Sequential and Parallel Algorithms for Embedding Problems on Classes of Partial k-Trees

Arvind Gupta* Naomi Nishimura†

Abstract

We present sequential and parallel algorithms for various embedding problems on bounded degree partial k-trees and k-connected partial k-trees; these include subgraph isomorphism and topological embedding, known to be NP-complete for general partial k-trees. As well as contributing to our understanding of the types of graphs for which these problems are tractable, this paper introduces new methods for solving problems on graphs. In particular, we make use of a tree-like representation of the graph (the tree-decomposition of the graph) to apply techniques used to solve problems on trees to solve problems on more general classes of graphs.

1 Introduction

In devising sequential and parallel algorithms for subgraph isomorphism and topological embedding of bounded degree partial k-trees and k-connected partial k-trees, we make advances in two streams of research. One stream of research is the identification of problems for which a bound on the tree-width of a graph allows a more efficient solution than in the general case; a partial k-tree is also known as a graph of bounded tree-width. The other stream is the identification of classes of graphs for which these embedding problems are tractable: the subgraph isomorphism problem, and consequently the more general problem of topological embedding, is known to be NP-complete for general graphs [GJ79], and remains so even for graphs of bounded tree-width [Sys82]. The algorithms presented in this paper make use of techniques developed in each of these streams, and in addition introduce general methods for enabling the techniques in one stream to be applied to problems in the other.

As we will see in Section 2, graphs of bounded tree-width can be characterized as the class of *partial k-trees*, and include many natural classes of graphs, such as trees, outerplanar graphs, series-parallel graphs, and Halin graphs. Of great interest for our algorithms is the fact that any graph in this class can be represented as a special type of tree, namely a *tree-decomposition*, of bounded width. Such graphs have the property that there are small separators which break the

*School of Computing Science, Simon Fraser University, Burnaby, Canada, V5A 1S6. email: arvind@cs.sfu.ca, FAX (604) 291-3045. Research supported by the Natural Sciences and Engineering Research Council of Canada, the Center for System Sciences and the Advanced Systems Institute.

†Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1. email: nishi@plg.uwaterloo.ca, FAX (519) 885-1208 Research supported by the Natural Sciences and Engineering Research Council of Canada.

graph into a tree-like structure. This property has led to efficient algorithms for a number of problems which are difficult to solve on general graphs. For example, Bodlaender [Bod90] uses this property to give a polynomial time algorithm to determine if two input graphs are isomorphic. He combines the use of separators with a dynamic programming approach to build information about the isomorphism. The basic idea of our algorithms is to attempt to process a graph with respect to its tree-decomposition. We use some techniques developed for handling trees and introduce some new techniques applicable to more general types of graphs.

Previous sequential algorithms for embedding problems on various classes of graphs include algorithms for subgraph isomorphism on trees [Mat78], two-connected outerplanar graphs [Lin89], two-connected series-parallel graphs [LS88], bounded degree partial k-trees [MT92], and k-connected partial k-trees [MT92], and algorithms for topological embedding on bounded degree partial k-trees [MT92] and k-connected partial k-trees [MT92]. Earlier work in a parallel setting includes algorithms for subgraph isomorphism [LK89, GKMS90, GN92] and topological embedding on trees [GN92] and a number of problems on bounded degree partial k-trees including the subgraph isomorphism problem [Bod88b], but not for topological embedding or for subgraph isomorphism on k-connected partial k-trees.

The contributions of our work to the understanding of embedding problems are two-fold. We present the first parallel algorithms for subgraph isomorphism of k-connected partial k-trees and for topological embedding of bounded degree partial k-trees and k-connected partial k-trees. Although there are different algorithms known for the other problems considered in this paper, one of our contributions is to provide a general framework that allows us to extract the similarities between trees and partial k-trees, subgraph isomorphism and topological embedding, and sequential and parallel settings. Our algorithms are conceptually simple, exploiting the relatively simple structure of a subtree isomorphism algorithm [Mat78], and thereby making clear the connection between trees and partial k-trees. At the same time, our methods are sufficiently flexible to allow us to introduce modifications that address the differences between the problems under consideration; we retain the same structure for sequential topological embedding algorithms, parallel subgraph isomorphism algorithms, and ultimately parallel topological embedding algorithms.

The outline of the remainder of the paper is as follows. In the next section we give definitions. Section 3 contains a description of some the difficulties encountered in attempting a straightforward adaptation of the subtree isomorphism algorithm to form algorithms for our problems, as well as techniques which can be applied to overcome these difficulties. The section ends with a general outline of all eight algorithms. In Sections 4 and 5 we present sequential algorithms for subgraph isomorphism and topological embedding, and in Section 6 we present their parallel counterparts. Finally, in Section 7 open problems are presented.

2 Preliminaries

For the problems discussed in this paper, each input graph will be a partial k-tree, properties of which will be discussed in Section 2.3. Of particular importance will be our ability to structure such a graph as a special type of tree called a *tree-decomposition*, discussed in greater detail in Section 2.2.

As our algorithms entail identifying a tree-decomposition, we introduce a graph that includes all tree-decompositions, called a *tree-decomposition graph*; the tree-decomposition graph will be discussed in greater detail in Section 3.2. In order to avoid confusion, we will refer to *nodes* in the original graphs, *vertices* in a tree-decomposition, and *tdg-vertices* in the tree-decomposition graph. Terminology relevant to many of these structures is discussed in Section 2.1.

2.1 Graphs

We assume a basic familiarity with graphs and trees, the reader is referred to a standard reference [BM76] for the appropriate background. We denote the vertex and edge sets of a graph G by $V(G)$ and $E(G)$. Unless otherwise specified, all our graphs will be connected.

In the course of our algorithms, we will divide a graph G into pieces using a subset A of $V(G)$. We will denote the set of connected components of $G \backslash A$ by $\mathcal{C}_G(A)$ where the subscript G will be dropped when it is clear from context. The *neighbourhood of A in B*, $nbhr_A(B)$, is the set of vertices of $B \subseteq V(G)$ which are adjacent to vertices of A. For G_1 and G_2 subgraphs of G, A *separates* G_1 and G_2 if the only nodes that G_1 and G_2 have in common are in A and there is no edge in G from a node in G_1 to one in G_2. In this case, we will refer to A as a *separator*. It is not difficult to see that A is a separator of G_1 and G_2 if and only if $nbhr_{G_1}(G_2), nbhr_{G_2}(G_1) \subseteq A$.

All trees in this paper will be rooted (and therefore directed). The size of a tree T, $|V(T)|$, is denoted by $|T|$. We will say that a vertex x is a *descendant* of a vertex y if there is a (directed) path from x to y of nonnegative length. We will distinguish between an arbitrary connected *subgraph* of T, and a *subtree* of T consisting of a vertex and all of its descendants. We use T_v to denote the subtree of T rooted at v. In addition, we will be concerned with pieces of the tree that arise from removing a subtree from another subtree. For $v \in V(T)$ and $w \in V(T_v)$, the subgraph $T_v \backslash T_w$ denotes the subgraph obtained by removing from T_v all proper descendants of w. In particular, this means that the node $w \in T_v \backslash T_w$. We will call $T_v \backslash T_w$ a *scarred subtree* of T. In this context, we will call v a *scar*, and we will say that T_v *is scarred at* w or, more succinctly, that v is scarred at w.

In addition, we will refer to (*induced*) subgraphs of a graph G. By a subgraph of G we will mean a graph G' such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$. For S a subset of nodes of G, the *node-induced subgraph* (or just *induced subgraph*) of G with respect to S, or $G_{[S]}$, is the subgraph of G consisting of all nodes in S and all edges in G between two vertices in S.

More generally, for a directed acyclic graph G and a node a of G, we will say that node b is a *child* of node a if there is an edge from b to a in the graph.

2.2 Tree-decompositions

The representation of graphs by the tree-like structures we present here was first introduced by Robertson and Seymour [RS86].

Definition: Let G be a graph. A *tree-decomposition* for G is a pair (T^G, χ^G) where T^G is a tree and $\chi^G : V(T^G) \rightarrow \{\text{subsets of } V(G)\}$ satisfying:

1. for every $e = (u, v) \in E(G)$, there is an $x \in V(T^G)$ such that $u, v \in \chi^G(x)$; and
2. for $x, y, z \in V(T^G)$, if y is on the path from x to z in T^G then $\chi^G(x) \cap \chi^G(z) \subseteq \chi^G(y)$.

The *width* of a tree-decomposition (T^G, χ^G), $tw(T^G, \chi^G)$, is $\max\{|\chi^G(x)| - 1 : x \in V(T^G)\}$ and the tree-width of a graph G, $tw(G)$, is the minimum width over all its tree-decompositions.

When clear from context, we may drop the superscript and write (T, χ). In addition, we may refer to $\chi^H(x)$ as the *label* of x.

Examples: The tree-width of a clique K_n is $n - 1$, the tree-width of a tree is 1 and the tree-width of a cycle C_n is 2. A graph which is k-connected will have tree-width at least k.

Notice that in the above definition labels of vertices of T^G only include nodes of G. We will, however, refer to edges in $\chi^G(x)$ for $x \in V(T^G)$ by which we mean the edges of the graph induced by $\chi^G(x)$. As well, we will assume that for every x, $\chi^G(x)$ is an (ordered) sequence instead of a set (although we will use set notation where convenient). For $x, y \in V(T^G)$, y a child of x, suppose $a, b \in \chi^G(x) \cap \chi^G(y)$. All our tree-decompositions will have the property that a and b occur in the same relative order in both $\chi^G(x)$ and $\chi^G(y)$.

We can extend the notion of a node-induced subgraph to a set of nodes appearing in $\chi^G(x)$ for nodes x in $V(T^G)$. Namely, for a subgraph S of T^G, we set $G_{\{S\}}$ to be the subgraph of G induced by the nodes in $\bigcup\{\chi^G(x) : x \in V(S)\}$.

Let (T^G, χ^G) be a tree-decomposition of a graph G, and let S be a connected subgraph of T^G. Then, for χ^G_S the restriction of χ^G to S, (S, χ^G_S) is a tree-decomposition of $G_{\{S\}}$.

Intuitively, a tree-decomposition gives insight into separators of the graph. In particular, for (T^G, χ^G) a tree-decomposition of G and $x \in V(T^G)$, $\chi^G(x)$ separates the subgraphs of G induced by $(T^G_{c_1}, \chi^G), \ldots, (T^G_{c_\ell}, \chi^G), (T^G \backslash T^G_x, \chi^G)$ where $c_1, \ldots, c_\ell$ are the children of x.

The following lemma is a consequence of the definition of a tree-decomposition and the discussion above. The separator is crucial to the development of our algorithms.

Lemma 2.1. *Let (T^G, χ^G) be a tree-decomposition of a graph G. Let y be a non-root vertex of T^G with parent x. Then $\chi^G(x) \cap \chi^G(y)$ separates $G_{\{T_y\}}$ from $G_{\{T \backslash T_x\}}$.*

Clearly the tree-decomposition of a graph G is not unique (see Figure 1) and in general, if G has tree-width k it has tree-decompositions of width between k and $|V(G)|$. For fixed k, there has been considerable effort devoted to finding algorithms which determine if a graph has tree-width at most k, and if so, find a tree-decomposition of bounded width.

Robertson and Seymour's work [RS86] gives a non-constructive proof of the existence of an $O(n^2)$ algorithm for this problem. Arnborg et al. [ACP89] exhibit an $O(n^{k+2})$ algorithm. Subsequent work on this problem was performed by Lagergren [Lag90] and Reed [Ree92] who achieved quasi-linear time algorithms. Recently, Bodlaender [Bod93] has settled the question for the sequential case by demonstrating a linear time algorithm. Discussion of parallel algorithms for the problem can be found in Section 6.

2.3 Graphs of bounded tree-width

In this paper, we will use the terms *partial k-tree* and *graph of bounded tree-width* interchangeably [Ros74]. A partial k-tree is a subgraph of a k-tree, which in turn is defined below.

Definition: Let $k > 0$. A *k-tree* is a graph from which it is possible to obtain a k-clique (K_k) by a sequence of eliminations of degree k vertices whose neighbours form a clique.

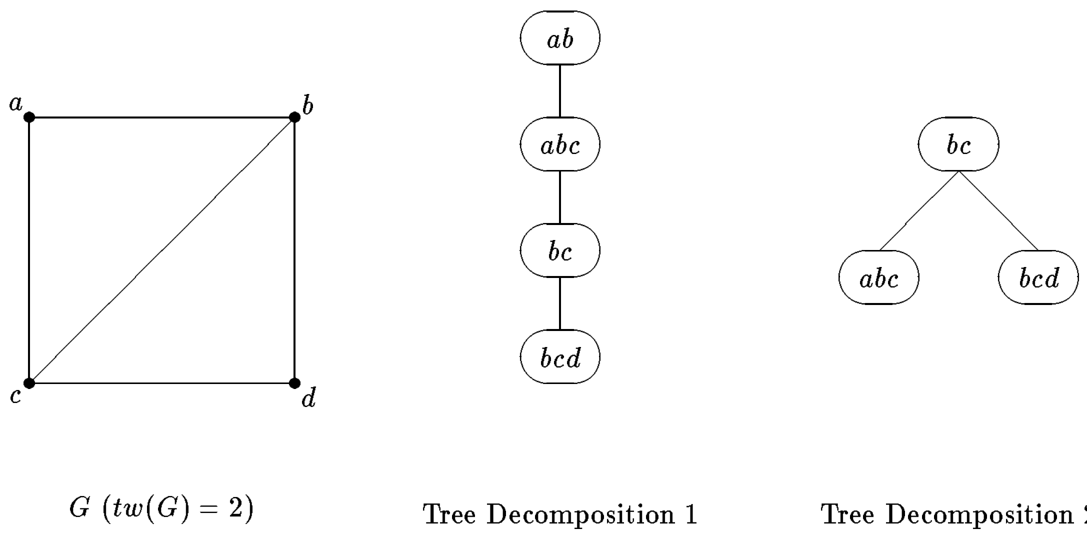


Figure 1: Two different tree decompositions of graph G

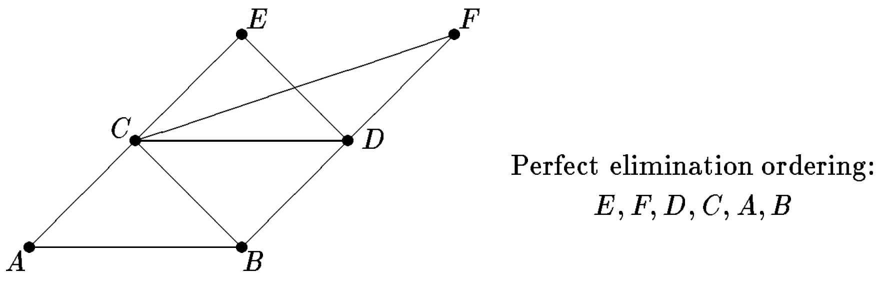


Figure 2: A 2-tree and perfect elimination ordering

For a k-tree G, a sequence of vertex eliminations satisfying the property in the above definition is called a *perfect elimination ordering* for G. Figure 2 gives an example of a 2-tree with a perfect elimination ordering. An alternate characterization of a k-tree is given by the following recursive construction.

