Embedding Graphs with Bounded Treewidth into Their Optimal Hypercubes

Volker Heun and Ernst W. Mayr

Fakultät für Informatik der Technischen Universität München,
80290 München, Germany
E-mail: heun@in.tum.de; mayr@in.tum.de

Received January 5, 1999

1. INTRODUCTION

Hypercubes are a very popular model for parallel computation because of their regularity and their relatively small number of interprocessor connections. Another important property of an interconnection network is its ability to simulate efficiently the communication of parallel algorithms. Thus, it is desirable to find suitable embeddings of graphs representing the communication structure of parallel algorithms into hypercubes representing the interconnection network of a parallel computer.

The embedding of graphs with a regular structure, such as rings, (multidimensional) grids, complete trees, binomial trees, pyramids, X-trees, and meshes of trees, has been investigated by many researchers, see,
e.g., [7–9, 13, 14, 18, 35, 36]. Unfortunately, the communication structure of a parallel algorithm can often be very irregular. Embeddings of such irregular graphs have only been studied in the special case when the graph is a binary tree.

For graphs whose structure is less regular, it is in general hard to decide whether there is a good embedding into a given host graph. Here, an embedding is considered to be good if it has small dilation and load. Given a graph \( G \) and a positive integer \( d \), it is \( \mathcal{NP} \)-complete to decide whether \( G \) is a subgraph of a \( d \)-dimensional hypercube, even if \( G \) is a tree [37]. Given a graph \( G \) and a positive integer \( d \), it is also \( \mathcal{NP} \)-complete to decide whether \( G \) has a dilation 2 embedding into the \( d \)-dimensional hypercube [38]. The unboundedness of the degree of the given graph is essential in the proof. Hence, it might be possible to construct efficient embeddings if the degree of a graph is bounded.

For arbitrary binary trees, one-to-one embeddings into their optimal hypercubes with constant dilation and constant node-congestion have been constructed in [4]. The exact value for the dilation was not given, but it ranges between 12 and 15. Another one-to-one embedding of an arbitrary binary tree into its optimal hypercube with constant dilation is given in [28], but in this paper neither node-congestion nor edge-congestion is considered. In [28], the existence of embeddings with smaller dilation was claimed. However, the proof given in that paper is incomplete and it is still open whether the claimed dilation could be achieved. The embedding given in [20] yields dilation 8 and constant node-congestion. This is the best-known bound on the dilation. Furthermore, this embedding can be efficiently computed on the hypercube itself. In [17], Havel has conjectured that every binary tree has a one-to-one embedding into its optimal hypercube with dilation 2. This conjecture is still open. In terms of lower bounds, a simple parity argument shows that the complete binary tree of size \( 2^d - 1 \) cannot be embedded into the \( d \)-dimensional hypercube with dilation 1, as shown in [34].

It has also been investigated whether balanced caterpillars of size \( 2^d \) are subgraphs of \( d \)-dimensional hypercube. Caterpillars are binary trees whose vertices with degree 3 lie on a single path in the binary tree. This path is called the backbone of the caterpillar. All other paths growing from the backbone are called legs. Since a tree is bipartite graph, we call a tree balanced if both sets of the bipartition of the vertices have the same cardinality. In [19], it was shown that balanced caterpillars with legs of unit length are subgraphs of their optimal hypercubes. This was generalized in [3]; it was shown that balanced caterpillars with the additional property that the length of its legs have the same parity are subgraphs of their optimal hypercubes.

In this paper, we present a one-to-one embedding of graphs with bounded treewidth into their optimal hypercubes based on the tree-decomposition
for the given graph. This is the first time that embeddings of graphs with
an extremely irregular structure into hypercubes have been investigated.
The dilation of the presented embedding is at most $3\lceil\log((d + 1)(t + 1))\rceil + 8$ and the node-congestion is bounded by $O(d(dt)^3)$, where $t$ denotes
the treewidth and $d$ the maximal degree of the given graph. Hence, for
graphs with constant treewidth and constant degree, we exhibit a one-to-one
embedding with constant dilation and node-congestion into their optimal
hypercubes. Furthermore, if the graph is given by a tree-decomposition, we
present an efficient construction of the embedding on the optimal hyper-
cube. It will be shown, that this embedding can be computed in time
$O(\max\{\text{T}_{LR}(n), (dt)^2 \log^2(n)\})$, where $\text{T}_{LR}(n)$ denotes the time complexity
for list ranking on the hypercube. If the given graph has constant treewidth,
a minimal tree-decomposition for this graph can be computed on the optimal
hypercube in time $O(\log^3(n) \log \log^2(n))$, using a result in [6].

On the other hand, we present some lower bounds which will give strong
evidence that each embedding of a graph with treewidth $t$ and maximal
degree $d$ into its optimal hypercube requires dilation $\Omega(\log(dt))$.

2. PRELIMINARIES

2.1. Embeddings

An embedding of a graph $G = (V_G, E_G)$ (called guest graph) into a graph
$H = (V_H, E_H)$ (called host graph) is a mapping $\varphi: G \rightarrow H$ consisting of two
mappings $\varphi_V: V_G \rightarrow V_H$ and $\varphi_E: E_G \rightarrow \mathcal{P}(H)$. Here, $\mathcal{P}(H)$ denotes the set of
paths in the graph $H$. The mapping $\varphi_E$ maps each edge $\{v, w\} \in E_G$
to a path $p \in \mathcal{P}(H)$ connecting $\varphi_V(v)$ and $\varphi_V(w)$. We call an embedding
one-to-one if the mapping $\varphi_V$ is 1-1.

The dilation of an edge $e \in E_G$ under an embedding $\varphi$ is the length
of the path $\varphi_E(e)$. Here, the length of a path is the number of its edges.
The dilation of an embedding $\varphi$ is the maximal dilation of an edge in $G$. The
number of vertices of a guest graph which are mapped onto a vertex $v$ in the
host graph, is called the load of the vertex $v$. The load of an embedding $\varphi$
is the maximal load of a vertex in the host graph. In this paper, unless
otherwise noted, we only consider embeddings with unit load. The ratio
$|V_H|/|V_G|$ is called the expansion of the embedding $\varphi$. The congestion of an
edge $e' \in E_H$ is the number of paths in $\{\varphi_E(e) \mid e \in E_G\}$ that contain $e'$.
The edge-congestion is the maximal congestion over all edges in $H$. The
congestion of a vertex $v \in V_H$ is the number of paths in $\{\varphi_E(e) \mid e \in E_G\}$
containing $v$. Again, the node-congestion is the maximal congestion over all
vertices in $H$. 
2.2. Hypercubes and Trees

A hypercube of dimension $d$ is a graph with $2^d$ vertices, labeled 1-1 with the strings in $\{0, 1\}^d$. Two vertices are connected iff their labels differ in exactly one position. The smallest hypercube into which we can embed a graph $G = (V, E)$ with load one is called its optimal hypercube. Thus, the dimension of the optimal hypercube is $\lceil \log(|V|) \rceil$ and every embedding of a graph $G$ into its optimal hypercube has expansion less than two.

In the following, we initially restrict our attention to finding a suitable mapping $\phi_V$, and we will use shortest paths in the hypercube for the mapping $\phi_E$. Nevertheless, it is still important to decide which paths we choose, since we are interested in obtaining an embedding with small node-congestion.

As usual, a tree is a connected acyclic graph with one distinguished vertex, called the root. A vertex of degree 1 is called a leaf if it is not the root. A vertex is called an internal vertex if it is not a leaf. Given a vertex $v$ in a tree, another tree vertex $w$ is called a successor of $v$ if $v$ lies on the simple path from the root to $w$. We call $v$ also an ancestor of $w$. A vertex $w$ is called a child of a vertex $v$ if $w$ is a successor of $v$ and $v$ and $w$ are adjacent. The vertex $v$ is also called the parent of $w$.

The level of a tree vertex $v$ is the number of vertices on the simple path from the root to $v$. For instance, the level of the root is 1. The height of a tree $T$ is the maximum level of a vertex in $T$. A subtree rooted at a vertex $v$ is the induced subgraph of all successors of $v$ in the tree. In the sequel, we mean by a subtree always a subtree rooted at some tree vertex. Let $v$ and $w$ be two tree vertices, we call $u$ the lowest common ancestor of $v$ and $w$ if $u$ is the root of a smallest subtree containing $v$ and $w$.

A complete $d$-ary tree $T$ of height $h$ is a tree such that each internal vertex has exactly $d$ children and all leaves have level $h$. A nearly complete $d$-ary tree $T$ of size $n$ is a subgraph of the smallest complete $d$-ary tree of size at least $n$ such that the following hold: $T$ has exactly $n$ vertices, all but one of the internal vertex has $d$ children, and the levels of any two leaves differ by at most one.

2.3. Graphs with Bounded Treewidth

A tree-decomposition $\mathcal{D}_G$ for a graph $G = (V, E)$ is a pair $(T, X)$ consisting of a tree $T = (S, F)$ and a set $X = \{X_s \subseteq V : s \in S\}$ such that the following three conditions are satisfied:

(i) $\bigcup_{s \in S} X_s = V$,
(ii) $\forall \{v, w\} \in E: \exists s \in S: \{v, w\} \subset X_s$,
(iii) $\forall v \in V$: the induced subgraph of $T$ on the set $\{s : v \in X_s\}$ is a tree.
The tree $T$ is also called a decomposition tree for $G$. The width of a tree-decomposition $\mathcal{D}_G = (T, X)$ is defined as $\max\{|X_s| - 1 : s \in S\}$. The treewidth of a graph $G$ is the minimum width over all tree-decompositions for $G$.

Given a graph $G$ and an integer $t$, it is in general $NP$-complete to decide whether $G$ has treewidth at most $t$ as shown in [2]. On the other hand, for some special classes of graphs, e.g., permutation graphs, cotriangulated graphs, convex graphs, chordal bipartite graphs, and circle graphs, there exist polynomial time algorithms to determine the treewidth (see, e.g., [24]). For graphs with constant treewidth an algorithm for constructing a minimal tree-decomposition is described in [6]. This algorithm works in time $O(\log^2(n))$ using $O(n)$ operations on a EREW PRAM.

The size of a tree-decomposition $\mathcal{D}_G = (T, X)$ is defined as the cardinality of $X$. Note that $|X| = |S|$, where $T = (S, E)$. Given a graph $G$ with treewidth $t$, we always can find a small tree-decomposition for $G$. This follows from the fact that the class of graphs with treewidth $k$ coincides with the class of partial $k$-trees (see, e.g., [24]).

