International Journal of Foundations of Computer Science
Vol. 24, No. 1 (2013) 67–93

DOI: 10.1142/S0129054113400054

A UNIFIED FRAMEWORK FOR BI(TRI)CONNECTIVITY AND CHORDAL AUGMENTATION*

N. S. NARAYANASWAMY† and N. SADAGOPAN‡

Department of Computer Science and Engineering, Indian Institute of Technology Chennai-600036, India
†*swamy@cse.iitm.ac.in*
‡*sadagopu@cse.iitm.ac.in*

Received 1 April 2011
Accepted 8 May 2012
Communicated by Taso Viglas

For a connected non-complete graph, a *vertex separator* is a subset of vertices whose deletion increases the number of connected components and the *vertex connectivity* of the graph refers to the size of a *minimum vertex separator*. A graph with the vertex connectivity k is said to be *k-vertex connected*. Given a k-vertex connected graph G, vertex connectivity augmentation determines a smallest set of edges whose augmentation to G makes it $(k+1)$-vertex connected. In this paper, we report our study on connectivity augmentation in 1-connected graphs, 2-connected graphs, and k-connected chordal graphs. We first represent the graph under consideration using a "tree-like" graph. This tree is unique and explicitly captures the connectivity information of the graph. Using this tree, our proposed data structure maintains the set of equivalence classes based on an equivalence relation on the set of leaves of the tree. This partition determines a set of edges to be augmented to increase the connectivity of the graph by one. Based on our data structure, we present a new combinatorial analysis and an elegant proof of correctness of our linear-time algorithm for an optimum connectivity augmentation. The novelty is in the data structure which is a unified framework for all three augmentations. As far as the run-time analysis is concerned, given the associated tree, our approach yields an augmentation set in linear time.

Keywords: Biconnectivity augmentation; triconnectivity augmentation; chordal augmentation.

1. Introduction

Connectivity augmentation in graphs is a classical topic of combinatorial optimization. Vertex connectivity augmentation of a graph adds the smallest set of edges to reach a given vertex connectivity. A special case of this study is to increase the connectivity by one. i.e., given a k-vertex connected graph G, find a minimum

*Preliminary version of this paper appeared in 17*th Computing: Australasian Theory Symposium* (*CATS*) 2011.

number of edges to be augmented to G to increase its connectivity to $k+1$. This was an open problem for the past three decades, only recently, it was shown by Vegh in [1] that it is polynomial-time solvable. Although, the complexity of this problem was open for several years, special cases of this problem have attracted many researchers. In particular, Eswaran and Tarjan [2] studied the biconnectivity augmentation which asks for identifying a minimum set of edges whose augmentation makes the given graph 2-vertex connected. The following results are presented in [2] for biconnectivity augmentation. From a connected graph G, a *block tree* T is constructed in linear time. A block tree is a tree such that each node is one of the following: a cut-vertex, a cut-edge, and a maximal 2-connected subgraph in G. Using T, Eswaran and Tarjan [2] presented the lower bound on the optimum biconnectivity augmentation number of G which is $\max\{\lceil \frac{l}{2} \rceil, \Delta_c(T) - 1\}$, where l is the number of leaves in T and $\Delta_c(T)$ denotes the maximum degree among all cut vertices of G. In [2], a constructive proof of tightness of the lower bound is also mentioned. Rosenthal and Goldner [3] developed an $O(n+m)$ linear-time sequential algorithm for biconnectivity augmentation based on the combinatorial analysis given in [2], where n is the number of vertices and m is the number of edges in the graph. Algorithm given in [3] involves three stages: stage-1 augments a set of edges such that the resulting graph is connected, the graph resulting from stage-2 is such that the degree of each vertex is at most half the number of leaves of the block tree and the graph obtained from stage-3 is biconnected. Later, Hsu and Ramachandran [4] discovered an error in stage-3 of the algorithm reported in [3] and they presented the corrected version in [4]. Subsequently, they proved that the bound mentioned in [2] is tight. In [4], a parallel algorithm for biconnectivity augmentation was also presented. Recently, Hsu [5] presented a simple sequential algorithm and an improved parallel algorithm for biconnectivity augmentation and these improvements are over the algorithm presented in [4]. It is important to note that the approach in [4], is an iterative algorithm which determines a set of edges to be augmented by recomputing the associated tree. As a result, the idea of [4] yields a not so elegant combinatorial analysis and makes the implementation a challenging task. The idea in [5], though avoids recomputation of the associated tree, the underlying combinatorial analysis is quite involved. It may be noted that, despite the efforts of [3–5], a simple linear-time algorithm without recomputing the associated tree is still open. This is the first motivation for our work.

Another fundamental graph theoretic problem in this line is triconnectivity augmentation which determines a minimum number of edges to be augmented to a biconnected graph to make it 3-vertex connected. This study was initiated by Watanabe and Nakamura in [6], where similar to biconnectivity augmentation, using the structural understanding of a biconnected graph, a 3-*block tree* is constructed. A 3-block tree is a tree in which each node is one of the following: a maximal 3-connected component, a 2-size vertex separator, and a polygon. The lower bound on the triconnectivity augmentation number involving the leaves and the maximum degree of a 3-block tree is presented in [6]. This lower bound is similar to the one

proposed in [2] for the biconnectivity augmentation. Watanabe and Nakamura [6] gave an $O(n(n+m)^2)$ time two stage sequential algorithm for triconnectivity augmentation. Later, Hsu and Ramachandran [7] developed a linear-time algorithm for this problem. In both the attempts, the associated 3-block tree is recomputed at each iteration of the algorithm, leading to a complicated combinatorial analysis. This calls for a simple triconnectivity augmentation algorithm avoiding the recomputation of the associated tree with elegant combinatorial analysis. This is the second motivation for our work.