Lemma 2.2. *Let $k > 0$. Then, the set of k-trees is the smallest set of graphs satisfying:*

1. *A k-clique is a k-tree.*
2. *Let G be a k-tree on n nodes and K be a k-clique in G. Then the $n + 1$ node graph G' formed by taking G and introducing a new vertex v adjacent to all of K is a k-tree.*

Every graph G is a partial k-tree for some k, since G is is trivially a partial $|V(G)|$-tree. The problem of determining the minimum k for which a graph is a partial k-tree is NP-complete [ACP89].

This problem is in fact equivalent to finding the minimum k for which G has tree-width k, as shown in the lemma below.

Lemma 2.3. *Let G be a graph. Then G is a partial k-tree if and only if $tw(G) = k$.*

There are a large number of natural classes of graphs which are partial k-trees for some k. Trees are 1-trees and both outerplanar graphs and the more general series-parallel graphs are partial 2-trees. Halin graphs are partial 3-trees. Chordal graphs with maximum clique size k are partial k-trees. Finally, any family of graphs closed under minors which does not contain a planar graph is a k-tree for k a function of the size of the excluded planar graph [RS84].

2.4 Topological embedding

The problem of topological embedding is a generalization of the problem of subgraph isomorphism. Intuitively, a subgraph isomorphism algorithm determines a mapping from nodes in G to nodes in H and from edges in G to edges in H. A topological embedding algorithm instead finds a mapping from nodes in G to nodes in H and from edges in G to paths in H, such that the images of the edges in G form node-disjoint paths in H. A more formal definition follows:

Definition: Let G and H be graphs. Then G is *topologically embeddable* in H, if there is a pair of functions (f_1, f_2) such that:

1. $f_1 : V(G) \to V(H)$ is one-to-one;
2. $f_2 : E(G) \to \{\text{paths in } H\}$;
3. if $e = (x, y) \in E(G)$ then $f_2(e)$ has end points $f_1(x)$ and $f_1(y)$; and
4. for $e, e' \in E(G)$, $e \neq e'$, $f_2(e)$ and $f_2(e')$ have no vertices in common except possibly for their endpoints.

Alternatively, if there is a subgraph of H that is isomorphic to G, we can talk about transforming H into G by removing nodes and edges not found in that subgraph. Each node of H is either preserved by the transformation, if it is in the image of $V(G)$ under the mapping, or it is deleted. If there is a topological embedding of G in H, we can transform H into G by removing nodes and edges, and then by shrinking paths into edges. The process of shrinking a path consists of contracting edges in the path. We can view the edges in the path as being contracted from one path endpoint or the other into the middle of the path, where each time an edge is contracted, the intermediate node on the path is *collapsed* into a path endpoint. In the transformation of H into G, there are nodes which are preserved (the image of the mapping of the $V(G)$), nodes which are collapsed (interior nodes in the paths), and nodes which are deleted.

To make explicit the mechanism of collapsed nodes in a topological embedding, we give an alternate definition of topological embedding, equivalent to that given above. In the earlier definition, the embedding is characterized by two functions, one from vertices in G to vertices in H, and the other from edges in G to paths in H. The notion of collapsed nodes is implicit in the function from edges in G to paths in H. In the definition below, the embedding is also characterized by two functions, this time one from vertices in G to vertices in H, and the other from intermediate nodes of paths in H to endpoints of paths in H. In this alternate definition, the collapsing of nodes is made explicit and the existence of node-disjoint paths is made implicit. In the remainder of the paper, we will use this alternate definition.

Lemma 2.4. *Let G and H be graphs. Then G is topologically embeddable in H, if and only if there is a pair of functions (f, g) such that:*

1. *$f : V(G) \to V(H)$ is one-to-one (we use FI to denote the nodes in the image of f);*
2. *$g : SG \to FI$, where $SG \subseteq V(H) \backslash FI$ (we use GI to denote the nodes in the image of g);*
3. *each node in SG has degree at most two in $G_{[SG \cup FI]}$;*
4. *the nodes in SG can be partitioned such that there is one (possibly empty) partition P_e for each edge $e \in E(G)$ and one partition for "unused" nodes; and*
5. *for edge $e = (x, y) \in E(G)$, there is a path $f(x), v_1, \ldots, v_k, f(y)$ in H such that $v_1, \ldots, v_k \in P_e$ and for some $i \leq k$, $g(v_1) = \cdots = g(v_i) = f(x)$ and $g(v_{i+1}) = \cdots = g(v_k) = f(y)$.*

Where appropriate, we will say that G is *topologically embeddable in H with respect to* (f, g).

3 Adapting tree techniques

Our algorithms make use of methods developed to solve problems on trees by making use of the tree-like structure of the tree-decomposition of a graph. To provide a framework in which to discuss techniques needed to create working algorithms, we briefly describe one method of solving subtree isomorphism sequentially and then delineate the problems encountered in attempting to adapt this algorithm to more general classes of graphs. We demonstrate our techniques with respect to the subgraph isomorphism problem on k-connected partial k-trees and then show how they can be applied to other embedding problems. In Section 6 we discuss further developments used in forming parallel algorithms.

Matula [Mat78] was the first to show a polynomial time procedure for the subtree isomorphism problem. He used a dynamic programming approach to work level by level up the tree, at each step combining information using bipartite matching. This approach has successfully been combined with other techniques to derive parallel algorithms for this problem [LK89, GKMS90] as well as parallel algorithms for a variety of other embedding problems on trees [GN92]. Bodlaender applied this methodology to a large set of problems, including the subgraph isomorphism problem for bounded degree graphs of bounded treewidth, both sequentially [Bod88a] and in parallel [Bod88b].

To examine the dynamic programming approach in greater detail, suppose we are given two trees T and T' and we wish to determine whether T is isomorphic to a subtree of T'. We proceed by working from leaves to root in T, finding in turn for each node v all possible mappings of T_v into T'. For a node v with children c_1 through c_k, the mappings of T_v will be determined using the previously computed mappings of T_{c_1} through T_{c_k}. Because each vertex is a separator of the tree, by identifying a node v we are immediately identifying a set of previously processed connected components, namely T_{c_1} through T_{c_k}, and a connected component yet to be processed, namely $T \backslash T_v$.

In the subtree algorithm, the structure of T can be used to determine the order in which nodes are processed. In particular, the rooting of a tree provides an orientation, making it meaningful to talk about processing children before a parent. This orientation is particularly important as information gained in processing children is used in processing a parent.

Viewed as a problem on graphs of tree width one, we can view the processing of nodes as taking place simultaneously in the tree-decompositions of the source tree T and the target tree T'. Since the trees themselves can act as tree-decompositions, we have the tree-decompositions of the source and target graphs as inputs.

When our inputs are graphs of tree-width greater than one, however, we no longer have as input canonical tree-decompositions of the two graphs. Although we can easily obtain tree-decompositions of the graphs (as discussed in Section 2), since tree decompositions are not necessarily unique, isomorphic graphs may not be characterized by isomorphic tree-decompositions. We can fix a tree decomposition for one of the two graphs and root it to give an orientation to that graph, but it is not clear how this orientation can be used in the other graph. It is in fact this problem that seems to induce "NP-completeness" in the general case for subgraph isomorphism of partial k-trees. We will see in Section 4, that we can use the fact that either the source graph is of bounded degree or k-connected to solve this problem. In the remainder of this section, we introduce tools created to address these problems, namely normalized tree-decompositions and tree-decomposition graphs.

3.1 Normalized tree-decompositions

In this section, we define a special type of tree-decomposition needed for the subgraph isomorphism algorithm, and then show how to form a tree-decomposition of this type.

The goal of the algorithm will be to construct such a decomposition for H and then attempt to construct one for G with the same underlying tree structure. In the process we will determine if G is a subgraph of H.

Definition: Let H be a graph of size greater than k and let (T^H, χ^H) be a tree-decomposition of H of width k. Then (T^H, χ^H) is a *normalized* tree-decomposition if:

1. The vertices of T^H can be labelled as *separator vertices* and *clique vertices* such that the root of T^H is a separator vertex, the leaves of T^H are clique vertices, the child of each separator vertex is a clique vertex and the child of each clique vertex is a separator vertex.

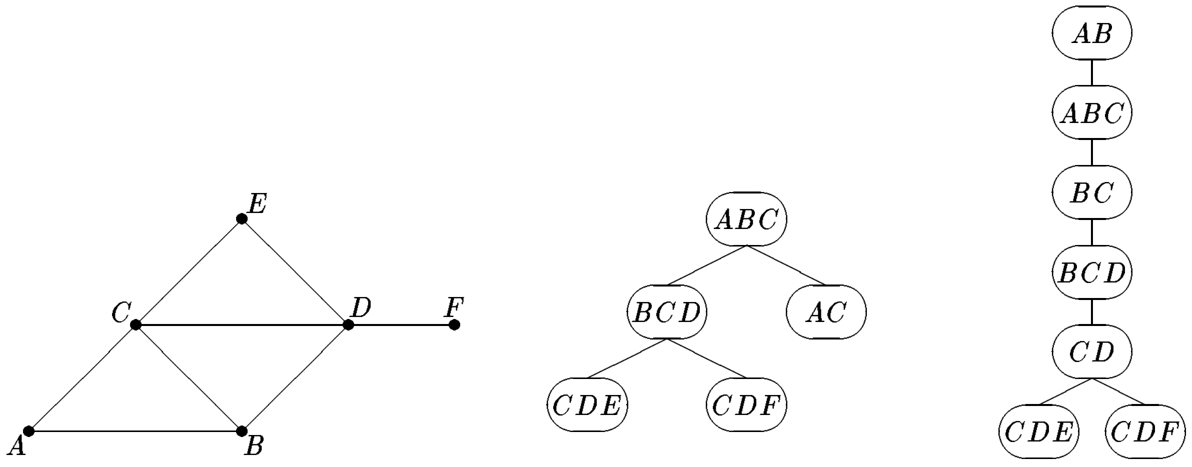
2. For any separator vertex $x \in V(T^H)$, $|\chi^H(x)| \leq k$.

3. For any clique vertex $y \in V(T^H)$, $|\chi^H(y)| \leq k+1$.

4. For any siblings x and y, $|\chi^H(x)| \neq |\chi^H(y)|$.

5. For any separator vertex x with child y, $\chi^H(y) = \chi^H(x) \bigcup \{a\}$ for some node $a \notin \chi^H(x)$, and the ordering on the nodes in $\chi^H(y)$ is the same as in $\chi^H(x)$, with a added at the end.

6. For any clique vertex y with child x, $\chi^H(x) \subseteq \chi^H(y)$, with the nodes in $\chi^H(x)$ in the same relative order as in $\chi^H(y)$.



Graph H A Tree Decomposition A Normalized Tree Decomposition

Figure 3: A tree-decomposition of H, before and after normalization

7. For any clique vertex y with child x, with $a \in \chi^H(y)$ but not in the the parent of y, a must be in $\chi^H(x)$.

A normalized tree-decomposition roughly corresponds to a perfect vertex elimination ordering for the underlying graph completed to a k-tree, where the node a in conditions 5 and 7 corresponds to the node being eliminated at a particular point in the sequence.

It is not difficult to transform any tree-decomposition into a normalized one. Figure 3 illustrates a graph H, a tree-decomposition of H, and the normalized tree-decomposition of H obtained by applying the construction given in Lemma 3.1.

Lemma 3.1. *Let H be a graph and (T^H, χ^H) be a tree-decomposition of H of width k. Then there is an $O(|V(T^H)|^2)$ time algorithm which takes (T^H, χ^H) and returns a normalized tree-decomposition (S^H, λ^H) of H whose width is also k.*

Proof:

We first set S^H to equal T^H and λ^H to equal χ^H, and then modify them to fit the conditions.
Step 1: We contract any edge between vertices x and y such that $\lambda^H(x) \subseteq \lambda^H(y)$.
Step 2: We set all existing vertices to be clique vertices; since (T^H, χ^H) was a tree-decomposition of H of width k, it is clear that condition 2 has been satisfied.
Step 3: Between each pair of clique vertices x and y, where x is the parent and y the child, we insert a constant number of new vertices in a path as follows. Let $S = \lambda^H(x) \cap \lambda^H(y)$ and let $n_1, \ldots n_j$ be the nodes in $\lambda^H(y) - S$. Because the underlying graph H is connected, we can conclude that S is nonempty; due to Step 1, we know that S is strictly smaller than either the label of x or the label of y. We create a child of x, a separator vertex with label S. The next two nodes on the path from x to y are a clique vertex and then a separator vertex, each with label $S \cup \{n_1\}$, the

following two a clique vertex and a separator vertex each with label $S \cup \{n_1, n_2\}$, and so on, until the path ends at y.

It is not difficult to see that conditions 2 and 3 are satisfied. To see that conditions 5 and 6 can be satisfied, it suffices to observe that we can set the orderings on the labels in the obvious way.

Step 4: We now ensure that condition 7 is satisfied. The condition is violated by a situation in which a separator vertex z is the parent of a clique vertex y, in turn the parent of separator vertex x, such that the unique node a in $\lambda^H(y) \backslash \lambda^H(z)$ is not contained in $\lambda^H(x)$. In this case, it is not difficult to see that $\lambda^H(x) \subseteq \lambda^H(z)$. We remove the edge between y and x, instead making x into a child of the parent of z.

Step 5: Condition 4 can be satisfied by examining all siblings to see if they share a label value, and if so, to combine such siblings into a single node with all subtrees of the original siblings.

Step 6: Finally, to satisfy condition 1, we add in a separator vertex as a root, choosing as its label an arbitrary subset of the label of the original root.

It is not difficult to see that each of the above steps can be completed in the stated running time. ∎

Combining the result of Bodlaender [Bod93] showing a linear time algorithm for finding a tree-decomposition of width k for a partial k-tree and the algorithm in Lemma 3.1, we obtain:

Corollary 3.2. *Let H be a partial k-tree with n nodes. Then, there is an $O(n^2)$ algorithm which, on input H returns a normalized tree-decomposition (T^H, χ^H) of H of width k.*

We will require the following fact about normalized tree decompositions of k-connected partial k-trees.

Lemma 3.3. *Let (T^G, χ^G) be a width k normalized tree-decomposition of a k-connected partial k-tree G. Let $v \in V(T^G)$ with children $w_1, \ldots, w_r$. Then each of the graphs $G_{\{T \backslash T_v\}}, G_{\{T_{w_1}\}}, \ldots, G_{\{T_{w_r}\}}$ are connected.*

Proof:

Since we can view a tree-decomposition as being rooted at any vertex of the tree T^G, it suffices to consider only the case in which v is the root of T^G. Similarly, since the proofs for v a separator vertex and v a clique vertex are similar, we present only the former.