**Lemma 2.1.** Let $G = (V, E)$ be a graph with treewidth $t$. There exists a tree-decomposition $\mathcal{D}_G = (T, X)$ for $G$ of width $t$ such that $|X| \leq |V|$.

A tree-decomposition $\mathcal{D}_G = (T, X)$ for a graph $G = (V, E)$ is called binary if $T$ is a binary tree. It is possible to transform each tree-decomposition into a binary tree-decomposition as the next lemma shows. We will need the proof later to construct an efficient transformation of a tree-decomposition into a binary tree on the hypercube.

**Lemma 2.2.** Let $G = (V, E)$ be a graph with treewidth $t$. There exists a binary tree-decomposition $\mathcal{D}_G = (T, X)$ for $G$ of width $t$ and $|X| \leq 2|V|$.

**Proof.** Consider a vertex $s$ of the decomposition tree $T = (S, F)$ with $n > 2$ children. Replace vertex $s$ by a nearly complete binary tree with $n$ leaves. For each internal vertex $v$ of this nearly complete binary tree define $X_v = X_s$. Identify the $n$ leaves of this nearly complete binary tree with the $n$ children of $s$. Repeating this procedure until all vertices have at most two children yields a binary tree-decomposition for $G$. Since the number of internal vertices of a nearly complete binary tree is less than the number of its leaves, the size of the binary decomposition tree is at most twice the size of the given decomposition tree.

To establish some lower bounds on the dilation in the next section, we present some fundamental properties of graphs with bounded treewidth. The following well-known lemma can be found, e.g., in [24].

**Lemma 2.3.** Let $G = (V, E)$ be a graph containing a $t + 1$-clique. Then the treewidth of $G$ is at least $t$. 

A graph $G = (V, E)$ has a $s$-separator $(A, B, S)$, if there exists a partition of $V$ into the sets $A$, $B$, and $S$ such that the following conditions are satisfied:

(i) $|S| = s$,
(ii) $\frac{1}{3}(|V| - s) \leq |A|, |B| \leq \frac{2}{3}(|V| - s)$,
(iii) $S$ separates $A$ from $B$, i.e., there are no edges between $A$ and $B$.

It is well known that every graph of treewidth $t$ has a $t + 1$-separator (see, e.g., [24]).

**Lemma 2.4.** Let $G = (V, E)$ be a graph with treewidth $t$, then $G$ has a $t + 1$-separator.

### 3. LOWER BOUNDS

First, we will show that every embedding of a graph with treewidth $t$ and maximal degree $d \geq t$ into its optimal hypercube has dilation at least $\Omega(\log(d))$.

**Lemma 3.1.** Let $T = (V, E)$ be a nearly complete $d$-ary tree of size $n$. Every one-to-one embedding of $T$ into its optimal hypercube has dilation at least $\Omega(\log(d))$.

**Proof.** Given a one-to-one embedding, consider two vertices $v_0, v_1 \in V$ which are mapped to the hypercube vertices whose labels differ in at least $\lceil \frac{1}{2} \log(n) \rceil$ positions. The existence of such a pair of hypercube vertices can be seen as follows. The number of vertices whose labels differ in at most $\lceil \frac{1}{2} \log(n) \rceil - 1$ positions from a fixed vertex $v_0$ is at most

$$\sum_{i=0}^{\left\lfloor \frac{1}{2} \log(n) - 1 \right\rfloor} \binom{\left\lfloor \log(n) \right\rfloor}{i} \leq 2^{\left\lfloor \log(n) \right\rfloor - 1} < n.$$  

Since we consider a one-to-one embedding into the optimal hypercube, exactly $n$ hypercube vertices are images of the mapping. Thus, there must exist a vertex $v_1$ mapped to a hypercube vertex whose label differs in at least $\lceil \frac{1}{2} \log(n) \rceil$ position from $v_0$.

It is easy to see that the distance in $T$ between any two vertices of $T$ is at most $O(\log(n)/\log(d))$. Now we can conclude that at least one edge on the path from $v_0$ to $v_1$ has dilation of at least

$$\Omega\left(\frac{\log(n)/2}{\log(n)/\log(d)}\right) = \Omega(\log(d)).$$
For each \( n, d, \) and \( t \) with \( t \leq d < n \), there exists a graph \( T_{t,d}^n \) of size \( n \), treewidth \( t \), and maximal degree \( d \) such that each one-to-one embedding of \( T_{t,d}^n \) into its optimal hypercube requires dilation at least \( \Omega(\log(d)) \).

Proof. We construct the graph \( T_{t,d}^n \) from a nearly complete \( d-1 \)-ary tree \( T_c \) of size \( n \). The graph \( T_{t,d}^n \) is a supergraph of \( T_c \), where an edge \( \{v, w\} \) between \( v \) and \( w \) is added if \( v \) and \( w \) are leaves of the same parent and if \( v \) and \( w \) are one of the \( t \) left-most children of their parent. Lemma 2.3 implies that the treewidth of \( T_{t,d}^n \) is at least \( t \). In contrast, it can easily be seen that the treewidth of \( T_{t,d}^n \) is also at most \( t \). Clearly, the size of \( T_{t,d}^n \) is \( n \), and the maximal degree of a vertex in \( T_{t,d}^n \) is \( d \). Since \( T_{t,d}^n \) contains \( T_c \) as a subgraph, Lemma 3.1 implies that the dilation of each one-to-one embedding of \( T_{t,d}^n \) into its optimal hypercube has dilation at least \( \Omega(\log(d)) \).

Given a graph \( G = (V, E) \), we denote by \( N(U) := \{v : \{u, v\} \in E \wedge u \in U\} \) the neighborhood of the set \( U \subseteq V \). The expansion of a set of vertices \( U \subseteq V \) is defined as \((|N(U)\setminus U|)/|U|\). An \( \alpha \)-expander is a regular graph \( G = (V, E) \) such that every set \( U \subseteq V \) of size at most \( |V|/2 \) has expansion \( \alpha \), where \( \alpha \) is some fixed constant. Using probabilistic arguments, it is easy to prove the existence of an infinite family of constant-degree \( \alpha \)-expanders for some constant \( \alpha \) (see, e.g., [10, 31, 32]). Moreover, almost every regular random graph is an expander. On the other hand, it is quite hard to construct an infinite family of constant-degree \( \alpha \)-expanders deterministically. The following slightly modified lemma is presented in [4] and shows a lower bound on the dilation for \( \alpha \)-expanders.

Lemma 3.2 ([4]). Every one-to-one embedding of a constant-degree \( \alpha \)-expander with \( n \) vertices into its optimal hypercube has a dilation of at least \( \Omega(\alpha \log(n)) \).

Lemma 3.3. Every constant-degree \( \alpha \)-expander of size \( n \) has treewidth \( \Theta(n) \).

Proof. Let \( G = (V, E) \) be an \( \alpha \)-expander of size \( n \) with degree \( d \) and treewidth \( t \). By Lemma 2.4, consider a \( t+1 \)-separator \( (A, B, S) \) for \( G \). Without loss of generality, we assume that

\[
\frac{n - (t + 1)}{3} \leq |A| \leq \frac{n}{2}.
\]

Since \( S \) separates \( A \) from \( B \), we have \(|N(A)| \leq t + 1\). On the other hand, \( G \) is an \( \alpha \)-expander, which implies that

\[
|N(A)| \geq \alpha \cdot |A| \geq \alpha \frac{n - t - 1}{3}.
\]
Altogether we get
\[ t \geq \frac{\alpha(n - 1) - 3}{\alpha + 3} = \Omega(n). \]

Since each graph of size \( n \) has obviously treewidth of at most \( n \), the claim follows.

Combining Lemma 3.2 and Lemma 3.3 yields the following theorem.

**Theorem 3.2.** There exists an infinite family \( \mathcal{G} \) of graphs with treewidth \( \Theta(n) \) and constant degree, where \( n \) is the size of the graph, such that every one-to-one embedding of a graph into its optimal hypercube has dilation at least \( \Omega(\log(n)) \).

### 4. THE EMBEDDING

In this section, we give an embedding of a graph represented by a tree-decomposition into its optimal hypercube. First, we introduce our basic tool on which our construction of the embedding heavily relies. Then we give an overview of the embedding algorithm. Next, we explain in detail how to partition a graph represented by a decomposition tree, which is the essential step in our algorithm. Finally, we give an estimation for the dilation and node-congestion of the embedding.

#### 4.1. A Useful Tool—The \((k, h, o)\)-Tree

To construct our embedding, we use the data structure of a \((k, h, o)\)-tree. The \((k, h, o)\)-tree is a complete \(2^k\)-ary tree of height \( h \) with integer node weights, also called the *capacities* of the nodes. The capacities depend on the level of a vertex, and on the parameters \( o, k \in \mathbb{N} \). We also distinguish between a *full* and a *partial* \((k, h, o)\)-tree, which differ only in the capacity of the root. A node at level \( \ell \) of a full or partial \((k, h, o)\)-tree has a capacity of \( 2^o((2^k - 1)(k(h - \ell + 1) - k + 1) + k + \delta_{i,j}(kh + 1)) \) or \( 2^o((2^k - 1)(k(h - \ell + 1) - k + 1) - k) \), respectively. Here, \( \delta \) denotes the Kronecker symbol and \( \delta_{i,j} = 1 \) iff \( i = j \) and 0 otherwise. In the following, we call vertices of a \((k, h, o)\)-tree nodes, and we denote the capacity of a node at level \( \ell \) by \( c(\ell) \). Unless stated otherwise, we mean by a \((k, h, o)\)-tree a full \((k, h, o)\)-tree. Note that a subtree of a \((k, h, o)\)-tree is itself a partial \((k, h, o)\)-tree.

**Lemma 4.1.** Let \( T \) be a subtree of a \((k, h, o)\)-tree rooted at a node at level \( \ell \), then \( T \) itself is a partial \((k, h - \ell + 1, o)\)-tree.

In the following lemma, we determine the overall capacity of a \((k, h, o)\)-tree that is the sum of all capacities.
Lemma 4.2. The overall capacity of a partial or full \((k, h, o)\)-tree is \(2^{kh+o} - 2^o(kh + 1)\) or \(2^{kh+o}\), respectively.

