zero. Therefore, the algorithm will always choose the next guest node from among the neighbours of previously mapped guest nodes, the frontier nodes. Storing a list of the frontier nodes
requires an operation of the order of the degree of the guest graph every time a pair of
nodes are mapped. In order to choose the best pair it is only necessary to consider the
frontier nodes.

If the guest graph is a tree then it is likely that several mapping pairs will have the
same gain. This is even more likely under our assumption of uniform edge communication
cost. This can be seen by examining the behaviour of the gain function. Assume that the
tree is rooted at the first guest node to be mapped and let $P(g)$ be the parent of $g$ in the
rooted tree. The gain of mapping any further guest node, $g$, will be zero as long as $P(g)$
is unmapped. Once $P(g)$ has been mapped, the gain of mapping $g$ will be determined by
the dilation of the edge $(g, P(g))$. Therefore, mappings of $g$ to all unmapped host nodes
equidistant from $\varphi(P(g))$ will have the same gain. Also, mappings of two guest nodes
$g_1$ and $g_2$ will have the same gain if the distance between $\varphi(g_1)$ and $\varphi(P(g_1))$ is equal to
the distance between $\varphi(g_2)$ and $\varphi(P(g_2))$. At any stage in the algorithm all pairs which
embed their edge with the same dilation will have the same gain.

It is also likely that more than one mapping pair will have the highest gain, forcing
the algorithm to make an arbitrary decision. As proposed by Chen and Gehringer, the
algorithm relies on an ordering of the nodes of the guest and host graphs to resolve such
cases. A well chosen ordering can produce a good embedding. For example, an $N$ node
ring can be mapped perfectly to an $N$ node hypercube if the hypercube nodes are ordered
in a hamiltonian circuit. The resolution of ties in the gain calculation is a significant part
of the greedy algorithm.

Lee and Aggarawal [26] also propose a mapping algorithm based on a greedy ap­
proach. The algorithm is very similar Chen and Gehringer’s algorithm. It starts with an
empty solution and extends it until a complete solution is reached. Lee and Aggarawal
suggest that mappings found with the greedy algorithm may be improved by using a
node swapping method similar to mincut-based search. First the greedy algorithm is
run, then an attempt is made to improve the embedding by swapping guest nodes which are incident on dilated edges.

3.3 Modifications to the Algorithm

For binary tree guest graphs Chen and Gehringer resolve ties in the gain calculation randomly. It is not obvious whether the algorithm could be improved by eliminating random choices or if the randomness actually contributes to the performance of the algorithm. In order to determine the role of randomness in the greedy algorithm, the random tie breaking method was compared with three other methods: breadth-first tree order, hypercube node order and hypercube level order. With breadth-first tree order the binary tree is rooted at the first node mapped and ties are resolved by choosing the guest node closest to the root. Hypercube node order and hypercube level order resolve ties based on an ordering of the host graph rather than the guest graph. Hypercube node order is determined by the binary node labels. With this order the algorithm will attempt to fill smaller subcubes before moving into larger subcubes. Hypercube level order is determined by the distance from node zero. Node numbers whose bits sum to \( l \) are in level \( l \). With level ordering the algorithm maps hypercube node zero first, then attempts to fill levels from lower to higher.

None of these tie-breaking methods is completely deterministic since the first guest node is still chosen arbitrarily from among those with the highest degree. Arbitrary choices are also made between guest nodes at the same depth for breadth-first tree order and host nodes at the same level for hypercube level order. Hypercube node order is the most deterministic of the three — selecting the first guest node is the only choice made randomly.

We also attempted to improve the algorithm by eliminating choices which indirectly create dilated edges. Dilated edges are created indirectly in several different ways, and
sometimes these situations are easily detectable. In order to determine whether a guest node, \( g \), should be mapped to a host node, \( h \) we can check the number of unmapped neighbours of \( g \) and \( h \). If \( g \) has more unmapped neighbours than \( h \) then the extra edges incident to \( g \) must eventually be dilated. To perform this check it is necessary to keep track of the number of unmapped neighbours of each node. Each time a pair of nodes are mapped this number must be updated for all neighbours of both nodes in the pair. This operation takes time proportional to the degree of the graphs, \( O(\log N) \), and is incurred each time a pair of nodes are mapped.

A second way in which edges can be indirectly dilated is also related to unmapped neighbours of mapped nodes. Assume node \( g \) is mapped to node \( h \) and that the number of unmapped neighbours of \( g \) is exactly equal to the number of unmapped neighbours of \( h \). If one of the unmapped neighbours of \( h \) is mapped to a guest node other than one of the unmapped neighbours of \( g \) then one of the edges incident on \( g \) will eventually be dilated. Checking for this condition requires an extra \( O(\log N) \) operation for every pair of candidate nodes considered when a choice is made. This is in addition to the time required to update the unmapped neighbour counts. Of course, even if the above two checks are done it is still possible a dilation causing choice will be forced.

There is a third type of choice which indirectly causes edges to be dilated but is not easily detectable. This happens when a “hole” is left in the hypercube. A hole is an unmapped host node which is completely surrounded by mapped nodes none of which has any unmapped neighbours. Eventually some guest node will have to be mapped to the hole thus creating a dilated edge. Avoiding the creation of holes is difficult. Holes are created incrementally by mapping elsewhere, one at a time, all nodes that could map to the hole with dilation one. If the last of these nodes is not a leaf it cannot be mapped to the hole without dilating at least one edge. Detecting holes when they are created is not useful unless some form of backtracking is introduced. A hole may also consist of
CHAPTER 3. A GREEDY APPROACH

a set of connected nodes rather than a single node, further complicating detection and avoidance.

The conditions described above ensure that this greedy algorithm cannot guarantee optimal solutions for binary tree to hypercube mapping problems. It is possible to avoid dilation causing moves early on in the mapping process but it is not possible to prevent either the creation of holes or the occurrence of situations in which all moves are dilation forcing moves. Furthermore, it is not obvious whether avoiding such moves early in the algorithm will actually improve the quality of the mappings. In order to determine this it is necessary to actually implement and compare the different methods.

In total we implemented and tested five versions of the greedy algorithm — named greed-1 through greed-5. The first, greed-1, is an implementation of the algorithm as described by Chen and Gehringer. Each guest node has a unique integer label and ties are broken according to the value of the label. These labels are assigned arbitrarily thus ties are broken randomly. The only change we made to greed-1 was to improve its running time by eliminating unnecessary gain calculations. In the second version, greed-2, the tie breaking mechanism was changed by assigning guest node labels in breadth-first order starting from the first guest node mapped. Greed-3 and greed-4 also differ from greed-1 in the method used to resolve ties in the gain calculations. Instead of guest node order, greed-3 uses hypercube level order and greed-4 uses hypercube node order. Greed-5 also uses hypercube node order to resolve ties (since this was found to be the best of the first three variations). Greed-5 tries to avoid moves which indirectly dilate edges. It does this by using two of the methods described above: avoid mapping guest nodes to host nodes with fewer unmapped neighbours, and avoid reducing the number of unmapped neighbours of mapped host nodes below the number of their guest node’s unmapped neighbours. We also implemented a version of Lee and Aggarwal’s combined greedy/swap algorithm. This algorithm, known as G-Swap, finds an embedding
CHAPTER 3. A GREEDY APPROACH

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greed-1</td>
<td>Break ties arbitrarily</td>
</tr>
<tr>
<td>Greed-2</td>
<td>Break ties by tree breadth first order</td>
</tr>
<tr>
<td>Greed-3</td>
<td>Break ties by hypercube level order</td>
</tr>
<tr>
<td>Greed-4</td>
<td>Break ties by hypercube node order</td>
</tr>
<tr>
<td>Greed-5</td>
<td>Avoid indirect dilation</td>
</tr>
<tr>
<td>G-Swap</td>
<td>Greed-5 with mincut-based search</td>
</tr>
</tbody>
</table>

Table 3.1: Description of Greedy Algorithms

using greed-5, then attempts to improve that embedding using the mincut-based search algorithm described in the previous chapter. Table 3.1 contains a brief description of each the greedy algorithms.

