Embedding all Binary Trees in the Hypercube

by

A.S. Wagner

Technical Report 90-34
November, 1990

Department of Computer Science
University of British Columbia
Vancouver, B.C. V6T 1W5

Embedding all binary trees in the hypercube*

A.S. Wagner

Department of Computer Science,
University of British Columbia,
Vancouver, British Columbia, Canada, V6T 1W5.

Abstract

An $\mathcal{O}(N^2)$ heuristic algorithm is presented that embeds all binary trees, with dilation 2 and small average dilation, into the optimal sized hypercube. The heuristic relies on a conjecture about all binary trees with a perfect matching. It provides a practical and robust technique for mapping binary trees into the hypercube and ensures that the communication load is evenly distributed across the network assuming any shortest path routing strategy. One contribution of this work is the identification of a rich collection of binary trees that can be easily mapped into the hypercube.

1 Introduction

The problem of allocating processes to processors in a multiprocessor system is known as the mapping problem [3]. If the communication pattern of the application is given as a task graph, where nodes denote processes and edges denote communication between processes, then this problem can be modeled as a graph embedding problem. There are many variations of this problem depending on whether the communication graph is static or dynamic, directed (i.e precedence constraints) or undirected, and weighted (nodes or edges) or unweighted. The problem also depends on the cardinality and topology of the communication graph versus that of the underlying network. We consider the problem of mapping statically known tree-structures, with unweighted edges, into a hypercube multiprocessor where the cardinality of the tree and the hypercube match.

Hypercube multiprocessors are one of the most popular types of parallel machines. They are cost-effective and current technology makes it both technically and economically feasible to build hypercubes with a large number of nodes. From a programming perspective, the hypercube's recursive structure and the fact that it contains as a subgraph

*This work was supported by the Natural Sciences and Engineering Research Council of Canada.

computational structures like rings, 2-d meshes, and higher dimensional meshes makes it suitable for many problems. These regular structures, however, do not capture the irregular computation and communication structure of many algorithms, like for instance binary trees. The problem of mapping these structures in the hypercube remains.

Trees are an important class of computational structure. As data structures they are easily manipulated and there are many operations efficiently performed on them. As computation graphs they are the structure underlying divide and conquer problem solving strategies, functional and logic programming, and algorithms for searching a problem's solution space, including NP-complete problems.

The technique presented in this paper is a two step process to embed all binary trees into the optimal sized hypercube. First, the tree is mapped into a member of a rich family of binary trees, trees containing a perfect matching, which we call strongly balanced trees. Second, an algorithm is given for mapping all strongly balanced trees into the hypercube. The second step relies on a conjecture about all strongly balanced trees.

In the next section we formally define the problem, the costs used to measure the goodness of a mapping, and review existing algorithms. In Section 3 we define and discuss the properties of strongly balanced trees. In Section 4 we give an algorithm for mapping all binary trees into strongly balanced binary trees. In Section 5 we present an algorithm for mapping all strongly balanced trees into the hypercube.

2 Preliminaries

We consider undirected graphs, $G = (V, E)$, specifically unrooted binary trees. The tree algorithms we describe, however, operate on binary trees rooted at an arbitrarily chosen leaf. To avoid confusion, with respect to trees, the term *node* will be used to refer to a vertex of a rooted tree. In a rooted tree, two nodes u, v are *related* when v is the descendant of u or vice versa. The *weight* of a subtree is taken to be the number of nodes in the subtree. Throughout, the degree of a node in a tree will be the degree of the vertex in the corresponding unrooted tree.

The *n-dimensional boolean hypercube*, H_n, is the graph with 2^n vertices labelled $\{0, 1, \ldots 2^n - 1\}$ with an edge between two vertices whenever the binary representations of their labels differs by a single bit. The term node will be used to refer to the label of a vertex of H_n. The *Hamming distance* between two nodes is the number of bits in which the nodes differ.

The earliest work in this area is by Havel and Morávek [7] who tried to characterize the spanning trees of the hypercube. They conjectured that:

Conjecture 1 *Every balanced binary tree with 2^n vertices spans H_n.*

A tree T is *balanced* when there exists a bipartition of the vertices of T (i.e. a two colouring) into two equal sized sets such that no edge lies entirely within one of the parts. Several

subfamilies of balanced binary trees were shown to span the hypercube. These include quasistars [7], double rooted complete binary tree [8], and balanced one-legged caterpillars [6] (see Figure 1). It has, however, been difficult to extend these results to richer families of

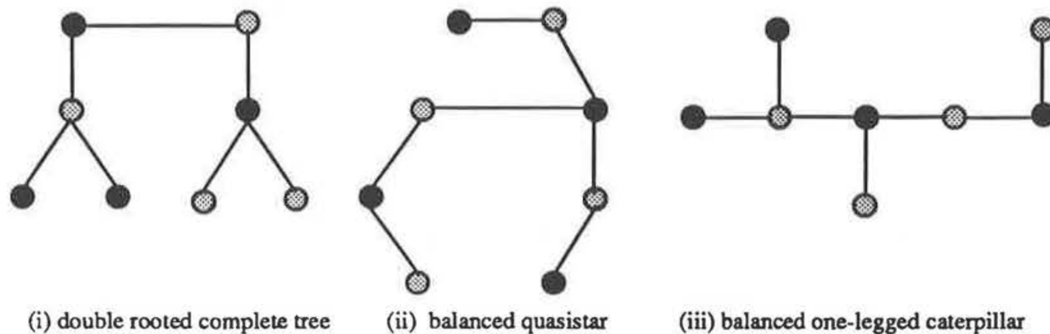

Figure 1: *Examples of families of trees which span hypercubes*

trees.

The problem of minimizing the number of dimensions required so that a given tree was a subgraph of the hypercube has also been considered. Havel and Morávek [7] and Afrati, Papadimitriou, and Papageorgiou [1] noticed that every tree is embeddable in a sufficiently large hypercube, but for a tree with N vertices, the minimum number of dimensions varies from $\log N$ to $N-1$. It was shown by Wagner and Corneil [11] that determining the minimum number of dimensions needed for an arbitrary tree is NP-complete. The reduction in [11] uses trees with small depth and large degree so the question of binary trees remains. Afrati, Papadimitriou, and Papageorgiou [1] described a divide and conquer algorithm which maps an N vertex binary tree in the hypercube with $\mathcal{O}(N^2)$ vertices. This was later improved by Wagner [12] who presented an algorithm for mapping binary trees in the hypercube with $\mathcal{O}(N \log N)$ vertices.