To show that each of the graphs $G_{\{T_{w_i}\}}$ is connected is equivalent to showing that the number of components in $\mathcal{C}_G(\chi^G(v))$ is equal to the number of children of v. By the definition of a tree-decomposition, clearly there cannot be more children than components.

Suppose instead that the number of children is smaller than the number of components. Then, there must be some pair of components that are subgraphs of the graph $G_{\{T_{w_\ell}\}}$, for some value of ℓ. Without loss of generality, we assume that the components C and D form $G_{\{T_{w_1}\}}$. Since w_1 is a clique vertex, by Condition 5 in the definition of normalized tree-decompositions there is a node $a \in \chi^G(w_1)$ such that $a \notin \chi^G(v)$. The node a must be in either C or D; without loss of generality, we assume that $a \in C$.

As C and D are both subgraphs of $G_{\{T_{w_1}\}}$, there must be a least one vertex in T_{w_1} which contains at least one node in D. We let y be the vertex of T_{w_1} that contains at least one node in D, say b, and such that no ancestor of y contains any node in D.

Since G is k-connected, it is not difficult to see that for each separator vertex $x \in T^G$, the size of $\chi^G(x)$ is equal to k. Moreover, there must be at least k node-disjoint paths in G from b to the k nodes in $\chi^G(v)$.

We now consider the child x of w_1 which is the ancestor of y. By Condition 7 in the definition of normalized tree-decompositions, we can conclude that $a \in \chi^H(x)$. Since x is a separator vertex, $\chi^G(x)$ contains at most $k-1$ nodes in D. By Menger's Theorem, there can be no more than $k-1$ node-disjoint paths from b to the nodes in $\chi^G(v)$.

Thus, we can not obtain the required k node-disjoint paths in G from b to the k nodes in $\chi^G(v)$, contradicting the assumption that G is k-connected. ■

3.2 Tree-decomposition graphs

As outlined above, our algorithm constructs a normalized tree-decomposition of H and then proceeds up this tree level by level, attempting to construct a similar tree-decomposition of G. Here we will construct a representation graph whose vertices correspond to vertices in such a tree-decomposition of G, if one exists. We give a high-level explanation of the construction, followed by formal definitions. In this section we focus on the subgraph isomorphism problem for k-connected partial k-trees; modifications needed for the other algorithms will be presented in Sections 4.2 and 5. Lemma 3.3 is the key to simplifying the k-connected case.

3.2.1 Definition of $TDG(G)$

We construct a graph called a *tree-decomposition graph* of G; we refer to the vertices of this graph as *tdg-vertices.*

Intuitively, a tdg-vertex corresponds to a potential vertex in a tree-decomposition of G; it contains information about the structure of that vertex as well as its placement in the tree-decomposition. A tree-decomposition graph consists of two types of tdg-vertices, *separator vertices* and *clique vertices*, intuitively corresponding to the k-clique to which a new node is attached and the resulting $(k+1)$-clique, respectively, in a node-elimination ordering of a k-tree.

A separator vertex α of the tree-decomposition graph of G is a triple $(Sep^\alpha, \sigma^\alpha, Part^\alpha)$, where Sep^α is a set of nodes of G called the *separator of* α, σ^α is a permutation on the elements of the separator of α called the *ordering on* Sep^α, and $Part^\alpha$ the *partition induced by* α. Furthermore, in $Part^\alpha = (P_0^\alpha, P_1^\alpha, \ldots, P_r^\alpha)$, we call P_0^α the *parent partition* and for $i > 0$, we call P_i^α a *child partition.* In these partitions, each P_i^α is a single connected component of $G \backslash Sep^\alpha$.

The separator is a set of nodes forming the label of a vertex in a tree-decomposition of G. The ordering on Sep^α indicates the ordering on the nodes, as necessitated by the definition of a label of a tree-decomposition vertex as a sequence rather than a set of nodes. The tdg-vertices represent all possible separators and all possible orderings.

In any tree-decomposition of G with a vertex x labelled by the nodes in Sep^α (with any ordering), the subgraph of non-descendants of x and the subtrees rooted at the children of x induce subgraphs of G, each consisting of a single connected component from $\mathcal{C}(Sep^\alpha)$ (see Lemma 3.3; recall that this is a consequence of the fact that G is k-connected). For convenience, each connected component can be represented by the smallest numbered node appearing in that component. $Part^\alpha$ specifies the assignment of connected components to subgraphs of the tree-decomposition. In particular, P_0^α

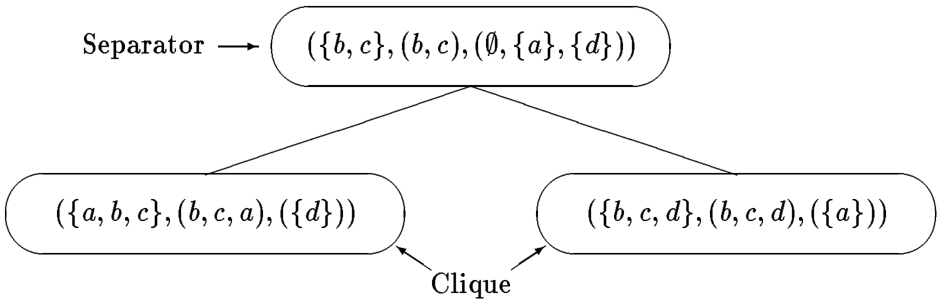

Separator
$(\{b,c\},(b,c),(\emptyset,\{a\},\{d\}))$
$(\{a,b,c\},(b,c,a),(\{d\}))$
$(\{b,c,d\},(b,c,d),(\{a\}))$
Clique

Figure 4: Part of $TDG(G)$

is the set of nodes in the connected component assigned to the subgraph of non-descendants of x and the P_i^α's for $i \neq 0$ to the subtrees rooted at the r children of x. The tdg-vertices represent all possible ways of making such assignments.

A clique vertex is defined similarly; the main difference is that in a clique vertex there must exist a node which did not exist in its parent. A clique vertex β is a tuple $(Clique^\beta, \gamma^\beta, Region^\beta)$, where $Clique^\beta$ is a set of nodes in G, γ^β is a permutation on the nodes in that set, and $Region^\beta$ is a set of regions defined below. The set of partitions, $Region^\beta$, consists of the set R_0^β of nodes forming the parent partition of $Clique^\beta$, plus pairs (R_i^β, S_i^β), where S_i^β is $Clique^\beta$ less one node x_i (x_i not the new node) and R_i^α is G less the component of $\mathcal{C}(S_i^\beta)$ which contains x_i.

Just as the tdg-vertices correspond to possible vertices in a tree-decomposition of G, the edges in the tree-decomposition graph correspond to potential edges in a tree-decomposition. To capture this notion, we group children of a node into clusters, such that the edge between the parent and only one of the children in a cluster can exist. We can think of the clusters as a level of OR gates between the parents and the children. Then, a clique vertex is in the jth cluster of a separator vertex (the parent) if the clique vertex contains all the nodes in the parent plus a new node in the jth partition, and if the sets of connected components match up in the appropriate way.

The requirements for a separator vertex to be in the cluster of a clique vertex are similar in general. In the case of a k-connected partial k-tree, there is only one possible choice for a separator and thus the clusters are trivial.

The following formalizes the intuitive definition given above. Figure 4 illustrates a part of the tree-decomposition graph for the graph G illustrated in Figure 1.

Definition: Let G be a k-connected graph of width k. The *tree-decomposition graph* of G, $TDG(G)$, is a directed graph $(VS \cup VC, E)$.

1. The set VS consists of all triples $\alpha = \{(Sep^\alpha, \sigma^\alpha, Part^\alpha)\}$ satisfying:

   (a) Sep^α is a subset of $V(G)$ of size k;

   (b) σ^α is a permutation of the elements of Sep^α; and

   (c) $Part^\alpha$ is a set $(P_0^\alpha, P_1^\alpha, \ldots, P_r^\alpha)$, $r \leq |\mathcal{C}(Sep^\alpha)|$ such that for each i, $0 \leq i \leq r$, P_i^α is a set of nodes of G, and the P_i^α's form a partition of $\mathcal{C}(Sep^\alpha)$ such that P_0^α contains zero or one connected component and the remaining P_i^α's contain exactly one connected component each.

2. The set VC is the set of all tuples $\beta = \{Clique^{\beta}, \gamma^{\beta}, Region^{\beta}\}$, such that:

   (a) $Clique^{\beta}$ is a subset of $V(G)$ of size $k+1$;

   (b) γ^{β} is a permutation of the elements of $Clique^{\beta}$, say $x_1, \ldots, x_{k+1}$;

   (c) $Region^{\beta}$ is a set $(R_0^{\beta}, (R_1^{\beta}, S_1^{\beta}), \ldots, (R_k^{\beta}, S_k^{\beta}))$, such that the following conditions are satisfied:

      i. for $1 \leq i \leq k$, $S_i^{\beta} = Clique^{\beta} \backslash \{x_i\}$;

      ii. $R_0^{\beta} = \cup\{C : C \in \mathcal{C}(G_{[x_1,\ldots,x_k]})$ and $x_{k+1} \notin C\}$;

      iii. $R_i^{\beta} = \cup\{C : C \in \mathcal{C}(G_{[S_i^{\beta}]})$ and $x_i \notin C\}$.

3. A clique node $\beta = (Clique^{\beta}, \gamma^{\beta}, Region^{\beta})$ with $Clique^{\beta}$ under the permutation γ^{β} equal $(x_1, \ldots, x_{k+1})$ is in the jth cluster of a separator node $\alpha = \{(Sep^{\alpha}, \sigma^{\alpha}, Part^{\alpha})\}$ (the parent) if and only if

   (a) $x_{k+1} \in P_j$.

   (b) Sep^{α} under the permutation σ^{α} is $(x_1, \ldots, x_k)$;

   (c) $R_0^{\beta} = \bigcup_{i \neq j} P_i^{\alpha}$;

   (d) $P_j^{\alpha} = \{x_{k+1}\} \cup \bigcup_{i>0} R_i^{\beta}$.

4. A separator node $\alpha = \{(Sep^{\alpha}, \sigma^{\alpha}, Part^{\alpha})\}$ is in the jth cluster of a clique node $\beta = (Clique^{\beta}, \gamma^{\beta}, Region^{\beta})$ (the parent) with $Clique^{\beta}$ under the permutation γ^{β} equal $(x_1, \ldots, x_{k+1})$ if and only if

   (a) Sep^{α} under the permutation σ^{α} is $(x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_{k+1})$;

   (b) $P_0^{\alpha} = \bigcup_{i \neq j} R_i^{\beta}$;

   (c) $R_j^{\beta} = \bigcup_{i>0} P_i^{\alpha}$.

As we noted earlier, it is easy to show that for a k-connected partial k-tree, there will be at most one separator node in the cluster of each clique node. We present these definitions in a more general form to serve as a basis for the definitions for other cases, to be considered later in the paper.

3.2.2 Facts about $TDG(G)$

In order to make use of tree-decomposition graphs in our algorithm, we first establish a number of properties concerning the size and structure of the graphs. Similar lemmas about tree-decomposition graphs of bounded degree partial k-trees can be found in Section 4.2.

Lemma 3.4. *Let $TDG(G)$ be the tree-decomposition graph of a k-connected partial k-tree G, and let $n = |V(G)|$. Then, $TDG(G)$ has size $O(n^{k+1})$ and can be created in time $O(n^{k+2})$.*

Proof: To create all the separator tdg-vertices in $TDG(G)$, we first identify all separators of the appropriate size, and for each separator determine all the connected components which remain when the separator is removed. For convenience, a connected component can be represented by the smallest numbered vertex in that component (unique numbers can be assigned to the nodes in G in a preprocessing step using a depth-first search). The number of separators is at most $\binom{n}{k}$, which is in $O(n^k)$. For each separator, the connected components can be found in time $O(n)$, for a total cost of $O(n^{k+1})$. At this point we have generated all possible values for the first component in a separator tdg-vertex.

Next, for each separator, we enumerate all possible orderings, giving a total of $O((k+1)!)$ orderings per separator, or $O((k+1)!n^{k+1})$ different possibilities for the first two components in a tdg-vertex. To determine all possibilities for the third component, recall that by Lemma 3.3 each partition consists of a single component. It suffices to specify which of the $O(n)$ components is the parent component, since the order of the children partitions is unimportant.

To count the number of clique tdg-vertices, we consider the number of possible choices for *Clique*, orderings, and for each of the constant number of subsets, the choice of components, for a total of $O(n^{k+1})$.

Finally, we determine the memberships in all clusters. For a particular clique tdg-vertex, a separator tdg-vertex in its jth cluster must have a particular separator and a particular ordering on that separator. Moreover, as a consequence of Lemma 3.3 and the fact that the nodes associated with the separator tdg-vertex form a subset of the nodes associated with the clique tdg-vertex, the partition induced by the clique tdg-vertex dictates the partition induced by the separator tdg-vertex. Since there is a single candidate for a given clique tdg-vertex and cluster, there are a total of $O(n^{k+1})$ candidates for all clique vertices (recall that the number of clusters of a clique tdg-vertex is constant). For a particular separator tdg-vertex, there are $O(n)$ different possible ways of extending the separator label to form a clique label, and for each of those $O(1)$ ways of partitioning since the parent partition is fixed and the other partitions are each a connected component. This yields a total time of $O(n^{k+2})$ for the whole algorithm. ∎

The next three lemmas show that tree-decompositions are structurally contained inside a tree-decomposition graph.

Lemma 3.5. *For G a graph, the tree-decomposition graph $TDG(G)$ is acyclic.*

Proof: Suppose that there is an edge from a clique tdg-vertex β (parent) to another separator tdg-vertex α (child). By the requirements for the existence of the edge, the size of P_0^β must be strictly smaller than the size of P_0^α. The acyclicity of the graph follows from this fact. ∎

Definition: Let G be a graph of width k and α a tdg-vertex of $TDG(G)$. The *graph induced by* α, $G_{(\alpha)}$ is the subgraph of G induced by $V(G)\backslash V(P_0)$.

The proof of Lemma 3.6 (and the analogous lemma for clique tdg-vertices, omitted from this paper) is a direct consequence of the conditions specifying an edge in the definition of a tree-decomposition graph.

Lemma 3.6. *Let $TDG(G)$ be a tree-decomposition graph of a graph G. Let α be a separator tdg-vertex with children $\beta_1, \ldots, \beta_r$. Let $Part^\alpha = (\mathcal{P}_0^\alpha, \ldots, \mathcal{P}_s^\alpha)$ be the partition induced by α. Then, there is a one-to-one correspondence between the β_i and the $\mathcal{P}_i^\alpha$, $1 \le i \le r$.*

3.3 General algorithm sketch

Since the same techniques are used in all eight algorithms discussed in this paper, the algorithms share a general structure. In this section we mention the points common to all the algorithms; in later sections we will highlight the differences between them.