Greed-5 deliberately tries to avoid dilation causing moves by keeping track of the number of unmapped neighbours of guest and host nodes. This also allows it to detect situations in which a move is forced. If guest node $g$ is mapped to host node $h$, a move will be forced when the number of unmapped neighbours of $g$ is equal to (or less than) the number of unmapped neighbours of $h$. Detecting these cases does not cost anything extra but can improve the execution time of the algorithm. When a forced move is found it can be made immediately thus eliminating an entire set of gain calculations.

3.4 Results

Algorithm 3.1 is an outline of Chen and Gehringer's greedy algorithm. Given an $N$ node tree, the outer while loop is executed $N$ times, and within the $i$th iteration, the gain is calculated $(N - i)^2$ times, resulting in an order $N^3$ algorithm. Based on the observation that nodes without mapped neighbours all have a gain of zero, many of the possible mapping pairs do not have to be considered. A careful rewrite of the algorithm can reduce it to $O(N^2 \log N)$.

Chen and Gehringer evaluate their algorithm using a hypercube host graph, and a
CHAPTER 3. A GREEDY APPROACH

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Number of Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
</tr>
<tr>
<td>Greed-1</td>
<td>1.82</td>
</tr>
<tr>
<td>Greed-2</td>
<td>1.63</td>
</tr>
<tr>
<td>Greed-3</td>
<td>1.56</td>
</tr>
<tr>
<td>Greed-4</td>
<td>1.56</td>
</tr>
<tr>
<td>Greed-5</td>
<td>1.22</td>
</tr>
<tr>
<td>G-Swap</td>
<td>0.93</td>
</tr>
</tbody>
</table>

Table 3.2: Greedy Algorithms – Extra Dilation

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Number of Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
</tr>
<tr>
<td>Greed-1</td>
<td>1.0744</td>
</tr>
<tr>
<td>Greed-2</td>
<td>1.0669</td>
</tr>
<tr>
<td>Greed-3</td>
<td>1.0622</td>
</tr>
<tr>
<td>Greed-4</td>
<td>1.0622</td>
</tr>
<tr>
<td>Greed-5</td>
<td>1.0405</td>
</tr>
<tr>
<td>G-Swap</td>
<td>1.0212</td>
</tr>
</tbody>
</table>

Table 3.3: Greedy Algorithms – Dilation Ratio

number of guest graphs: linear arrays, rings, hypercubes and almost fully connected graphs. They also present results for running the algorithm on trees, but do not indicate what type or distribution of trees were considered.

Lee and Aggarwal evaluate their combined greedy/swap approach using a hypercube host graph and a set of guest graphs from a low-level computer vision application. A total of five eight-node graphs and four sixteen-node graphs are tested. They give results for maximum dilation only. Our version of the combined greedy/swap algorithm of the same order, $N^2 \log N$, as the mincut-based search routine.

Tables 3.2 through 3.3 show the results of running the greedy algorithms on binary tree guest graphs and hypercube host graphs with 16 to 512 nodes. The binary trees were generated using the tree generator described in chapter one. Each algorithm was tested on 2000 binary trees of each size. Averages are shown for extra dilation in Table
3.2 and for dilation ratio in Table 3.3.

Figure 3.1 is a graph of the dilation ratio averages for each algorithm and for each size of binary tree. Of the four tie breaking methods, the random method and the breadth-first method produced the worst dilation ratio. The other two methods are almost equal with hypercube node order (Greed-4) performing slightly better than hypercube level order.

It is interesting to note that for all of the different greedy algorithms, the dilation ratio eventually decreases as the size of the graph increases (see Figure 3.1). Even the simplest version of the greedy algorithm avoids making bad choices (one which dilates an edge) until near the end. This is because all good choices must be exhausted before a bad choice will be made. What is striking is the small number of edges which are dilated. By the time all good choices are used up there are relatively few nodes remaining to be mapped, thus limiting the number of edges which are dilated. This, however, does not limit the maximum dilation of the embedding, since these edges may have the maximum possible dilation. The net result is that the total dilation grows at rate which is slower than the rate at which the size of the graph grows.

The eventual decrease in the dilation ratio may be explained by the relative sparseness of the binary tree. The degree of the hypercube increases logarithmically with the number of nodes, while the degree of the binary tree remains constant. The greedy algorithms are able to proceed farther before being forced to make a bad choice when the difference in degree is larger. The relative sparseness becomes significant at about dimension 7 where the dilation ratio begins to decrease.

The results show that, on average, greed-5 performs better than any of the other greedy algorithms. It is also better in terms of the number of optimal (defined in terms of Havel's conjecture) solutions found as shown in Table 3.4. Table 3.5 shows average execution times for each of the algorithms on an INMOS T800 Transputer. As well as
CHAPTER 3. A GREEDY APPROACH

Table 3.4: Greedy Algorithms - Percentage Optimal Solutions

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Number of Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
</tr>
<tr>
<td>Greed-1</td>
<td>34.25</td>
</tr>
<tr>
<td>Greed-2</td>
<td>38.80</td>
</tr>
<tr>
<td>Greed-3</td>
<td>42.05</td>
</tr>
<tr>
<td>Greed-4</td>
<td>42.05</td>
</tr>
<tr>
<td>Greed-5</td>
<td>63.50</td>
</tr>
<tr>
<td>G-Swap</td>
<td>75.75</td>
</tr>
</tbody>
</table>

Table 3.5: Greedy Algorithms - Average Time (milliseconds)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Number of Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
</tr>
<tr>
<td>Greed-1</td>
<td>0.0000</td>
</tr>
<tr>
<td>Greed-2</td>
<td>0.0000</td>
</tr>
<tr>
<td>Greed-3</td>
<td>0.0000</td>
</tr>
<tr>
<td>Greed-4</td>
<td>0.0000</td>
</tr>
<tr>
<td>Greed-5</td>
<td>0.0051</td>
</tr>
<tr>
<td>G-Swap</td>
<td>0.0369</td>
</tr>
</tbody>
</table>

producing better solutions, greed-5 is also faster than the other greedy algorithms. This is due to the fact that the number of candidate mapping pairs can be greatly reduced when the unmapped neighbours of mapped nodes are known. Avoiding dilation causing moves whenever possible does, in fact, produce better embeddings.

The combined greed/swap algorithm improved upon a significant number of the solutions found by the greed-5 algorithm alone. It may be possible to reduce the order of this algorithm from \( N^2 \log N \) by using a more efficient algorithm to find swaps which improve the embedding. This algorithm produces, on average, better solutions than any of the other greedy algorithms.

All of our tests have been performed on binary trees but, for the most part, the algorithms in this chapter can be applied to guest graphs other than binary trees. Some of the reductions in the number of gain calculations require that the guest graph be
acyclic. The speed of the greed-5 and greed/swap algorithms would also deteriorate if the degree of the guest graph was increased. It is difficult to say exactly how average dilation would be affected by higher degree and/or cyclic guest graphs but, it seems likely that the greedy algorithm would run out of good choices sooner, thus producing higher average dilation for such graphs. It would be interesting to find out if and at what point dilation ratio starts to decrease for higher degree trees.

![Figure 3.1: Dilation Ratio — Averages for Greedy Algorithms](image)
Monien and Sudborough [34] and Bhatt et al. [6] describe separator-based approaches to the problem of mapping binary trees to hypercubes. A separator is a set of nodes or edges which, when removed from a graph, divide it into two or more unconnected sets of nodes. Separators can be used to recursively split the nodes of a graph such that neighbour nodes in the original graph are assigned to nearby separator sets. Using this approach both [34] and [6] were able to bound the maximum dilation of binary tree to hypercube embeddings. Both Monien and Sudborough’s algorithm (M&S) and Bhatt’s et al. algorithm (BCLR) embed binary trees into hypercubes with maximum dilation $O(1)$. The BCLR algorithm has expansion 1 and $O(1)$ node congestion, while M&S has expansion $O(1)$.

4.1 Outline of the Algorithm

The BCLR algorithm has two basic steps, first separators are used to decompose the original binary tree ($T$) into a thistle tree ($TT$), then the thistle tree is embedded into a hypercube. A thistle tree is a complete binary tree which has a weight associated with
CHAPTER 4. A GRAPH SEPARATOR BASED APPROACH

Figure 4.1: Fifteen Node Thistle Tree

each node. The weight determines how many nodes of $T$ can be embedded into a node of $TT$. The thistle tree weight is set to $k - l$ where $k$ is the height of $TT$ and $l$ is the level of the node in the tree (see Figure 4.1). The total number of nodes which can be embedded into a thistle tree is $2^{k+1} - k - 2$. The thistle tree is embedded into a $2^{k+1}$ node hypercube. If the size of $T$ is between $2^{k+1} - k - 2$ and $2^{k+1}$ then the extra $k + 2$ nodes are excluded at the beginning of the algorithm and added to the embedding at the end.