In the context of the synchronous simulation of one multiprocessor interconnection by another, the mapping of trees into hypercubes has been studied as a graph embedding problem. An *embedding*, $< \varphi, \phi >$, of a *guest* graph G into a *host* graph H is a one-one mapping, $\varphi : V(G) \longrightarrow V(H)$, along with a mapping $\phi : E(G) \longrightarrow \{\text{paths in } H\}$, from edges in G to paths in H. The *dilation* of an edge, $e \in E(G)$, is the length of path $\phi(e)$ in H. The *dilation* of an embedding is the maximum dilation over all edges. The *expansion* of an embedding is $\lceil |V(G)|/|V(H)| \rceil$. An embedding with expansion one is said to be an embedding of a graph into the optimal sized host. The *congestion* of an edge, $e \in E(H)$, is the number of paths in which e appears under ϕ. The *congestion* of an embedding is the maximum congestion over all edges of H.

Bhatt, Chung, Leighton, and Rosenberg [2] (BCLR) describe an algorithm which em-

beds all binary trees in the hypercube with $\mathcal{O}(1)$ dilation, $\mathcal{O}(1)$ expansion, and $\mathcal{O}(1)$ congestion. This was improved by Monien and Sudbrough [10] whose embedding had dilation 3 and expansion $\mathcal{O}(1)$ or dilation 5 and expansion 1. Two conjectures appearing in BCLR are:

Conjecture 2 *There exists an expansion 1, dilation 2 embedding of all binary trees in the hypercube.*

Conjecture 3 *There exists an expansion 2, dilation 1 embedding of all binary trees in the hypercube.*

It can be shown that conjecture 1 implies the last two conjectures.

Finally, with respect to the multiprocessor mapping problem, most researchers have considered total or average dilation rather than maximum dilation. The *total dilation* (or *average dilation*) of an embedding is the sum (or average) over all edge dilations. This is usually measured with respect to the optimal size host. Intuitively, in an asynchronous multicomputer, total dilation measures the overall communication load on the network. In contrast to the previously mentioned theoretical results, Smedley [9] and Chen, Stallmann and Gehringer [4] have experimentally investigated different heuristics for embedding trees in the hypercube. These results indicate that, for randomly generated trees, it is often easy to find an embedding with small average dilation, less than 1.5. However, the maximum dilation of the embedding is often high, close to the diameter of the hypercube. The best heuristic algorithm, with respect to average dilation, is reported by Smedley [9] to be a variation of a greedy algorithm.

The algorithm presented in this paper uses a heuristic to embed all binary trees in the optimal sized hypercube with dilation two and average dilation close to that reported by Smedley [9]. The heuristic used in this case is quite different from the simulated annealing, min-cut, or greedy search heuristics investigated by Chen *et al* [4]. In this case the term heuristic is used because the success of our algorithm relies on the conjecture that strongly balanced binary trees span the hypercube. Experimentation with a large number of randomly generated binary trees, about 80,000, ranging in size from 32 to 1024 vertices has not produced a counterexample.

The objective of this work has been to minimize both maximum and average dilation. Not only does this minimize the load on the network, but more importantly, it ensures that this load is evenly distributed.

In most multiprocessor systems the routing algorithm is fixed, either in hardware or by the operating system. Therefore, the paths of an embedding cannot be chosen so as to minimize the congestion. In fact, the standard hypercube routing algorithm [1] increases

[1] Forward the message on the link whose dimension is the highest dimension in which the source and destination nodes differ.

the chance of congestion since lower dimensional edges are used more often than higher dimensional edges.

Consider the effect of dilation on congestion where now the only assumption made about the routing strategy is that it routes along shortest paths. In the following discussion let $< \varphi, \phi >$ be an embedding of a guest graph G into a host graph H. Furthermore, assume that the degree of vertices in H is larger than the degree of vertices in G. Obviously, an embedding with maximum dilation one, also has congestion one.

Suppose that the maximum dilation of $< \varphi, \phi >$ is two. Consider an edge $\{u, v\} \in E(H)$ and suppose that $\{u, v\}$ is on the path $\phi(e)$ for some $e \in E(G)$. It follows, since the length of $\phi(e)$ is at most two, that an endpoint of e is mapped to u or v of H. Therefore, since φ is a one-one mapping, the congestion, the number of edges, e of G with $\{u, v\}$ in $\phi(e)$, depends on the degree of G. In the case of a binary tree, assuming that the links of the network are bidirectional, the congestion is at most six (see Figure 2-(a)). Once dilation

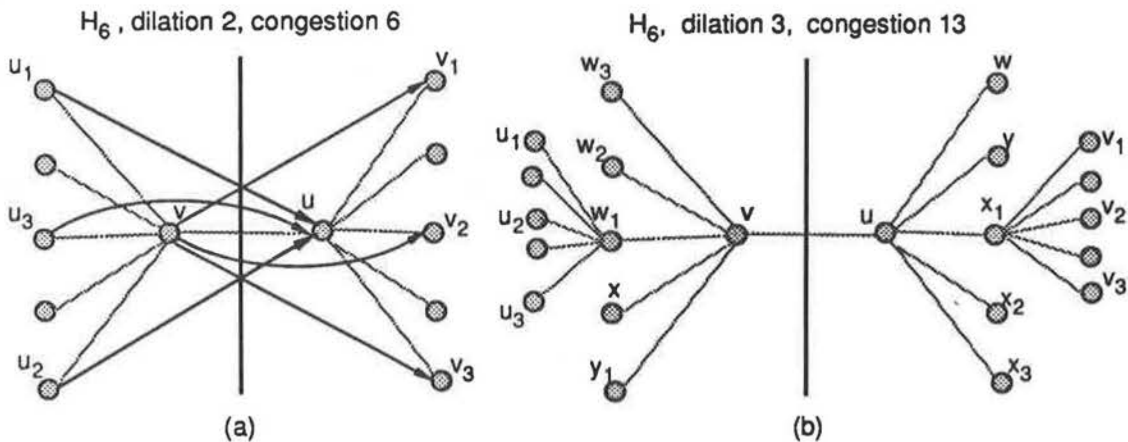

Figure 2: *(a) Depicts an example of congestion in H_6 for a dilation 2 embedding of a binary tree. In the tree vertices u and v are adjacent to u_1, u_2, u_3 and v_1, v_2, v_3, respectively. The arcs are tree edges. (b) Depicts an example of congestion in H_6 for a dilation 3 embedding of a binary tree. Shown are u, v, w, x, y and the tree edges to which they are adjacent u_i, v_i, w_i, x_i and y_i, for $i = 1, 2, 3$. The arcs corresponding to tree edges have been omitted.*

is greater than two, congestion is more a factor of the host graph rather than the guest graph. For a dilation D embedding, D larger than two, it is no longer the case that an endpoint of an edge e in $E(G)$ is mapped onto an endpoint of the path, $\phi(e)$. Congestion now is related to the number of vertices within distance $D-1$ of the endpoints of an edge in $E(H)$. For example in H_n, if $D = 3$ then congestion can be $\mathcal{O}(n)$ (see Figure 2-(b)).

These observations underscore the importance of having dilation one or two embed-

dings. Only then, for any shortest path routing algorithm, is it possible to guarantee embeddings with constant congestion.