Proof. We prove only the claim for a full \((k, h, o)\)-tree, since the claim for a partial \((k, h, o)\)-tree follows then immediately. By the definition of the capacities we get

\[
\sum_{\ell=1}^{h} 2^k(\ell-1) \cdot c(\ell)
= \sum_{\ell=1}^{h} 2^k(\ell-1)[2^o((2^k - 1)(k(h - \ell + 1) - k + 1) - k)] + 2^o(kh + 1)
= 2^o \left[ (k - 2^k) \left( \frac{2^{kh} - 1}{2^k - 1} - k \sum_{\ell=0}^{h-1} \ell 2^\ell \right) \right.
\quad - \left. k \frac{2^{kh} - 1}{2^k - 1} + kh + 1 \right]
= 2^o \left[ (kh - k + 1)(2^{kh} - 1) + kh + 1 \right.
\quad - \left. (2^k - 1)k \frac{(h - 1)2^{kh+1} - h2^{kh} + 2^k}{(2^k - 1)^2} - k \frac{2^{kh} - 1}{2^k - 1} \right]
= 2^o \left[ kh2^{kh} - kh - (k - 1)2^{kh} + (k - 1) + kh + 1 \right.
\quad - \left. k \frac{2^{kh}h2^{kh} - h2^{2kh} + 2^{2kh} + 2^k}{2^k - 1} \right]
= 2^o \left[ kh2^{kh} - (k - 1)2^{kh} + k - kh2^{kh} + k2^{kh} - k \right]
= 2^{kh+o}.
\]

4.2. Embeddings Using the \((k, h, o)\)-Tree

We now describe a mapping of a \((k, h, o)\)-tree into its optimal hypercube such that each node of the \((k, h, o)\)-tree occupies as many vertices of the hypercube as given by its capacity. Each node in a \((k, h, o)\)-tree can be represented by a string in \(\{0, 1\}^k\) as follows. The empty string \(\varepsilon\) represents the root of the \((k, h, o)\)-tree. If \(\alpha\) represents a node \(v\) of the \((k, h, o)\)-tree, then the strings \(\alpha\beta\) for \(\beta \in \{0, 1\}^k\) represent the \(2^k\) children of \(v\) from
left to right. We define the following sets of hypercube locations, where \( \alpha \) represents some arbitrary node in a \((k, h, o)\)-tree:

\[
S_\alpha := \{ \alpha \beta \gamma \delta \in \{0,1\}^{kh+o} : \beta \in (0+1)^*1 \land |\beta| \leq k \\
\land \gamma \in 0^*10^* \land \delta \in \{0,1\}^o \}
\]

\[
R := \{ \gamma \delta \in \{0,1\}^{kh+o} : \gamma \in (0^*10^* + 0^*) \land \delta \in \{0,1\}^o \}.
\]

We also define the set \( S := \bigcup S_\alpha \). The vertices of the given graph mapped to the node of a \((k, h, o)\)-tree represented by \( \alpha \) will finally be mapped to hypercube locations in the set \( L_\alpha := S_\alpha \) if \( \alpha \neq e \), and \( L_e := S \cup R \) otherwise.

We will now show that the capacity of a node in the \((k, h, o)\)-tree is equal to the cardinality of the set of vertices in the hypercube to which it is mapped.

**Lemma 4.3.** Let \( \alpha \) be a representation of a \((k, h, o)\)-tree node at level \( \ell \), then \( |S_\alpha| = 2^\alpha((2^k - 1)(k(h - \ell + 1) - k + 1) - k) \). Furthermore, \( |R| = 2^\alpha(kh + 1) \) and \( |L_\alpha| = c(\ell) \).

**Proof.** The equality on \(|R|\) follows immediately from the definition of \( R \).

Since \( \alpha \) represents a node of the \((k, h, o)\)-tree at level \( \ell \), we have \(|\alpha| = (\ell - 1)k\). By definition of \( S_\alpha \), we obtain \(|\gamma| = kh - |\alpha \beta| = k(h - \ell + 1) - |\beta|\). Thus, we get

\[
|S_\alpha| = 2^\alpha \sum_{\beta \in (0+1)^*1 \atop |\beta| \leq k} |\gamma| = 2^\alpha \sum_{\beta \in (0+1)^*1 \atop |\beta| \leq k} (k(h - \ell + 1) - |\beta|) = 2^\alpha \sum_{i=0}^{k-1} 2^i[k(h - \ell + 1) - (i + 1)] = 2^\alpha \left( (2^k - 1)(k(h - \ell + 1)) - \frac{1}{2} \sum_{i=1}^{k} i2^i \right) = 2^\alpha \left( (2^k - 1)(k(h - \ell + 1)) - \frac{1}{2}(2^{k+2} - (k + 1)2^{k+1} + 2) \right) = 2^\alpha((2^k - 1)(k(h - \ell + 1) - k + 1) - k).
\]

Furthermore, it can easily be verified that for every \( \alpha \neq \alpha' \in \{0,1\}^k \)
\( S_\alpha \cap S_{\alpha'} = \emptyset \) and \( S_\alpha \cap R = \emptyset \), and hence \( L_\alpha \cap L_{\alpha'} = \emptyset \). Hence, for each string \( s \in S \cup R \) there is a unique decomposition \( s = \alpha \beta \gamma \delta \) as used in the definition of \( S_\alpha \) and \( R \).
implies immediately that their labels differ in at most two positions. Hence, the labels of any two vertices that differ in at most two positions cannot be the same for any two vertices in the same set. Therefore, the labels of any two vertices in the same set differ in at most two positions.

**LEMMA 4.4.** Let \( T \) be a subtree of a \((k, h, o)\)-tree of height \( h'\) and \( A_T \) the set of representations of nodes belonging to \( T \), then \( \bigcup_{\alpha \in A_T} \{L_\alpha \} \) is contained in a \( k \cdot h' + o\)-dimensional subcube.

The following lemma is fundamental to prove the claimed bounds on the dilation of our embeddings.

**LEMMA 4.5.** Let \( \alpha_1 \) and \( \alpha_2 \) represent two nodes in a \((k, h, o)\)-tree such that their lowest common ancestor represented by \( \alpha \) is at distance \( \Delta \) from both nodes. Let \( v \in L_{\alpha_1} \) and \( w \in L_{\alpha_2} \), then \( v \) and \( w \) differ in at most \((\Delta + 1)k + o + 2\) positions.

**Proof.** We first consider the case where both \( v \) and \( w \) belong to the set \( S \). The diagram in Fig. 1 shows the unique decomposition of \( v \) and \( w \). In this picture, \( \alpha \) represents the lowest common ancestor of the \((k, h, o)\)-tree nodes represented by \( \alpha_1 \) and \( \alpha_2 \). Therefore, \( \alpha_1 = aa' \) and \( \alpha_2 = aa'' \). Without loss of generality, we assume \(|\alpha'| \leq |\alpha''|\). Since the lowest common ancestor of the \((k, h, o)\)-tree nodes represented by \( \alpha_1 \) and \( \alpha_2 \) is at distance of at most \( \Delta \) from both nodes, we get \(|\alpha'| \leq |\alpha''| \leq k \cdot \Delta \). The definition of the sets \( L_\alpha \) implies that \(|\beta'|, |\beta''| \leq k, |\delta'| = |\delta''| = o\), and that \( \gamma' \) and \( \gamma'' \) contain exactly one 1 each. Hence, the labels of \( v \) and \( w \) differ in at most \((\Delta + 1)k + o + 2\) positions.

We now consider the case that \( v \in R \) and that \( w \in S \), i.e., \( v = \gamma' \delta' \) and \( w = \alpha' \beta'' \gamma'' \delta'' \). Note that in this case necessarily \( \alpha = \epsilon \). Since the distance between corresponding \((k, h, o)\)-tree nodes of \( v \) and \( w \) is \( \Delta \), we have \(|\alpha''| \leq k \cdot \Delta \). Hence there are at most \( k \cdot \Delta + k + 1 = (\Delta + 1)k + 1 \) 1’s in the first \( k \cdot h \) positions in \( w \). By definition of the set \( R \), \( v \) has at most one 1 in the first \( k \cdot h \) positions. Thus, \( v \) and \( w \) differ in at most \((\Delta + 1)k + 1 + o = (\Delta + 1)k + o + 2\) positions.

Finally, if both vertices belong to the set \( R \), then the definition of \( R \) implies immediately that their labels differ in at most \( o + 2\) positions. 

**4.3. The Embedding Algorithm**

Given a graph \( G = (V, E) \), we denote the number of vertices in \( G \) by \( n \), and the maximal degree of a vertex in \( G \) by \( d \). By Lemmas 2.1 and 2.2, we assume for simplicity that we have a binary tree-decomposition \( \mathcal{D}_G = (S, F) \).
of width $t$ and size $\leq 2n$ for $G$. We will see that it is not necessary to bound the size of the tree-decomposition to obtain the claimed dilation.

In the following, we assume that $d \geq 3$, since all graphs whose vertices have degree at most two can be embedded one-to-one with dilation of at most two into the 2-dimensional grid of size $2 \times \lceil \frac{n}{2} \rceil$, which is obviously a subgraph of the optimal hypercube.

Our embedding of graphs with bounded treewidth into optimal hypercubes is achieved in two steps. First, we embed the graph into a $(k, h, o)$-tree. This will be explained in detail in the following. Then, we use the mapping presented in the previous section to complete the embedding.

To obtain a small dilation, adjacent vertices of the graph are mapped to nodes which are close in the $(k, h, o)$-tree. Our goal is to obtain an embedding of the graph into a $(k, h, o)$-tree such that adjacent vertices are mapped to two nodes of the $(k, h, o)$-tree with distance at most 1 from their lowest common ancestor of the $(k, h, o)$-tree. Our method leads to an embedding of a graph $G$ into the hypercube with dilation $2k + o + 2$.

The embedding of the graph $G$ into the $(k, h, o)$-tree will be performed as follows. First, we fill up the root of the $(k, h, o)$-tree with arbitrarily chosen vertices of $G$ and remove these vertices from $G$, yielding $G'$. Then, we mark the unmapped neighbors of the mapped vertices in the resulting graph $G'$. We associate this graph $G'$ with the root of the $(k, h, o)$-tree. Now we partition this graph $G'$ into $2^k$ parts using its decomposition tree such that the marked vertices are distributed evenly over all $2^k$ parts and that the number of edges between different parts is not too large. Such a partition will be given in the next subsection. Associate to each of the children one of the parts of the decomposed graph $G'$. We will call edges between different parts partition edges and vertices incident to such edges partition vertices.

In the next step, we fill up the children of the root with the marked vertices in the associated subgraph of $G'$. Additionally, we map the partition vertices of the previous decomposition to the $2^k$ children of the root. Finally, we fill up the nodes of the $(k, h, o)$-tree with arbitrarily chosen vertices of the associated subgraph until the capacity of this node exceeds.