Having seen the existence and importance of the *associated* trees for optimum biconnectivity and triconnectivity augmentations, a natural question is to construct a *tree-like* structure for every k-connected graph and design a lower bound similar to a 3-block tree. It turns out that beyond 2-connected graphs, the notion of *tree-like* structure in general is not clear. For example, even for 3-connected graphs the existence of such a tree is not clear as described in [8]. Hsu [8] presented a different lower bound by understanding the structure of vertex separators in 3-connected graphs and an augmentation algorithm for 3-connected graphs. The algorithm in [8] precisely augments the number which equals the new lower bound and hence the new lower bound is tight for 4-connectivity augmentation of 3-connected graphs. In general, the combinatorial analysis and the complexity of the vertex connectivity augmentation for an arbitrary k-connected graph was one of the most challenging open questions of this area. In other words, whether the decision version of the problem, is in P or NP-complete was open for a long time in the literature. Only recently, Vegh in [1] presented a polynomial-time algorithm for an optimum $(k+1)$-connectivity augmentation of k-connected graphs. Analogous to vertex connectivity augmentation, the general k-edge connectivity augmentation problem allowing multiple edges between a pair of vertices was solved by Watanabe and Nakamura [9]. Interestingly, these problems are also of practical interest and are well motivated from the study of network reliability and fault-tolerant computing [10].

1.1. *Our work*

Our goal is to explore the possibility of identifying a framework using which optimum connectivity augmentation can be done in several special graph classes. The two important issues to be considered in designing such a framework are: it must avoid the recomputation of the associated tree and it must yield an elegant algorithm with simple combinatorial analysis. It is important to identify such a framework which might give an handle on the existence of the associated trees for k-connected graphs in general. It is interesting and tempting to analyze this observation which might yield an improved algorithm for connectivity augmentation in k-connected graphs over the algorithm reported in [1].

Our Contributions:

- Given a connected undirected unweighted graph, our framework first constructs the associated tree by understanding the structural decomposition of a graph

with respect to its minimum vertex separators. For example, given a 1-connected graph G, we construct the associated biconnected component tree where each node is either a cut vertex or a biconnected component of G. The highlights of this tree are it is unique and it captures all minimum vertex separators in the given graph. In all three augmentations reported in this paper, we focus our combinatorial analysis on the associated tree to obtain an optimum connectivity augmentation set. As a first step, we propose an equivalence relation on the set of leaves of the tree. The set of equivalence classes is our data structure, which is precisely a partition of the set of leaves of the associated tree. This partition determines the edge to be augmented to the graph at each iteration. Moreover, this partition guarantees a recursive sub-problem and it is sufficient to maintain this partition for finding an optimum connectivity augmentation set in graphs. To the best of our knowledge, such a data structure has not been maintained to solve augmentation problems, and we believe that this is the main contribution of this paper.

- The three applications of this data structure reported in this paper are biconnectivity augmentation of 1-connected graphs using biconnected component trees, triconnectivity augmentation of 2-connected graphs using 3-block trees, and $(k+1)$-connectivity augmentation of k-connected chordal graphs using clique separator trees. For each of the augmentations reported here, we discuss the construction of the associated tree, followed by the lower bound results for optimum connectivity augmentation. Based on our framework we provide a new proof of tightness and an elegant linear-time algorithm for augmentation.
- It is important to compare our work with the results of [2, 4]. Given a 1-connected graph G, in [2, 4] a *block tree* T is constructed from G. It is also mentioned that an optimum biconnectivity augmentation for G can be obtained from T. An iterative algorithm in [4] identifies the edge to be added between a pair of leaves in T. Once an edge is added into T, it again computes the block tree. Since the addition of an edge creates a biconnected component, to get the tree after each iteration, the algorithm in [4] recomputes the block tree by dynamically maintaining the list of cut-vertices and the set of biconnected components. Recomputing the block tree and maintaining the additional information about cut-vertices makes the implementation of this algorithm a challenging task. Though [5] avoids the recomputation of the tree, it does demand a rooted normalized tree. Moreover, the underlying combinatorial analysis is quite involved. In contrast, our algorithm with the help of our new data structure avoids recomputation of the tree. This new approach yields a simple algorithm for finding an optimum biconnectivity augmenting set.
- It may be noted that, this paper is the first attempt in studying augmentation in k-connected chordal graphs, for any k. This result is important because k-connected chordal graphs contain a well-known subclass, namely, k-trees which are a generalization of trees. Note that trees are 1-trees. It is natural to look at connectivity augmentation in k-trees and its super classes. In particular, our

methods naturally extend to connectivity augmentation in k-connected chordal graphs, and this does not seem to be easily feasible using the approach of Tarjan *et al.* [2] and Hsu *et al.* [4]. We also present a new algorithm for triconnectivity augmentation of biconnected graphs using our new data structure. Our approach for triconnectivity augmentation does not involve recomputation of the 3-block tree, and is fundamentally different from the results reported in [6, 7].

Outline of the Paper: We first introduce graph-theoretic terminologies and notation used in this paper. In Section 2, we introduce our framework and discuss optimum biconnectivity augmentation in trees. We report three applications of our data structure, namely, connectivity augmentation in 1-connected graphs, 2-connected graphs, and k-connected chordal graphs in Sections 3.1, 3.2, and 3.3, respectively.

Graph-Theoretic Preliminaries: We follow standard graph-theoretic definitions and notation [11, 12]. Let $G = (V, E)$ be an undirected unweighted graph where $V(G)$ is the set of vertices and $E(G) \subseteq \{\{u, v\} \mid u, v \in V(G),\ u \neq v\}$ is the set of edges. The *neighborhood* of a vertex v in G is the set $N_G(v) = \{u \mid \{u, v\} \in E(G)\}$. The degree of a vertex v, denoted as $deg_G(v) = |N_G(v)|$. $\delta(G)$ and $\Delta(G)$ denote the minimum and maximum degree of vertices, respectively in G. A path P on the vertex set $V(P) = \{u = v_1, v_2, \ldots, v_n = v\}$ (where $n \geq 2$) has its edge set $E(P) = \{\{v_i, v_{i+1}\} \mid 1 \leq i \leq n-1\}$. Such a path is denoted by P_{uv}. For a connected non-complete graph G, a vertex separator $S \subseteq V(G)$ is such that the induced subgraph, denoted by $G \setminus S$, on the vertex set $V(G) \setminus S$ has more than one connected component. The vertex connectivity of G, written $\kappa(G)$, is the size of a minimum vertex separator. G is k-connected if $\kappa(G) = k$. For a k-connected graph G, a connectivity augmentation set is a smallest set of edges whose augmentation to G makes it $(k+1)$-connected. We use $E_{min}(G)$ to denote such a set of minimum cardinality.