The algorithms given in this paper produce dilation 2 embeddings of all binary trees with 2^n nodes into H_n. We restrict the discussion to trees with 2^n nodes since these are the most difficult trees — other sized trees can be "filled out" to this size. This is optimal, with respect to maximum dilation, since every embedding of an unbalanced tree must dilate at least one edge.

The algorithm is a two step process. The first step produces a maximum dilation 2 embedding of all binary trees into the family of strongly balanced trees. In this step, we try to minimize total dilation and edge congestion. In the second step we take the tree, which we now conjecture is a spanning tree of the hypercube, and map it into the hypercube.

3 Definitions

Let T be a tree with an even number of vertices. T is called a *strongly balanced tree*, or simply an *SB-tree*, when it has a perfect matching. A *perfect matching* of G is a subset of the edges of G such that every vertex of G is the endpoint of exactly one edge in the matching.

An edge of T is *even*, or *odd*, if its removal disconnects T into two components with an even, or odd, number of vertices. There is the following relationship between a perfect matching and the odd edges in a tree.

Lemma 1 *Let T be a tree with an even number of vertices. If T contains a perfect matching M then M equals the set of odd edges in the tree.*

Proof:
Suppose edge $e \in M$. If all edges incident to e are removed then all of the resulting components must contain an even number of vertices. That is, e is only incident to even edges. Since T has an even number of vertices, every vertex is incident to at least one odd edge. Therefore, e is an odd edge.

If e is an odd edge not in M then it is incident to some edge of the matching. But this contradicts the fact that, by the previous argument, edges in M are odd and are incident to only even edges. Therefore e is in M.

□

Using this lemma there is the following characterization of strongly balanced trees.

Lemma 2 *Let T be a tree with an even number of vertices N. The following are equivalent:*

1. T is strongly balanced.

2. *Every vertex is incident to exactly one odd edge.*

3. *The number of odd edges is one more than the number of even edges.*

4. *Removing an even edge disconnects T into two strongly balanced trees.*

Proof:
(1) $\Rightarrow$ (2) by lemma 1. (2) $\Rightarrow$ (3) since every vertex is incident to one odd edge and there are exactly $N/2$ odd edges, thus leaving $N/2 - 1$ even edges. (3) $\Rightarrow$ (4), by induction on N, even, using the fact that every tree has at least $N/2$ odd edges. Finally, it is obvious that (4) $\Rightarrow$ (1).

□

It follows from Lemma 2 that every strongly balanced tree is also balanced. Moreover, statement 3 of Lemma 2 implies that every part of T is also balanced. That is, locally there is not an excess or deficit of one colour or another in a two colouring of T, hence the term strongly balanced. Intuitively, what makes these trees good candidates for spanning trees of H_n is that there are fewer restrictions on mapping the tree into H_n. In any two colouring of the tree, colour deficits do not occur, so a deficit in one part of the tree does not have to be matched with an excess elsewhere in the tree.

4 Trees to strongly balanced trees

In this section we present an algorithm to produce a dilation 2 embedding of a binary tree into a strongly balanced binary tree. The algorithm also attempts to minimize the average dilation of the embedding.

In the rest of this section let T be a binary tree with an even number of vertices. In T identify the following two types of vertices. A *3-0* vertex is a vertex incident to three odd edges and a *1-x* vertex is a vertex incident to one odd edge and zero, one or two even edges. Since T has an even number of vertices, no vertex in T is incident to an even number of odd edges. Therefore, all vertices of T are either 3-0 or 1-x vertices.

Let T be rooted at an arbitrarily chosen leaf. Note that because T has an even number of vertices the parity of an edge is independent from where the tree is rooted. Consider a path from u to v, $u = u_0u_1 \ldots u_k = v$, in T satisfying the following properties;

1. u is a 3-0 node and v is a leaf or a node of degree 2,

2. the path does not contain two consecutive even edges.

Now consider the following operation that dilates the edges incident to the path (see Figure 3).

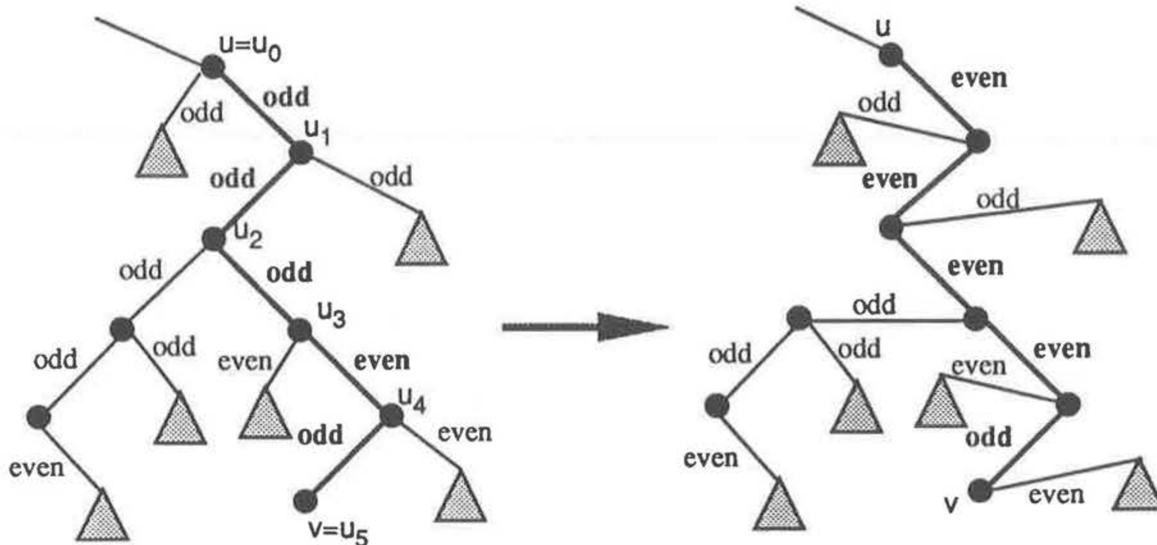

Figure 3: *The result of a shift operation on the path shown in bold.*

Procedure *shift (node u,v)*
  for $i \leftarrow 0$ **to** $k-1$ **do**
    if *(sibling(u_{i+1}) $\neq$ NIL)*
      remove edge $\{u_i, sibling(u_{i+1})\}$
      add edge $\{u_{i+1}, sibling(u_{i+1})\}$

In procedure shift, edges and siblings are with respect to the *original* tree and do not change during the operation.

In order to show that this operation converts all 3-0 nodes to 1-x nodes there are two cases to consider. First, suppose that edge $\{u_i, \text{sibling}(u_{i+1})\}$ is odd. Since there are not two consecutive even edges in the path, u_i must be a 3-0 node, implying that edge $\{u_i, u_{i+1}\}$ is odd. The shift operation will add an odd number of nodes to the odd weight subtree rooted at u_{i+1}. Therefore, $\{u_i, u_{i+1}\}$ becomes an even edge. Second, suppose that edge $\{u_i, \text{sibling}(u_{i+1})\}$ is even. The shift operation adds the even weight subtree rooted at sibling(u_{i+1}) to u_{i+1}. Therefore, the parity of $\{u_i, u_{i+1}\}$ is unchanged. All subsequent shifts are in the subtree rooted at u_{i+1} and so do not affect the parity of edge $\{u_i, u_{i+1}\}$. Therefore, the shift procedure converts all 3-0 nodes in the path to 1-x nodes while leaving 1-x nodes unchanged.