We will repeat this process until we have reached the leaves of the $(k, h, o)$-tree. In this case, we map all remaining vertices which belongs to the associated subgraph to their corresponding leaves of the $(k, h, o)$-tree.

Since we are looking for an embedding into the optimal hypercube, we must choose the parameters of the $(k, h, o)$-tree such that $kh + o = \lfloor \log(n) \rfloor$. It will be shown that we can choose $k = \lfloor \log(2(d+1)(t+1)) \rfloor$. Hence, we must choose $o$ such that $o \equiv \lfloor \log(n) \rfloor \mod k$. Thus, the height of the $(k, h, o)$-tree is determined by

$$h = \frac{\lfloor \log(n) \rfloor - o}{k}.$$
4.4. Partitioning a Graph with Bounded Treewidth

In this subsection, we present a technique to partition a graph with bounded treewidth based on its decomposition tree. We start with a fundamental lemma on partitioning strings.

Let \( s \in \{1, \ldots, m\}^* \). By \( |s_i| \) we denote the number of occurrences of the symbol \( i \) in \( s \), and by \( |s| \) we denote the length of the string \( s \). Hence, we have \( |s| = \sum_{i=1}^{m} |s_i| \). The following lemma can be found in \([1, 5, 11, 16]\).

**Lemma 4.6.** Let \( s \in \{1, \ldots, m\}^* \). There exists a decomposition of \( s \) into substrings \( s_i \), for \( i = 0, \ldots, m \), such that \( s = s_0 \cdots s_m \), and a partition of \( \{0, \ldots, m\} \) into sets \( J_1 \) and \( J_2 \), such that

\[
\left\lfloor \frac{|s|}{2} \right\rfloor \leq \sum_{j \in J_1} |s_j| \leq \left\lceil \frac{|s|}{2} \right\rceil
\]

and

\[
\left\lfloor \frac{|s|}{2} \right\rfloor \leq \sum_{j \in J_2} |s_j| \leq \left\lceil \frac{|s|}{2} \right\rceil
\]

for \( k = 1, \ldots, m \) and \( \ell = 1, 2 \).

**Proof.** For lack of space, we only give the proof for \( m = 2 \) (which is all we use). We assume that \( n \) is even, a similar argument holds if \( n \) is odd. Without loss of generality, the number of 1’s in the left half of \( s \) is at least as large as the number of 1’s in the right half. Sliding a window of length \( \frac{n}{2} \) across \( s \), there must exist a position of the window such that the number of 1’s in the window is

\[
\left\lfloor \frac{|s_1|}{2} \right\rfloor.
\]

The number of 2’s in the window then is

\[
\frac{n}{2} - \left\lfloor \frac{|s_1|}{2} \right\rfloor = \frac{|s_1| + |s_2|}{2} - \left\lfloor \frac{|s_1|}{2} \right\rfloor = \left\lceil \frac{|s_2|}{2} \right\rceil.
\]

The two endpoints of the window give the required cuts for the strings. \[\]

We will need this proof later to construct such a bisection of a string \( s \) efficiently. Note that Lemma 4.6 is optimal in the sense that there exist strings that cannot be divided as required using fewer substrings.

Since a vertex in the decomposition tree represents several vertices of the given graph \( G = (V, E) \), we introduce a weight function \( w: S \rightarrow \mathbb{N} \) on the decomposition tree \( T = (S, F) \) of a tree-decomposition \( (T, X) \) for \( G \). For our purposes, the following weight function is sufficient (but another one might improve the dilation). For a tree vertex \( s \), let \( w(s) = |X_s \setminus X_{\text{par}(s)}| \).
except for the root \( r \) of \( T \), and let \( w(r) = |X_r| \). Here, \( \text{par}(s) \) denotes the parent of a vertex \( s \) in a tree. The weight of a subtree is defined as the sum of the weights of vertices which belong to this subtree. Note that \( w(T) = |V| \). We remark that condition (iii) in the definition of a tree-decomposition and our definition of the weight function implies that subtrees with weight 0 can be removed from the decomposition tree without removing some edge of the represented graph.

As described in the previous subsection, we also have marked vertices in the subgraph associated to a \((k, h, o)\)-tree node. We recall that a vertex is marked if at least one neighbor is mapped to a \((k, h, o)\)-tree node in a previous stage. The marked graph vertices will be represented by a second weight function \( w' \) on the decomposition tree. For each vertex \( s \) in the decomposition tree, let \( w'(s) \) be the number of marked vertices in \( X_s \setminus X_{\text{par}(s)} \). Again, \( w'(T) \) is equal to the number of marked vertices in \( G \).

In order to partition decomposition trees using Lemma 4.6, we need a linear representation of such forests like inorder strings. In the following, the string listing the vertices of \( T \) in inorder is called the inorder string of \( T \). Because each vertex of the decomposition tree represents several vertices of the given graph \( G \), we mean by an inorder string of a decomposition tree its inorder string, where each vertex \( s \) of the decomposition tree is replaced by the string \( 1^{w(s)} \cdot 2^{w'(s)} \). Obviously, there is a one-to-one correspondence between the unmarked and marked vertices in the graph \( G \) and the 1’s and 2’s, respectively, in the modified inorder string of the decomposition tree for \( G \).

Since we are interested in partitioning a decomposition tree cutting at most a logarithmic number of edges, and since the decomposition tree may be highly imbalanced, we cannot simply split the vertices into two parts as given by the lower and upper half of the inorder string. Instead we must reorganize the trees in the forests. In the following, we restrict our discussion to a single tree.

A tree vertex is called a left or right vertex if it is the left or right child of its parent, respectively. For simplicity, the root is a left vertex. An edge is called a left or right edge if it is the edge from a vertex to its left or right child, respectively. We call a left or right vertex folded if the weight of its left or right subtree, respectively, is less than the weight of its right or left subtree, respectively. Otherwise, we call the vertex unfolded. A tree is called unfolded if all its vertices are unfolded and folded otherwise. Our goal is to transform a given binary tree into an unfolded tree. We refer to this transformation simply as unfolding. Clearly, a binary tree can be transformed into an unfolded binary tree in linear time using two depth first search traversals of the tree. An efficient implementation to unfold binary trees on the hypercube will be given in the next section. Note that in general a subgraph of an unfolded tree is not necessarily unfolded.
After the unfolding, a path from the root to a vertex with positive weight can use at most \(\lfloor \log(w(T)) \rfloor \)-times an edge from a left vertex to a right vertex or from a right vertex to a left vertex, since each such edge at least halves the weight of the remaining subtree.

This construction together with Lemma 4.6 can be used to partition a graph containing marked vertices represented by a binary decomposition trees into \(2^k\) parts. But before we state the lemma on partitioning graphs represented by decomposition trees, we first need a technical lemma (see, e.g., [15]):

**Lemma 4.7.** Let \(f: \mathbb{R} \to \mathbb{R}\) be a continuous and monotonically increasing function such that \(f(x) \in \mathbb{Z} \Rightarrow x \in \mathbb{Z}\), then \(\lceil f(x) \rceil = \lceil f(\lfloor x \rfloor) \rceil\).

Now we are ready to prove the following fundamental lemma:

**Lemma 4.8.** Let \(G = (V,E)\) be a graph and \(D_G = (T,X)\) a tree-decomposition of width \(t\) for \(G\) and let \(V' \subseteq V\) be a set of marked vertices in \(G\). Then \(G\) can be decomposed into \(2^k\) subgraphs \(G_i = (V_i,E_i)\) \((i \in \{1, \ldots, 2^k\})\) represented by \(2^k\) decomposition trees \(T_i\) of width at most \(t\) such that the following conditions are satisfied:

(i) \(\bigcup_{i=1}^{2^k} V_i = V\),

(ii) \(\bigcup_{i=1}^{2^k} E_i \subseteq E\),

(iii) \(|V_i| \leq \left\lceil \frac{|V|}{2^k} \right\rceil\),

(iv) in each \(G_i\) are at most

\[
\left\lceil \frac{4(t + 1)(d + 1)(2^k - 1) \log(|V'|) + |V'|}{2^k} \right\rceil
\]

partition and marked vertices.

**Proof.** Proof by induction on \(k\). Let \(T = (S,F)\) be the decomposition tree for \(G\) and let \(w, w': S \to \mathbb{N}\) be two weight functions defined as above.

**Induction base \((k = 1)\).** We first unfold the decomposition tree \(T\) as described above. As mentioned earlier, we replace each vertex \(s \in S\) in the inorder string of the unfolded decomposition tree by the string \(1^{w(s) - w'(s)}2^{w'(s)}\). Using Lemma 4.6 for \(m = 2\) on the modified inorder string of the unfolded decomposition tree, we cut the inorder string at most twice. Note that cutting the string listing the vertices of the tree in inorder corresponds to cutting some edges of the path from the root to a leaf, and note that these edges alternate between right and left. Since we have replaced a vertex \(s\) of a decomposition tree by a string \(x\) of length \(w(s)\), it is possible that the cut through the inorder string could also cut the string \(x\) which means that a tree vertex could be cut too (cf. Fig. 2). It can
FIG. 2. Decomposition of a tree by cutting its inorder string.

easily be seen that a cut through the modified inorder string of the decomposition tree can cut at most one tree vertex. If we cut a tree vertex of the decomposition tree, each part gets a copy of this cut vertex. This yields two forests of decomposition trees which can be extended to two decomposition trees $T_1$ and $T_2$ of width at most $t$ by adding some tree edges. The graphs $G_i$ are the graphs represented by the decomposition trees $T_i$ for $i = 1, 2$. Obviously, conditions (i) and (ii) hold.

As stated above, at most $2\lceil \log(w(T)) \rceil$ edges and vertices of the decomposition tree $T$ could be cut. Consider a cut tree edge $(r, s)$. Since only vertices in the set $X_r \cap X_s$ and their neighbors could be partition vertices, the number partition vertices produced by cutting a tree edge is at most $(t + 1)(d + 1)$. This is illustrated in Fig. 3. The vertices in the set $X_r \cap X_s$ are drawn black and partition edges are indicated by dashed lines. Cutting a vertex of the decomposition tree produces also at most $(t + 1)(d + 1)$ partition vertices. Hence at most

$$\left\lceil 2(t + 1)(d + 1) \log(w(T)) + \frac{w(T)}{2} \right\rceil = \left\lceil 2(t + 1)(d + 1) \log(|V|) + \frac{|V'|}{2} \right\rceil$$

partition and marked vertices could be in each part $G_i$, which is claimed in condition (iv).

Condition (iii) is obviously satisfied by Lemma 4.6.