2. Biconnectivity Augmentation in Trees: A New Approach

Given a tree, we present a new approach to find an optimum augmenting set which makes a tree biconnected. We first discuss the lower bound on the size of any optimum augmenting set, followed by a new proof of tightness. In this proof, we identify an equivalence relation on the set of leaves of the tree. Using the partition of the set of leaves, namely, the set of equivalence classes we describe an approach to find an optimum biconnectivity augmenting set. We also show that it is sufficient to maintain this partition at each iteration of the algorithm. This framework also guarantees a tree at each iteration and hence, we obtain a recursive sub-problem efficiently. It is now natural to extend our tree augmentation approach to augmentation in graphs which have tree like structures. Since trees are a subclass of 1-connected graphs, a natural extension is to study augmentation in 1-connected graphs by representing them using associated trees. The standard approach in the literature [2, 4] for biconnectivity augmentation of a 1-connected graph is to maintain a tree, namely,

a block tree that captures the biconnected components, cut vertices, and bridges of a 1-connected graph. After the choice of an edge to be added is made, a new tree is computed, and the procedure stops when the tree becomes a single node. We work with biconnected component trees which are similar to block trees, to find an augmenting set using our proposed framework. Our framework does not recompute these associated trees at each iteration and this new approach is fundamentally different from the results reported in [2, 4]. This new data structure and combinatorial analysis gave an insight into the study of connectivity augmentation in other graphs which have an associated tree like structures to represent their vertex connectivity information. In particular, our methods naturally extend to connectivity augmentation in k-connected chordal graphs, and this does not seem to be easily feasible using the approach of Tarjan *et al.* [2] and Hsu *et al.* [4]. We also present a new algorithm for triconnectivity augmentation of biconnected graphs using our data structure.

The Lower Bound on Biconnectivity Augmentation in Trees: Given a tree T, we now present the lower bound on the size of any optimum biconnectivity augmenting set. It is a well-known fact that in any 2-connected graph, for any pair of vertices, there exists two vertex disjoint paths between them. This fact is useful in determining the lower bound on the optimum biconnectivity augmentation number. Let l denote the number of leaves in T. Clearly, to biconnect T, we must augment at least $\lceil \frac{l}{2} \rceil$ edges. Another lower bound is due to the number of components created by removing a cut vertex of T. Note that the number of components created by removing a cut vertex x of T is precisely the degree of x in T. This shows that in any biconnectivity augmentation of T, for each cut vertex x, one must find at least $deg_T(x) - 1$ new edges in the augmenting set. Therefore, we must augment at least $\Delta(T) - 1$ edges to biconnect T. Therefore, by combining the two lower bounds, the number of edges to biconnect T is at least $\max\{\lceil \frac{l}{2} \rceil, \Delta(T) - 1\}$. This lower bound is indeed tight as shown in [2, 4].

In the next section, we present a new proof of tightness. In this proof, we identify an equivalence relation on the set of leaves of the tree, and show that adding edges among appropriately chosen leaf pairs naturally results in a recursive sub-problem in which the lower bound value is one less. The main contribution here is the identification of the equivalence relation which consequently guarantees an easy construction of the recursive sub-problem. The equivalence relation and therefore the set of equivalence classes yield an elegant algorithm to compute $E_{min}(G)$, an optimum biconnectivity augmenting set. This approach is fundamentally different from the results presented in [4, 5]. We present an algorithm by focusing our combinatorial arguments on the set of equivalence classes. We further prove that the number of edges augmented by our algorithm is precisely the lower bound mentioned in this section.

2.1. *An equivalence relation for connectivity augmentation*

We now define a relation R on the set $L(T) = \{x_1, x_2, \ldots, x_l\}$ of leaves in T. Leaf x is related to leaf y, $y \neq x$ if there exists at most one vertex of degree at least 3 in P_{xy} and it is written as xRy. If x is not related to y in R then it is written as $x\tilde{R}y$. In other words, the path P_{xy} contains at least two vertices z and z' such that $deg(z) \geq 3$ and $deg(z') \geq 3$.

Lemma 1. *R is an equivalence relation.*

Proof. Clearly, the path P_{xx} contains no vertex of degree at least 3 and hence xRx. Moreover, this is true for all $x \in L(T)$. This implies that R is reflexive. Also if xRy then it follows yRx as T is an undirected tree. Clearly, R is symmetric. To prove that R is transitive, for $x, y, w \in L(T)$, if xRy and yRw, we need to prove that xRw. Suppose $x\tilde{R}w$. This implies that the path P_{xw} contains at least two vertices z and z' such that $deg(z) \geq 3$ and $deg(z') \geq 3$. Since $x, y, w \in L(T)$, it follows that either the path P_{xy} or the path P_{yw} contains at least two vertices of degree at least 3. This implies that either $x\tilde{R}y$ or $y\tilde{R}w$. However, this is a contradiction to the given fact that xRy and yRw. Hence our assumption that $x\tilde{R}w$ is wrong. What follows is that the path P_{xw} contains at most one vertex z such that $deg(z) \geq 3$. Therefore, xRw. Hence R is transitive. Therefore, R is an equivalence relation. □

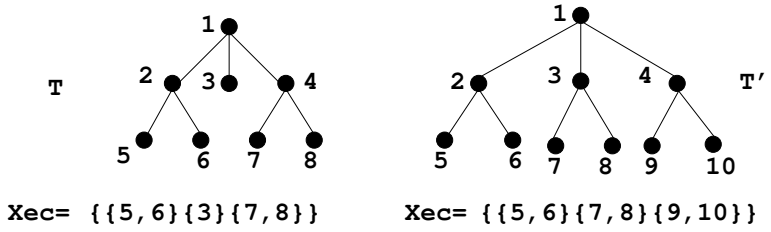


Fig. 1. Examples illustrating equivalence classes.

Since R is an equivalence relation, R induces a set X_{ec} of equivalence classes on the set $L(T)$ of leaves in T. An example is illustrated in Fig. 1. The following fact highlights a structural property of each equivalence class in X_{ec}.

Fact: *Each equivalence class is associated with a unique vertex* (*representative*) *of degree at least* 3 *in* T.