The algorithm to convert T to a strongly balanced binary tree performs shift operations on paths in T. Let *sbT* be the tree modified by the shift procedure. Initially *sbT* equals T, where T and *sbT* are global variables.

Procedure *shiftpath(subtree rooted at r)*
  1. **if** *(r is a leaf) return*
  2. **else**

3. *Find a path from r to a leaf l in T such that the path does not have two consecutive even edges.*
4. *Let u be the first 3-0 node on the path, or NIL otherwise.*
5. **if** *($u \neq NIL$) shift(u,l) in sbT.*
6. **for** *(all v, v a sibling of a node on the path from r to l)* **do** *shiftpath(v)*

In step 3, a path to l not containing two consecutive even edges is constructed by following, when possible, an odd edge. Since every node is incident to at least one odd edge, this strategy never chooses two consecutive even edges.

Shiftpath converts all 3-0 nodes on the path from r to l to 1-x nodes. It also is called recursively for each subtree rooted at a node incident to the chosen path. Therefore, when started at the root of T, it visits every node exactly once and on termination sbT contains no 3-0 nodes. It follows, by lemma 2, that sbT is a strongly balanced tree. If T has N nodes then this is an $\mathcal{O}(N)$ algorithm.

Let Δ_2 and Δ_3 denote, respectively, the number of degree two and degree three nodes in a binary tree. In the previous algorithm, because only edges incident to the path are dilated, at most one of the edges of a degree three node connecting it to a child is dilated. Also, the edge connecting a degree two node to its child is never dilated. Therefore, the total dilation is at most $N-1+\Delta_3$. It is easily derived that

$$\text{average dilation} \leq 1.5 - \frac{\Delta_2 + 1}{2(N-1)}$$

For trees with no vertices of degree two, all degree three vertices are type 3-0, and the average dilation is almost 1.5. For trees containing degree two vertices, a lower bound on the number of dilated edges is the number of 3-0 vertices in the tree. In practice, the algorithm dilates somewhere between the number of 3-0 vertices and Δ_3. Often, it is not possible to achieve the lower bound since for a given tree the shift operation must shift an edge incident to every degree three vertex on the path, including any 1-x vertices.

Once the tree has been converted to a SB-tree the folding algorithm given in the next section is used to find a dilation one embedding of the SB-tree in the hypercube. The average dilation of the overall embedding depends entirely on the dilation introduced by the shiftpath operation. Table 1 shows the average dilation of an embedding produced by shiftpath and folding in comparison to the best known greedy algorithm for embedding a tree in the hypercube [2]. As Table 1 shows, the average dilation of shiftpath and folding is close to that of the best known greedy heuristic. In the previous experiments the maximum dilation of the embeddings produced by the greedy algorithm was large, within one of the diameter of the host hypercube. In comparison, the maximum dilation produced by shiftpath and folding is two.

[2]The results shown in this table are from Smedley[9]

algorithm	Number of Vertices						
	16	32	64	128	256	512	1024
greedy	1.0405	1.0620	1.0657	1.0656	1.0574	1.0495	1.0434
folding	1.0986	1.1173	1.1330	1.1526	1.1583	1.1657	1.1920

Table 1: *Experimental results showing the average dilation for embedding 2000 randomly generated binary trees into the optimal sized hypercube.*

One possible improvement to shiftpath is to try to minimize the number of edges incident to 1-x nodes that are dilated by the algorithm. To this end, define the *cost* of a path to be the number of 1-x nodes along the path from the first 3-0 node to the leaf or node of degree two. Now, in Step 3 of shiftpath, choose the minimum cost path satisfying the conditions for a shift operation.

For any shortest path routing strategy, shiftpath and folding produces an embedding of T in the hypercube with congestion at most five. This follows from the fact that no vertex of T is incident to more than two dilated edges. If routes can be chosen then it is possible to obtain a congestion two embedding. By the shift operation, the edge from a sibling to the parent is replaced by an edge between two siblings. Therefore, in sbT, the path corresponding to this dilated edge can be the new edge between siblings followed by the edge to the parent.

It is also possible to eliminate 3-0 vertices by adding vertices to T. Choose a 3-0 vertex u and construct a path from one leaf to a second leaf so that the path contains u. Choose this path so that it does not contain two consecutive even edges. Such a path can be found by starting at u and constructing two edge disjoint paths from u to a leaf. As in shiftpath, each path from u to a leaf can be found by starting at u and following, when possible, an odd edge. Now, append a new vertex to each endpoint (i.e. leaves) of the overall path. The effect of this operation is that it flips the parities of all the edges in the path. Therefore, since there are no consecutive even edges, it converts any 3-0 vertices on the path to 1-x vertices and does not introduce any new 3-0 vertices. This operation can be repeated until there are no longer any 3-0 vertices in T.

In total, by adding at most twice the number of 3-0 vertices, the tree will be strongly balanced. Since the number of 3-0 vertices is strictly less than half the size of the tree, the resulting strongly balanced tree is at most 1.5 times the size of the original tree. In practice, given a tree of arbitrary size it is possible to combine the operations of dilating edges and adding vertices to obtain an embedding with small total dilation.

The algorithm given is this section is a simple one but ensures that the load is evenly distributed throughout the hypercube. It will guarantee edge congestion at most five, using any shortest path routing algorithm, and average dilation of less than 1.5. In the next section we consider the problem of finding dilation one embeddings of strongly balanced trees in the hypercube.

5 Embedding strongly balanced binary trees into the hypercube

Given a strongly balanced binary tree the second step is to embed this tree in the hypercube. In the following sections an embedding is taken to be a dilation 1 embedding of graphs with 2^n vertices for some n. That is a labeling of vertices of the graph so that the Hamming distance between adjacent nodes is one. The algorithm described in this section produces an embedding of a strongly balanced binary tree with 2^n vertices in H_n. The technique is based on a factoring of a graph with respect to the cartesian product of graphs.

The cartesian product, $G = G_1 \otimes G_2$, of two undirected graphs G_1 and G_2 is the undirected graph with vertex set $V(G) = V(G_1) \times V(G_2)$. There is an edge $\{(u_1, v_1), (u_2, v_2)\}$ in G whenever $u_1 = u_2$ and $\{v_1, v_2\} \in E(G_2)$ or $v_1 = v_2$ and $\{u_1, u_2\} \in E(G_1)$. A graph G is said to be *factorable* if there exists graphs G_1 and G_2 such that $G = G_1 \otimes G_2$. A recursive definition of H_n equivalent to that given in Section 2 is the cartesian product of K_2 (i.e. the complete graph on two vertices) with itself n times. Informally, one can view H_n as consisting of two copies of H_{n-1} with an edge between all corresponding nodes.