FIG. 3. Partition vertices created by cutting a decomposition tree edge.
Induction step ($k-1 ightarrow k$). Using the induction hypotheses for $k = 1$, we obtain a partition into two graphs $G^1$ and $G^2$ represented by two decomposition trees, where

$$|V'| \leq \left\lfloor \frac{|V|}{2} \right\rfloor$$

and $G^1$ as well as $G^2$ contains at most

$$\left\lfloor 2(t+1)(d + 1) \log(|V|) + \frac{|V'|}{2} \right\rfloor$$

marked and partition vertices.

Now, we also mark all partition vertices. Again, we unfold both decomposition trees and modify the inorder string as above. Using on each graph $G^i$ the induction hypotheses for $k-1$, we obtain a decomposition of $G$ into $2^k$ graphs $G_i$ represented by $2^k$ decomposition trees $T_i$ of width $t$. Clearly, conditions (i) and (ii) hold.

The induction hypotheses and Lemma 4.7 imply that (iii) is fulfilled:

$$|V'_i| \leq \left\lfloor \frac{1}{2^{k-1}} \left\lfloor \frac{|V|}{2} \right\rfloor \right\rfloor = \left\lfloor \frac{|V|}{2^k} \right\rfloor.$$

Using Lemma 4.7, the number of marked and partition vertices in each $G_i$ can be bound by

$$\leq \left\lfloor \frac{4(t+1)(d + 1)(2^{k-1} - 1) \log\left(\frac{|V'|}{2^k}\right) + \left\lfloor \frac{4(t+1)(d + 1)\log(|V|) + |V'|}{2^k} \right\rfloor}{2^{k-1}} \right\rfloor$$

$$\leq \left\lfloor \frac{4(t+1)(d + 1)(2^k - 2) \log(|V|) + 4(t+1)(d + 1) \log(|V'|) + |V'|}{2^k} \right\rfloor$$

$$\leq \left\lfloor \frac{4(t+1)(d + 1)(2^k - 1) \log(|V|) + |V'|}{2^k} \right\rfloor.$$

4.5. Dilation of the Embedding

It remains to show that we never map more vertices of the graph to a node of the $(k, h, o)$-tree as its capacity allows. Let $n(\ell)$ denote the maximal number of marked and partition vertices mapped to a node of the $(k, h, o)$-tree at level $\ell$. Hence, we must show that $n(\ell) \leq c(\ell)$ for all $1 \leq \ell \leq h$. By $s(\ell)$ we denote the maximal size of the subgraph which is associated to a node of the $(k, h, o)$-tree at level $\ell$. Since we embed the graph $G = (V, E)$ into its optimal hypercube, we have $n = |V| \leq 2^{kh+o}$. In each iteration the
size of the associated subgraph to a node of the \((k, h, o)\)-tree shrinks by a factor of \(2^k\) so that
\[
s(\ell) \leq \left\lceil \frac{2^{kh+o}}{2^{k(\ell-1)}} \right\rceil.
\]
Since we fill up the root of the \((k, h, o)\)-tree with arbitrary nodes of the graph \(G\), the inequality for \(\ell = 1\) holds. On the other hand, we can never map more than \(c(h)\) vertices to a leaf of the \((k, h, o)\)-tree, since in each stage we distribute the unmapped vertices of the graph evenly to the children of the considered node of the \((k, h, o)\)-tree.

Now we bound for \(2 \leq \ell \leq (h-1)\) the number of partition and marked vertices mapped to a node of the \((k, h, o)\)-tree. Let \(G_i = (V_i, E_i)\) associated to a node \(x\) of the \((k, h, o)\)-tree at level \(\ell - 1\). By Lemma 4.8 each graph associated to a child of \(x\), and thus at level \(\ell\), contains at most \(\left\lceil \frac{1}{2}(4(t + 1)(d + 1)(2^k - 1) \log(|V_i|) + |V_i'|) \right\rceil\) marked and partition vertices.

Obviously, \(|V'_i| \leq s(\ell - 1)\). The number of marked vertices of a graph associated to a \((k, h, o)\)-tree node at level \(\ell - 1\) is at most \(d \cdot c(\ell - 1)\). Hence, we get \(|V'_i| \leq d \cdot c(\ell - 1)\). Altogether, we get the following bound on \(n(\ell)\) which should be less equal \(c(\ell)\).

\[
n(\ell) \leq \left\lceil \frac{d \cdot c(\ell - 1) + 4(t + 1)(d + 1)(2^k - 1)\log(s(\ell - 1))}{2^k} \right\rceil
\]
\[
\leq \left\lceil \frac{d}{2^k} \left(2^o(2^k - 1)k(h - \ell + 2) - 2^o(2^k - 1)(k - 1) - k2^o \right.
\]
\[
+ \delta_{t, 2}2^o(hk + 1) + 4(t + 1)(d + 1)\log\left(\frac{2^{kh+o}}{2^{k(\ell-2)}}\right) + 1
\]
\[
\leq 2^o d\left( k(h - \ell + 2) - k + 1 + \delta_{t, 2} \frac{hk + 1}{2^k} \right)
\]
\[
+ 4(t + 1)(d + 1)[hk + o - k\ell + 2k] + 1
\]
\[
\leq 2^o d\left( k(h - \ell + 1) + 1 + \delta_{t, 2} \frac{hk + 1}{2^k} \right)
\]
\[
+ 4(t + 1)(d + 1)[k(h - \ell + 1) + o + k] + 1. \quad (1)
\]

Hence \(n(\ell)\) is less than \(c(\ell) = 2^o(2^k - 1)k(h - \ell + 1) - 2^o(2^k - 1)(k - 1) - 2^o k\) if the following inequalities are true:

\[
2^o d + 4(t + 1)(d + 1)(o + k) + 2^o(2^k - 1)(k - 1)
\]
\[
+ 2^o k + 1 + \delta_{t, 2}d(hk + 1) \frac{2^o}{2^k}
\]
\[
\leq (h - \ell + 1)k[2^o(2^k - 1) - 2^o d - 4(t + 1)(d + 1)].
\]
A simple computations shows that this is equivalent to
\[
2^o \left( d - 2^k + 1 + \frac{1}{2^o} \right) + 2^o 2^k k + 4(k + o)(t + 1)(d + 1) + \delta_{k, 2^o} \frac{d(hk + 1)}{2^k} \\
\leq (h - \ell + 1)k[2^o(2^k - d - 1) - 4(t + 1)(d + 1)].
\]
As mentioned above, we now choose \( k = \lfloor \log(2(d + 1)(t + 1)) \rfloor \) and \( o \in [4:2^k + 3] \) such that \( o \equiv \lfloor \log(n) \rfloor \mod k \). Thus, the inequality is fulfilled if
\[
2^o \left( (d+1) + \frac{1}{2^o} - 2(d+1)(t+1) \right) \\
+ 2^o \cdot k(t+1)(d+1) + 4(k + o)(t + 1)(t + 1) + \delta_{k, 2^o} \frac{d(hk + 1)}{2(d+1)(t+1)} \\
\leq (h - \ell + 1)k[2^o(2(t+1)(d+1) - (d+1)) - 4(t+1)(d+1)].
\]
Dividing the previous inequality by \( k \) yields the following inequality:
\[
(d+1) + \frac{1}{k} + (t+1)(d+1) \left[ 2 \cdot 2^o \left( 1 - \frac{1}{k} \right) + 4 \cdot \frac{k + o}{k} \right] + \delta_{k, 2^o} \frac{h+1}{d+1} \\
\leq (h - \ell + 1)[(t+1)(d+1)(2 \cdot 2^o - 4) - 2^o(d+1)]
\]
For \( 3 \leq \ell \leq h - 1 \), the above inequality (2) is satisfied if the following inequality holds (note, that \( o \geq 4 \) and \( (h - \ell + 1) \geq 2 \):
\[
2^o \left( d + \frac{1}{k} \right) + (t+1) \left[ 2 \cdot 2^o \left( 1 - \frac{1}{k} \right) + 4 \cdot \frac{k + o}{k} \right] \\
\leq (t+1)(4 \cdot 2^o - 8) - 2 \cdot 2^o.
\]
This is equivalent to
\[
2 \cdot 2^o + \frac{2^o}{k} + \frac{1}{k(d+1)} \leq (t+1) \left[ 2 \cdot 2^o \left( 1 + \frac{1}{k} \right) - 8 - 4(k + o) \right].
\]
Since \( t \geq 1 \), it is sufficient if the following inequalities are satisfied:
\[
24 + \frac{8o}{k} + \frac{1}{k(d+1)} \leq 2 \cdot 2^o + \frac{3}{k} 2^o.
\]
Since \( o \geq 4 \), it suffices to fulfill the following inequality:
\[
\frac{8o}{k} + \frac{1}{k(d+1)} \leq 8 + \frac{3}{k} 2^o.
\]
Because \( o \geq 4 \) implies \( 8o \leq 3 \cdot 2^o \), this inequality is true.
For $\ell = 2 \leq h - 1$, from inequality (2) follows that the following inequality should be satisfied.

$$(d + 1) \frac{2^\omega}{k} + \frac{1}{k} + (t + 1)(d + 1) \left[ 2 \cdot 2^\omega \left( 1 - \frac{1}{k} \right) + 4 \frac{k + \omega}{k} \right] + 2^\omega \frac{h - 1 + 2}{2(t + 1)}$$

$$\leq (h - 1)[(t + 1)(d + 1)(2 \cdot 2^\omega - 4) - 2^\omega(d + 1)]$$

Division by $(d + 1)2^\omega$ yields

$$\frac{1}{k} + \frac{1}{2^\omega k(d + 1)} + (t + 1) \left[ 2 \left( 1 - \frac{1}{k} \right) + 4 \frac{k + \omega}{k2^\omega} \right] + \frac{1}{2(t + 1)(d + 1)}$$

$$\leq (h - 1) \left[ (t + 1) \left( 2 - \frac{4}{2^\omega} \right) - 1 - \frac{1}{2(d + 1)(t + 1)} \right].$$

Since $k \geq 4$, $o \geq 4$, $d \geq 3$ and $t \geq 1$, it is sufficient to satisfy the following inequality:

$$(t + 1) \left[ 2 + \frac{4 + \omega}{2} + \frac{81}{256} \right] - \frac{17}{16} \leq (h - 1). \quad (3)$$

For each fixed $t \geq 1$, the left-hand side of inequality (3) converges as $o \to \infty$. It can easily be verified from Fig. 4 that for $o \geq 4$ the left-hand side of inequality (3) is less then 2 except for $(t = 1, o = 4)$. A simple recomputation of the right-hand side of inequality (1) for $h = 3$, $t = 1$, and $o = 4$ yields that it is less than $c(2)$ for $d \geq 3$. Hence, for $h \geq 3$ we obtain $n(\ell) \leq c(\ell)$ for all $\ell \in [2:h]$. This implies the required bound on the dilation for our embedding. For $h \leq 2$, we trivially have a dilation...
2k + o embedding of any graph with at most $2^{2k+o}$ vertices into its optimal hypercube.