Each algorithm first creates a normalized tree-decomposition of H. In the parallel algorithms, the tree-decomposition of H undergoes further processing to ensure sublinear running time.

Next, the tree decomposition graph of G is formed. Each algorithm entails searching in $TDG(G)$ for a tree-decomposition of G which mimics that of H. The way in which H can be transformed into G depends on whether the algorithm is a subgraph isomorphism algorithm or a topological embedding algorithm. Thus, the exact definition of $TDG(G)$ depends on which problem is being solved. Recall that the problems are NP-complete for G a general partial k-tree; to obtain polynomial time algorithms, we exploit the nature of G as being of bounded degree or k-connected. The definition of $TDG(G)$ differs for these two cases.

To be able to talk about the similarity between a tdg-vertex in $TDG(G)$ and a vertex in T^H, we define a special type of tdg-mapping between the two structures. Ultimately, we wish to show that if there exists a tdg-mapping between any tdg-vertex and the root of T^H, then we can conclude that there exists a subgraph of H isomorphic to G, in the case of the subgraph isomorphism problem, or that G can be topologically embedded in H, in the case of the topological embedding problem.

Finally, each algorithm proceeds by processing the nodes of T^H from bottom to top, at each point determining for each tdg-vertex in $TDG(G)$ whether or not there is a tdg-mapping from the tdg-vertex to the node of T^H. The results of this determination for a particular node of T^H are stored in a bit vector or array for that node. The size of the array and the amount of information to be stored in an entry depend on the particular problem being solved. In each case, the values in the array for a particular vertex of T^H will depend on values in the arrays at children of that vertex.

The running time of the algorithm is then the time to create a tree-decomposition of H, added to the time to normalize that tree-decomposition (and add further structure, if needed), added to the time to create $TDG(G)$, and finally added to the time to fill in each array in T^H. This last quantity will consist of the number of vertices in T^H multiplied by the time to fill in all entries in one array, in the case of a sequential algorithm, and in the case of a parallel algorithm, the depth of the newly structured T^H multiplied by the time to fill in all entries in one array.

4 Sequential subgraph isomorphism

In this section we present an $O(n^{k+4.5})$ time algorithm to determine, given a k-connected partial k-tree G and a partial k-tree H, whether H contains a subgraph isomorphic to G. We then show how this algorithm can be modified to yield an $O(n^{k+2})$ time algorithm for the subgraph isomorphism problem for G a bounded degree partial k-tree, and polynomial time algorithms for topological embedding. Although there are other algorithms known for these problems [MT92], we present ours as natural generalizations of the approaches taken in solving subgraph isomorphism on trees which, in turn, generalize naturally to sequential topological embedding algorithms and parallel algorithms for all the problems under consideration.

4.1 Subgraph isomorphism for the k-connected case

In this section, we present details of the subgraph isomorphism algorithm for a k-connected source graph. In order to be able to discuss the mapping of clique (respectively, separator) tdg-vertices in $TDG(G)$ to clique (respectively, separator) vertices of T^H, we need to define an isomorphism between the subgraphs of G induced by the two sets of nodes.

Definition: Given a $TDG(G)$ clique vertex β and a vertex x in T^H, we will say that β is *order preserving isomorphic to* x if $k+1 = |Clique^\beta| = |\chi^H(x)|$ and $G_{[Clique^\beta]}$ is isomorphic to a subgraph of $H_{[\chi(x)]}$, such that the sequence of nodes of G in $Clique^\beta$ ordered by the permutation γ^β map to the nodes in $\chi^H(x)$, in order, in that isomorphism.

The definition with respect to separator vertices is analogous, and hence omitted. More generally, we require isomorphisms from subgraphs of $TDG(G)$ to subtrees of a tree-decomposition of H; the definition with respect to separator vertices is analogous.

Definition: Suppose there is a clique tdg-vertex β of $TDG(G)$ and a vertex x of T^H such that $G_{(\beta)}$ is isomorphic to a subgraph of $H_{\{T_x\}}$. Let f be an isomorphism between these two graphs. Then, f is a *tdg-isomorphism* from β to x if:

1. f is an order preserving isomorphism of β into x; and
2. for each child α of β there is a unique child y of x such that (the appropriate restriction of) f is a tdg-isomorphism from α to y.

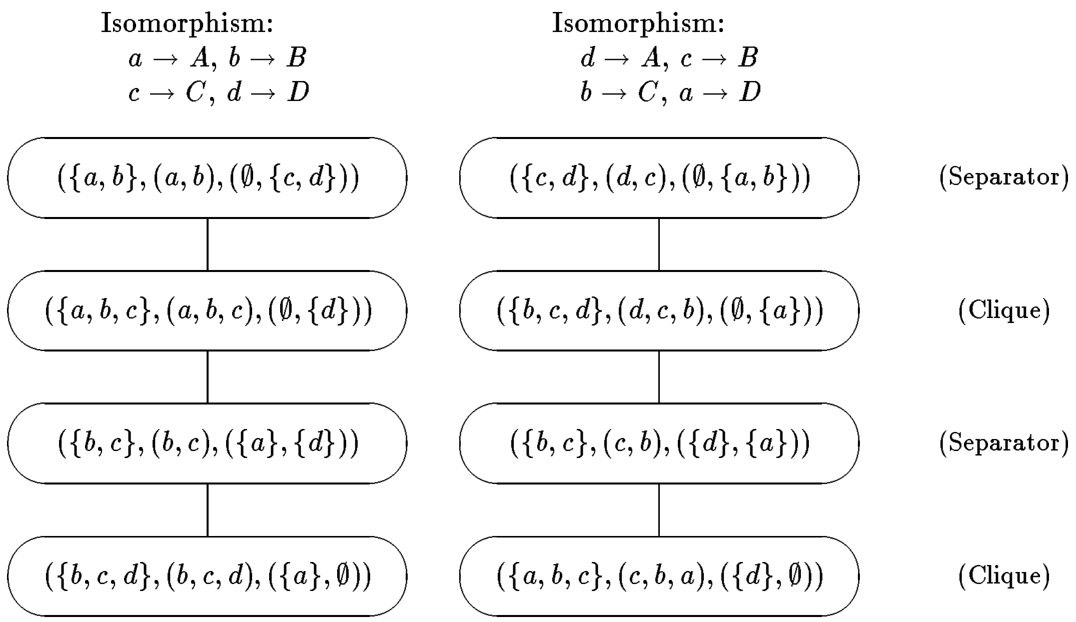
Theorem 4.1. *Let G and H be partial k-trees for k greater than zero, and let G be k-connected. Let $n = |V(G)| + |V(H)|$. Then there is an $O(n^{k+4.5})$ time algorithm to determine whether or not G is isomorphic to a subgraph of H.*

Proof of Theorem:
Our algorithm first creates a normalized tree-decomposition of H, (T^H, χ^H), and after that a tree-decomposition graph of G, $TDG(G)$. We then wish to determine, for every tdg-vertex and every vertex of T^H, whether or not there is a tdg-isomorphism from the tdg-vertex to the vertex of T^H. If there is a tdg-isomorphism from any sink in $TDG(G)$ to the root of T^H, we will then conclude that there exists a subgraph of H which is isomorphic to G.

With every vertex x of T^H we associate a bit vector, B_x, of length $|TDG(G)|$ with one bit for each tdg-vertex of $TDG(G)$. The bit corresponding to tdg-vertex α of $TDG(G)$ is denoted by $B_x[\alpha]$. We will set $B_x[\alpha]$ to 1 if and only if there is a tdg-isomorphism from α to x, proceeding up from the leaves of T^H. For a vertex x of T^H and tdg-vertex α of $TDG(G)$, $B_x[\alpha]$ is determined after all $B_y[\beta]$ have been determined where y is a child of x and β is any tdg-vertex of $TDG(G)$.

We now show how to determine whether or not $B_x[\alpha] = 1$. If x has no children, then $B_x[\alpha] = 1$ if and only if α is order preserving isomorphic to x, and $Part^\alpha = (P_0)$ (which implies that α has no children). This can be determined in constant time.

We now consider the situation in which x has at least one child. Let $y_1, \ldots, y_m$ be the children of x and $\beta_1, \ldots, \beta_r$ be the children of α. When we process x and α, vectors for all children of x have been completed; in particular, we have obtained all tdg-isomorphism information concerning


Isomorphism:
$a \to A$, $b \to B$
$c \to C$, $d \to D$
Isomorphism:
$d \to A$, $c \to B$
$b \to C$, $a \to D$
$(\{a,b\},(a,b),(\emptyset,\{c,d\}))$
$(\{c,d\},(d,c),(\emptyset,\{a,b\}))$
(Separator)
$(\{a,b,c\},(a,b,c),(\emptyset,\{d\}))$
$(\{b,c,d\},(d,c,b),(\emptyset,\{a\}))$
(Clique)
$(\{b,c\},(b,c),(\{a\},\{d\}))$
$(\{b,c\},(c,b),(\{d\},\{a\}))$
(Separator)
$(\{b,c,d\},(b,c,d),(\{a\},\emptyset))$
$(\{a,b,c\},(c,b,a),(\{d\},\emptyset))$
(Clique)

$TDG(G)$ contains both segments given above. The isomorphism depends on the tree decomposition of H.

Figure 5: Parts of $TDG(G)$ which map to (T^H, χ^H)

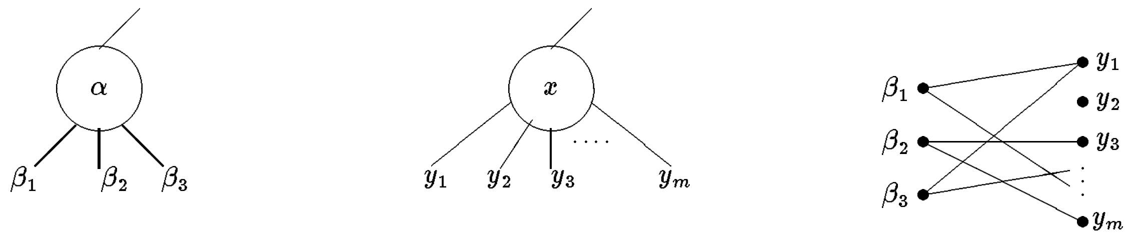
children of α and children of x. More formally, for $1 \leq i \leq m$, $1 \leq j \leq r$, $B_{y_i}[\beta_j]$ has been determined.

We construct a bipartite graph (X, Y) where there is a vertex in X for each β_i and a vertex in Y for each y_j. There is an edge from β_i to y_j if $B_{y_j}[\beta_i] = 1$. Then we set $B_x[\alpha]$ to 1 if there is a matching in (X, Y) in which all of X is used and there is an order preserving isomorphism from α to x which is consistent with the matching.

An illustration of a matching problem can be seen in Figure 6, where a tdg-vertex α has three children and a vertex x in the tree-decomposition of H has m children.

By finding tdg-isomorphisms from tdg-vertices in $TDG(G)$ to vertices in T^H, the process of filling in the entries in the bit vectors can be seen as selecting tdg-vertices that represent a tree-decomposition of G corresponding to that of H. Since there are nodes in H which do not appear in G, it is possible that there is not a one-to-one correspondence between selected tdg-vertices and vertices of T^H. In particular, consider a separator vertex z with parent y and grandparent z, all in T^H. By definition, there exists in H a node a which is in $\chi^H(y)$ and $\chi^H(z)$, but not in $\chi^H(x)$. If no node in G maps to a, then $TDG(G)$ will not contain tdg-vertices for all of x, y, and z.

In order to handle this situation, it is necessary to check whether or not the desired tdg-isomorphism was not already achieved at the previous level of separator tdg-vertices (respectively, clique tdg-vertices). Accordingly when for some grandchild z of x, $B_\alpha[z] = 1$ and α is tdg-isomorphic



In $TDG(G)$, α has 3 children
Bold lines indicate clusters

In (T^H, χ^H), x has m children

Setting up the matching
(β_i, y_j) an edge iff $B_{y_j}[\beta_i] = 1$

Figure 6: Setting up a matching between α and x

to x with the same tdg-isomorphism as that from α to z, we set $B_x(\alpha) = 1$. There are $O(n)$ grandchildren to check for each x.

Figure 5 illustrates two possible mappings obtained between $TDG(G)$ and (T^H, χ^H) when the input graphs are graph G from Figure 1 and graph H from Figure 3. For each of the mappings, the corresponding isomorphism is illustrated.

To prove that the algorithm is correct, it is sufficient to show by induction that for any α and x, there is a subgraph of $H_{\{T_x\}}$ isomorphic to $G_{(\alpha)}$ if and only if there is a tdg-isomorphism from α to x. The proof follows from the definition of the tdg-isomorphism and the algorithm; the details are omitted from this paper.

To determine the running time of this algorithm, we recall we can construct a normalized tree decomposition of H in time $O(n^2)$ (Corollary 3.2). Further, we recall that the size of $TDG(G)$ is $O(n^{k+1})$ and that $TDG(G)$ can be constructed in time $O(n^{k+2})$ (Lemma 3.4). For x a vertex of T^H and α a tdg-vertex of $TDG(G)$, determining $B_x[\alpha]$ can be accomplished in time $O(n^{2.5})$ (due to the cost of bipartite matching) giving a time bound of $O(n^{k+4.5})$. ∎

4.2 Subgraph isomorphism for the bounded degree case

We give a brief outline of the changes to the above algorithm for the case where G is of bounded degree but not necessarily k-connected. Recall that in Lemma 3.3 we asserted that when a separator was removed from G, each remaining piece was connected. When we created the tree-decomposition graph of G, to form a partition into parent and child partitions, it was sufficient to identify which one connected component belonged to the parent partition. Now, we have to consider the cost of assigning multiple connected components to a particular partition. Moreover, in the previous case we could assume that the size of a clique was always $k+1$ and that of a separator was always k; in this more general case, we can no longer make such an assumption.

4.2.1 Modification of $TDG(G)$

To handle the situation in which a tdg-vertex is to be mapped to a vertex of T^H containing a different number of vertices, we make the following changes to $TDG(G)$:

1. Sep^α is an array of size k of which zero or more entries may be blank;
2. $Clique^\beta$ is an array of size $k+1$ of which zero of more entries may be blank;
3. P_0^α contains zero or more connected components and the remaining P_i^α's contain one or more connected components each (recall that the P_i's form a partition of $\mathcal{C}_G(Sep^\alpha)$);
4. S_i^β is a subset of the numbers $\{1,\ldots,k\}$ corresponding to a subset of $Clique^\beta$ with the new node understood to be included. Since $Clique^\beta$ may contain blanks, for any $j \in S_i^\beta$, the jth item in $Clique^\beta$ may be a blank.

We can make use of the fact that G has bounded degree to keep $TDG(G)$ from becoming too large. Lemma 4.2 establishes the size and creation time.