Algorithmically, a labelling of G in H_n can be obtained by recursively finding factors G' and K_2 such that G spans $K_2 \otimes G'$. This leads to the following reformulation of the embedding problem. Given G, partition the vertices of G into two equal size sets, G^1 (the upper cube) and G^0 (the lower cube), and find a bijection φ from G^1 to G^0 such that:

Property 1 *For all $\{x, y\} \in E(G)$, if $x \in G^1$ and $y \in G^0$ then $\varphi(x) = y$.*

This property ensures that the cartesian product structure exists. Define φ to be a mapping satisfying this property and let $\varphi(G)$ be the homomorphic image of G under φ. That is the graph G' where $V(G') = \{v \in V(G) | \exists u \in V(G) \ni \varphi(u) = v\}$ and edges $E(G') = \{\{u, v\} | \{u, v\} \in G^0\} \cup \{\{\varphi(u), \varphi(v)\} | \{u, v\} \in G^1\}$ (see Figure 4).

Lemma 3 *There exists φ such that $\varphi(G)$ embeds in H_{n-1} if and only if G embeds in H_n.*

Proof

Assume that φ exists. Given $\varphi(u) = v$ and an embedding of $\varphi(G)$ in H_{n-1}, label u with 1 concatenated with the label of v in $\varphi(G)$ and label v with 0 concatenated with the label of v in $\varphi(G)$. It is easy to show that this results in an embedding of G in H_n. Conversely, φ can be found by projecting the part of G in one copy of H_{n-1} (i.e. G^1) onto the corresponding vertices in the second copy (i.e. G^0).

□

The algorithm begins with a tree T_n with 2^n vertices and recursively finds a φ which collapses T_n to a tree T_{n-1}. It tries to maintain the invariant that on collapsing T_i to T_{i-1},

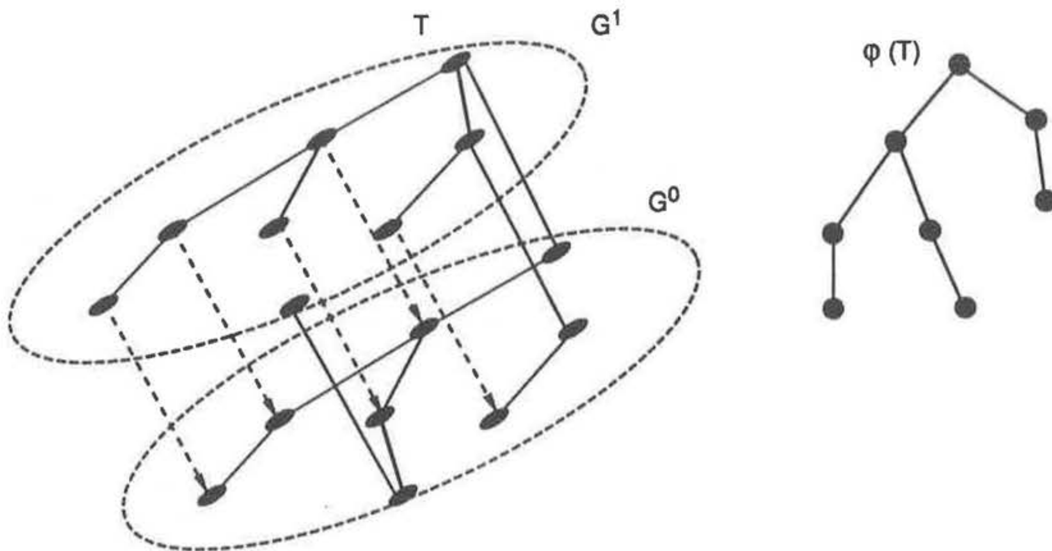

Figure 4: *Mapping of a tree between the upper and lower cubes.*

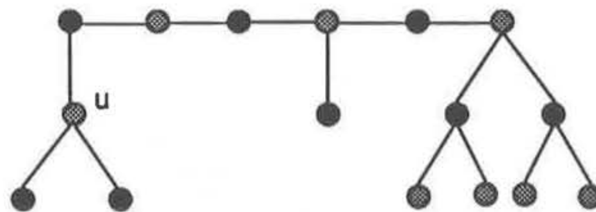

Figure 5: *An example of a balanced binary tree which is not foldable*

T_{i-1} equals $\varphi(T_i)$ and T_{i-1} is a binary tree. If the tree can be successfully collapsed then after $n-1$ iterations we obtain the trivial embedding of T_1 in H_1. An embedding of T_n in H_n can then be constructed by using lemma 3.

A tree T is *foldable* if there exists a φ resulting in a binary tree. We call $\varphi(T)$ the *folded* tree, and say that φ *folds* T. A tree T is *SB-foldable* when there exists a φ resulting in a SB-tree. Every strongly balanced tree is foldable (this will follow from the algorithm in the next section), however, not every balanced tree is foldable. A counterexample is shown in Figure 5. Vertex u cannot be mapped to any other vertex in the tree without violating property 1 or the fact that $\varphi(T)$ must be a binary tree.

This formulation of the embedding problem is related to the work of Feigenbaum and Haddad [5] on factorable extensions of graphs. A graph G' is a *factorable extension* of G if

a set of edges or vertices can be added to G such that $G' = G_1 \otimes G_2$ for some G_1 and G_2. They showed that every tree with N vertices, N even, can be factored into $K_2 \otimes G_2$ where $V(G_2) = {}^1\!/_2 N$. In contrast, we consider only extensions of trees which are themselves trees.

The algorithm given in the next section takes as input an SB-tree and is guaranteed to find, if it exists, an SB-fold. The success of this embedding strategy relies on the conjecture that

Conjecture 4 *All SB-trees are SB-foldable.*

If this holds then from our previous remarks it follows that every SB-tree can be embedded in the hypercube and that there exists a dilation 2 embedding of every binary tree in the optimal sized hypercube.

5.1 Folding an SB-tree

The algorithm used to fold a strongly balanced binary tree is essentially a search procedure. For every pair of vertices in the tree the procedure will check whether there exists a φ associating the two vertices. First, we describe the search procedure and then show how to add a structure to the procedure to record the intermediate choices. Finally, we show how φ can be extracted from this structure.

Consider a strongly balanced binary tree T with an even number of vertices arbitrarily rooted at a vertex of degree two or one. If T is SB-foldable then there must exist a vertex which becomes associated with the root of T. This suggests the following routine to check whether or not a tree, or in general a subtree, rooted at node u can be SB-folded.

Function *fold(node u) : boolean*
1. **if** *(u = NIL) return TRUE*
2. fold ← FALSE
3. **for** *all nodes v of T, v a descendant of u* **do**
fold ← fold ∨ foldpath(u,v)
4. return fold

Function foldpath returns TRUE only if the subtree rooted at u can be SB-folded so that u and v are associated by φ. Given a path from u to v, foldpath tries to associate u with v, the next node along the path with the parent of v and so on.