A careful inspection of the previous estimation shows that the dilation of the embedding does not depend on the size of the given decomposition tree. As we will see in the following section, only the running time of the algorithm depends on the size of the given decomposition tree.

Altogether, we obtain the following theorem.

**Theorem 4.1.** Let $G = (V, E)$ be a graph with treewidth $t$ and maximal degree $d$. There exists a one-to-one embedding of $G$ into its optimal hypercube with dilation at most $3 \lceil \log((d + 1)(t + 1)) \rceil + 8$.

**Proof.** We have shown that such a graph can be embedded one-to-one into its optimal hypercube with dilation of at most $2k + o + 2$, where $k = \lceil \log(2(d + 1)(t + 1)) \rceil$ and $o \geq 4$ such that $o \equiv \log(n) \mod k$. Thus, we have $o \leq k + 3$. Hence, the dilation is bounded by $3k + 5 \leq 3 \lceil \log((d + 1)(t + 1)) \rceil + 8$. □

### 4.6. Congestion of the Embedding

In the remainder of this section, we will bound the node-congestion of the given embedding. We restrict our attention to bound the node-congestion, since the edge-congestion is less than or equal to the node-congestion.

Consider two adjacent vertices of the given graph which are mapped to hypercube locations labeled $v$ and $w$. We decompose the label into four segments $A$, $B$, $C$, and $D$. Segment $D$ consists of the last $o$ bits. The lengths of the segments $A$, $B$, and $C$ are multiples of $k$. Segment $C$ is the longest suffix before segment $D$ such that it contains at most one 1 in both $v$ and $w$ and its length is a multiple of $k$. Segment $B$ consists of the $2k$ positions before segment $C$, and segment $A$ is the remainder. See Fig. 5 for an illustration of this decomposition. Recall that the hypercube locations of vertices $v$ and $w$ can differ only in segments $B$ and $D$, and in at most 2 positions of segment $C$. See Fig. 5 for the case $v, w \in S$, the positions where the labels can differ are indicated by shading. Also note that segments $A$ and $B$ can be empty.

For a path $p_{v, w}$ from $v$ to $w$, we call $v$ the lower endpoint of $p_{v, w}$, if $v = \alpha_v \beta_v \gamma_v \delta_v \in S$, $w = \alpha_w \beta_w \gamma_w \delta_w \in S$ (cf. Fig. 5) and $|\gamma_v| < |\gamma_w|$, or if $v \in S$ and $w \in R$. Otherwise, if $v, w \in S$ and $|\gamma_v| = |\gamma_w|$ or if $v, w \in R$, we

![FIG. 5. Hypercube locations of two adjacent tree vertices.](image-url)
arbitrarily select one endpoint of the path $p_{v,w}$ to be the lower endpoint. The endpoint of the path $p_{v,w}$ not being the lower endpoint is called the upper endpoint.

To construct a shortest path $p_{v,w}$ from $v$ to $w$ in the hypercube, we proceed in four phases. Without loss of generality, we assume that $v$ is the lower endpoint; otherwise, we execute the routing about to be described in reverse order. In the first phase, we flip the bit position in segment $C$ which has to be changed from 0 to 1, if it exists. In the second phase, we flip those bit positions in segment $D$ that need to be changed. In the third phase, we first flip in segment $B$ 0’s to 1’s from right to left that must be changed. Then we flip 1’s to 0’s from left to right whenever necessary. Finally, we flip the bit position in segment $C$ which has to be changed from 1 to 0, if it exists.

**Theorem 4.2.** Let $G = (V,E)$ be a graph with treewidth $t$ and maximal degree $d$. There exists a one-to-one embedding of $G$ into its optimal hypercube with dilation at most $3\lceil\log(d+1)(t+1)\rceil + 8$ and node-congestion $O(d(dt)^3)$.

**Proof.** We only have to prove the claim on the node-congestion. To obtain an upper bound, we consider a fixed hypercube location $u$. In what follows, we will bound the number of hypercube locations which can be an upper or lower endpoint of a path hitting $u$. If this number is bounded by $c$, the node-congestion is bounded by $d \cdot c$, since the degree of a vertex is at most $d$ and hence at most $d$ paths can originate at a single hypercube vertex. We distinguish four different cases, depending on the number of 1’s in segment $C$ for a given path.

**Case 1.** First, we consider paths that could hit $u$ such that both endpoints have no 1’s in their segment $C$. While flipping the bits in segments $D$ and $B$, the lower endpoint of the path can differ from $u$ only in the shaded positions in row (a) of Fig. 6, implying that $c$ is increased by $2^{2k+o}$.

**Case 2.** Now we assume that the label of $v$ has one 1 in segment $C$ and segment $C$ of $w$’s label consists of 0’s only. While flipping bits in segments $B$ and $D$, $u$ and the label of the lower endpoint differ again only in the shaded positions in row (b) of Fig. 6, and again $c$ is increased by at most $2^{2k+o}$.

![FIG. 6. Possible lower and upper endpoints of a path hitting $u$.](image_url)
Case 3. Next, let segment $C$ of $v$’s label consist of 0’s only and segment $C$ of $w$’s label contain a 1 at position $i$. After flipping the bit at position $i$ in $v$’s label, $u$ and the lower endpoint agree except in the shaded position in row (c) of Fig. 6. Thus $c$ increases by at most one. While flipping the bits in segments $B$ and $D$, the upper endpoint and $u$ can only differ in the shaded segments in row (a) of Fig. 6. Hence, $c$ is increased by at most $2^{2k+o}$.

Case 4. Finally, we consider the case that labels of both endpoints contain a 1 in segment $C$. After flipping the bit in segment $C$, $u$ and the lower endpoint differ in the shaded positions in row (e) of Fig. 6. Hence, $c$ increases by at most 2. While flipping the bits in segments $B$ and $D$, $u$ and the lower endpoint differ in the shaded positions in row (d) of Fig. 6. Since the length of the shaded positions is bounded by $2k + o + 2$, $c$ increases by at most $2^{2k+o+2}$ during the second and third phase. After the third phase, the label of an upper endpoint of a path hitting $u$ can only differ in the shaded positions of row (e) of Fig. 6. Again, $c$ is increased by 2.

Altogether, we have shown that for any hypercube location $u$ at most $c = O(2^{2k+o+2})$ hypercube locations could be an upper or lower endpoint of a path hitting $u$, implying, since the load of the embedding will be 1, that the node-congestion is at most $O(d2^{2k+o+2}) = O(d2^{2k}) = O(d(d^3))$. 

5. IMPLEMENTATION ON THE HYPERCUBE

In this section, it will be shown that the algorithm of the previous section can be efficiently implemented on the optimal hypercube. This section is organized as follows. First, we present some fundamental hypercube algorithms, which we will need in the following. Next, we discuss some reasonable assumptions to obtain an efficient implementation. Finally, we describe the preprocessing step, the main algorithm, and a procedure for partitioning a graph represented by a tree-decomposition.

5.1. Fundamental Hypercube Algorithms

First, we mention that we call in the following vertices in the hypercube processors. In this subsection, we consider a hypercube of size $n$.

Let $H = (M, \circ)$ be a semigroup. Given at each processor $i \in \{0, \ldots, n - 1\}$ of the hypercube an element $m_i \in M$, we can compute at each processor $i$ of the hypercube the value $\bigcirc_{j=0}^{i} m_j$ in time $O(\log(n))$ as shown in [33]. This is called a parallel prefix computation. Sometimes we are interested in the postfix sum instead of the prefix sum. Clearly, this is computationally equivalent.
A variation of the parallel prefix computation is the segmented parallel prefix computation. Additionally, we have a sequence \( 0 = j_0 < \cdots < j_m < n \) of \( m \leq n \) integers, where each \( j_i \) is stored at processor \( j_i \) as a marker. On the hypercube we can compute at each processor \( i \) the values \( \bigcap_{j=\max(j, 2)}^m \{ j \} \), in time \( O(\log(n)) \) as proved in [33]. It can be shown that the segmented parallel prefix problem can be reduced to a parallel prefix problem as shown in [25, 27].

Performing a series of \( \ell \) independent (segmented) parallel prefix computations can be implemented using pipelining in time \( O(\ell + \log(n)) \) as shown in [27]. We refer to this operation as pipelined (segmented) parallel prefix computations. We note that various broadcast operations can be realized using (pipelined) segmented parallel prefix computations.

Concentrating \( m \leq n \) elements stored one element per processor into a contiguous block of \( m \) hypercube locations preserving their order can be implemented in time \( O(\log(n)) \) as shown in [29]. We call this algorithm concentration routing and the inverse operation inverse concentration routing. If the elements should be stored in reverse order we call it reverse concentration routing. This can be implemented in time \( O(\log(n)) \) too.

Another fundamental special case of permutation routing is monotone routing: provided that each processor is the source and the destination of at most one element, the problem is to route the elements preserving their order from their source to their destination processor. This can be implemented by a concentration routing followed by a inverse concentration routing. Hence, monotone routing can be solved in time \( O(\log(n)) \).

Let \( T_{\text{Sort}}(m, d) \) denote the time required to sort \( m \) items on a \( d \)-dimensional hypercube where at most one item is stored per processor. As proved in [12], we obtain for Sharesort \( T_{\text{Sort}}(m, \lceil \log(m) \rceil) = O(\log(m) \log\log^2(m)) \). For sorting a small number of items, it was shown in [30] that for sparse enumeration sort we get

\[
T_{\text{Sort}}(m, d) = O\left(\frac{d \log(m)}{d - \log(m) + 1}\right).
\]

Note that these sorting algorithms are also the fastest known online algorithms for (partial) permutation routing. In the following, we use \( T_{\text{Sort}}(m) \) as an abbreviation for \( T_{\text{Sort}}(m, \lceil \log(m) \rceil) \).

Given a linked list of length \( n \), list ranking is the problem to determine for each item in the list the number of its successors in the list. We denote by \( T_{\text{LR}}(n) \) the time for solving the list ranking problem on a \( \lceil \log(n) \rceil \)-dimensional hypercube for a list of length \( n \) stored one item per processor. The fastest known algorithm, given in [20], requires time \( T_{\text{LR}}(n) = O(\log^2(n) \log \log \log(n) \log^3(n)) \).