Lemma 1 [6] Every $N$-node binary tree $T$ can be mapped onto an $N$-node complete binary tree $C$ so that at most $6 \log \frac{N}{T} + 18$ nodes of $T$ are mapped onto a node of $C$ at distance $l$ from the root, and so that any two nodes adjacent in $T$ are mapped to nodes at most distance three apart in $C$.

A binary tree $T$ is embedded into the thistle tree with dilation no more than eight. The tree is recursively bisected and successive sets of separator nodes are placed within successively lower levels of $TT$ according to Lemma 1. So far dilation is three but the thistle tree has an excess of nodes at the higher levels and a corresponding dearth at the lower levels. The extra $T$ nodes can be moved downward such that every thistle tree at level $l$ has $k - l$ nodes and the dilation is increased by no more than five.

The next step is to map the thistle tree onto the hypercube. First, the nodes of $T$ are redistributed so that there are at most two per thistle tree node. Then each $TT$ node
is mapped to two hypercube nodes. The thistle tree is mapped onto the hypercube by assigning labels to the nodes of $TT$ according to an inorder traversal of $TT$ (see Figure 4.2). This assigns each node of the lower subcube to a node of the thistle tree. Each $TT$ node is mapped to a second hypercube node — the corresponding node from the upper subcube. The embedding of $TT$ into $Q$ increases dilation by at most three.

If $T$ is of size between $2^{k+1} - k - 2$ and $2^{k+1}$ then the extra $k + 2$ nodes remain to be embedded. This is done by mapping the extra nodes next to their neighbours and rippling the displaced nodes along a path to an empty hypercube node. Within an $n$-dimensional hypercube $n$ node-disjoint paths can be found to connect any $n$ sources to any $n$ sinks [18] so embedding $k + 1$ of the extra nodes in this manner will increase dilation by one. The final extra node is mapped in the same way, increasing dilation by one again for a final dilation of thirteen.

Monien and Sudborough [34] claim to be able to embed arbitrary binary trees into their optimal hypercube with dilation five. Their algorithm starts by partitioning the original tree into a type two collision tree, $T^{(2)}$, which is a thistle tree which allows twice as many $T$ nodes at each $TT$ node. A binary tree $T$ is mapped to $T^{(2)}$ such that all edges in $T$ connect grandparents to grandchildren or cousins to cousins in $T^{(2)}$. The collision tree is, in turn, mapped onto the hypercube such that grandchildren are at most five away from grandparents and cousins are at most five away from cousins. This gives a
Algorithm: BCLR (Constructs a mapping of a binary tree $T$ onto a hypercube $Q$)

1. Decompose the tree $T$ into a thistle tree $TT$
   - Find a $O(\log n)$ node bisector of $T$
   - Place the bisector at the appropriate thistle tree node
   - Recursively bisect the two components of $T$ until only leaves remain

2. Embed the thistle tree into a hypercube $Q$
   - Mark one $T$ node in each $TT$ node as the central node
   - Distribute all other $T$ nodes along the trace
   - Embed the central nodes into the lower subcube of $Q$ according to the inorder embedding
   - Embed the trace nodes into the upper subcube of $Q$ according to the inorder embedding

3. Embed any extra nodes using the rippling procedure

Algorithm 4.1: BCLR Separator-based Algorithm

maximum dilation of five.

The M&S algorithm was not tested because they do not describe in detail how they achieve the partitioning of $T$ into the collision tree. Specifically, they do not include the proofs of their partitioning lemmas in their discussion. Type two collision trees have size $2^k - 2k - 4$ and embed into cubes of size $2^k$. Monien and Sudborough do not consider the case of trees of size between $2^k - 2k - 4$ and $2^k$. If these extra nodes are placed using the rippling technique given in [6] dilation will be increased by three.

4.2 Description of the Algorithm

In this section we describe, in detail, our implementation of the BCLR algorithm. The algorithm has two basic steps, decomposing $T$ into a thistle tree, then embedding the thistle tree into the hypercube. A third step is required if the tree has between $2^k - 2k - 4$ and $2^k$ nodes. This third step embeds the extra $k + 2$ nodes of $T$ into the hypercube. Algorithm 4.1 is an outline of the BCLR algorithm.


4.2.1 Binary Tree Decomposition

The binary tree $T$ is recursively bisected until it has been decomposed into single nodes. The separator nodes of each subtree are placed at the corresponding node of $TT$. First $T$ is divided into two subtrees $T_1$ and $T_2$ with the separator nodes placed at the root of $TT$. Then $T_1$ is bisected and the separator nodes placed at the left child of the root of $TT$. Similarly, $T_2$ is bisected and the separator nodes placed at the right child of the root of $TT$. This process is repeated for the subtrees of $T_1$ and $T_2$ until only single nodes remain. Before a subtree is bisected at level $l$, all nodes which are adjacent to the separator at level $l-2$ are removed from the subtree and placed at the thistle tree node. This ensures that nodes adjacent in $T$ will be mapped to nodes in $TT$ no more than three apart.

A three-colour separator is used to distribute the nodes which are next to separator nodes evenly among the subtrees. The size of this separator is given by Theorem 1.

Theorem 1 [9, 8] Colour each of the nodes of an $N$-node binary, $T$, with one of $k$ colours. Let $n_i$ be the number of nodes of colour $i$. By removing $k \log N$ or fewer nodes $T$ can be bisected into two components of sizes $[N/2]$ and $[N/2]$ such that each component has at least $[n_i/2]$ nodes of colour $i$.

Originally all nodes are coloured with colour A. At level $l$ all nodes of colour $l \mod 2$ are removed from the current tree and placed at the thistle tree node. Then every node of colour A which is next to separator at the previous level is coloured with colour $l \mod 2$. Finally, a three colour separator is used to bisect the remaining subgraph and the separator nodes are placed at the thistle tree node.

Only one change was made in the implementation of the BCLR algorithm. The number of nodes required to form a three colour bisector of an $N$-node binary tree is at most $3 \log N$. However, observation of the algorithm on trees with up to 512 nodes shows that rarely are more than $\log N$ nodes required. The recursive bisection part of
CHAPTER 4. A GRAPH SEPARATOR BASED APPROACH

Figure 4.3: A Collinear Layout of a Binary Tree

the algorithm was modified to place exactly \( k - l \) nodes in separators at level \( l \). On the rare occasion in which more nodes are required, the extra nodes are passed down the next level and coloured so that they are included in a separator within the next two levels. This eliminates the step of moving nodes from the higher level over-loaded thistle tree nodes to the under-utilized lower level thistle tree nodes.

4.2.2 Colour Separators

Two steps are required to find a \( 3 \log N \) node colour bisector of a three-coloured binary tree. First the nodes of the tree are placed in a collinear layout such that if the layout is split between any two nodes no more than \( \log N \) edges will be cut (see Figure 4.3). Then at most three such cuts are found which divide the tree into two components of sizes \( \lceil N/2 \rceil \) and \( \lfloor N/2 \rfloor \) such that each component has half of the nodes of each colour. One endpoint from each of the cut edges form the bisector set.

A collinear layout of an \( N \)-node binary tree, \( T \), is constructed as follows \([4]\). First an edge is removed from \( T \) such that neither subtree contains more than \( \lfloor \frac{2}{3}N \rfloor + 1 \) nodes. If either subtree contains more than \( \lfloor N/2 \rfloor \) nodes, that component is separated again by removing one edge in exactly the same way. This divides \( T \) into two or three components. If there are two components they are placed side by side in the layout. If there are three components, the one which is connected to both of the others is placed in the middle.
The subtrees of $T$ are recursively divided and placed until single nodes remain. It can be shown that the height of this layout is no more than $\log N$.

Theorem 2 is used to bisect the collinear layout of the nodes of $T$.