By the definition of our relation, leaf x and leaf y are related if there exists at most one vertex z of degree at least 3 in P_{xy}. This implies that z is the first vertex of degree at least 3 in P_{xy} and every other vertex in P_{xz} and P_{yz} is of degree at most 2 in T. Since R partitions the set of leaves into the set of equivalence classes,

we observe that, for each equivalence class $X \in X_{ec}$ there is an associated unique vertex, denoted by $w(X)$ such that $w(X)$ is the nearest vertex of degree at least 3 on the path from each element of X. We refer to $w(X)$ as the representative associated with X.

Before we present our algorithm we highlight one more fact which describes a special tree. If the tree T is such that T has exactly one vertex of degree at least 3 then by the definition of R, we get exactly one equivalence class containing all the leaves in T. The tree in this case is a star like tree. We call such a special tree as *star*.

A Generic Approach to Augmentation Algorithm: To decide upon the edge to be augmented at each iteration, we perform the following: from the set of equivalence classes, we identify two equivalence classes X and Y such that degree of its representatives are maximum and second maximum, add the edge $\{u, v\}$, $u \in X$ and $v \in Y$ and update the set of equivalence classes. The tail condition of this procedure is when there is exactly one equivalence class and we know from our earlier discussion that there is exactly one special tree namely *star* and augmentation for *star* is done separately. An example illustrating the trace of Algorithm 1 is also given.

2.2. *Augmentation using equivalence classes*

In this section, we present our linear-time algorithm to find optimum biconnectivity augmentation sets in trees. Let $X_{ec} = \{X_1, \ldots, X_r\}$ denote the set of equivalence classes with $w(X_i)$ being the associated representative of $X_i \in X_{ec}$. Let $M_0 = \max\{\lceil \frac{l}{2} \rceil, \Delta(T) - 1\}$. We use $\Delta(T)$ and Δ interchangeably and the tree to which it is associated will be clear from the context. Let Δ^i denote the value of $\Delta(T)$ at the end of i-th iteration of the algorithm. Similarly, l_i denotes the number of leaves in the tree at the end of the i-th iteration. At the end of i-th iteration of the algorithm, let $M_i = \max\{\lceil \frac{l_i}{2} \rceil, \Delta^i - 1\}$. The following key lemma is presented in [4] and this key observation leads to an optimum algorithm reported in [4]. We present our key observation in Lemma 3 and this observation yields an elegant combinatorial analysis and an algorithm for finding an optimum connectivity augmentation set.

Lemma 2. [4] *Let T be a tree with $l \geq 3$. If $\Delta(T) > \lceil \frac{l}{2} \rceil$, then there exists at most two cut-vertices $v_1, v_2 \in V(T)$ whose degree is $\Delta(T)$.*

The following result is a key observation and is used in the proof of Lemma 4.

Lemma 3. *Let T be a tree with $l \geq 3$ and z be a vertex in $V(T)$. If $deg(z) = \Delta > \lceil \frac{l}{2} \rceil$ then there exists an equivalence class $X \in X_{ec}$ such that $w(X) = z$.*

Proof. Suppose there does not exist X such that $w(X) = z$. Then there are at least two leaves in each of the trees in the forest obtained by removing z. Also, by our hypothesis each component is neither an isolated vertex nor a path in $T \setminus \{z\}$.

This implies that each component is a tree with at least two leaves and each of these leaves, except the $N_T(z)$ is indeed a leaf in T. Since $deg(z) = \Delta > \lceil \frac{l}{2} \rceil$, it follows that the number of leaves is at least $2\Delta > l$. A contradiction. Therefore, there exists an equivalence class $X \in X_{ec}$ such that $w(X) = z$. □

Algorithm 1 Biconnectivity Augmentation in Trees: *tree-augment*(*Tree* T)

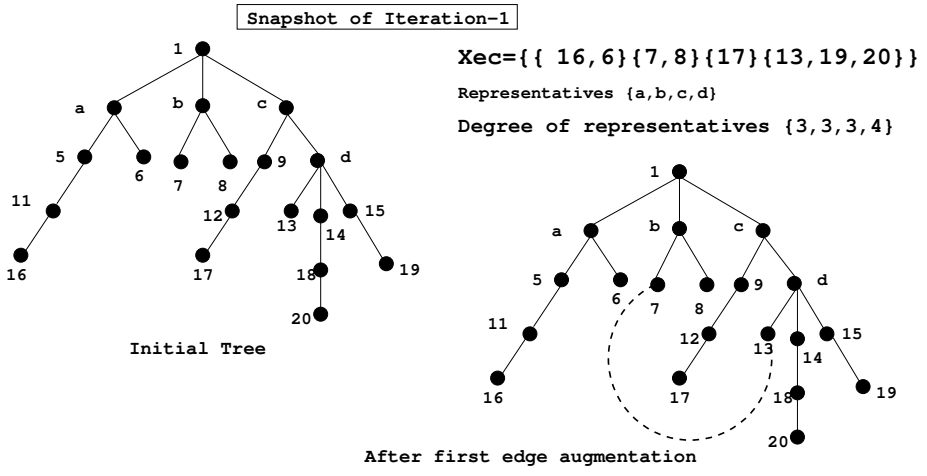
if there are exactly two leaves x and y **then**
  Add the edge $\{x, y\}$ and return the biconnected graph
else
  Compute the set X_{ec} of equivalence classes
  if $|X_{ec}| > 1$ **then**
    `/* T is not a star */`
    Y_{ec}=non-star-augment(X_{ec})
    star-augment(Y_{ec})
  else
    `/*`T `is a star */`
    star-augment(X_{ec})
  end if
end if