We also will use the Euler contour path technique. For each vertex \( v \) of a binary tree, we introduce three vertices \( \text{LEFT}(v) \), \( \text{RIGHT}(v) \), and \( \text{LOWER}(v) \)
We define recursively how to link together the list vertices. Consider the subtree rooted at some vertex \( v \). First, if \( v \) has a left child, link \( \text{LEFT}(v) \) with the beginning of the Euler contour path of its left subtree and the end of this Euler contour path with the list vertex \( \text{LOWER}(v) \), and link \( \text{LEFT}(v) \) with \( \text{Lower}(v) \) otherwise. Next, if \( v \) has a right child, link \( \text{Lower}(v) \) with the beginning of the Euler contour path of its right subtree and link the end of this Euler contour path with \( \text{RIGHT}(v) \), and link \( \text{LOWER}(v) \) with \( \text{Right}(v) \) otherwise. Clearly, using list ranking and sorting, a parallel prefix operation on the Euler contour path can be performed in time \( O(T_{\text{Sort}}(n) + T_{\text{LR}}(n)) \).

5.2. Preliminary Remarks

In the rest of this section, we assume that the graph \( G \) has size \( n \), treewidth \( \leq t \), maximal degree \( d \), and that the size of the given decomposition tree for \( G \) is at most \( n \). Our algorithm can be easily modified to work with decomposition trees of size \( N>n \); this yields a slow-down of \( \frac{N}{n} \). As mentioned above, we will give our implementation on the optimal hypercube, i.e., a hypercube of dimension \( \lceil \log(n) \rceil \).

In the following, we assume that at most one vertex of the decomposition tree is stored per processor. The edges of the decomposition tree are given as a link \( \text{par}(s) \) of each vertex \( s \) to its parent except for the root. The set \( X_s \) and the information about edges in \( G \) between vertices in \( X_s \) are also stored in the processor containing \( s \). Whenever a vertex \( s \) of the decomposition tree is remapped to another processor, the set \( X_s \) and the information whether two vertices in \( X_s \) are adjacent will be remapped too. Hence, we must perform a permutation routing to remap this information which requires time \( O(t^2 \cdot T_{\text{Sort}}(n)) \). All other reasonable representations can be converted to this representation on the hypercube in time \( O(t^2 \cdot T_{\text{Sort}}(n) + T_{\text{LR}}(n)) \).

We refer to elements in \( X_s \), for some vertex \( s \) of the decomposition tree, as mini-vertex. Note that a single graph vertex is usually represented by a large number of mini-vertices. Nevertheless, the number of mini-vertices is bounded by \( (t+1)n \) due to the assumptions above. For each graph vertex \( v \), we distinguish one of the mini-vertices representing \( v \) as major mini-vertex. The mini-vertex representing a graph vertex \( v \) which is contained in the set \( X_s \) and is not contained in \( X_{\text{par}(s)} \) is called major mini-vertex.

In Lemma 4.8, we have constructed a partition of a graph using its tree-decomposition. Differently from the proof of Lemma 4.8, we do not add edges to obtain a new decomposition tree. Instead, we will work with a forest of binary decomposition trees. On the other hand, we will not really unfold the remaining decomposition trees after a partitioning. Instead, the unfolding procedure will be applied to virtual decomposition trees in which
the already mapped graph vertices have not been removed. Obviously, this will not affect the quality of our embedding but it leads to a more efficient implementation of our embedding on the hypercube as we will see.

5.3. The Preprocessing

Our hypercube algorithm consists of two main stages, the preprocessing stage and the main stage. During the preprocessing, we compute some important parameters which will be used during the main stage. The detailed algorithm for the preprocessing stage is given in Fig. 7.

First, we describe the transformation of the decomposition tree into a binary tree. To determine the in-degree of each vertex, we sort the tree vertices by the value \( \text{par}(s) \). Using two segmented parallel prefix operations, we can determine for each vertex the number of its siblings. Therefore, the degree of each vertex can be determined using a permutation routing. Now we can easily extend the tree-decomposition into a binary tree-decomposition as described in Lemma 2.2. For each new tree vertex \( s \), we have to initialize \( X_s \). Thus, step 1 of the preprocessing can be computed in time \( O(t^2 \cdot T_{\text{Sort}}(n)) \).

First, we note that the weight of each vertex of the decomposition tree can be computed in time \( O(t^2 \cdot T_{\text{Sort}}(n)) \). To compute, for each vertex, the weight of its subtree in step 2 of the preprocessing, we mark \( \text{LEFT}(v) \) with \( w(v) \) and \( v \)'s other list vertices with 0. By computing the parallel prefix sum along the Euler contour path, we obtain, for each tree vertex \( v \), the weight of its subtree by subtracting the value computed at \( \text{LEFT}(v) \) from the value computed at \( \text{RIGHT}(v) \), and adding \( w(v) \). To compute the depth of each vertex, we mark \( \text{LEFT}(v) \) with 1, \( \text{LOWER}(v) \) with 0, and \( \text{RIGHT}(v) \) with \(-1\) for all vertices \( v \). The depth of a vertex \( v \) is the value \( \text{LEFT}(v) \) as determined by parallel prefix summation. The maximum depth of a vertex also determines the height of the tree. Finally, if, for all vertices \( v \), we

1. Transform the decomposition tree into a binary decomposition tree
2. For each vertex of the binary decomposition tree, compute the weight of its subtree, its depth, and its inorder number; also compute the height of the tree
3. For each vertex of the binary decomposition tree, gather the following information about its neighborhood in the binary decomposition tree:
   3.1 the inorder numbers for its children and its parent
   3.2 the size of the subtrees of its children and grandchildren
4. Unfolding the binary decomposition tree
5. Compute the new inorder numbering of the unfolded binary decomposition tree
6. Route each vertex \( s \) of the decomposition tree together with the set \( X_s \) to the location in the hypercube given by its inorder number
7. Repeat step 3 for the unfolded binary decomposition tree

FIG. 7. Preprocessing stage of the hypercube algorithm.
mark \( \text{LOWER}(v) \) with 1 and the other list vertices with 0, a parallel prefix sum operation produces the inorder numbering of the tree. Altogether, step 2 can be implemented in time \( O(t \cdot T_{\text{Sort}}(n) + T_{\text{LR}}(n)) \).

Step 3 and step 7 consist of a constant number of routing problems which can all be solved using sorting. To accomplish step 4, we simply have to determine whether a tree vertex becomes a left or right vertex in the unfolded binary decomposition tree. This depends only on how often the path from the root to the vertex branches to a child with a smaller subtree than its sibling. If this number is even the vertex becomes a left vertex in the unfolded binary tree, and a right vertex otherwise. Thus, we mark \( \text{LEFT}(v) \) with 1, \( \text{RIGHT}(v) \) with -1, and \( \text{LOWER}(v) \) with 0 if the size of \( v \)'s subtree is less than that of its sibling, and we mark all these list vertices with 0 otherwise. Again, using a parallel prefix sum along the Euler contour path, we find how often the path from the root to a vertex branches to the smaller subtree. Hence, step 4 can be implemented in time \( O(T_{\text{LR}}(n) + t^2 \cdot T_{\text{Sort}}(n)) \).

As described above, step 5 can also be implemented using parallel prefix operations on the Euler contour path. As mentioned in the previous sub-section, step 6 can be implemented in time \( O(t^2 \cdot T_{\text{Sort}}(n)) \). Altogether, the preprocessing can be done in time \( O(T_{\text{LR}}(n) + t^2 \cdot T_{\text{Sort}}(n)) \).

5.4. The Main Part

The main algorithm is as stated in Fig. 8. Note that step 1.1 will not be executed for the root, since at this moment there are no partition and marked vertices present. Clearly, step 2 can be solved in time \( O(d \cdot T_{\text{Sort}}(n)) \).

Each iteration of the loop in step 1 performs the assignment of vertices to the nodes at the corresponding level of the \((k, h, o)\)-tree. Thus, the \( \ell \)th iteration is performed in \( 2^k(\ell-1) \) distinct \((k(h-\ell+1)+o)\)-dimensional subcubes, where each subcube contains a forest of binary decomposition subcubes, where each subcube contains a forest of binary decomposition

---

1 For each level \( \ell \in \{1, \ldots, h\} \) do:
   1.1 Map the partition and marked vertices to the corresponding \((k, h, o)\)-tree nodes
   1.2 Fill the \((k, h, o)\)-tree nodes up to capacity using additional vertices
   1.3 For each mapped vertex, compute its location in the hypercube
   1.4 Mark the neighbors of mapped vertices
   1.5 Partition each graph into \( 2^k \) parts of the nearly the same size using the procedure Biject-Graph (During the partition the actual cube will be divided into \( 2^k \) subcubes and each newly created forest of binary decomposition trees will be routed to its own subcube)

2 Route the vertices to their computed locations in the hypercube and determine, for each vertex, the locations of its neighbors

---

FIG. 8. Main part of hypercube algorithm.
tree of size at most 
\[ \left\lfloor \frac{n}{2^{k-1}} \right\rfloor. \]

For the following discussion, we consider a fixed iteration of the loop and we denote by \( m \) the size of the actual subcube. Note that \( k(h - \ell + 1) \leq \log(m) \leq \log(n) \).

To map the partition and marked vertices in step 1.1, it is sufficient to determine the major mini-vertices and its number. Then, we determine the slack of the capacity of the considered \((k, h, o)\)-tree node and select the remaining graph vertices using a pipelined parallel prefix operation. Hence, step 1.2 can be realized in time \( O(t + \log(m)) \). Using a pipelined parallel prefix operation, the mapped graph vertices can be numbered with labels in \( L_\alpha \), for the corresponding \( \alpha \). Thus, step 3 can be implemented in time \( O(t + \log(m)) \).

To mark the neighbors of mapped vertices in step 1.4, we simply broadcast the mapped graph vertices. By construction, the number of mapped vertices is at most \( c(\ell - 1) \leq O(k2^{k+o}(h - \ell + 2)) = O(2^{2k} \log(m)) \). Using a pipelined segmented parallel prefix operation, this broadcasting can be done in time \( O(2^{2k} \log(m)) = O((dt)^2 \log(m)) \). Since in each tree vertex are stored at most \( t \) mini-vertices, we can determine the subset of mapped mini-vertex stored in that tree vertex in time \( O((dt)^2 \log(t) \log(m)) \). To decide whether a mini-vertex should be marked, i.e., whether it is a neighbor of a mapped mini-vertex, can be done in time \( O(t^2) \). Altogether, step 1.4 requires time \( O((dt)^2 \log(t) \log(m)) \). Note that for a marked graph vertex all its representing mini-vertices get marked.