They are several conditions which must be satisfied if the path from u to v is to result in a SB-fold.

1. Unless v is adjacent to u, the degree of v must be at most two. Otherwise in the folded tree the degree of u is larger than three, violating the condition that $\varphi(T)$ must be a binary tree.

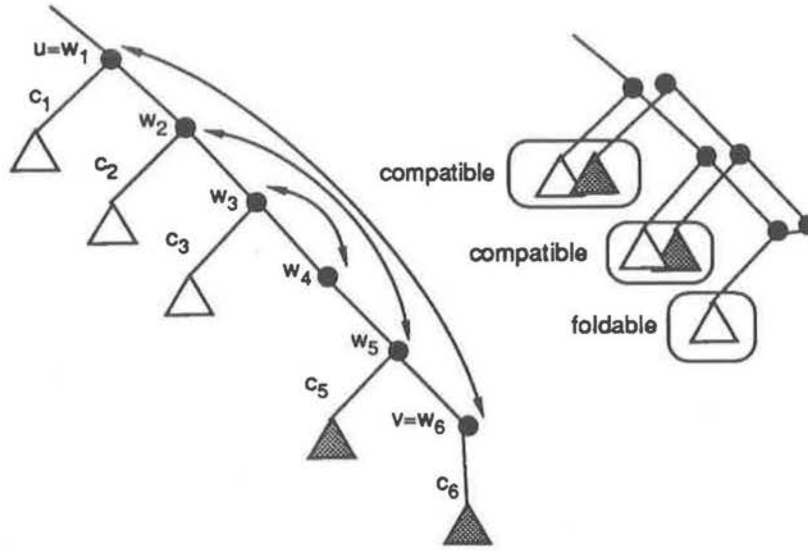

Figure 6: *An example showing the foldpath test*

2. There must be an even number of nodes in the path. Otherwise, the middle node will not be associated with a node in the path and yet it is adjacent to two nodes which are associated. This cannot result in a fold because it does not satisfy property 1.

3. The parity of the edges incident to a pair of associated nodes on the path must be the same. Also, in $\varphi(T)$, the association resulting from pathfold must not have two incident odd edges in the new path. This last condition ensures that $\varphi(T)$ will be strongly balanced.

Given these routines to check that it is a valid path define foldpath as follows (see Figure 6). Let $u = w_1 w_2 \ldots w_k = v$ denote a path in the tree from node v to an ancestor u. Let c_i denote the child of w_i not on the path from u to v, where $c_i =$ NIL if the child does not exist.

Function *foldpath (node u,v) : boolean*

1. **if** *(not valid path) return FALSE*

2. foldpath $\leftarrow$ *TRUE*

3. **if** *(u adjacent to v and degree of v is 3)*

Let c'_2 *be the second child of v.*

foldpath $\leftarrow$ $[compatible(c_1,c_2) \wedge fold(c'_2)] \vee [compatible(c_1,c'_2) \wedge fold(c_2)]$

4. **else**

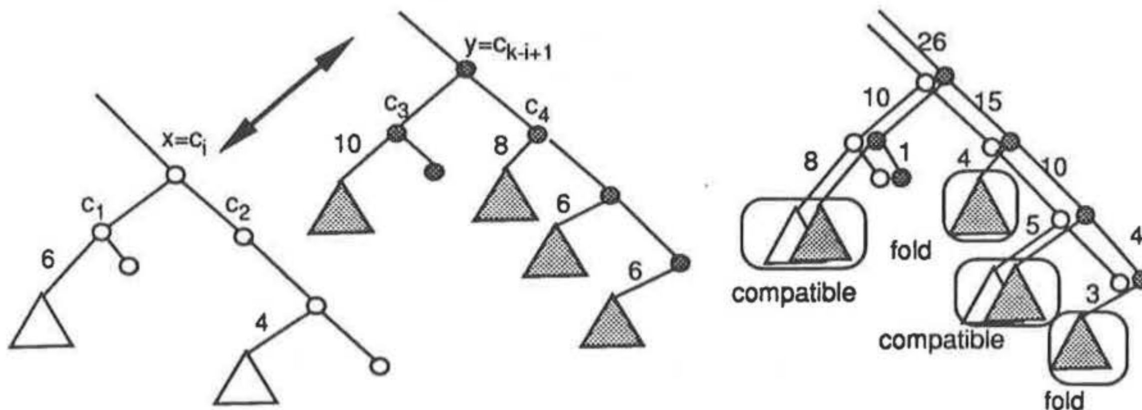

Figure 7: *An example showing the compatible test. The weights of the subtrees are shown on the edges.*

5. **for** $i \leftarrow 1$ **to** $k/2-1$ **do**
   foldpath $\leftarrow$ *foldpath* $\wedge$ *compatible*(c_i, c_{k-i+1})
6. *foldpath* $\leftarrow$ *foldpath* $\wedge$
   $[[$ *fold*$(c_{k/2}) \wedge$ *fold*$(c_{k/2+1})] \vee$ *compatible*$(c_{k/2}, c_{k/2+1})]$.
7. *return foldpath*

Function compatible returns TRUE if the subtrees rooted at c_i and c_{k-i+1} can be overlaid so that the parts of these subtrees not associated with each other can be SB-folded (see Figure 7). A recursive function to check this condition follows. For two nodes x and y let c_1 and c_2 be the children of x and c_3 and c_4 be the children of y.

Function *compatible (node x,y) : boolean*

1. **if** *(x or y is NIL)*
   compatible $\leftarrow$ *fold*$(x) \wedge$ *fold*(y)
2. **else**
   compatible $\leftarrow$ [*compatible* $(c_1,c_3) \wedge$ *compatible*(c_2,c_4)]
   $\vee$ [*compatible* $(c_1,c_4) \wedge$ *compatible*(c_2,c_3)]
3. **if** *(strong balance check = FALSE)*
   compatible $\leftarrow$ *FALSE*

The strong balance condition checks that after associating nodes x and y, the parity of at least one of the incident edges is even.

If the strong balance condition is ignored, then as long as the parity of the edges joining x and y to their parents are the same, compatible always returns TRUE. This follows from the fact that because T is strongly balanced all nodes in T are 1-x nodes. Similarly, in

foldpath, if the condition for strong balance is ignored then a valid foldpath can always be found by following, when possible, an odd edge. It follows that every SB-tree is foldable.

The SB-foldability of T is checked by calling function fold at the root of the tree. Implicitly, functions foldpath and compatible associate pairs of nodes in T. Function foldpath(u,v), for $u = w_1 w_2 \dots w_k = v$, associates w_i and w_{k-i+1}, for $i = 1, k/2$. Function compatible(x,y) associates nodes x and y. Together, these associations correspond to a φ which folds T. It can be extracted from the calls to fold, foldpath, and compatible by storing the results of intermediate calls to these functions. In function

1. fold(u): store a node v for which foldpath(u,v) is TRUE.