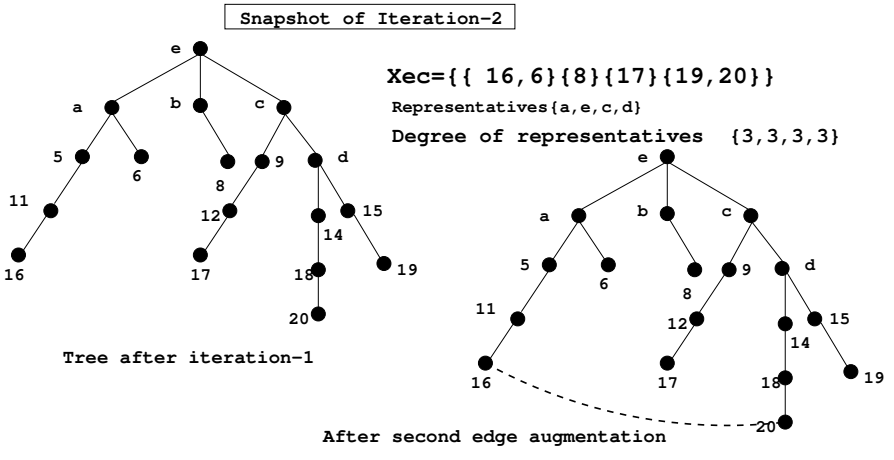
Algorithm 2 Biconnectivity Augmentation in stars: *star-augment*(*eclass-list* X_{ec})

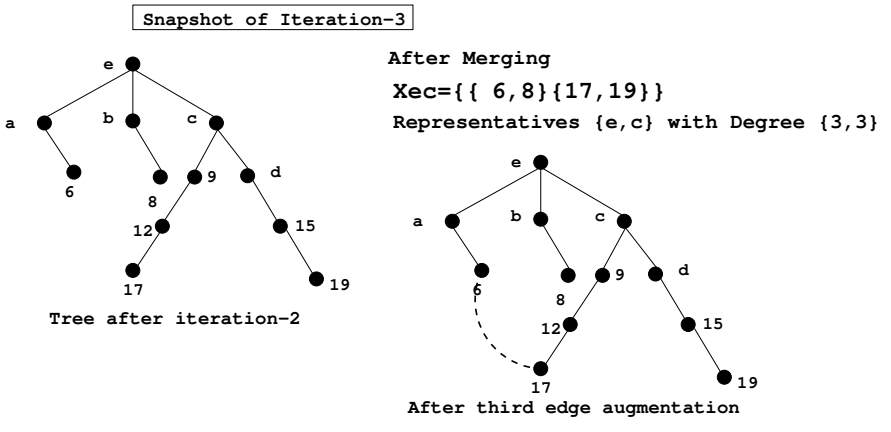
Let $X = \{x_1, \ldots, x_l\}$ be the set of leaves in T such that $X \in X_{ec}$
Add $|X| - 1$ edges to T, i.e., add $\{\{x_i, x_{i+1}\} \mid 1 \leq i \leq |X| - 1, x_i \in X\}$.

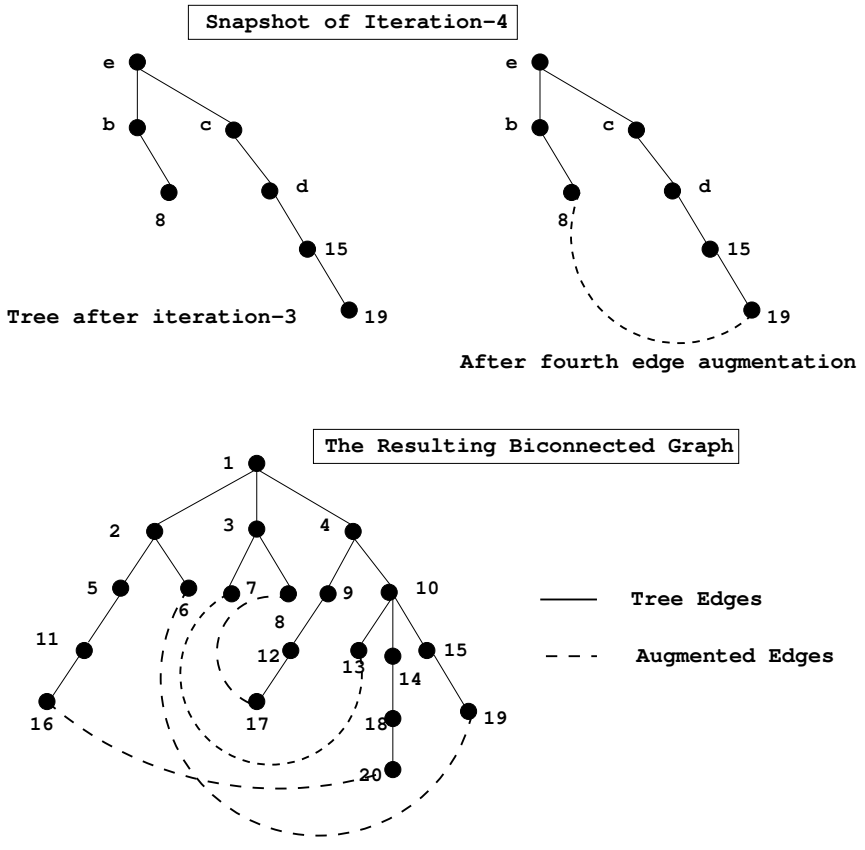
Algorithm 3 Biconnectivity Augmentation in non-star Trees: *non-star-augment*(*eclass-list* X_{ec})

1: **while** $|X_{ec}| \geq 2$ **do**
2: Let X_1 and X_2 are two equivalence classes in X_{ec} such that $deg(w(X_1)) \geq deg(w(X_2)) \geq deg(w(X_i)), i \geq 2$
3: Add the edge $\{x, y\}, x \in X_1, y \in X_2$. Remove x from X_1 and y from X_2.
4: Remove the path from x to $w(X_1)$ and y to $w(X_2)$ from T `/* This yields a tree after augmentation */`
5: Update X_{ec}
6: **end while**
7: **return** X_{ec} `/*`X_{ec} `has single equivalence class and the associated tree is a star */`