In the next subsection, we will show that the partitioning of the remaining graph can be implemented in time \( O(k(dt)^2 \log(m)) \) and, therefore, step 5 can be implemented within the same time bound. Since there are \( h = O((\log(n))/k) \) stages, altogether the time complexity of the main part of the algorithm is bounded by \( O((dt)^2 \log^2(n)) \).

5.5. The Partitioning

In this subsection, we describe in detail how to partition a graph represented by a binary decomposition tree in step 1.5 of the main algorithm. We will only show how to bisect a graph represented by a forest of binary decomposition trees in time \( O((dt)^2 \log(m)) \), where \( m \) is the actual size of the used subcube. Applying this bisection to each newly created part in parallel, we obtain after \( k \) iterations a partition into \( 2^k \) parts in time \( O(k(dt)^2 \log(m)) \). The detailed procedure for bisecting a graph is listed in Fig. 9.

The bisection of the inorder string is based on the method given in the proof of Lemma 4.6. We first compute for each prefix of the inorder string
Embedding graphs with bounded treewidth

1. Decompose the forest of binary decomposition trees into two forests, such that each newly created forest contains one half of the marked vertices as well as one half of all vertices of the graph to be embedded.

2. Update the information about the size and weight of subtrees of the children and grandchildren:
   2.1 Compute and broadcast the size and weight of the removed subtree(s) in each tree.
   2.2 Update the information about the size and weight of subtrees of its children and grandchildren.

3. Unfold the newly created forests:
   3.1 For each vertex, compute the number of vertices on the path from the root to the vertex, for which the weight of the left subtree became smaller than the weight of the right subtree (this number is called the index of the vertex).
   3.2 Unfold each tree in the forest by concentrating the vertices with odd index in reverse order and concentrate the even indexed vertices at the end.

4. Update the inorder numbering:
   4.1 For each vertex, compute its new inorder number.
   4.2 For each vertex, update the information about its parent, children and grandchildren.


the number of marked major mini-vertices using a parallel prefix operation in time $O(t + \log(m))$. Using monotone routing, we can compute in parallel the number of marked mini-vertices in each window of appropriate length.

Now we determine which vertices of the given graph are partition vertices. A vertex $v$ of the given graph is a partition vertex iff $v \in X_r \cap X_s \cup N(X_r \cap X_s)$ for some partition edge $(r, s)$. Note, that there are at most $O(\log(m))$ partition edges. Using the information about the neighborhood of a decomposition tree vertex, it can be easily determined in parallel whether a decomposition tree vertex is incident to a partition edge. Hence, the vertices in $X_r \cap X_s$ for each partition edge $(r, s)$ can be computed in time $O(t \log(m))$ by sending $X_r$ to the tree vertex $s$ and $X_s$ to the tree vertex $r$ using $t$ times sparse enumeration sort. Now, we broadcast which vertices belong to $X_r \cap X_s$ for each partition edge $(r, s)$. Because there are at most $O(t \log(m))$ such vertices, this can be done using pipelined parallel prefix in time $O(t \log(m))$. Next, the partition vertices can be determined in time $O(t^2)$. Note that for each partition vertex all its representing mini-vertices are also recognized as partition mini-vertices. As described in subsection 5.4, the neighbors of the partition vertices can be marked in time $O((dt)^2 \log(m))$. Altogether, step 1 requires time at most $O((dt)^2 \log(m))$.

For simplicity, we now consider only one binary decomposition tree of the forest after the bisection, and we note that all computations are executed in parallel for all such trees. The operations required are (pipelined) parallel prefix computation, broadcasting, concentration routing, and reverse concentrating routing. Since the trees are stored in inorder, we can perform these operations using (pipelined) segmented parallel prefix computations.
and monotone routings. In case of reverse concentration routing we have
to be more careful, since the reversal takes place only within each tree.
We determine, for all trees in parallel, a largest subcube within the interval
belonging to the tree, and we use this subcube to perform the reverse con-
centrating routing. Since the size of this subcube is at least a quarter of the
length of the interval, a constant number of phases will suffice.

We call a vertex \textit{pruned} iff it lost at least one child by the bisection. Since
a subtree is removed from a tree only at the right end or left end of its
inorder string, there are at most two pruned vertices in each newly created
tree. If we have removed subtrees at both ends of the inorder string, we
split the tree into the left and right subtree of the root. After unfolding
these trees independently, we then recombine them. Thus, we may assume
that each tree contains at most one pruned vertex, and we call the path
from the root to a pruned vertex the \textit{trunk} of the tree. In the following, we
also assume without loss of generality that the pruned vertex is the left-most
vertex of its tree.

The next step is to recompute, for each vertex, the size and weight of
its subtree. Each vertex knows the size and weight of its left and right sub-
tree before partitioning and the (old) inorder numbers of the left-most and
right-most vertex after partitioning. Hence, each vertex can easily determine
whether it lies on the trunk and whether it is the pruned vertex. The pruned
vertex broadcasts the size and weight of its subtree that was removed. Sub-
sequently, each vertex on the trunk subtracts these values from the size and
weight, respectively, of its own subtree, that of its left child, and that of its
left-most grandchild. It follows that step 2 require time $O(n \log n)$.

After the partitioning some originally unfolded vertices become folded.
Note that all folded vertices belong to the trunk. The next step is to unfold
the tree again as described in Section 4.4. Let the \textit{index} of a vertex $v$ be
the number of folded vertices on the path from the root to $v$. It is easy to
see that a vertex belonging to the trunk and all vertices in its right subtree
have the same index.

First, we consider an arbitrary folded vertex $v$ in a tree $T$. If we swap
the children of each vertex in $v$'s subtree, the vertex $v$ become unfolded
and no new folded vertex is introduced in $T$. This operation corresponds
to reversing the inorder substring of $v$'s subtree within the inorder string
of $T$. Hence, if we reverse all inorder substrings of the subtrees rooted at
a folded vertex, the corresponding tree is unfolded.

Let $v$ be a vertex belonging to the trunk of a tree $T$, and let $s$ be the
inorder string of $v$'s right subtree. After the above operation the inorder
substring $uv$ is unchanged if $v$'s index is even, and it is reversed if $v$'s index
is odd. As the pruned vertex is the left-most vertex in $T$, the vertices with
an odd index move to the front of the inorder string, since the subtrees
containing these vertices move to the left of the path from the root to the
FIG. 10. Unfolding an unfolded tree after a removal of a left-most subtree.

pruned vertex. Altogether, we obtain the inorder string of the unfolded tree if we concentrate all vertices with an odd index in reverse order at the beginning, and concentrate the vertices with an even index at the end of the inorder string. This permutation can be realized in time \( O(t^2 \log m) \) using \( t^2 \) times algorithms for concentration and reverse concentration routing. Again, it is clear that time \( O(t^2 \log m) \) suffices for step 3.

This part of the algorithm is illustrated in Fig. 10, where a larger right subtree of a vertex is marked by shading. Thus, the vertices numbered 1, 5, 8, and 9 are folded. To unfold this tree, these vertices and their descendants in the right subtrees must swap their subtrees.

To compute the new inorder number for each vertex, we only need a parallel prefix computation, because the tree vertices are now stored in inorder. Also, since each vertex knows the size of the subtrees of its children and grandchildren, it can easily compute the inorder number of its parent and its children. Thus, again the operations of step 4 require at most time \( O(\log m) \).

Altogether, we have proved the following theorem:

**Theorem 5.1.** Let \( G = (V, E) \) be a graph of size \( n \) with treewidth \( t \) and maximal degree \( d \). Given a tree-decomposition for \( G \) of width \( t \) and size \( O(n) \), there exists a one-to-one embedding of \( G \) into its optimal hypercube with dilation \( 3[\log((d + 1)(t + 1))] + 8 \) and node-congestion \( O(d(t^2)) \). This
embedding can be computed in time

\[ O\left( T_{LR}(n) + (dt)^2 \log^2(n) \right) \]

\[ = \begin{cases} 
O((\log^2(n) \log \log \log(n) \log^*(n)) & \text{if } (dt)^2 = O(\log \log \log(n) \log^*(n)) \\
O((dt)^2 \log^3(n)) & \text{if } (dt)^2 = \Omega(\log \log \log(n) \log^*(n))
\end{cases} \]

on the optimal hypercube.

Provided that the given graph has constant treewidth, it is possible to compute a tree-decomposition efficiently. Recall that in general it is \( \mathcal{NP} \)-hard to decide the treewidth of a given graph. As shown in [6], a minimal tree-decomposition for a graph with constant treewidth can be computed on an EREW PRAM in time \( O(\log^2(n)) \). Using sorting, this algorithm can be immediately transformed into a hypercube algorithm with running time \( O(\log^3(n) \log \log^2(n)) \). Thus, we have also proved the following theorem.

**Theorem 5.2.** Let \( G = (V, E) \) be a graph of size \( n \) with constant treewidth and constant degree. There exists a one-to-one embedding of \( G \) into its optimal hypercube with constant dilation and constant node-congestion. This embedding can be computed in time \( O(\log^3(n) \log \log^2(n)) \) on the optimal hypercube.

### 6. CONCLUSION

We have presented an efficient algorithm for an embedding of a graph with treewidth \( t \) and maximal degree \( d \) into its optimal hypercube with unit load, dilation of at most \( 3\lceil \log((d + 1)(t + 1)) \rceil + 8 \) and node-congestion of at most \( O(d(dt)^3) \). This is the first time that an embedding of a class of highly irregular graphs into their optimal hypercube is presented. For graphs with constant treewidth and constant degree, our construction yields an embedding with constant dilation and constant node-congestion into the optimal hypercube. Moreover, this embedding can be very efficiently computed on the optimal hypercube itself.

It remains open to study dynamic embeddings of graphs with bounded treewidth into hypercubes. Observe that any dynamic embedding of a binary tree into its optimal hypercube with unit load and small dilation must use randomization or migration as a result in [26] shows. In [22], we presented a dynamic deterministic embedding of binary trees with unit load, nearly optimal expansion, and constant dilation using migration. This embedding allows also a very efficient implementation on the hypercube itself. It turns out, that this algorithm can be extended to dynamic embeddings of graphs
with bounded treewidth provided that the growth of the graph is represented by its growing decomposition tree.

REFERENCES