**Theorem 2** [18] For every two-ended string of $n$ pearls, $n_i$ of which are coloured $i$, $1 \leq i \leq k$, it is possible to divide the pearls into two sets, using no more than $k$ cuts, such that each set has a total of $\lceil n/2 \rceil$ or $\lfloor n/2 \rfloor$ pearls, and $\lceil n_i/2 \rceil$ and $\lfloor n_i/2 \rfloor$ pearls of colour $i$ for all $i$, $1 \leq i \leq k$.

Since the number of colours, $k$, is fixed at three, the cuts can be found in polynomial time simply by searching all possible combinations of three or fewer cuts until a valid set of cuts is found.

There are $\left( \binom{N}{1} + \binom{N}{2} + \binom{N}{3} \right)$ different sets of three cuts to be searched, making the layout bisection order $N^3$. The linear layout is $O(N)$, so finding a colour separator is an order $O(N^3)$ procedure.

### 4.2.3 Embedding the Thistle Tree

At every thistle tree node one of the $T$ nodes is marked as the central node and the remaining nodes are distributed along the trace: where the trace of a $TT$ node starts at its left child and follows the rightmost path from there down to a leaf. After this operation every $TT$ node has at most two $T$ nodes — the central node and at most one trace node. The thistle tree nodes are mapped onto hypercube nodes according to the inorder embedding shown in Figure 4.2.

The inorder embedding maps a height $k$ tree into a dimension $k$ hypercube. This is half the size of the dimension $k + 1$ hypercube which is required for the $2^{k+1} - k - 2$ thistle tree nodes. The central node is embedded into its corresponding inorder hypercube node. The central nodes occupy the lower subcube of the hypercube. The trace nodes
are embedded into the corresponding nodes of upper subcube. The inorder embedding has the property that the descendants of a node that lie no more than \( l \) levels below it occupy an \( l + 1 \) dimensional hypercube. The central nodes of \( T \) are no more than eight apart so they lie within a nine-dimensional hypercube.

### 4.2.4 Node Rippling

If the size of \( T \) is between \( 2^{k+1} - k - 2 \) and \( 2^{k+1} \) then the extra \( k + 2 \) nodes are mapped last. Each extra tree node is placed at a hypercube node next to one of its neighbours. Any displaced tree nodes are rippled along a path in the hypercube to an empty hypercube node. Greenberg [19] shows how to find \( n \) node-disjoint paths from any \( n \) sources to any \( n \) sinks within an \( n \)-dimensional hypercube. This method is used to find a set of non-intersecting paths along which to ripple the displaced nodes. The displaced nodes are the sources and the empty hypercube nodes are the sinks. Algorithm 4.2 is an outline of the node rippling procedure.

First the source/sink pair \((s, k)\) with the smallest Hamming distance is connected by one of the shortest paths between them. For \( n \geq 2 \) there are at least two sources so there must be a half cube which contains \( s \) and \( k \) and all nodes on the path. Call the subcube containing \( s \) and \( k \) the upper half, the other subcube the lower half, and the dimension between them \( i \). The next step is to find a path from every unconnected source and sink in the upper half to a surrogate node in the lower half. The entire process is repeated for the lower half until all the sources and sinks have been connected.

The surrogate of a source will either be its image through dimension \( i \), or if that image is also a source, then the image of one of its neighbours in the upper half. These neighbours must be disjoint from each other and also disjoint from the path from \( s \) to \( k \). There are at most \( n - 2 \) sources which require neighbours. For each of these sources it is always possible to find two neighbours disjoint from any of the other sources and also
CHAPTER 4. A GRAPH SEPARATOR BASED APPROACH

Procedure: Node Rippling

1. (Mark all sources and sinks)
   Mark every unmapped hypercube node as a sink
   For each unmapped tree node t
   {
     Find a neighbour of t which is mapped to a hypercube node q
     Mark q as a source
   }

2. (Find $n$ disjoint paths between $n$ source/sink pairs in $Q_n$)
   Choose the source $s$ and sink $k$ with the smallest Hamming distance
   Find a shortest path $p$ in $Q$ from $s$ to $k$
   Choose a dimension $i$ not on $p$
   Divide $Q$ into two subcubes, $S_u$ and $S_i$, along dimension $i$ such that $S_u$ contains $s$ and $k$
   For every source (sink) $s$ in $S_u$ find a surrogate in $S_i$ as follows:
   {
     Let $M_i(s)$ be the image of $s$ through dimension $i$
     If $M_i(s)$ is not a source (sink)
       Use $M_i(s)$ as the surrogate
     Else
       Use $M_i(n)$ as the surrogate — where $n$ is a neighbour of $s$ in $S_u$
       such that $M_i(n)$ is not a source (sink)
       $n$ is disjoint from $p$
       $n$ is disjoint from all other sources (sinks)
       $n$ is disjoint from all immediate neighbours of all other sources (sinks)
       Connect the source (sink) to its surrogate
   }
   Use the surrogates as new sources and sinks
   If not all sources and sinks are connected
   Repeat 2 with $S_i$ and the new sources and sinks

3. (Embed the unmapped $T$ nodes)
   For each path $p$ between source $s$ and sink $k$
   {
     Embed $t$ on $p$ next to $s$
     Ripple the displaced $T$ nodes along $p$ to $k$
   }

Algorithm 4.2: Node Rippling Procedure
disjoint from any of their two neighbours [19]. At least one of these neighbours must also be disjoint from the path from \( s \) to \( k \). Therefore every source and sink in the upper half has a surrogate in the lower half. The paths between sources and sinks through their surrogates form the \( n \) disjoint paths.

The rippling procedure is \( O((\log N)^2) \). There are \( \log N \) levels of recursion since one source and sink pair is connected at each level. At each level, \( i \), it may be necessary to find surrogates for \( (2 \log N) - i \) nodes.

### 4.3 Results

Tables 4.1 and 4.2 show the results of running the separator algorithm on trees ranging in size from 16 to 512 nodes. The algorithm produces dilation thirteen in the worst case, so for hypercubes of dimension less than thirteen, the worst dilation is equal to the dimension (ie. diameter). On average, the dilation is less than the dimension and appears to grow more slowly than the dimension. The total dilation, however, starts relatively high and grows with the size of the tree. This is because the algorithm makes no attempt to limit the number of edges which are dilated. Tables 4.3 and 4.4 show the results of running a random embedding algorithm on the same trees. The graph in Figure 4.4 shows how worst case dilation grows with respect to dimension for the two algorithms.

We expect that the performance of Monien and Sudborough's algorithm would be similar with respect to total dilation since, again, no attempt is made to limit the number of edges which are dilated.

The BCLR algorithm is \( O(N^3 \log N) \). The binary tree is bisected a total of \( \log N \) times, and the bisecting procedure is \( O(N^3) \). The results show that in practice the algorithm has a much better performance than \( N^3 \). This is because the colour separators are usually found in much less than \( O(N^3) \) time.
### Table 4.1: BCLR Separator Algorithm - Averages over 2,000 Trees

<table>
<thead>
<tr>
<th>Measure</th>
<th>Hypercube Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td>Dilation</td>
<td>3.39</td>
</tr>
<tr>
<td>Extra Dilation</td>
<td>14.29</td>
</tr>
<tr>
<td>Dilation Ratio</td>
<td>1.8801</td>
</tr>
<tr>
<td>Time (ms)</td>
<td>0.01</td>
</tr>
</tbody>
</table>

### Table 4.2: BCLR Separator Algorithm - Worst Case over 2,000 Trees

<table>
<thead>
<tr>
<th>Measure</th>
<th>Hypercube Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td>Dilation</td>
<td>4</td>
</tr>
<tr>
<td>Extra Dilation</td>
<td>26</td>
</tr>
<tr>
<td>Dilation Ratio</td>
<td>2.73</td>
</tr>
<tr>
<td>Time (ms)</td>
<td>0.01</td>
</tr>
</tbody>
</table>

### Table 4.3: Random Embedding - Averages over 2,000 Trees

<table>
<thead>
<tr>
<th>Measure</th>
<th>Hypercube Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td>Dilation</td>
<td>3.56</td>
</tr>
<tr>
<td>Extra Dilation</td>
<td>17.03</td>
</tr>
<tr>
<td>Dilation Ratio</td>
<td>2.0557</td>
</tr>
<tr>
<td>Time (ms)</td>
<td>0.00</td>
</tr>
</tbody>
</table>

### Table 4.4: Random Embedding - Worst Case over 2,000 Trees

<table>
<thead>
<tr>
<th>Measure</th>
<th>Hypercube Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td>Dilation</td>
<td>4</td>
</tr>
<tr>
<td>Extra Dilation</td>
<td>25</td>
</tr>
<tr>
<td>Dilation Ratio</td>
<td>2.67</td>
</tr>
<tr>
<td>Time (ms)</td>
<td>0.00</td>
</tr>
</tbody>
</table>
Figure 4.4: Dilation — Averages for Separator and Random Algorithms
Chapter 5
Tree Folding

In [36, 38] Wagner describes an algorithm for embedding binary trees into hypercubes. There are two versions of the algorithm: the first version [36] has dilation 1, load-factor 1 and expansion $O(\log N)$; the second version [38] has expansion 1, load-factor 1, low dilation and reasonable average dilation. Since we are concerned with expansion 1 embeddings only the second version of this algorithm was tested.