Lemma 4. *Let T be a tree such that T is not a star. Then $M_1 = M_0 - 1$.*

Proof. Given that T is a tree such that T is not a star implies that $|X_{ec}| \geq 2$. Since, the maximum degree does not increase from one iteration to the next, and the number of leaves strictly reduces, clearly, $M_1 \leq M_0$. We prove that $M_1 = M_0 - 1$ by contradiction. Suppose $M_1 \neq M_0 - 1$. This implies that $M_1 = M_0$. We know that $l_1 = l_0 - 2$. Since $M_1 = M_0$ and $l_1 = l_0 - 2$ it must be the case that $\Delta^1 - 1 > \lceil \frac{l_1}{2} \rceil$. We prove this observation by contradiction. Suppose, $\Delta^1 - 1 \leq \lceil \frac{l_1}{2} \rceil = \lceil \frac{l_0}{2} \rceil - 1$. Consequently, $M_1 = \max\{\Delta^1 - 1, \lceil \frac{l_1}{2} \rceil\} = \lceil \frac{l_0}{2} \rceil - 1$. Further, $M_1 = M_0$ implies that $\lceil \frac{l_0}{2} \rceil - 1 = M_1 = M_0 \geq \lceil \frac{l_0}{2} \rceil$, and this is a contradiction. Therefore, our observation that $\Delta^1 - 1 > \lceil \frac{l_1}{2} \rceil$ is true. Further, since $l_1 = l_0 - 2$ and $M_1 = M_0$, it must be the case that $\Delta^1 = \Delta^0$. Therefore, $\Delta^0 - 1 > \lceil \frac{l_0}{2} \rceil - 1$ and hence, $\Delta^0 > \lceil \frac{l_0}{2} \rceil$. Therefore, now applying Lemma 2, we know that there can be at most two vertices $v_1, v_2 \in V(T)$ of degree Δ^0. Further, from Lemma 3, a cut-vertex of maximum degree is the representative associated with an equivalence class. Since the biconnectivity augmentation adds an edge between $x \in X_1$ and $y \in X_2$ such that $deg(w(X_1)) \geq deg(w(X_2)) \geq deg(w(X_i)), 2 \leq i \leq r$, it follows that the degrees of the associated representatives also reduce by one. Since the maximum degree

vertices have degree more than $\lceil \frac{l}{2} \rceil$, there are at most two of them (by Lemma 2), both are associated representatives of two equivalence classes (by Lemma 3), and the algorithm adds an edge between two vertices in these two equivalence classes, it follows that the maximum degree reduces by 1. That is, $\Delta^1 = \Delta^0 - 1$ and this contradicts our earlier conclusion that $\Delta^1 = \Delta^0$. Therefore, our starting premise, $M_1 = M_0$ is wrong, $M_1 = M_0 - 1$. □

Theorem 5. *Let T be a tree. The minimum biconnectivity augmentation of T uses $M_0 = \max\{\lceil \frac{l}{2} \rceil, \Delta(T) - 1\}$ edges.*

Proof. The algorithm guarantees that at the end of each iteration we always have a tree. Further, we know from Lemma 4 that if $|X_{ec}| \geq 2$, then $M_1 = M_0 - 1$. Therefore, let us assume that *while-loop* of the function *non-star-augment* is called p times, after which *star-augment* is called once. Then $M_p = M_0 - p$. In other words, in the tree obtained after p calls to *while-loop* of *non-star-augment* there is a single equivalence class. In such a situation, we know from our earlier discussion that the resulting tree obtained from *non-star-augment* is a star. We know that for star $\Delta^p = l_p$, and by definition $M_p = \Delta^p - 1$. *star-augment* function biconnects this tree using M_p edges. Therefore, the total number of edges added is $M_p + p = M_0 - p + p = M_0$. It is also easy to see by induction on p that the set of edges added is a biconnectivity augmentation set. The base case is when $p = 0$, and clearly, *star-augment* biconnects the tree T. After the first iteration, the resulting tree goes through a strictly smaller number of iterations, and we assume by induction on the number of iterations that the algorithm returns a biconnectivity augmentation set using $M_0 - 1$ edges. The first iteration counts one more edge that ensures an additional path between the unaccounted vertices of degree at most two, thus guaranteeing biconnectivity. Note that the unaccounted vertices are two paths in the tree each originating at a distinct leaf from two distinct equivalence classes, namely X_1 and X_2. □

2.3. *A linear-time implementation of Algorithm* 3 *using equivalence-classes-ADT*

The important steps to be analyzed in our algorithm are computing the set of equivalence classes, finding two equivalence classes such that the degree of its representatives are maximum and second maximum, and updating of equivalence classes after edge additions. We below mention possible situations that may arise on execution of lines (3) and (4) of Algorithm 3, and the specific tasks to be taken in updating the set of equivalence classes.

- For an equivalence class X, if $deg(w(X)) = 2$ and $|X| = 1$ then by definition, $w(X)$ is no longer a representative vertex of X. The *update* in this case is the identification of the new representative of X. The update is done by identifying a nearest vertex z of degree at least 3 from $w(X)$. Since $deg(z) \geq 3$, there must be an equivalence class Y (possibly empty) such that $w(Y) = z$. We say that X

Table 1. Operations defined on *equivalence-classes-ADT*.

Set-up-eclass()	This creates the set of equivalence classes from the tree. This is the first method to be called to populate the data structure
Locate(L, w)	Return from L, the location of equivalence class whose representative is w
Insert(L, pos, l)	Insert the leaf node l into the equivalence class whose position in L is *pos* and return the updated list L
Retrieve(L, pos)	Return from L, an element of the equivalence class whose position is *pos*
Parent-Representative-Indicator(L, x)	Return from L, the *parent-indicator* of leaf x

merges with the equivalence class Y. To identify such a z efficiently, we maintain an additional information at each leaf x to ensure that the search for z does not happen on the path from $w(X)$ to x. We call this additional information as *parent-indicator* for each leaf x in T. For a leaf $x \in X$, the *parent-indicator* of x is a vertex x' such that $x' \in N_G(w(X))$ and $x' \in P_{xw(X)}$. The purpose of *parent-indicator* is that the search for z must avoid $P_{x'x}$ as every vertex in $P_{x'x}$ is of degree two.

- The other possible situations are $|X| = 0$ or $|X| \geq 2$ or $|X| = 1$ and $deg(w(X)) \geq 3$. In this case, the *update* must reorganize the equivalence classes based on the degree of the representatives.

An ADT for Equivalence Classes: We propose an abstract data type(ADT), namely, *Equivalence-Classes-ADT* to *maintain* the set of equivalence classes. Using the operations listed in Table 1, we present a linear-time implementation of the steps in Algorithm 3. We use two basic data types namely, *eclass-list* to refer the data type of X_{ec} and *node* to refer the data type of a vertex x. Let L be an object of type *eclass-list*.