Lemma 4.2. *Let $TDG(G)$ be the tree-decomposition graph of a bounded degree partial k-tree G. Let d be the maximum degree of a node in G, and let $n = |V(G)|$. Then, $TDG(G)$ has size $O(n^{k+1})$ and can be created in time $O(n^{k+2})$.*

Proof: As this proof is very similar to that of Lemma 3.4, we omit some of the details in order to highlight the differences.

To create all the separator tdg-vertices in $TDG(G)$, we first identify all separators of the appropriate size, and for each separator determine all the connected components which remain when the separator is removed. The number of separators will be at most $\sum_{i=1}^{k}\binom{n}{i}$, which is in $O(n^k)$. For each separator, the connected components can be found in time $O(n)$, for a total cost of $O(n^{k+1})$. At this point we have generated all possible values for the first component in a separator tdg-vertex.

Next, for each separator, we enumerate all possible orderings, giving a total of $O((k+1)!)$ orderings per separator, or $O((k+1)!n^{k+1})$ different possibilities for the first two components in a tdg-vertex. To determine all possibilities for the third component, we consider all partitions of the components into r different subsets, and for each partition, all possible choices of a single part of the partition as the parent partition (recall that the order of the children partitions is unimportant). Since the degree of the graph is bounded, we are determining all possible partitions of a constant number of connected components into $r+1$ nonempty sets (if the parent partition is nonempty), which we multiply by $r+1$ to account for the selection of a particular piece as the parent partition, plus the number of possible partitions of d connected components into r nonempty sets (if the parent partition is empty). This gives us a constant number of possible partitions for each setting of the first two components in the vertex.

To count the number of clique tdg-vertices, we consider the number of possible choices for $Clique$, orderings, and for each of the constant number of subsets, the choice of components, for a total of $O(n^{k+1})$.

Finally, we determine the memberships in all clusters. For a particular clique tdg-vertex, a separator tdg-vertex in its jth cluster must have a particular separator and a particular ordering on that separator. There are at most $O(1)$ possible candidates (differing by the way in which connected components are partitioned), or $O(n^{k+1})$ for all clique vertices (recall that the number of clusters of a clique tdg-vertex is constant).

For a particular separator tdg-vertex, there are $O(n)$ different possible ways of extending the separator label to form a clique label, and for each of those $O(1)$ ways of partitioning, for a total of $O(n^{k+1})$ candidates for all separator vertices.

To check whether or not a candidate is in a cluster requires checking partitioning information, which can be accomplished in $O(n)$ time, for a total of $O(n^{k+2})$ time for the whole algorithm. ■

4.2.2 Modification of the tdg-mapping

In order to incorporate the changes in the definition of the tree-decomposition graph, we alter slightly our definition of an order preserving isomorphism.

Definition: Given a $TDG(G)$ clique vertex β and a vertex x in T^H, we will say that β is *order preserving isomorphic to* x if the array for $Clique^\beta$ is of length $|\chi^H(x)|$, and $G_{[Clique^\beta]}$ is isomorphic to a subgraph of $H_{[\chi(x)]}$, such that for all ℓ the node of G in position ℓ in the array for $Clique^\beta$ maps to the ℓth node in $\chi^H(x)$ in that isomorphism.

The definition with respect to separator vertices is analogous, and hence omitted; the definition of a tdg-isomorphism follows from the above definition.

4.2.3 Modification of the algorithm

We now consider the differences between the algorithm for the k-connected case and the algorithm for the bounded degree case.

Theorem 4.3. *Let $k, d > 0$ and let G and H be partial k-trees, G with degree at most d. Let $n = |V(G)| + |V(H)|$. Then there is an $O(n^{k+2})$ time algorithm to determine whether or not G is isomorphic to a subgraph of H.*

Proof of Theorem:

Since the algorithm is very similar to that given in the proof of Theorem 5.1, we omit the similarities in favor of highlighting the differences between the two algorithms.

The key difference between the algorithms is in the determination of whether or not $B_x[\alpha] = 1$ for a tdg-vertex α with at least one child. As before, let $y_1, \ldots, y_m$ be the children of x and $\beta_1, \ldots, \beta_r$ be the children of α. Assume that for $1 \leq i \leq m$, $1 \leq j \leq r$, $B_{y_i}[\beta_j]$ has been determined. The bit vectors for all children of x have been completed; in particular, we have obtained all tdg-isomorphism information concerning children of α and children of x. Also as before, we construct a bipartite graph (X, Y) where there is a vertex in X for each β_i and a vertex in Y for each y_j. There is an edge from β_i to y_j if $B_{y_j}[\beta_i] = 1$. Then we set $B_x[\alpha] = 1$ if and only if there is a matching in (X, Y) in which all of X is used and there is an order preserving isomorphism from α to x which is consistent with the matching.

To solve a matching problem, we first partition X into the set X_s of vertices of degree at most $|X|$ in (X, Y) and the set X_ℓ of vertices of degree greater than $|X|$ in (X, Y). Since both X_s and $Y_s = nbhr_{X_s}(Y)$ are of constant size, it can be determined in constant time whether or not a matching using all of X_s exists in (X_s, Y_s).

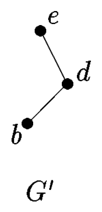


Figure 7: A input graph for subgraph isomorphism, bounded degree case

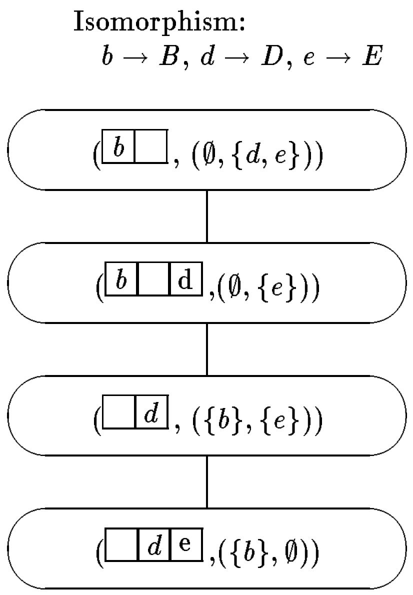

Isomorphism:
$b \to B$, $d \to D$, $e \to E$
$(\boxed{b}\boxed{\ }, (\emptyset, \{d, e\}))$
$(\boxed{b}\boxed{\ }\boxed{d}, (\emptyset, \{e\}))$
$(\boxed{\ }\boxed{d}, (\{b\}, \{e\}))$
$(\boxed{\ }\boxed{d}\boxed{e}, (\{b\}, \emptyset))$

Figure 8: Part of $TDG(G')$ which maps to (T^H, χ^H)

If there is no such matching in (X_s, Y_s), then we can conclude that there is no such desired matching in (X, Y). If on the other hand there is a matching in (X_s, Y_s), we can remove from Y all matched vertices in Y_s to create Y_ℓ. Since the number of matched vertices in Y_s is exactly $|X|$, each node in X_ℓ has degree greater than $|X| - |X_s| = |X_\ell|$ in Y_ℓ, guaranteeing that a matching exists. It can be found using a simple greedy algorithm.

For inputs the graph G' illustrated in Figure 7 and the graph H illustrated in Figure 3, Figure 8 illustrates a possible mapping obtained between (T^H, χ^H) and $TDG(G')$.

It is not difficult to see that each matching problem can be solved in constant time, yielding an overall execution time of $O(n^{k+2})$, as claimed. ∎

5 Sequential topological embedding

In an algorithm with a similar structure, the tree-decomposition graph and the array used in processing H can be modified to allow us to solve the topological embedding problem. As mentioned in Section 2.4, if G is isomorphic to a subgraph of H, we can view each node of H as being either in the image of the mapping of the nodes of G to the nodes of H, or extraneous to the mapping. We can view the topological embedding of G in H as the determination of a subgraph of H such that each node in the subgraph is either in the image of the mapping of the nodes of G to the nodes of H, or is an intermediate node in a path to which one of the edges of G is mapped. A node of H which is the intermediate node in such a path is said to have been *collapsed* into one of the endpoints of the path; special care has to be taken to ensure that all collapsing can be detected by the algorithm.

5.1 Modification of $TDG(G)$

To account for collapsed nodes of H, we modify $TDG(G)$ to include tdg-vertices which mimic the structure of the section of T^H involving the addition of a node which is collapsed. In particular, a collapsed node may be the new node added in a clique in T^H; to account for this possibility, we allow a separator node α to have as a child a clique node β such that $Sep^\alpha = Clique^\beta$. The modifications made to the definition of $TDG(G)$ are as below; the same modifications are made for both the bounded degree case and the k-connected case:

1. A separator node α is in a cluster of a clique node β if Sep^α is a proper subset of $Clique^\beta$ (and the other conditions in $TDG(G)$ are satisfied).
2. A clique node β is in a cluster of a separator node α if $Clique^\beta = Sep^\alpha$ or if $Clique^\beta$ contains one new node (and the other conditions in $TDG(G)$ are satisfied).
3. For a clique node β in the cluster of a separator node α such that $Clique^\beta = Sep^\alpha$, α is also in a cluster of β. We call the edge from child α to parent β an *up-edge*.

It is not hard to see that the size of $TDG(G)$ increases by at most a constant factor, and if we ignore up-edges, $TDG(G)$ is acyclic, as needed.

5.2 Modification of the tdg-mapping

To determine whether or not a tdg-vertex α can tdg-map to a vertex x of T, we must be able to map each node in Sep^α (or $Clique^\alpha$) into a node in $\chi^H(x)$ and each edge in $G_{[Sep^\alpha]}$ (or $G_{[Clique^\alpha]}$) into a path in H. Due to the presence of collapsed nodes in H, particular care must be taken in the combining of mappings obtained for children of x. The arrays at each x hold information sufficient to allow the correct combination of children mappings, with several entries for each α.

For each x and α, the array contains entries for each partition of the edges in $G_{[Sep^\alpha]}$ (or $G_{[Clique^\alpha]}$) into the sets *Above*, *Below*, and *Inside*. An edge is in the set *Below* if the entire path to which that edge is mapped is contained in $H_{\{T_y\}}$ for some child y of x. If an edge is in the set *Inside*, the edge can be mapped to a path in $H_{\{T_x\}}$, of which at least one edge is contained in $H_{[\chi(x)]}$. *Above* contains all edges that require the use of edges in $H_{\{T \setminus T_x\}}$. When filling in the

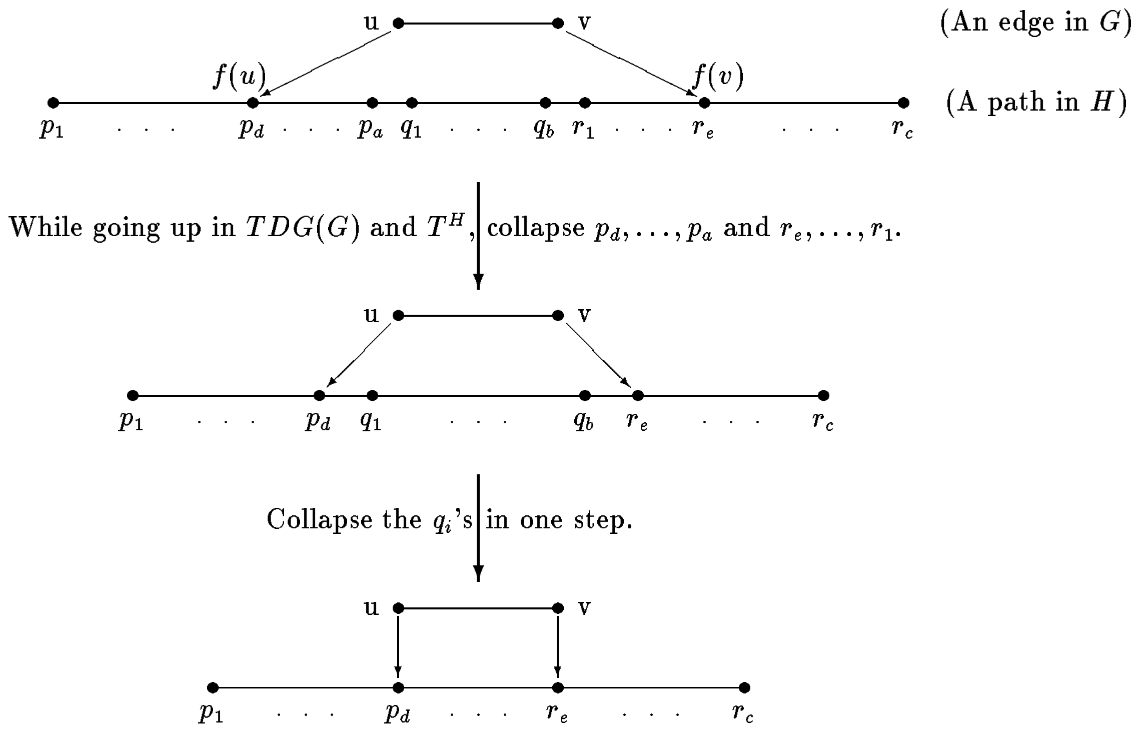

u
v
(An edge in G)
$f(u)$
$f(v)$
(A path in H)
p_1 · · · p_d · · · p_a q_1 · · · q_b r_1 · · · r_e · · · r_c
While going up in $TDG(G)$ and T^H, collapse $p_d, \ldots, p_a$ and $r_e, \ldots, r_1$.
p_1 · · · p_d q_1 · · · q_b r_e · · · r_c
Collapse the q_i's in one step.
p_1 · · · p_d · · · r_e · · · r_c

Figure 9: Mapping an edge to a path

array for x using the arrays for the children of x, it is necessary to be sure that the partitions are consistent, namely that each edge appears somewhere in H.

Since the edges in G must be mapped to node-disjoint paths in H, it is important to keep track of the collapsed nodes to be sure that each is used in no more than one path. The detection of a path in H can be seen as occurring in several states. Suppose the edge (u, v) in G is mapped to the path $p_1, \ldots, p_d, \ldots, p_a, q_1, \ldots, q_b, r_1, \ldots, r_e, \ldots, r_c$ in H. First u is mapped to a node p_i, say p_d, when tdg-vertex α_1 is tdg-mapped to vertex x_1 in T^H. Then, in a series of ancestors of α_1 tdg-mapping into a series of ancestors of x_1, p_d is collapsed into p_{d+1}, p_{d+1} is collapsed into p_{d+2}, and finally p_{a-1} is collapsed into p_a. Similarly, v is tdg-mapped to some r_j, say r_e, when tdg-vertex α_2 is tdg-mapped to vertex x_2 in T^H, and subsequently r_e is collapsed into r_{e-1} and so on until r_2 is collapsed into r_1. There is an ancestor β of α_1 and α_2 that maps into an ancestor y of x_1 and x_2, such that y contains both p_a and r_1. In the mapping of β to y, q_1 through q_d are collapsed into p_a and q_{d+1} through q_b are collapsed into r_1 to complete the path. It is not difficult to see that the segments p_1 through p_{d-1} and r_{e+1} through r_c are determined through a similar series of collapses.