5.1 Outline of the Folding Algorithm

The tree-folding algorithm takes advantage of the recursive structure of the hypercube to embed an $2^n$-node graph $G$ into the $2^n$-node hypercube $Q_n$. While the algorithm is described for graphs with exactly $2^n$ nodes, it can be easily modified to apply to all graphs. The folding algorithm uses a special type of graph contraction operation known as a fold. Each time a graph is folded the number of nodes is reduced by half. The graph $G$ is embedded into $Q$ by repeatedly folding $G$ until only one node is left. Folding a graph corresponds to splitting a hypercube into two subcubes. In the process of folding unique $n$-bit labels are assigned to each of the nodes of $G$. These labels correspond to the nodes of the hypercube thus embedding $G$ into $Q$. The label of a node $g$ will be denoted $\varphi(g)$. 
A fold is a special type of graph contraction operation. An $N$-node graph $G$ is folded as follows. Group all of the nodes of $G$ into $N/2$ disjoint pairs $(u_i, l_i)$ $(1 \leq i \leq N/2)$. For each pair $i$ fold $u_i$ onto $l_i$ as follows: delete $u_i$ from the graph, delete the edge $(u_i, l_i)$ if it exists, and change all edges which were incident on $u_i$ to be incident on $l_i$ (see Figure 5.1). Folding all $N/2$ pairs of nodes yields a new graph with $N/2$ nodes. Repeatedly folding a graph will eventually produce a graph with a single node.

Every time a graph is folded half of the nodes $(u)$ are embedded into the upper subcube, and the other half $(l)$ are embedded into the lower subcube. That is, for each pair $i$, $u_i$ and $l_i$ will be labeled such that bit $d$ of $\varphi(u_i)$ is set to 1, bit $d$ of $\varphi(l_i)$ is set to 0, and $\varphi(u_i)$ and $\varphi(l_i)$ differ only in bit $d$ (where $d$ is the dimension along which the upper and lower subcubes are divided). Every fold is along a different dimension so $d$ varies from 0 to $\log N$ and after $\log N$ folds every node will have a unique $\log N$ bit label. Algorithm 5.1 is an outline of the folding algorithm. In the first step the graph is repeatedly folded in half. In the second step the labels are assigned by undoing the folds in reverse order.

Each time a graph is folded it is partially mapped to the hypercube. This is not a
CHAPTER 5. TREE FOLDING

General Folding Algorithm
(Constructs a mapping \( \varphi \) of a \( 2^n \)-node graph \( G \) into an \( n \)-dimensional hypercube \( Q \))

Step 1: (Repeatedly folds \( G \) until one node \( v_0 \) is left)
   For each dimension \( d \) from 0 to \( n \)
   {
      Group the nodes of \( G \) into a set of disjoint pairs \((u, l) \in F_d\)
      (The set of pairs \( F_d \) is known as the fold in dimension \( d \))
      For each pair in \( F_d \) fold \( u \) onto \( l \) as follows
      {
         Delete the node \( u \) from \( G \)
         Delete the edge \((u, l)\) (if it exists) from \( G \)
         Change all edges which were incident on \( u \) to be incident on \( l \)
      }
   }

Step 2: (Uses the folds \( F_d \) to embed \( G \) into \( Q \))
   Let \( \varphi(v) \) be the hypercube node that \( v \in V(G) \) is mapped to
   Set \( \varphi(v_0) \) to 0 (\( v_0 \) is the single remaining node of \( G \) after the folding step)
   For each dimension \( d \) (from \( n \) to 0)
   {
      For each pair \((u, l) \in F_d\)
      {
         Set \( \varphi(u) \) to \( \varphi(l) \)
         Set bit \( d \) of \( \varphi(u) \) to 1
      }
   }

Algorithm 5.1: General Folding Algorithm
CHAPTER 5. TREE FOLDING

partial mapping in the usual sense that a subset of the nodes are mapped. A partial mapping in this case means that a subset of the bits of the labels of every node is set. It is still possible to refer to the dilation of a partial mapping. The dilation of an edge \((u, v)\) under a partial mapping is equal to the number of bits in which \(\varphi(u)\) and \(\varphi(v)\) differ. Only those bits which have been set under the mapping are counted.

An \(N = 2^n\)-node binary tree is embedded into the hypercube by folded it in half \(n\) times. In the \(i\)th iteration \((1 \leq i \leq n)\) a constant time operation is required to fold each of the \(2^{n-1}\) pairs of nodes. The general tree-folding algorithm is therefore order \(N \log N\).

The general folding algorithm embeds any \(2^n\)-node graph into an \(n\)-dimensional hypercube with expansion 1. If the folds are chosen arbitrarily no bound can be placed on the dilation. The problem then is to choose the folds in order to minimize the dilation. It is not feasible to search all possible folds for one with the minimum dilation since there are \(\binom{N}{2}\) different ways to perform a single fold of an \(N\)-node graph. One approach is to use heuristics to limit the search, a second approach, described in the next section, is to find a class of binary trees for which low dilation folds are easily found.

5.2 Folding Strongly Balanced Binary Trees

Wagner [38] has conjectured that a class of binary trees known as strongly balanced binary trees are embeddable into the hypercube by the folding algorithm with expansion 1 and dilation 1 in polynomial time. The conjecture is that a very restricted type of fold, known as a path-fold, will suffice to fold strongly balanced trees with dilation 1. These folds can be found in polynomial time. Every binary tree can be converted with dilation 2 in linear time to a strongly balanced tree, so under the conjecture the algorithm embeds arbitrary binary trees with dilation 2.

A binary tree is strongly balanced if it has an even number of nodes and none of its degree three nodes has odd parity, where the parity of a degree three node is defined
as follows. The weight of a node $n$, denoted $W(n)$, in a rooted tree is equal to 1 plus the number of descendants of $n$. The parity of a degree three node $v$ is determined by calculating the weights of its three neighbours. If all three neighbours have an odd weight then $v$ is odd. If two of the neighbours have an even weight (the third may have an even or odd weight), then $v$ is even. The other possibility, two neighbours with odd weight and one with even weight cannot occur in a binary tree with an even number of nodes. If all of the degree three nodes of a binary tree are even then the tree is strongly balanced.

The algorithm folds a strongly balanced tree in half such that the resulting graph is also a strongly balanced tree and the dilation of the fold is 1. First the tree is rooted by choosing an arbitrary node of degree 1 or 2 for the root. Then the rooted tree is folded in two stages. In the first stage the tree is traversed in depth-first order and a path-fold is found for every even weight node. In the second stage the path-folds are combined to fold the tree in half. The entire two stage procedure is applied recursively to search for a complete dilation 1 embedding of a $2^n$-node strongly balanced binary tree into the $n$-dimensional hypercube. Algorithm 5.2 is an outline of the algorithm for finding a fold of a strongly balanced tree.