2. foldpath(u,v): in this routine, choices occur only when the two adjacent, middle nodes in the path are associated. It suffices to record the combinations of folds and calls to compatible, if any, which result in a fold.

3. compatible(x,y): store which of the two possible matchings of children returned TRUE.

For a tree with N nodes, this information can be conveniently stored in the upper half of an N by N matrix. Call this matrix F. In F store the fold information about node u on the diagonal, $F(u, u)$. Compatibility is checked only if nodes u and v are not related while foldpath is called between related nodes in T. This allows us to store either the foldpath or compatible information in $F(u, v)$. For the sake of simplicity, we extract φ from F by executing the algorithm once more using the information from F. In practice of course, a φ can be found directly during the execution of the algorithm.

This results in an association of nodes in T but it is still necessary to determine for a given u, v whether $\varphi(u) = v$ or $\varphi(v) = u$. Define an edge $\{u, v\}$ of T to be a *cross-edge* if u and v were associated. Now notice that between any pair of associated nodes u and v, the path from u to v in T traverses exactly one cross-edge. This cross-edge is the "middle" edge of the path in a call to foldpath. The path between pairs of nodes associated by foldpath and the path between nodes associated by any resulting calls to compatible, contain this cross-edge. Nodes not associated by this foldpath are folded in their own subtree where again this property holds.

Now choose a node u in T and let G^1 be the set of all nodes in T reachable from u by traversing an even number of cross-edges. All remaining nodes are in G^0. It follows that, in every pair of associated nodes, the two nodes are in different Gs and that $\varphi(u) = v$ for every edge between the G's. Therefore, the resulting mapping satisfies property 1 and φ is an SB-fold.

Moreover, by slightly modifying foldpath, the converse is also true. That is, if T rooted at r is SB-foldable then fold(r) returns TRUE. First however, foldpath must be modified to allow for the degree of the root of T to become three. This is the only restriction

introduced by rooting the tree. So now, in foldpath(u,v), assume that the path is valid when u is the root, and v has degree three. Let c_1 be the child of u and let c_2 and c_3 be the children of of v. Add the following statement between steps 2 and 3 of foldpath.

> **if** *(u is the root and u is adjacent to v)*
> $foldpath \leftarrow fold(c_1) \wedge fold(c_2) \wedge fold(c_3)$
> **else**
> $foldpath \leftarrow [compatible(c_1,c_2) \wedge fold(c_3)] \vee [compatible(c_1,c_3) \wedge fold(c_2)]$

The fact that fold now returns TRUE whenever T is SB-foldable is a result of the following lemma.

Lemma 4 *If φ folds T and $\varphi(u) = v$ then for the path $u = w_1 w_2 \ldots w_k = v$ in T, k is even and for $i = 1, k/2$, $\varphi(w_i) = w_{k-i+1}$.*

Proof:
First, k cannot equal 3 since this would contradict property 1. Second, if k is larger than 3, the statement of the lemma holds since otherwise $\varphi(T)$ would contain a cycle.

□

The connection between φ in T and calls to foldpath and compatible can now be stated. Suppose that T has been arbitrarily rooted. The following is true with respect to φ and the rooted tree.

1. If $\varphi(u) = v$, where u and v are not related in T, then compatible(u,v) is TRUE.

2. Suppose $\varphi(u) = v$, where u and v are related. Assume, without loss of generality, that u is an ancestor of v. Furthermore, suppose that in the subtree rooted at u there does not exist a w such that $\varphi(w) = x$ (or $\varphi(x) = w$) where x is not a descendant of u. In this case, foldpath(u,v) is TRUE. Call u a *cut* node of T with respect to φ.

The proof of these two statements will follow from lemma 4 by induction on the recursive calls made within each function. The base case for the first statement is the situation in which u and v are not related and the subtrees rooted at u and v do not contain cut nodes with respect to φ. It follows; by contradiction, from lemma 4, that φ must map all nodes in the subtree rooted at u (or vice versa) to the subtree rooted at v.

For the base case of the second statement, consider subtrees rooted at a cut node that do not contain any other cut nodes. Suppose u is the root of this subtree, where $\varphi(u) = v$ and consider foldpath(u,v). Foldpath(u,v) returns TRUE only if there is an appropriate set of calls to compatible and fold, on nodes incident to the path from u to v, that all return TRUE. But, since u is the only cut node in the subtree, φ must map the nodes

incident to the path onto each other. This implies that the only recursive calls made by foldpath are to compatible. Each call to compatible satisfies the conditions for the base case of the first statement and is therefore TRUE. It follows that foldpath(u,v) is TRUE.

The verity of these two statements now follows, by induction, on the calls made in each function and the fact that foldpath and compatible examine all possible ways in which to associate nodes.

In summary, we have the following theorem.

Theorem 1 *Given a binary tree T with an even number of nodes. T is SB-foldable if and only if, for T arbitrarily rooted at node u, fold(u) is true.*

We now analyse the complexity of the algorithm. The algorithm searches in a depth first manner using matrix F to store the intermediate results on subtrees of T. Consider the general inductive step where the information from the two subtrees T_1 and T_2 with roots v_1 and v_2, respectively, is combined in subtree T_3 rooted at u (see Figure 8). Let the number of nodes in T_3, T_1 and T_2 be N, N_1 and N_2, respectively. Assume that T_3 has an

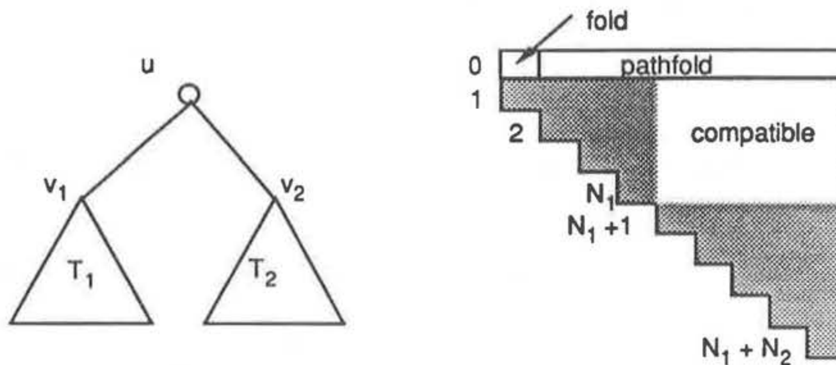

Figure 8: *Combining the information from the recursive calls. The shaded regions depict information calculated from the recursive call for each subtree.*

even number of nodes and that the matrix F has been filled in for the nodes of T_1, labelled 1 to N_1, and those in T_2, labelled $N_1 + 1$ to $N_1 + N_2$. To combine the information from the two subtrees it is necessary to first calculate all of the compatibilities between vertices in T_1 and those in T_2. In total, this requires $N_1 \times N_2$ calls to compatible. Using this information the pathfolds can now be calculated. Each pathfold calls compatible at most $N/2$ times and there are $N - 1$ pathfolds which must be calculated. The fold condition can be updated during the pathfold operations. Combining these estimates it follows that

$$T(N) = T(N_1) + T(N_2) + (N-1)N/2 + N_1 \times N_2.$$

The product $N_1 \times N_2$ is maximized when $N_1 = N_2$, therefore

$$\begin{aligned} T(N) &= {}^1\!/_2 T(N/2) + N^2/2 - N/2 + N^2/4 \\ &= {}^1\!/_2 T(N/2) + \mathcal{O}(N^2). \end{aligned}$$

Solving this recurrence gives an $\mathcal{O}(N^2)$ algorithm for checking the foldability of a strongly balanced tree.