Data Structures Used: The main data structure which is populated by *Set-up-eclass*( ) is a table of records, which we call *table-eclass*. For each vertex of degree at least 3 in T there is a record in *table-eclass*. Each record has three fields, the label of a vertex w of degree at least 3, the degree of w in T, and the subset of leaves in the equivalence class associated with w. The method *Set-up-eclass*( ) fills entries in *table-eclass* by performing Depth First Search(DFS) on T. During the DFS, for each vertex $w, deg(w) \geq 3$, we find the equivalence class of w. We also store the *parent-indicator* of each element in the equivalence class associated with vertex w (i.e. for each leaf in T). Since each vertex is visited at most twice during the DFS,

construction of *table-eclass* takes at most $2|E(T)|$ and hence $O(n)$ time. With this table, ADT methods *Insert*, *Retrieve*, and *Parent-Representative-Indicator* can be implemented in constant time.

To index *table-eclass* to locate the record labelled w, in constant time, we use *index-table*[ ] array to store the position of w in *table-eclass* table. This implements the ADT method *Locate* in constant time.

The subroutines *get-top-two-representative* and *update-eclass* implement lines (2) and (5), respectively of Algorithm 3. These subroutines make use of two additional data structures. The data structure *initial-active-eclass*[ ] array contains vertices x in non-increasing order of their degrees such that there is a non empty equivalence class associated with x. This can be achieved by running radix sort on the associated set and it runs in $O(n)$ time. *sorted-vertex*[i] which contains a list of vertices x of degree i in T. Initially, *sorted-vertex*[i] is filled using elements of *initial-active-eclass*[ ]. We present the pseudo code of Algorithm 3 in procedure *MAIN*.

Psuedo code for the implementation of Algorithm 3.

Procedure-MAIN()

(1) Compute the set of equivalence classes
(2) get-top-two-representative()
(3) edge-addition() `/* this subroutine incurs constant time */`
(4) remove-path-from-leaf-to-representative()`/* this implements line(4) of Algorithm 3, total time spent over all iterations is` $O(n)$ `*/`
(5) update-eclass()

Procedure 1: get-top-two-representative()

1: `/* max1=initial-active-eclass[0], max2=initial-active-eclass[1] This subroutine incurs constant time*/`
2: Remove the first element from the list associated with *sorted-vertex*[$max1$], say x
3: max1-pos-table=locate(x)
4: Retrieve(L,max1-pos-table)
5: Decrement the $deg(x)$ by one in the table
6: Remove the first element from the list associated with *sorted-vertex*[$max2$], say y
7: max2-pos-table=locate(y)
8: Retrieve(L,max2-pos-table)
9: Decrement the $deg(y)$ by one in the table

Procedure 2: update-eclass(L): eclass-list

1: `/* This procedure updates` X_{ec} `after edge addition. Here we outline the specific tasks to be taken during update` X_{ec}`.` $X_1, X_2 \in X_{ec}$ `such that` $deg(w(X_1)) = max1$ `and` $deg(w(X_2)) = max2$`. Every line incurs constant time */`

2: We first check whether X_1 or X_2 merge with other element in X_{ec}. Merging happens, if $|X_i| = 1$ and $deg(w(X_i)) = 2, i \in \{1, 2\}$.
3: If **merging=true** (for our discussion assume X_1 merges), perform the following 4 steps
4: Follow the path P from $w(X_1)$ till we hit the first vertex w' of degree at least 3 such that P does not contain *parent-indicator* of leaf $x \in X_1$ `/* For this step, the total time spent over all iterations is` $O(n)$
5: Merge X_1 with the equivalence class associated with w'.
6: Make an entry for w' in *sorted-vertex*[ ], if not exists. Reset *max*1 and *max*2 using *sorted-vertex*[ ] and *initial-active-eclass*[ ]
7: Update *parent-indicator* of x.
8: If **merging=false**, reset *max*1 and *max*2 using *sorted-vertex*[ ] and *initial-active-eclass*[ ]

Run-Time Analysis: The overall time complexity of *non-star-augment* algorithm is given as follows: Depth First Search(DFS) on the given tree to fill *table-eclass* incurs $O(n)$ time. Running radix sort and creating an entry in *sorted-vertex*[ ] takes $O(n)$ time. With supporting data structures, we incur constant time effort for the subroutine *get-top-two-representative*. For the subroutine *update-eclass*, we are analyzing the total effort involved over all iterations. If there is a *merging* then to identify the nearest vertex of degree at least 3, the total time spent for the above operation over all iterations is $O(n)$, as we visit each vertex exactly once. This is possible because of *parent-indicator* information stored at each leaf. If there is no *merging* then we incur constant time effort to reorganize the set X_{ec}. As per our algorithm, each vertex of degree at least 3 is visited at most the size of equivalence class associated with that vertex. Since the sum of the sizes of all equivalence classes is at most the sum of degrees in the tree, the time spent in identifying an augmentation set is $O(n)$. When the tree is a star, *star-augment* incurs an additional $O(n)$ time. Therefore, the total time complexity of *tree-augment* algorithm is $O(n)$, linear in the input size.

3. Augmentation in Graphs Using Tree Augmentation Approach

In this section, we generalize tree augmentation results and present an augmentation in 1-connected graphs. We then present triconnectivity augmentation in biconnected graphs. Finally, we present an augmentation in k-connected chordal graphs.

3.1. *Biconnectivity augmentation*

As mentioned earlier, the approach is to represent a 1-connected graph by an appropriate tree, and an optimum augmentation set is determined using the results presented in Section 2.