A path-fold of a node $v_1$ in a rooted tree is a fold of an even length path which starts at $v_1$ and proceeds down the tree. The path-fold of the $n$-node path $p = (v_1, \ldots, v_n)$ is constructed by folding $v_{n-i+1}$ onto $v_i$ for all $i \leq \frac{n}{2}$ and then folding the subtrees of each pair. Figure 5.2 shows a path-fold of a four node path: $v_2$ is folded onto $v_3$, $v_4$ is folded onto $v_1$, and the subtree of $v_4$ ($s_4$) is folded onto the subtree of $v_1$ ($s_1$). A valid path-fold is one which has dilation 1 and creates no odd degree three nodes. The algorithm searches for a valid path-fold for all even weight nodes $v$ in the tree. Each even length downward paths $p$ from $v$ is considered if $p$ is of length 2 or if the last node of $p$ has degree less than three. All such paths are examined until the first valid path-fold is found. Algorithm 5.3 is an outline of the procedure for finding a valid path-fold.
Algorithm: Fold a Strongly Balanced Binary Tree
(Finds a dilation 1 fold $F$ of an $N$-node strongly balanced binary tree $T$
into an $N/2$-node strongly balanced binary tree $T'$)

Stage 1: (Finds valid path-fold for every even weight node in $T$)
Pick an arbitrary leaf node $r$ of $T$ and root $T$ at $r$
Calculate the weight $W(v)$ of each node $v \in V(T)$
where $W(v) = s + 1$
and $s$ is the number of descendants of $v$
Visit each even-weight node $v$ in depth-first order
{
  Search all even-length downward paths from $v$
  until a valid path-fold is found
  using the path-fold procedure (Algorithm 5.3)
}

Stage 2: (Combines the path-folds to fold $T$ into $T'$)
Apply the path-fold of the root
Repeat until all nodes have been folded
{
  Apply the path-fold for every node $v$
  where $v$ has not yet been folded
  and $v$ is adjacent to a node which has been folded
  If the parent of $v$ was placed in the upper subcube
    Then place $v$ in the lower subcube
  If the parent of $v$ was placed in the lower subcube
    Then place $v$ in the upper subcube
}

Algorithm 5.2: Fold a Strongly Balanced Binary Tree
In order to construct a valid path-fold of a path $p$ the subtrees of each pair of folded nodes $(v_1, v_j)$ are also folded. We define the subtree of a node $v$ on $p$ as consisting of all children of $v$ which are not on $p$ and all descendants of those children. A fold of two subtrees $s_1$ and $s_2$ (called a subtree fold) is formed by pairing the nodes at each level of $s_1$ with the nodes at the same level of $s_2$, for each level, starting at the root. Figure 5.3 shows the subtree fold of the two subtrees $s_1$ and $s_4$ from Figure 5.2. A valid subtree fold is one which has dilation 1 and creates no odd degree three nodes. All combinations of pairs at each level are examined until a valid subtree fold of $s_1$ and $s_2$ is found, or until it is determined that no such fold exists. Any node in $s_1$ or $s_2$ which already has its own valid path fold may be excluded from the subtree fold. Otherwise all nodes of the subtrees of each pair from $p$ must be included in a subtree fold. These subtree folds are, in turn, included as part of the path-fold of $v_1$. Algorithm 5.4 is an outline of the procedure for finding a valid subtree fold.

Once the valid path-folds have been found, a subset of them are combined to fold the tree in half. First the path-fold of the root is applied to the tree. This produces a new tree with some folded nodes and some unfolded nodes. For each unfolded node $u$, the
Procedure: Find a Valid Path-Fold
(Finds a valid path-fold $F$ for an even-length downward path, $p = (v_1, \ldots, v_n)$)
(Returns FALSE if $p$ has no valid path-fold)

For each pair $(v, v') \in \{(v_1, v_n), (v_2, v_{n-1}), \ldots, (v_{n/2}, v_{(n/2)+1})\}$

\{ 
  If folding $v$ with $v'$ produces an odd degree three node
  or a node with degree greater than 3
  Then Return FALSE
  Find a valid subtree fold $F_*$ of $(v, v')$ using the procedure in Algorithm 5.4
  If a valid subtree fold was found
    Then Add $F_*$ to $F$
    Else Return FALSE
\}

Algorithm 5.3: Valid Path-Fold Procedure

![Figure 5.3: Subtree Fold of Two Subtrees](image)
CHAPTER 5. TREE FOLDING

Procedure: Find a Valid Subtree-Fold

(Recursively builds a valid subtree fold $F_s$ of two nodes $v$ and $v'$)
(Returns FALSE if $v$ and $v'$ do not have a valid subtree fold)

If folding $v$ with $v'$ creates an odd degree three node
   Then Return FALSE
If $v$ and $v'$ are leaves
   {
      Add the pair $(v, v')$ to $F_s$
      Return TRUE
   }

Group the children of $v$ and $v'$ into a disjoint set of pairs and singletons such that
1. Every singleton has a valid path fold
2. Every pair has a valid subtree fold
3. No pair contains two nodes with the same parent
If such a grouping exists Then
   {
      Add $(v, v')$ to $F_s$
      Return TRUE
   }
Else Return FALSE

Algorithm 5.4: Valid Subtree-Fold Procedure
path-fold of $u$ is applied if $u$ is adjacent to an unfolded node (see Algorithm 5.2). This process is repeated until every node has been folded.

The original depth first traversal takes $O(N)$ time for an $N$-node tree. At each node every even length path may have to be examined so this is another $O(N)$ operation. Finding valid subtree folds may also require an $O(N)$ operation making the folding operation $O(N^3)$. The tree is folded $\log N$ times so the entire algorithm is $O(N^3 \log N)$.

### 5.3 Folding Arbitrary Binary Trees

In order to use the tree-folding algorithm on an arbitrary binary tree $T$ it is necessary to first convert $T$ into a strongly balanced tree. Any odd degree three node $v$ can be converted into an even node by shifting one edge incident on $v$ to a node adjacent to $v$. Call the three neighbours of $v$ $v_1, v_2$ and $v_3$. The edge $(v, v_1)$, for example, can be shifted by deleting $(v, v_1)$ and adding either $(v_1, v_2)$ or $(v_1, v_3)$. Any one of the three edges may be shifted, each in two possible ways. If shifting an edge creates a degree four node the shift can be propagated along a path until a degree 1 or degree 2 node is reached. Figure 5.4 is an example of converting a binary tree into a strongly balanced binary tree by shifting edges.

Any binary tree $T$ can be converted into a strongly balanced tree $T_{sb}$ by shifting an edge or set of edges for every odd degree three node. If $T_{sb}$ is embedded into the hypercube with dilation 1, then every shifted edge will have dilation 2 (provided no edge is shifted more than once). A set of shifts is said to be overlapping if any edge is shifted more than once. The ideal would be to convert $T$ by shifting as few edges as possible with non-overlapping shifts.

We know of no polynomial time algorithm which finds the minimum number of non-overlapping shifts required to make an arbitrary binary tree strongly balanced. Wagner [38] uses a method which guarantees that no shifts overlap and does a reasonable job
of minimizing the number of shifts. First $T$ is rooted at a randomly selected node $r$. The tree is traversed in depth-first order and every time an odd degree three node $v$ is encountered one of the downward edges of $v$ is shifted along a path to the nearest degree 1 or 2 descendant of $v$. When a node $v$ is converted edges lower in the tree are the only ones which get shifted. However, since $v$ becomes a degree 2 node as a result any shifts which begin at an ancestor of $v$ will terminate at $v$. This algorithm converts a tree without shifting any edge more than once. Therefore if all strongly balanced trees are embeddable with dilation 1 then all binary trees are embeddable with dilation 2.

The conversion operation is only $O(N)$ and is performed only once, so it does not affect the order of the complete tree-folding algorithm.

## 5.4 Results

The tree-folding algorithm was run on 6 sets of 2,000 randomly generated trees ranging in size from 16 to 512 nodes. The average results are summarized in Table 5.1 and the worst case results in Table 5.2. The worst case results show that dilation was never more than 2. This provides empirical evidence of the conjecture that all strongly balanced trees are
embeddable with dilation 1. Further evidence was gathered by running the algorithm on 4 sets of 10,000 trees of sizes 16 to 256 nodes. Each of these trees was converted to a strongly balanced tree which was in turn embedded into the hypercube with dilation 1.

Table 5.2 also shows that in the worst case for every size of tree half of the edges were dilated. This is because the first tree of each sample is the \(2^n\)-node complete tree with no degree 2 nodes. At least half of the edges must be shifted in order make this tree strongly balanced. No tree required more edge shifts than this. This gives an empirical upper bound on the total dilation which is equal to the number of edges minus one.