This example is illustrated in Figure 9.

The above example contains the two ways in which paths are determined: the collapsing of a series of nodes occurs when the edge is in the set *Above* and the mapping of β to y when the edge is in the set *Inside*. When an edge is in the set *Inside*, it suffices to map that edge into a node-disjoint path. In terms of the definition below, f is the mapping of u to p_a and v to r_1, and g is the mapping which collapses the q_i's into p_a and r_1.

Definition: Given a $TDG(G)$ vertex α and a vertex x in T^H, we will say that α *embeds in x subject to Inside, f, and g*, for *Inside* a subset of the edges in $G_{[Clique^\alpha]}$ (or $G_{[Sep^\alpha]}$), if and only if the graph with vertex set $Clique^\alpha$ (or Sep^α) and edge set *Inside* is topologically embeddable in $H_{[\chi(x)]}$ with respect to (f, g).

For convenience, we will use the following notation:

Definition: Given a $TDG(G)$ vertex α that embeds in a vertex x in T^H subject to *Inside*, f, and g, we use $FI(x, \alpha)$ to denote the nodes in the image of f, $G(x, \alpha)$ to denote the domain of g and $GI(x, \alpha)$ to denote the image of g.

Handling the situation in which an edge (u, v) is in the set *Above* requires more book-keeping. As we saw in the previous example, the node-disjoint path to which (u, v) is mapped may be partially determined by a series of collapses. An intermediate node on the path can be collapsed into another intermediate node or into an endpoint; moreover, it is also possible for an endpoint to be collapsed into an intermediate node. If there is a function f governing the mapping from α to x and another function f' governing the mapping from a child of α to a child of x, then either $f(u)$ is collapsed into $f'(u)$ or $f'(u)$ is collapsed into $f(u)$. It is important to be able to distinguish which is the endpoint and which an intermediate node. Moreover, in ensuring that the mappings chosen at a particular α and x are consistent with mappings chosen at children of α and x, we must be sure that $G(x, \alpha)$ and $FI(x, \alpha)$ are disjoint from each other as well as from the sets of nodes collapsed in children of α and x. Each node in $G(x, \alpha)$ has degree two with respect to the embedding thus far, so that the collapsing constitutes the mapping of an edge to a path, rather than to a more complicated structure.

To make the consistency checks possible, we keep track of which nodes in $\chi^H(x)$ were collapsed in descendants. In determining whether or not these sets of nodes are disjoint for various children, it is only necessary to consider nodes in $\chi^H(x)$. That is, since T^H is a tree decomposition, we know that the only nodes that could possibly be contained in both $H_{\{T_y\}}$ and $H_{\{T_z\}}$ are those in $\chi^H(y) \cap \chi^H(z)$, which must be contained in $\chi^H(x)$, for y and z children of x.

In addition, at a particular point in the mapping we record the *path degrees* of the nodes in $\chi^H(x)$ with respect to the mapping thus far. The path degree of a node is the number of partially-created paths for which the node is a current endpoint. For our purposes, it will suffice to know whether the number is 0, 1, or more than 1, denoted 2. It is important to note that although in the course of collapsing nodes a particular node u in G may be mapped to many different nodes in H, only one of the nodes in H will be the image of the node in G with respect to the topological embedding of G in H. That single node will have degree at least as great as the degree of u in G; any other node to which u is mapped must be an intermediate node in a path, and hence must have degree two in $H\backslash\{$nodes not in the image of f or in the domain of $g\}$. If the current endpoint is known not to be the image of u (in particular, we have collapsed a high-degree node into it), we mark it with a path degree of -1. We can then use the stored path degrees to ensure that only legal collapsings take place.

Definition: Suppose that α is a tdg-vertex in $TDG(G)$ and x a vertex in T^H. Then, α *tdg-embeds in x subject to P, f, g, S, and PD*, where

1. P is a partition of the edges in $G_{[Clique^\alpha]}$ (or $G_{[Sep^\alpha]}$) into sets *Above*, *Below*, and *Inside*;
2. S is a subset of $\chi^H(x)$; and
3. $PD = (pd_1, \ldots, pd_{|\chi^H(x)|})$ is a tuple of values in $\{-1, 0, 1, 2\}$, the ith entry being the path degree of the ith node in $\chi^H(x)$;

if the following conditions are satisfied:

1. α embeds in x subject to *Inside*, f, and g;
2. for each child β_i of α, and for distinct children y_j of x, each β_i tdg-embeds in a distinct y_j subject to some P^i, f^i, g^i, S^i, and PD^i;
3. for each edge in α, if the edge is in *Above*, it is in *Above* in each β_i in which it exists; if it is in *Inside*, it is in *Above* or *Inside* in each β_i in which it exists; and if it is in *Below*, it is in *Inside* or *Below* in exactly one β_i;
4. the S^i's, $G(x, \alpha)$, $(\bigcup_i FI(y_j, \beta_i))\backslash FI(x, \alpha)$ and $FI(x, \alpha)$ are all disjoint;
5. $S = \bigcup_i (S^i) \cup G(x, \alpha) \cup (\bigcup_i FI(y_j, \beta_i))\backslash FI(x, \alpha)$;
6. for distinct nodes u and v in $Clique^\alpha$ (or Sep^α), $f^i(u) \neq f^j(v)$ for any $i \neq j$;
7. in the subgraph of H induced by $G(x, \alpha) \cup FI(x, \alpha) \cup (\bigcup_i FI(y_j, \beta_i))\backslash FI(x, \alpha)$ the following conditions are satisfied:
   (a) each node in the partition of "unused nodes" in $G(x, \alpha)$ has
      i. degree two in this subgraph, and
      ii. path degree zero in each child of x to which a child of α maps;
   (b) for each u in $Clique^\alpha$ (or Sep^α), if there are at least two j's such that $f^j(u) \neq f(u)$, then for each such j the node $f^j(u)$ has
      i. degree one in the graph,
      ii. a path from $f^j(u)$ to $f(u)$ in which each intermediate node is in the partition of "unused nodes" in $G(x, \alpha)$,
      iii. path degree at least zero in each child of x to which a child of α maps,
      iv. summing over all such children, path degree one, and
      v. in PD, $f(u)$ has path degree equal to the number of j's plus the sum of all path degrees of $f(u)$ in all children of x to which children of α map;

(c) for each u in $Clique^{\alpha}$ (or Sep^{α}), if there is one j such that $f^j(u) \neq f(u)$, then the node $f^j(u)$ either has all the properties above, or

    i. degree one in the graph,

    ii. a path from $f^j(u)$ to $f(u)$ in which each intermediate node is in the partition of "unused nodes" in $G(x, \alpha)$, and

    iii. in PD, $f(u)$ has path degree -1

(d) for all other nodes, the path degree is the sum of the path degrees of the node in all children of x to which children of α map.

5.3 Modification of the algorithm

The proof of Theorem 5.1 is very similar to the proof of Theorem 4.1; accordingly, only the differences are highlighted. Theorem 5.2 is stated without proof, as its proof follows from proofs of Theorems 5.1 and 4.3.

Theorem 5.1. *Let $k, d > 0$ and let G and H be partial k-trees, where G is k-connected. Let $n = |V(G)| + |V(H)|$. Then there is an $O(n^{k^2+3k+6.5})$ time algorithm to determine whether or not G can be topologically embedded in H.*

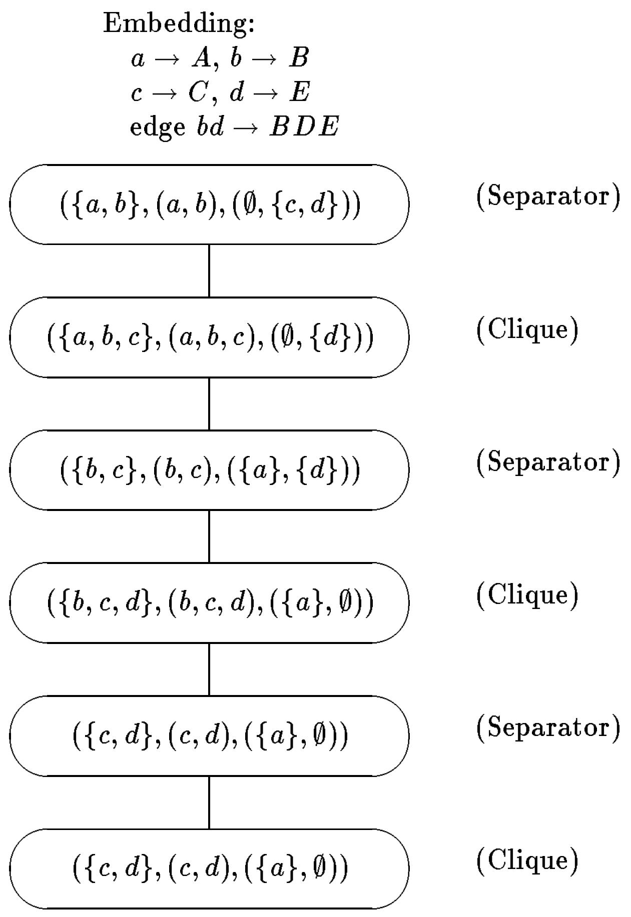
Proof of Theorem 5.1:

Our algorithm first creates a normalized tree-decomposition of H, (T^H, χ^H), and a tree-decomposition graph of G, $TDG(G)$. We then wish to determine, for every tdg-vertex and every vertex of T^H, whether or not the tdg-vertex tdg-embeds in the vertex of T^H. If any sink in $TDG(G)$ tdg-embeds in the root of T^H, we will then conclude that G can be topologically embedded in H.

With every vertex x of T^H we associate an array B_x, with one entry $B_x[\alpha, P, f, g, S, PD]$ for each tdg-vertex of $TDG(G)$, each partition P, each function f, each function g, each subset S, and each tuple PD. We will set $B_x[\alpha, P, f, g, S, PD]$ to 1 if and only if α tdg-embeds in x subject to P, f, g, S, and PD, proceeding up from the leaves of T^H. For a vertex x of T^H and tdg-vertex α of $TDG(G)$, $B_x[\alpha, \ldots]$ is determined after all $B_y[\beta, \ldots]$ have been determined where y is a child of x and β is any tdg-vertex of $TDG(G)$.

We now show how to determine whether or not $B_x[\alpha, P, f, g, S, PD] = 1$, where x is a vertex of T^H and α is a tdg-vertex of $TDG(G)$. If α has no children, then $B_x[\alpha, \ldots] = 1$ if and only if α embeds in x subject to *Inside*, f, and g, which can be determined in constant time.

We now consider the case in which α has at least one child. Let $y_1, \ldots, y_m$ be the children of x and $\beta_1, \ldots, \beta_r$ be the children of α. Assume that for $1 \leq i \leq m$, $1 \leq j \leq r$, $B_{y_i}[\beta_j, \ldots]$ have been determined. The arrays for all children of x have been completed; in particular, we have obtained all tdg-embedding information concerning children of α and children of x. We determine the value of the array $B_x[\alpha, P, f, g, S, PD]$ for fixed values of P, f, g, S, and PD. We now conduct a series of consistency checks and then set up a separate matching problem for each choice of (1) partition of the edges Below into children of α, (2) partition of S into $G(x, \alpha)$, $(\bigcup_i FI(y_j, \beta_i)) \backslash FI(x, \alpha))$, and the contributions associated with each child of α (for the child of x to which a child of α maps); (3) partition of $G(x, \alpha)$ into pieces for each edge in *Inside* and the "unused nodes" partition; (4) partition of each pd_i into "contributions" from each child of α or from α itself, for nonnegative values of pd_i, and for $pd_i = -1$ identification of a child of α associated with this collapse, and (5) assignment



Embedding:
$a \to A$, $b \to B$
$c \to C$, $d \to E$
edge $bd \to BDE$
$(\{a,b\},(a,b),(\emptyset,\{c,d\}))$
(Separator)
$(\{a,b,c\},(a,b,c),(\emptyset,\{d\}))$
(Clique)
$(\{b,c\},(b,c),(\{a\},\{d\}))$
(Separator)
$(\{b,c,d\},(b,c,d),(\{a\},\emptyset))$
(Clique)
$(\{c,d\},(c,d),(\{a\},\emptyset))$
(Separator)
$(\{c,d\},(c,d),(\{a\},\emptyset))$
(Clique)

Figure 10: Parts of $TDG(G)$ mapping to (T^H, χ^H)

of a distinct pair (β_i, u), for $u \in Sep^\alpha$ (or $Clique^\alpha$), to each node in $(\bigcup_i FI(y_j,\beta_i)) \backslash FI(x,\alpha)$. We construct a bipartite graph (X,Y) where there is a vertex in X for each β_i and a vertex in Y for each y_j. There is an edge from β_i to y_j if $B_{\beta_i}[y_j,\ldots] = 1$ subject to each of the preceding fixed partitions and straightforward consistency checks to be sure that values are consistent with those in the choices of partitions above. Then we set $B_\alpha[x] = 1$ if and only if for one of these problems there is a matching in (X,Y) in which all of X is used and α embeds in x subject to *Inside*, f, and g.

It is not difficult to see that each of the conditions in the definition of tdg-embedding is satisfied. Condition 1 is explicitly checked by the algorithm and condition 2 by the nature of the matching problem. The third condition is satisfied by the choice of partition (1), and the fourth and fifth conditions by the choice of partition (2), plus a check that S and f have been constructed in such a way that $FI(x, \alpha)$ is disjoint from S. Condition 6 is satisfied by partition (5). Finally, to see that the last condition is satisfied, we note that the subgraph under consideration is fixed by choices of f, S, and partitions (2), (3), (4), and (5), regardless of which children of α map to which children of x, allowing all conditions to be verified prior to the matching step.

Figure 10 illustrates part of a matching obtained between $TDG(G)$ and (T^H, χ^H), for G as in Figure 1 and H as in Figure 3.

To determine the running time of this algorithm, we recall we can construct a normalized tree decomposition of H in time $O(n^2)$ (Corollary 3.2). Further, we recall that the size of $TDG(G)$ is $O(n^{k+1})$ and that $TDG(G)$ can be constructed in time $O(n^{k+1})$ (Lemma 3.4). For x a vertex of T^H, α a tdg-vertex of $TDG(G)$, P, f, g, S and PD, the number of matching problems set up is the product of the following counts for the various partitions: (1) $O(n^{k^2})$, for the placing of $O(k^2)$ edges in $O(n)$ different sets; (2) $O(n^k)$, for the placing of $O(k)$ nodes in $O(n)$ different sets; (3) $O(1)$, for the placing of k nodes in $O(k^2)$ different sets; (4) $O(n^2)$, for each of $O(1)$ pd_i's, selecting out of $O(n)$ choices; and (5) $O(n^k)$, for choosing one of $O(nk)$ values for each of $O(k)$ nodes. Since each checking step can be done in time $O(n)$, each matching can be solved in time $O(n^{2.5})$ and the total number of values of x, α, P, f, g, S and PD is in $O(n^{k+2})$, the total cost of the algorithm is $O(n^{k^2+3k+6.5})$ ∎

The following theorem is stated without proof, as it is very similar to the previous theorem. The main differences lie in the cost of matching, which is constant, and the counts for the various partitions. Since G is a bounded degree graph, the counts are all constant.