5.2 Discussion

The algorithm for embedding all binary trees in the hypercube depends on conjecture 4, that is all SB-trees are SB-foldable. If this is true, the previous algorithm can be called recursively, $n = \log N$ times from T_n, $T_{n-1} = \varphi(T_n), \ldots, T_2$, to obtain an embedding of a strongly balanced tree in the hypercube. The complexity of the overall algorithm is still $\mathcal{O}(N^2)$. If conjecture 4 is true then all strongly balanced trees with 2^n nodes span a hypercube. In turn, as shown in Section 4, this implies that conjectures 2 and 3 from Section 2 are true.

The folding algorithm given in this section has been tested on a large number of trees. A test set of random trees was generated, converted to strongly balanced trees and then embedded in the hypercube. About 80,000 trees of each size were tested. Tree size was a power of 2 and ranged from 16 to 1024. The tests

1. randomly chose a leaf as the root for the strongly balanced tree to hypercube embedding, and
2. the selection of the trees was skewed towards trees with a large number of degree three nodes. Previous experience has led us to believe that bushy trees are the most difficult to embed [9].

In effect, considerably more smaller trees were tested since, for tree T_n with 2^n vertices, the algorithm also embeds all intermediate trees, from T_n down to T_2. No counterexample to conjecture 4 was found.

Further research investigated the foldability of the subtrees. That is, are all rooted trees with an even number of nodes SB-foldable? A rooted tree with an odd number of nodes is a SB-tree when there exists a matching containing all nodes but the root. Unfortunately, not all rooted trees with an even number of nodes are SB-foldable. A counterexample for a tree with 22 nodes is shown in Figure 9. However, this tree is foldable when rooted at a different vertex.

It is still possible to use this algorithm even if Conjecture 4 is not true. If there are SB-trees which are not SB-foldable then the algorithm may at some point fail. At that point there are two alternatives. If the tree is not already strongly balanced or has one

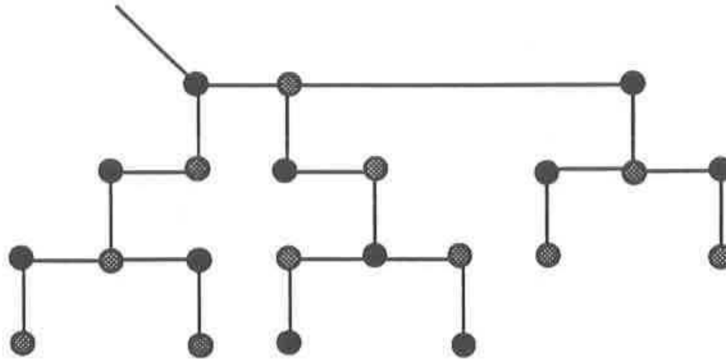

Figure 9: *An example of a rooted binary tree which is not foldable*

or more degree two vertices then a different SB-tree can be generated by the shiftpath algorithm given in Section 4. It is also possible to apply the folding algorithm directly to the original tree, dilating only the edges necessary to obtain a fold. Of course, this may result in edges with dilation larger than two.

6 Conclusions

We have given a two step algorithm for embedding all binary trees in the optimal sized hypercube with dilation two. The algorithm is the best possible in terms of dilation and, for random trees, produces close to the average dilation of the best known heuristics. It is the only tree embedding algorithm that achieves both small maximum dilation and small average dilation. This results in embeddings, which for any shortest path routing algorithm, evenly distributes the load across the hypercube. Experimentation has shown it to be both a practical and efficient algorithm. It is robust in the sense that there are several alternatives should the algorithm fail on a particular tree.

The second step of the algorithm is based on our conjecture that all SB-trees are SB-foldable. If true, all strongly balanced binary trees with 2^n nodes span the hypercube. This in turn would imply, by the algorithm given in Section 4, that the two conjectures appearing in BCLR are true. More importantly, unlike conjectures 1, 2, or 3; conjecture 4, can be tested in polynomial time.

Finally, this work has identified a rich collection of trees that are easily mapped into the hypercube. This is particularly significant given the very restricted families of trees that are known to span the hypercube.

Future work will try to use these results to find good embeddings of other graphs, such as n-ary trees and planar graphs.

References

[1] F. Afrati, C. H. Papadimitriou, and G. Papageorgiou. The complexity of cubical graphs. *Information and Control*, 66:53–60, 1985.

[2] S. N. Bhatt, F. R. K. Chung, T. Leighton, and A. L. Rosenberg. Optimal simulations of tree machines. In *Proceedings of the 27th FOCS Symposium*, pages 274–282. IEEE, Computer Society Press, 1985.

[3] S. Bokhari. On the mapping problem. *IEEE Trans. on Computers*, C-30(3):202–214, March 1981.

[4] W.-K. Chen, M. F. Stallmann, and E. F. Gehringer. Hypercube embedding heuristics: an evaluation. to appear Journal of Parallel and Distributed Computing, 1990.

[5] J. Feigenbaum and R. Haddad. On factorable extensions and subgraphs of prime graphs. *SIAM Journal on Discrete Math*, (2):197–218, 1989.

[6] I. Havel and P. Liebl. One-legged caterpillars span hypercubes. *Jounal of Graph Theory*, 10:69–77, 1996.

[7] I. Havel and J. Morávek. B-valuations of graphs. *Czechoslvak Mathematical Journal*, 22:338–351, 1972.

[8] L. Nebeský. On cubes and dichotomic trees. *Časopis pro Pěstování Matematiky*, 99:164–167, 1974.

[9] G. Smedley. Algorithms for embedding binary trees into hypercubes. Master's thesis, University of British Columbia, 1989.

[10] I. H. Sudborough and B. Monien. Simulating binary trees on hypercubes. In *3rd Aegean Workshop on Computing: VLSI Algorithms and Architectures*, 1988. Corfu, Greece.

[11] A. Wagner and D. Corneil. Embedding trees in the hypercube is NP-complete. *SIAM Journal of Computing*, 19(4):570–590, June 1990.

[12] A. S. Wagner. Embedding arbitrary binary trees in a hypercube. *Journal of Parallel Distributed Computing*, (7):503–520, June 1989.