The dilation ratio grows very slowly in both the average and worst cases. Implying that the total dilation grows slightly faster than the number of edges as the size of the graph increases. It may be possible to improve the average case if a better tree conversion algorithm can be found. As well as growing slowly the average dilation starts at a relatively low value and outperforms most of the other algorithms in this respect.

The tree-folding algorithm embeds trees with a very low dilation and a relatively low average dilation. On average the algorithm appears to run faster than indicated by its worst case analysis. This is because the search for a valid path-fold usually has to examine only a few possibilities. The algorithm uses the recursive nature of the hypercube and also takes advantage of the structure of the binary tree. Its only disadvantage is that it relies on a conjecture and therefore may yet fail to embed some trees. This, however, appears to be a rare event. Also there is still considerable flexibility in the algorithm which would allow such trees to be embedded by increasing the dilation.

The results show a clear correlation between sparseness and total dilation for the folding algorithm. Trees with fewer degree three nodes tend to require fewer edge shifts to make them strongly balanced. Since only shifted edges are dilated, sparse trees have lower total dilation. The graph in Figure 5.5 plots average dilation ratio against sparseness for 512 node trees. The results are similar for all other sizes of tree.
Figure 5.5: Total Dilation — Averages for 512 Node Trees
# CHAPTER 5. TREE FOLDING

<table>
<thead>
<tr>
<th>Measure</th>
<th>Hypercube Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td>Dilation</td>
<td>1.91</td>
</tr>
<tr>
<td>Extra Dilation</td>
<td>2.12</td>
</tr>
<tr>
<td>Dilation Ratio</td>
<td>1.0986</td>
</tr>
<tr>
<td>Time (ms)</td>
<td>0.02</td>
</tr>
</tbody>
</table>

Table 5.1: Tree Folding Algorithm - Averages over 2,000 Trees

<table>
<thead>
<tr>
<th>Measure</th>
<th>Hypercube Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td>Average Dilation</td>
<td>2</td>
</tr>
<tr>
<td>Extra Dilation</td>
<td>7</td>
</tr>
<tr>
<td>Dilation Ratio</td>
<td>1.47</td>
</tr>
<tr>
<td>Time (ms)</td>
<td>0.03</td>
</tr>
</tbody>
</table>

Table 5.2: Tree Folding Algorithm - Worst Case over 2,000 Trees
Chapter 6

Results and Conclusions

6.1 Summary

We set out, at the beginning, to examine possible solutions to the embedding problem. The problem of embedding a guest graph $G$ into a host graph $H$ arises in the process of mapping a parallel program to a multicomputer. The communication structure of the program is represented by $G$, the interconnection network of the multicomputer is represented by $H$ and the goal is to embed $G$ into $H$ such that some measure of the communication cost is minimized. Several different types of embedding algorithms have been proposed but to date no attempt had been made to provide a comprehensive comparison of these algorithms. Our approach was to evaluate the algorithms empirically using hypercube host graphs and binary tree guest graphs.

The hypercube was used as a host graph because it has several properties which make it a good choice for multicomputer interconnection. In fact, several different hypercube based parallel machines have been constructed. Binary trees were used as guest graphs because they are useful and recurring computational structures. Binary trees are interesting also because they lie on the edge of what is known about the complexity of the embedding problem. The embedding problem for general trees and hypercubes is known to be NP-complete; for binary trees the complexity is still open. In addition, there exists
a good mix of theoretical and heuristic approximation algorithms for the binary tree to hypercube embedding problem.

The communication cost of an embedding is reflected by its dilation or its average dilation: large dilation or average dilation implies greater slow down due to communication overhead. Which of these two measures is more important depends on the nature of communication in the multicomputer. We evaluated the algorithms in terms of both of these measures. Dilation ratio, the average dilation divided by the optimal average dilation was used as a measure of average dilation.

Five of the algorithms are adaptations of general heuristics. Simulated annealing is a combinatorial optimization technique which has been used by several researchers to solve various forms of the mapping problem. We implemented a version of simulated annealing developed by Ramanujam et al. [31]. Two different algorithms both based on the Lin-Kernighan minimum cut heuristic have been developed by Ercal [15]. One is recursive mincut bipartitioning and the other is mincut-based search. These are known as Mincut and MBS-2 respectively. MBS-2 is a modified version of mincut-based search which matches the parities of the weights of the guest and host nodes. The mincut and simulated annealing algorithms were originally designed to map large guest graphs to smaller host graphs by assigning more than one guest node per host node such that the computational load is balanced. These algorithms are equally applicable to the case where the sizes of the guest and host graphs are equal.

Chen and Gehringer [13] proposed a simple greedy approach to the embedding problem. We refined and improved their greedy algorithm to take advantage of the structure of binary trees. Our version of the greedy algorithm, called Greed-5, avoids creating dilated edges wherever possible. MBS-2 and greed-5 were combined in a way suggested by Lee and Aggarwal [26]: the greedy algorithm is used to produce an embedding which is used as an initial embedding for the mincut-based search algorithm.
These five algorithms, Annealing, Mincut, MBS-2, Greed-5 and Greed/MBS, are general embedding algorithms and are not specific to the binary-tree to hypercube mapping problem. Annealing and MBS-2 are the most general of the algorithms, both are applicable to any type of guest or host graph. The simpler versions of the greedy algorithm are also applicable to general host and guest graphs, but greed-5 requires that the guest graph be acyclic and of low degree. Mincut will embed general guest graphs but relies on the recursive structure of the hypercube host graph.

The remaining two algorithms are more specific to the binary-tree to hypercube mapping problem. The separator-based algorithm of Bhatt et al. [6], known as BCLR, is applicable to guest graphs of bounded degree and small-sized separators. This algorithm has a constant bound on the dilation. Wagner's [38] tree-folding algorithm was designed specifically for the binary-tree to hypercube mapping problem.

### 6.2 Results

The algorithms were tested with hypercube host graphs of sizes ranging from 16 to 1,024 nodes. For each size cube a random sample of 2,000 binary trees was generated to be used as guest graphs. Each algorithm was tested using exactly the same set of trees. Some of the slower algorithms were not tested on the larger hypercubes. Averages and maximums of dilation and dilation ratio were collected for each algorithm and each size of hypercube.

Table 6.4 and the graph in Figure 6.1 show the results for dilation ratio. The dilation ratio was computed for every embedding and the averages taken for each size hypercube. On average, the greedy algorithms perform better than any of the other types of algorithm. The folding algorithm produced the next lowest average dilation ratio after the greedy algorithms. Next was MBS-2, then mincut, annealing and finally BCLR. The dilation ratio's of BCLR algorithm are too large to appear in the graph in Figure 6.1,
CHAPTER 6. RESULTS AND CONCLUSIONS

this is not unexpected since this algorithm was never intended to produce small average dilation. The combined greedy MBS algorithm produces only a slight improvement over the greed-5 algorithm with a significant increase in the time.

The average dilation and dilation ratio both increase with the size of the tree for all of the algorithms except the greedy algorithms. The change in average dilation depends on the rate at which the extra dilation changes in relation to the number of edges since:

\[ d_a = d_t / |E(G)| \]
\[ d_t = d_a + |E(G)|. \]

If extra dilation increases faster than the number of edges then average dilation also increases; if the extra dilation increases more slowly than the number of edges then average dilation decreases. The results show that extra dilation grows faster than the size of the tree for all except the greedy algorithms. Dilation ratio is total dilation over optimal total dilation. Since optimal total dilation stays very close to constant for all sizes of trees, dilation ratio changes in the same way as extra dilation.

The dilation ratios were also compared directly tree by tree. These direct comparisons show that for all trees larger than 128 nodes the greed/MBS algorithm embedded every tree with the lowest dilation ratio (Table 6.1). For trees smaller than 256 nodes the other algorithms fare a little better, but greed/MBS still finds the lowest dilation for the largest percentage of trees. The results are similar, but not quite as good, for the greed-5 algorithm (Table 6.2).

Table 6.5 and the graph in Figure 6.2 show the worst case results for dilation ratio. The highest dilation ratio for each size hypercube is shown. Again the greedy algorithms produced consistently better results than any of the other algorithms. The worst case dilation ratio remains fairly constant for all of the algorithms except simulated annealing.