Theorem 5.2. *Let $k, d > 0$ and let G and H be partial k-trees, G with degree at most d. Let $n = |V(G)| + |V(H)|$. Then there is an $O(n^{k+2})$ time algorithm to determine whether or not G can be topologically embedded in H.*

6 Parallelizing the algorithms

The sequential algorithms given in the previous sections were designed in such a way that straightforward parallelization is possible: after a normalized tree-decomposition of H has been obtained and a tree-decomposition graph of G created, it is possible to process the vertices of T^H level by level rather than one at a time. The complexity of the algorithms then depends on the costs of obtaining a tree-decomposition of a graph, normalizing a tree decomposition, and creating a tree decomposition graph, and then on the cost of processing a particular vertex of T^H multiplied by the depth of T^H.

The overall complexity of a parallel algorithm depends in various ways on the choice of tree-decomposition algorithm used. Among the best known tree-decomposition algorithms, there are tradeoffs between running time, processor count, depth of the resulting tree-decomposition, and width of the resulting tree-decomposition. Since our algorithms for the k-connected case depend on a resulting tree-decomposition of width k and since the number of processors needed to form a tree decomposition graph is a function of the width of the tree decomposition, we cannot make use of either Lagergren's algorithm, which using $O(n)$ processors and $O(\log^3 n)$ time to produce a tree-decomposition of width $6k + 5$ [Lag90], or Reed's algorithm, which uses $n/\log n$ processors and $O(\log^2 n)$ time to obtain a tree-decomposition of width $3k$ [Ree92]. Instead we would opt to use either the fastest known parallel tree-decomposition algorithm, running in time $O(\log n)$ and using

$O(n^{3k+4})$ processors [Bod88b], or an earlier algorithm using $O(n^{2k+5})$ processors and $O(\log^2 n)$ time [CH88] each producing a width k tree decomposition.

The choice of tree-decomposition algorithm is further complicated by the importance of the depth of the resulting tree-decomposition on the running time of the subgraph isomorphism algorithm. As the tree-decomposition is processed level by level, the depth is a multiplicative factor in the running time. To obtain a sublinear running time, we could use a result of Bodlaender, who obtains a logarithmic depth tree decomposition of width $3k+2$ in $O(\log n)$ time using $O(n^{3k+4})$ processors [Bod88b], resulting again in the problems associated with a decomposition of width greater than k.

We instead consider a type of algorithm slightly more complicated than the straightforward one described above. Instead of reducing the depth of the tree-decomposition of H, we instead apply a technique which, in a logarithmic number of iterations, breaks T^H into constant size pieces.

We note that since the tree decomposition algorithms are written for CRCW PRAMs, our running times assume concurrent writing. Since the algorithms depend on matching, for which in the general case RNC but not NC algorithms are known, the algorithms for the k-connected case are randomized and those for the bounded degree case are deterministic.

The remainder of the section is organized as follows. First, in Section 6.1, we present a parallel algorithm for normalizing a tree decomposition. Next, Section 6.2 describes the technique used to break T^H into constant size pieces. The parallel algorithm for creating $TDG(G)$ is given in Section 6.3. Sections 6.4 and 6.5 give details of the parallel subgraph isomorphism and topological embedding algorithms.

6.1 Normalizing a tree decomposition

In this section, we show how the algorithm presented in Lemma 3.1 can be parallelized in a straightforward manner. The proof refers to step numbers used in the earlier lemma; the reader is referred to that lemma for further details.

Lemma 6.1. *Let H be a graph and (T^H, χ^H) be a tree-decomposition of H of width k with $n = |V(T^H)|$. Then there is an $O(\log n)$ time $O(n)$-processor algorithm which takes (T^H, χ^H) and returns a normalized tree-decomposition (S^H, λ^H) of H whose width is also k.*

Proof (outline):
Step 1: We contract any edge between vertices x and y such that $\lambda^H(x) \subseteq \lambda^H(y)$. This step is accomplished in $O(\log n)$ time with $O(n)$ processors by using pointer jumping: for y the parent of x, if $\lambda^H(x) \subseteq \lambda^H(y)$ or $\lambda^H(y) \subseteq \lambda^H(x)$, we reassign the parent of x to be the parent of y.
Step 2: We set all existing vertices to be clique vertices; since (T^H, χ^H) was a tree-decomposition of H of width k, it is clear that condition 2 has been satisfied.
Step 3: Between each pair of clique vertices x and y, where x is the parent and y the child, we insert a constant number of new vertices in a path as follows. Let $S = \lambda^H(x) \cap \lambda^H(y)$ and let $p_1, \ldots, p_j$ be the nodes in $\lambda^H(y) - S$. Because H is connected, S is nonempty. As well, due to Step 1, S is strictly smaller than both $\lambda^H(x)$ and $\lambda^H(y)$. The first vertex on the path, the child of x, is a separator vertex with label S. The next two vertices are a clique vertex and then a separator vertex, each with label $S \cup \{p_1\}$, the following two a clique vertex and a separator vertex each with label $S \cup \{p_1, p_2\}$, and so on, until the path ends at y.

It is not difficult to see each such path can be created in constant time with one processor. Since there are $O(n)$ paths, this gives a total of $O(n)$ processors.

Step 4: We now ensure that condition 7 is satisfied. The condition is violated by a situation in which a separator vertex z is the parent of a clique vertex y, in turn the parent of separator vertex x, such that the unique node a in $\lambda^H(y)\backslash\lambda^H(z)$ is not contained in $\lambda^H(x)$. In this case, it is not difficult to see that $\lambda^H(x) \subseteq \lambda^H(z)$. We remove the edge between y and x, instead making x into a child of the parent of z.

Step 4 clearly takes constant parallel time with $O(n)$ processors.

Step 5: Condition 4 can be satisfied by examining all siblings to see if they share a label value, and if so, combining such siblings into a single vertex with all subtrees of the original siblings.

This step can be accomplished in $O(\log n)$ time with $O(n)$ processors using sorting [Col88].

Step 6: Finally, to satisfy condition 1, we add in a separator vertex as a root, choosing as its label an arbitrary subset of the label of the original root.

It is not difficult to see that each of the above steps can be completed in the stated running time. ∎

6.2 Breaking T^H into constant size pieces

In this section we give a brief description of a technique, originated by Jordan [Jor69] and Brent [Bre74] for breaking T^H into constant size pieces in logarithmic time. Full details of the technique, including proofs of all the lemmas, can be found in an earlier paper on the subject [GN92].

The technique consists of recursively dividing a tree into smaller subgraphs of two different types, namely unscarred and scarred subtrees (the reader is referred to Section 2.1 for the definition of a scarred subtree and related concepts). Lemma 6.2 concerns the subsequent division of an unscarred subtree and Lemma 6.3 the subsequent division of a scarred subtree; both are slight generalizations of results by Brent.

Lemma 6.2. *Let T be a tree with at least two nodes. Then there is a unique node v of T with children $c_0, \ldots, c_{k-1}$ such that:*

1. $|T\backslash T_v| \le \frac{|T|}{2}$, *(or equivalently* $|T_v| > \frac{|T|}{2}$*); and*
2. $|T_{c_i}| \le \frac{|T|}{2}$, *for all* $0 \le i \le k$.

Lemma 6.3. *Let T be a tree, $|T| > 2$, and ℓ be a leaf of T. Then there is a unique ancestor v of ℓ such that if c is the child of v for which $\ell \in T_c$ then:*

1. $|T\backslash T_v| \le \frac{|T|}{2}$, *(or equivalently* $|T_v| > \frac{|T|}{2}$*); and*
2. $|T_c| \le \frac{|T|}{2}$.

We obtain a division of the tree into subgraphs by starting with a tree T and recursively applying the two lemmas depending on whether or not a subgraph has a scar. In both Lemma 6.2 and Lemma 6.3, the node v is called the *Brent break* of T. In Lemma 6.3 we can view the leaf ℓ as a place holder for a scar. It is clear that in the division, each subgraph has at most one scar. In

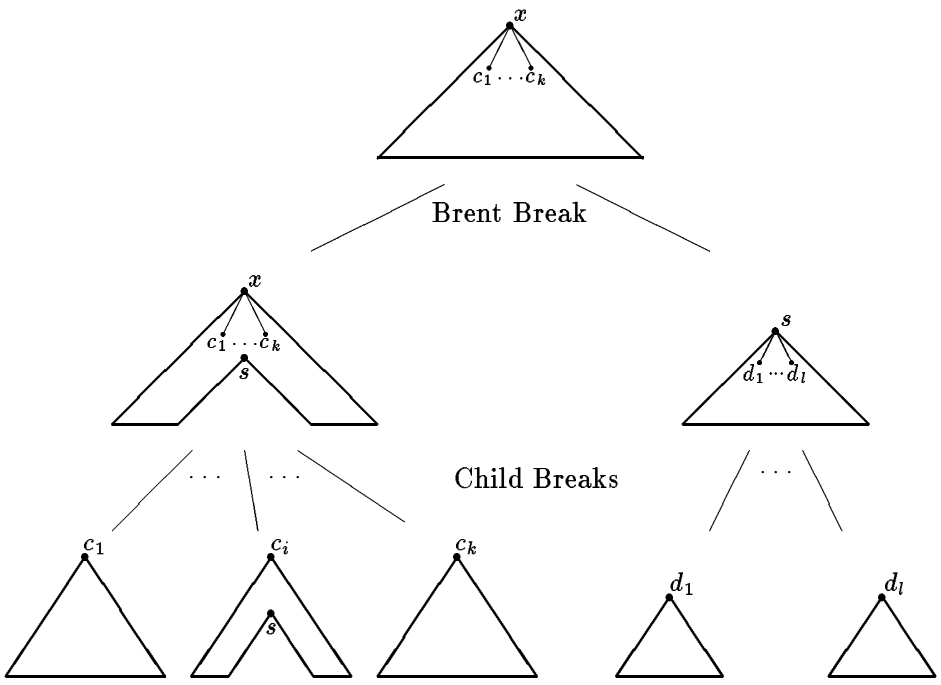


Figure 11: Part of a Brent tree

practice, we will view the applications of both Lemma 6.2 and Lemma 6.3 as two-step operations: first T is split into subgraphs $T \backslash T_v$ and T_v and then T_v is split into subtrees $T_{c_0}, \ldots, T_{c_{k-1}}$ where $c_0, \ldots, c_{k-1}$ are the children of v. The first step will be called a *Brent break* and the second a *child break*; we further distinguish between breaks that are *simple* (where, as in Lemma 6.2, none of the resulting subgraphs are scarred) or *scarred*, as in Lemma 6.3. The set of subgraphs obtained by the application of a Brent break and then a child break form disjoint sets of the nodes of the original subgraph. We use the term *Brent restructuring* to apply to one Brent break followed by one child break, resulting in disjoint subsets of the nodes. It is not difficult to see that $O(\log n)$ recursive applications of the lemmas will result in trees of constant size. For many problems, these can then be processed directly in constant time.

We can create a tree, the *Brent tree of* T, or $\mathcal{B}_T$, containing all the information about a sequence of Brent structuring steps on a tree. We will associate with each vertex a *level number*, where the root of the tree is at level 1 and the child of a vertex at level ℓ is at level $\ell + 1$. The root of the tree contains all of T, the children and grandchildren of the root contain the subgraphs obtained by a single application of Brent restructuring, and each subsequent pair of levels contains subgraphs obtained by a further application of Brent restructuring. A portion of a Brent tree is illustrated in Figure 11. The proof of the following result can be found in an earlier paper [GN92].

Lemma 6.4. *The total determination of $\mathcal{B}_T$ can be achieved in $O(\log |T|)$ time using $O(|T|^4)$ processors.*

6.3 Creation of $TDG(G)$

We present a parallel algorithm for the creation of $TDG(G)$. Where details, such as counting arguments, are identical to those appearing in the sequential algorithm, they are omitted from the following lemma. The reader is referred instead to the proof of Lemma 3.4 for such details. We begin with the k-connected case.

Lemma 6.5. *Let G be a k-connected partial k-tree with $n = |V(G)|$. Then $TDG(G)$, the tree-decomposition graph of G, can be created in time $O(\log n)$ using $O(n^{k+2})$ processors.*

Proof (outline): To create all the separator tdg-vertices in $TDG(G)$, we first identify all separators of size at most k in G, and for each separator S determine $\mathcal{C}_G(S)$. We can perform this step in $O(\log n)$ time by assigning $O(n)$ processors for the connected component problem arising from the removal of each of the $O(n^k)$ separators (using Shiloach and Vishkin's algorithm on partial k-trees, which each have a linear number of edges [SV82]), for a total of $O(n^{k+1})$ processors. At this point we have generated all possible values for the first component in a separator tdg-vertex.

Next, for each separator, we enumerate all possible orderings of the nodes in constant time. To determine all possibilities for the third component, recall that by Lemma 3.3 each partition consists of a single component. It suffices to specify which of the $O(n)$ components is the parent component, since the order of the children partitions is unimportant. This gives a total of $O(n^{k+1})$ different separator tdg-vertices which can be created in $O(\log n)$ time with $O(n^{k+1})$ processors.

To count the number of clique tdg-vertices, we consider the number of possible choices for *Clique*, orderings, and for each of the constant number of subsets, the choice of components. Using counting argument similar to the one given above gives a total of $O(n^{k+2})$ different tdg-vertices.

Finally, we must determine the memberships in all clusters. For a particular clique tdg-vertex, a separator tdg-vertex in its jth cluster must consist of a particular node set and a particular ordering on that set. The choice of parent component is fixed by the clique tdg-vertex. Since the number of clusters is constant for a clique tdg-vertex, this can be performed in constant time with $O(1)$ processors per clique tdg-vertex.

For a particular separator tdg-vertex, there is a cluster for each child partition. Within the cluster, there are O(size of the partition) different ways of extending the separator label to form a clique label. Since the partitions are disjoint this gives a total of $O(n)$ different children in all the clusters. By assigning $O(n)$ processors to each separator tdg-vertex the clusters can all be established in constant time. ∎

By applying similar techniques we can obtain a corresponding lemma for the bounded degree case.

Lemma 6.6. *Let G be a degree d partial k-tree for some positive d. Then $TDG(G)$, the tree-decomposition graph of G, can be created in time $O(\log n)$ using $O(n^{k+2})$ processors.*

6.4 Modification of the tdg-mapping

We give details of how the tdg-mapping is modified for subgraph isomorphism for k-connected partial k-trees. The modifications for the bounded degree case and for topological embedding are similar.