The BCLR algorithm has the best asymptotic behaviour with respect to dilation. However, Table 6.6 and Figure 6.3 show that the folding algorithm has, on average, the lowest dilation. The dilation was computed for every embedding and the averages taken for each size hypercube. For the range of trees tested the BCLR algorithm produces
relatively high dilation. It would be reasonable to expect dilation to continue to grow linearly with the hypercube dimension for all of the algorithms except for the BCLR and folding algorithms. The folding algorithm has a maximum dilation of 2 for every tree tested. BCLR will eventually produce a maximum dilation of at most 15. Among the algorithms with linear dilation, the greedy algorithms, especially Greed/Swap, produced the best results.

6.3 Conclusions

Our experiments show that the greedy approach produces the lowest average total dilation of all algorithms tested. This is because the greedy algorithm uses a search space representation which enables it to take advantage of the relative sparseness of binary trees. None of the other algorithms utilize this property. For the greedy algorithms dilation ratio actually starts to decrease as the trees grow large. This is because the relative sparseness of the binary tree increases with the size of the graphs — the degree of the binary tree remains constant while the degree of the hypercube grows logarithmically.

The combined greedy/MBS algorithm has average dilation lower than that of greed-5 but is correspondingly slower. Even the simplest of the greedy algorithms has lower average dilation than any of the other types of algorithms. Simulated annealing, which has often been used to solve the embedding problem has the worst performance in terms of both average dilation and time.

We would expect that the greedy algorithms would not perform as well on more general graphs since these algorithms seem to depend on the relative sparseness of the binary tree. It would be interesting to examine the behaviour of the greedy algorithms on trees of higher degree and on cyclical graphs. It would also be possible to extend the greedy approach to solve the mapping problem for guest graphs which are larger than the host graph.
For dilation, the folding algorithm produces very close to optimal results. The results are optimal in the sense that no tree was embedded with dilation greater than 2, however, some of these trees have known dilation 1 embeddings. It may be that not all strongly-balanced are foldable with dilation 1, but this appears to be a rare event. Even if such trees do exist the folding algorithm can be modified to embed them with possibly increased dilation. The only other algorithm which produces constant dilation is the BCLR algorithm, but the constant is relatively high.

It is possible to trade off dilation against average dilation. The greedy algorithms produce embeddings with a low number of dilated edges but arbitrarily large dilation. A rippling procedure similar to that used in the BCLR algorithm may be used to reduce the dilation of the highly dilated edges while increasing the total number of dilated edges, thus reducing the overall dilation while possibly increasing average dilation.

A similar type of trade-off occurs with the folding algorithm. Here dilation is limited but there exist trees for which the dilation ratio must be at least 1.5. That is, there is at least one binary tree of every size $2^n$ which requires that half of its edges be shifted to make the tree strongly balanced. This is the $2^{n-1}$ node complete binary tree with an extra node attached to the root.

Our experience indicates that the binary tree to hypercube mapping problem is very difficult. Besides examining approximation algorithms we also attempted to find a polynomial algorithm without success. Further work needs to be done to firmly establish the complexity of this problem.
CHAPTER 6. RESULTS AND CONCLUSIONS

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Number of Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
</tr>
<tr>
<td>Annealing</td>
<td>490</td>
</tr>
<tr>
<td>Mincut</td>
<td>354</td>
</tr>
<tr>
<td>MBS-2</td>
<td>1505</td>
</tr>
<tr>
<td>BCLR</td>
<td>0</td>
</tr>
<tr>
<td>Folding</td>
<td>766</td>
</tr>
</tbody>
</table>

Table 6.1: Lowest Dilation Ratio by Direct Comparison (Greed/MBS)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Number of Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
</tr>
<tr>
<td>Annealing</td>
<td>508</td>
</tr>
<tr>
<td>Mincut</td>
<td>366</td>
</tr>
<tr>
<td>MBS-2</td>
<td>1530</td>
</tr>
<tr>
<td>BCLR</td>
<td>0</td>
</tr>
<tr>
<td>Folding</td>
<td>786</td>
</tr>
</tbody>
</table>

Table 6.2: Lowest Dilation Ratio by Direct Comparison (Greed-5)
CHAPTER 6. RESULTS AND CONCLUSIONS

Table 6.3: Dilation — Worst Case Over 2,000 Trees of Each Size

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Number of Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
</tr>
<tr>
<td>Annealing</td>
<td>3</td>
</tr>
<tr>
<td>Mincut</td>
<td>4</td>
</tr>
<tr>
<td>MBS-2</td>
<td>4</td>
</tr>
<tr>
<td>BCLR</td>
<td>4</td>
</tr>
<tr>
<td>Greed-5</td>
<td>4</td>
</tr>
<tr>
<td>Greed/MBS</td>
<td>3</td>
</tr>
<tr>
<td>Folding</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 6.3: Dilation — Worst Case Over 2,000 Trees of Each Size
CHAPTER 6. RESULTS AND CONCLUSIONS

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annealing</td>
<td>1.0985</td>
<td>1.1718</td>
<td>1.2440</td>
<td>1.2933</td>
<td>1.4011</td>
<td>1.5784</td>
<td>—</td>
</tr>
<tr>
<td>Mincut</td>
<td>1.1230</td>
<td>1.1751</td>
<td>1.2256</td>
<td>1.2776</td>
<td>1.3193</td>
<td>1.3670</td>
<td>—</td>
</tr>
<tr>
<td>MBS-2</td>
<td>1.0365</td>
<td>1.1096</td>
<td>1.1778</td>
<td>1.2440</td>
<td>1.2940</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>BCLR</td>
<td>1.8801</td>
<td>2.0855</td>
<td>2.2470</td>
<td>2.3884</td>
<td>2.4566</td>
<td>2.5073</td>
<td>—</td>
</tr>
<tr>
<td>Greed-5</td>
<td>1.0405</td>
<td>1.0620</td>
<td>1.0657</td>
<td>1.0657</td>
<td>1.0574</td>
<td>1.0495</td>
<td>1.0434</td>
</tr>
<tr>
<td>Greed/MBS</td>
<td>1.0212</td>
<td>1.0377</td>
<td>1.0437</td>
<td>1.0450</td>
<td>1.0411</td>
<td>1.0358</td>
<td>—</td>
</tr>
<tr>
<td>Folding</td>
<td>1.0986</td>
<td>1.1173</td>
<td>1.1330</td>
<td>1.1526</td>
<td>1.1583</td>
<td>1.1657</td>
<td>1.1920</td>
</tr>
</tbody>
</table>

Table 6.4: Dilation Ratio — Average Over 2,000 Trees of Each Size

Figure 6.1: Dilation Ratio — Averages
CHAPTER 6. RESULTS AND CONCLUSIONS

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Number of Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
</tr>
<tr>
<td>Annealing</td>
<td>1.47</td>
</tr>
<tr>
<td>Mincut</td>
<td>1.47</td>
</tr>
<tr>
<td>MBS-2</td>
<td>1.40</td>
</tr>
<tr>
<td>BCLR</td>
<td>2.73</td>
</tr>
<tr>
<td>Greed-5</td>
<td>1.40</td>
</tr>
<tr>
<td>Greed/MBS</td>
<td>1.27</td>
</tr>
<tr>
<td>Folding</td>
<td>1.47</td>
</tr>
</tbody>
</table>

Table 6.5: Dilation Ratio — Worst Case Over 2,000 Trees of Each Size

![Graph showing dilation ratio across different algorithms and dimensions](image)

Figure 6.2: Dilation Ratio — Worst Cases
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Number of Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
</tr>
<tr>
<td>Annealing</td>
<td>1.99</td>
</tr>
<tr>
<td>Mincut</td>
<td>2.12</td>
</tr>
<tr>
<td>MBS-2</td>
<td>2.01</td>
</tr>
<tr>
<td>BCLR</td>
<td>3.39</td>
</tr>
<tr>
<td>Greed-5</td>
<td>1.89</td>
</tr>
<tr>
<td>Greed/MBS</td>
<td>1.73</td>
</tr>
<tr>
<td>Folding</td>
<td>1.91</td>
</tr>
</tbody>
</table>

Table 6.6: Dilation — Average Over 2,000 Trees of Each Size

Figure 6.3: Maximum Dilation — Averages
CHAPTER 6. RESULTS AND CONCLUSIONS

Figure 6.4: Time — Averages
Bibliography