In the sequential algorithm, we worked up T^H from the leaves to the root, at each point keeping track of mapping information between all tdg-vertices and a particular vertex in T^H. More specifically, the information for a tdg-vertex α and a vertex x of T^H indicated under which conditions $G_{(\alpha)}$ could be mapped to to $H_{\{T_x\}}$ (in other words, when α was tdg-isomorphic to x).

In the parallel algorithm, we work up the Brent tree of T^H, $\mathcal{B}_T$, keeping track of mapping information between tdg-vertices α and vertices X of the Brent tree. If the subgraph of H induced by X is unscarred, then the mapping can be computed as in the sequential case. If on the other hand the subgraph is scarred, then we compute for all possible scars in G the mapping from the scarred subgraph of G induced by α and the scar to X, with scars mapping to scars in the appropriate order. When a mapping is obtained from α with scar β to x with scar y, we say that (α, β) is *tdg-scarred-isomorphic* to (x, y). A similar concept can be defined for topological embedding.

6.5 Modification of the algorithms

We begin by describing a parallel algorithm for determining the subgraph isomorphism problem for k-connected partial k-trees. Subsequently we list the corresponding results for the other three problems under consideration.

Theorem 6.7. *For $k > 0$, let H be a partial k-tree and let G be a k-connected partial k-tree. Let $n = |V(G)| + |V(H)|$. Then there is an $O(\log^3 n)$ time, $O(n^{max\{3k+4, 2k+7.5\}})$ processor randomized CRCW PRAM algorithm to determine whether or not G is isomorphic to a subgraph of H.*

Proof (Outline):

We begin by creating a normalized tree-decomposition (T^H, χ^H) of H, then a Brent tree $\mathcal{B}_T$ of T, and finally a tree-decomposition graph of G.

With each Brent vertex X of $\mathcal{B}_T$, the Brent tree of a normalized tree-decomposition (T^H, χ^H) of H, we associate an array BB_X. For X an unscarred tree with root labelled x, the array BB_X is equivalent to B_x in the sequential case. If X is a scarred tree, say with root labelled x and scar labelled y, BB_X is a $|TDG(G)| \times |TDG(G)|$ bit matrix with entry $BB_X[\alpha, \beta] = 1$ if and only if (α, β) is tdg-scarred-isomorphic to (x, y).

We process the vertices of the Brent tree level by level, computing BB_X for each Brent vertex X. We must show how information at one level of the Brent tree can be combined to determine the array entries at the next level of the Brent tree. The transition from a vertex X in $\mathcal{B}_T$ to its children represents either a child break or a Brent break; we consider each case in turn.

If the level from X to its children represents a child break, then either zero or one child of X represents a scarred subgraph of T^H. If no child is scarred, this step is similar to that in Theorem 4.1.

Now suppose that the level from X to its children represents a child break, X is labelled by a tree $T_x \backslash T_z$, and its children $Y_1, \ldots, Y_r$ are labelled with trees $T_{y_1} \backslash T_z$, $T_{y_2}, \ldots, T_{y_r}$ where $y_1, \ldots, y_r$ are the children of x in T and z is a descendant of y_1. We can assume that all entries in $BB_{Y_1}, \ldots, BB_{Y_r}$ have been computed. For each child γ of a tdg-vertex α and each tdg-vertex β in a child partition of γ, we wish to determine whether or not $BB_X[\alpha, \beta] = 1$. To do so, we set up a matching similar to that in the proof of Theorem 4.1, except that we place the additional restriction that $BB_{Y_1}(\gamma, \beta) = 1$.

We now consider the case in which the level from X to its children represents a Brent break. Let the two children of X, Y_1 and Y_2, represent $T_x^H \backslash T_y^H$ and $T_y^H \backslash T_z^H$, (or T_y^H if it is unscarred) for x, y, and z vertices in T^H. Assuming that both children of X represent scarred trees, $BB_X[\alpha, \beta]$ will be 1 if there is a γ such that $BB_{Y_1}[\alpha, \gamma] = 1$ and $BB_{Y_2}[\gamma, \beta] = 1$ with γ a descendant of α and β a descendant of γ.

To determine the running time of the algorithm, we recall that we can form a tree-decomposition of H in time $O(\log^2 n)$ using n^{2k+5} processors, normalize the tree-decomposition in time $O(\log n)$ using $O(n)$ processors (Lemma 6.1), create $\mathcal{B}_T$ in time $O(\log n)$ using $O(n^4)$ processors (Lemma 6.4) and create $TDG(G)$ in time $O(\log n)$ using $O(n^{k+2})$ processors (Lemma 6.5). To process the vertices of the Brent tree, we note that each array entry for each of the $O(\log n)$ levels of the tree can be processed in parallel. We consider the child and Brent breaks separately.

To process a child break, the time to compute a single entry of BB_X is dominated by the time to set up and perform the matching. Using a randomized algorithm, this can be accomplished in time $O(\log^2 n)$ with $O(n^{4.5})$ processors [MVV87]. Since there are $O(|TDG(G)|^2)$ entries in BB_X, and at most n Brent vertices at a single level of the Brent tree, all of which can be processed in parallel, the total number of processors needed is in $O(|TDG(G)|^2 \cdot n \cdot n^{4.5})$.

We now consider the case in which the level from X to its children represents a Brent break. Let the two children of X, Y_1 and Y_2, represent $T_x^H \backslash T_y^H$ and $T_y^H \backslash T_z^H$, (or T_y^H if it is unscarred) for x, y, and z vertices in T^H. Assuming that both children of X represent scarred trees, $BB_X[\alpha, \beta]$ will be 1 if there is a γ such that $BB_{Y_1}[\alpha, \gamma] = 1$ and $BB_{Y_2}[\gamma, \beta] = 1$ with γ a descendant of α and β a descendant of γ. The value of $BB_X[\alpha, \beta]$ can be determined in constant time using $O(|TDG(G)|)$ processors, one for each possible value of γ. For all array entries for all vertices at the same level in the Brent tree, the Brent breaks can be processed in constant time using $O(n^{3k+4})$ processors.

From the discussion above, the total time to process the entire Brent tree is $O(\log^3 n)$ with $O(n^{max\{3k+4, 2k+7.5\}})$ processors. ∎

When G is a bounded degree partial k-tree, the algorithm differs mainly in the type of matching problem needed to process a child break. As in the sequential algorithms, a constant time algorithm can be used. In this case, the creation of the tree-decomposition of H will dominate the time and processor count.

Theorem 6.8. *For $k > 0$, let H be a partial k-tree and let G be a bounded degree partial k-tree. Let $n = |V(G)| + |V(H)|$. Then there is an $O(\log n)$ time, $O(n^{3k+4})$ processor CRCW PRAM algorithm to determine whether or not G is isomorphic to a subgraph of H.*

For the problem of topological embedding, when G is of bounded degree, we are able to obtain an algorithm of the same complexity.

Theorem 6.9. *For $k > 0$, let H be a partial k-tree and let G be a bounded degree partial k-tree. Let $n = |V(G)| + |V(H)|$. Then there is an $O(\log n)$ time, $O(n^{3k+4})$ processor CRCW PRAM algorithm to determine whether or not G can be topologically embedded in H.*

When G is k-connected, a larger processor count is required.

Theorem 6.10. *For $k > 0$, let H be a partial k-tree and let G be a k-connected partial k-tree. Let $n = |V(G)| + |V(H)|$. Then there is an $O(\log^3 n)$ time, $O(n^{k^2+4k+9.5})$ processor randomized CRCW PRAM algorithm to determine whether or not G can be topologically embedded in H.*

Proof (Outline):
The general outline of the proof is similar to that of Theorem 6.7, with technical details associated with topological embedding as in Theorem 4.1. We omit the points common to one theorem or the other, in the interest of emphasizing the differences. In particular, we focus on the fact that instead of processing a $|TDG(G)| \times |TDG(G)|$ bit matrix BB_X for each vertex X of the Brent tree, we now must consider all possible entries of the form $BB_X[(\alpha, P^\alpha, f^\alpha, g^\alpha, S^\alpha, PD^\alpha), (\beta, P^\beta, f^\beta, g^\beta, S^\beta, PD^\beta)]$ where P, f, g, S, PD are similar to those in the proof of Theorem 4.1.

It is not difficult to see that the total number of array entries for a single vertex X is in $O(|TDG(G)|^2)$. We set up one matching problem for each of the consistency checks outlined in the proof of Theorem 4.1. The running time and the processor count are dominated by the processing of the child breaks. The total number of matchings set up to compute one entry of BB_X is in $O(n^{k^2+2k+2})$. Since there are $O(|TDG(G)|^2)$ different array entries for any one Brent vertex, $O(n)$ Brent vertices at one level of the Brent tree, and $O(n^{4.5})$ processors required to set up and perform one matching, all consistency checks for all matchings for all nodes at one level of the Brent tree can be processed in parallel using $O(n^{k^2+4k+9.5})$ processors. Since the time to process one level is dominated by the time needed to perform a single matching, the total amount of time to process all $O(logn)$ levels of the Brent tree is in $O(\log^3 n)$, as claimed. ■

7 Conclusions and open problems

We have presented algorithms for subgraph isomorphism and topological embedding on two classes of partial k-trees. Since these problems are NP-complete for partial k-trees, it is evident that the result cannot be fully generalized. However, it is possible that some degree of generalization might be possible; this is left as an open question. Moreover, although the results in this paper delineate the boundary between the tractable and the intractable with respect to these embedding problems for classes of partial k-trees, there is still the need for the delineation of such a boundary for other classes of graphs with respect to these problems.

Subgraph isomorphism is only one of many problems known to have a polynomial time solution for trees but to be NP-complete for general graphs. It might be possible to attempt to apply to other problems of this type the methods developed in this paper for adapting tree techniques to techniques for general graphs.

The search for polynomial time algorithms for classes of partial k-trees is itself a topic of interest. It would be instructive to determine for what other problems a bound on degree or a restriction to k connectedness would yield a more efficient algorithm than in the general case.

References

[ACP89] S. Arnborg, D. Corneil, and A. Proskurowski, "Complexity of finding embeddings in a k-tree," *SIAM Journal of Algebraic and Discrete Methods* **8**, pp. 277-284, 1987.

[ALS91] S. Arnborg, J. Lagergren, and D. Seese, "Problems easy for tree-decomposable graphs," *Journal of Algorithms* **12**, 2, pp. 308ff, 1991.

[Bod88a] H. Bodlaender, "Dynamic programming on graphs with bounded treewidth," *Proceedings of the 15th International Colloquium on Automata, Languages and Programming*, pp. 105–118.

[Bod93] H. Bodlaender, "A linear time algorithm for finding tree-decompositions of small treewidth," *Proceedings of the 25th Annual ACM Symposium on the Theory of Computing*, pp. 226–234, 1993.

[Bod88b] H. Bodlaender, "NC-algorithms for graphs with bounded tree-width," Technical Report RUU-CS-88-4, University of Utrecht, 1988.

[Bod90] H. Bodlaender, "Polynomial Algorithms for Graph Isomorphism and Chromatic Index on Partial k-trees," *Journal of Algorithms* **11**, pp. 631–643, 1990.

[BM76] J. Bondy, and U.S.R. Murty,*Graph Theory with Applications*, North-Holland, 1976.

[Bre74] R. Brent, "The parallel evaluation of general arithmetic expressions," *Journal of the ACM* **21**, 2, pp. 201–206, 1974.

[CH88] N. Chandrasehkaran and S.T. Hedetniemi, "Fast parallel algorithms for tree decomposition and parsing partial k-trees," *Proceedings 26th Annual Allerton Conference on Communication, Control, and Computing*, 1988.

[Col88] R. Cole, "Parallel merge sort," *SIAM Journal on Computing* **17**, pp. 770–785, 1988.

[GJ79] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*, Freeman, San Francisco, 1979.

[GKMS90] P. Gibbons, R. Karp, G. Miller, and D. Soroker, "Subtree isomorphism is in random NC," *Discrete Applied Mathematics* **29**, pp. 35–62, 1990.

[GN92] A. Gupta and N. Nishimura, "The parallel complexity of tree embedding problems," *to appear in Journal of Algorithms.* A preliminary version has appeared in *Proceedings of the Ninth Annual Symposium on Theoretical Aspects of Computer Science*, pp. 21–32, 1992.

[Jor69] C. Jordan, "Sur les assemblages de lignes," *Journal Reine Angew. Math.* **70**, pp.185–190, 1869.

[Lag90] J. Lagergren, "Efficient parallel algorithms for tree-decompositions and related problems," *Proceedings of the 31st Annual IEEE Symposium on the Foundations of Computer Science*, pp. 173–181, 1990.

[Lin89] A. Lingas, "Subgraph isomorphism for biconnected outerplanar graphs in cubic time," *Theoretical Computer Science* **63**, pp. 295–302, 1989.

[LK89] A. Lingas and M. Karpinski, "Subtree isomorphism is NC reducible to bipartite perfect matching," *Information Processing Letters* **30** pp. 27–32, 1989.

[LS88] A. Lingas and M. M. Syslo, "A polynomial-time algorithm for subgraph isomorphism of two-connected series parallel graphs," *Proceedings of the 15th International Colloquium on Automata, Languages, and Programming* pp. 394–409, 1988.

[MT92] J. Matoušek and R. Thomas, "On the complexity of finding iso- and other morphisms for partial k-trees," *Discrete Mathematics* **108**, pp. 343–364, 1992.

[Mat78] D. Matula, "Subtree isomorphism in $O(n^{5/2})$," *Annals of Discrete Mathematics* **2**, pp. 91–106, North-Holland, 1978.

[MVV87] K. Mulmuley, U. Vazirani, and V. Vazirani, "Matching is as easy as matrix inversion," *Proceedings of the 19th Annual ACM Symposium on the Theory of Computing*, pp. 345–354, 1987.

[Ree92] B. Reed, "Finding approximate separators and computing tree width quickly," *Proceedings of the 24th ACM Symposium on the Theory of Computing*, pp. 221–228, 1992.

[Ros74] D.J. Rose, "On simple characterization of k-trees," *Discrete Mathematics* **7**, pp. 317–322, 1974.

[RS84] N. Robertson and P. Seymour, "Graph Minors III. Planar tree-width," *Journal of Combinatorial Theory (Ser. B)* **36**, pp. 49–64, 1984.

[RS86] N. Robertson and P. Seymour, "Graph Minors II. Algorithm aspects of tree-width," *Journal of Algorithms* **7**, pp. 309–322, 1986.

[SV82] Y. Shiloach and U. Vishkin, "An $O(log n)$ parallel connectivity algorithm," *Journal of Algorithms* **3**, pp. 57–67, 1982.

[Sys82] M. M. Syslo, "The subgraph isomorphism problem for outerplanar graphs," *Theoretical Computer Science* **17**, pp. 91–97, 1982.