The Associated Tree Description: We represent the given 1-Connected graph by a tree, namely, the biconnected component tree. A biconnected component

is a maximal 2-connected subgraph of a given graph. A biconnected component tree T is a tree constructed from the given graph G as follows: each vertex in T denotes either a biconnected component of G or a cut-vertex of G. For a vertex $x \in V(T)$, the associated label $label(x) = \{c\}$, c is a cut-vertex in G or $label(x) = S$, $S \subseteq V(G)$ such that $G[S]$ is a maximal 2-connected subgraph in G. $V(T) = B \cup B'$ where $B = \{x \mid label(x)$ is a biconnected component in $G\}$ and $B' = \{x \mid label(x)$ is a cut vertex in $G\}$. The adjacency between a pair of vertices in T is defined as follows: for $x, y \in V(T)$, $\{x, y\} \in E(T)$ if one of the following is true (see Fig. 2 for an illustration)

- $x \in B$ and $y \in B'$ such that $label(y) \subset label(x)$
- Both $x, y \in B'$ and there is a $\{c, c'\}$ cut-edge in G such that $label(x) = \{c\}$ and $label(y) = \{c'\}$
- $x \in B$ and $y \in B'$ such that $label(x)$ is a trivial biconnected component($|label(x)| = 1$) and there is a $\{u, v\}$ cut-edge in G such that $label(x) = \{u\}$ and $label(y) = \{v\}$.

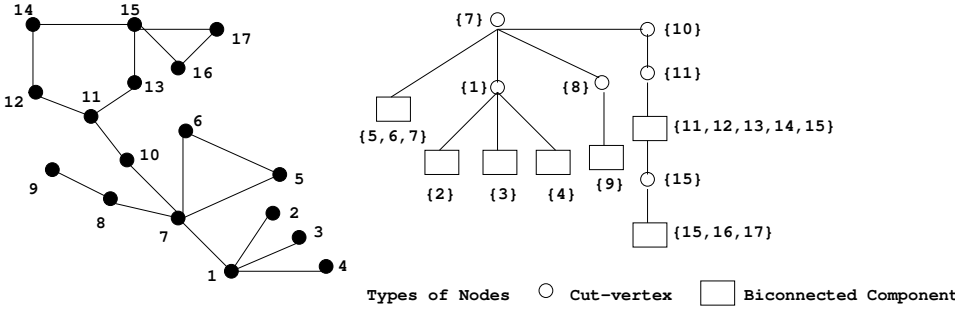


Fig. 2. A 1-connected graph and the associated biconnected component tree.

The Lower Bound on Biconnectivity Augmentation in 1-Connected Graphs: We use a well-known fact that in any 2-connected graph, for any pair of vertices, there exists two vertex disjoint paths between them. Let X denote the set of biconnected components having *exactly one* cut-vertex in G. Note that by our construction of T, each element of X corresponds to the *label* of a leaf in T. Clearly, to biconnect G we must augment at least $\left\lceil \frac{|X|}{2} \right\rceil$ edges and therefore we need at least $\lceil \frac{l}{2} \rceil$ edges, l denotes the number of leaves in T. Another lower bound is due to the number of components created by removing a cut vertex of G. Note the one-one correspondence between the number of components created by a cut vertex of G and the degree of a vertex $x \in B'$. This shows that in any biconnectivity augmentation of G, for each $x \in B'$, one must find at least $deg_T(x) - 1$ new edges in the augmenting set. Therefore, we must augment at least $\Delta_c(T) - 1$ edges to biconnect G, where $\Delta_c(T) = \max_{x \in B'} deg(x)$. Therefore, by combining the two lower

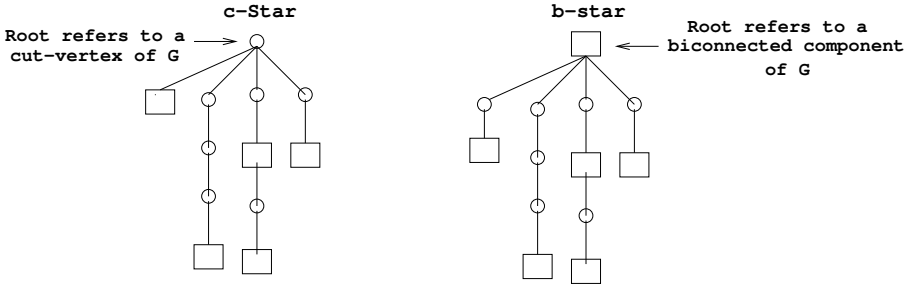


Fig. 3. An example of a c-star and a b-star.

bounds, the number of edges to biconnect G is at least $\max\{\lceil \frac{l}{2} \rceil, \Delta_c(T) - 1\}$. This lower bound is indeed tight as shown in [2, 4].

Note the similarity between this lower bound and the one presented in the previous section. $\Delta_c(T)$ appears here instead of $\Delta(T)$. Because of a similarity between the two lower bounds and the associated representation is a tree, all combinatorial analysis presented in the previous section naturally extends in the study of biconnectivity augmentation in 1-connected graphs. Hence, we get a new proof of tightness which leads to a new linear-time algorithm for finding a biconnectivity augmentation set in a 1-connected graph. The equivalence relation and therefore the set of equivalence classes yields an elegant algorithm that avoids completely the recomputation of the associated tree at each iteration, unlike, the results presented in [3, 4].

The Sketch of Algorithms 4 and 5: Compute the set X_{ec} of equivalence classes of the biconnected component tree T. If $|X_{ec}| \geq 2$ then find an augmenting set using the subroutine *non-star-augment* of Section 2. The tail condition is when there is exactly one equivalence class and such a tree is a star shaped tree. In this case, we get two special trees depending on the label of the representative vertex z. We know that $label(z)$ is either a set containing a cut-vertex of G or a maximal 2-connected subgraph in G. If $label(z)$ is a set containing a cut-vertex of G, then we call T as a *c-star*. If $label(z)$ is a maximal 2-connected subgraph in G, then we call T as a *b-star*. An example is shown in Fig. 3. An Augmentation for tail condition is done separately. With this new approach we simplify our biconnectivity augmentation algorithm by calling *non-star-augment* subroutine with biconnected component tree as the input. Let $M_0 = \max\{\lceil \frac{l}{2} \rceil, \Delta_c(T) - 1\}$. Observe that $\Delta_c(T)$ appears in this expression instead of $\Delta(T)$. With this observation we see that Lemmas 2, 3, and 4 are true in the biconnected component tree T as well. We only present a proof of Theorem 6.

Theorem 6. *Let T be a biconnected component tree. The minimum biconnectivity augmentation of G uses $M_0 = \max\{\lceil \frac{l}{2} \rceil, \Delta_c(T) - 1\}$ edges.*

Algorithm 4 Augmentation in 1-connected graphs: 1-*connect-augment*(*Graph* G)

Compute the biconnected component tree T of G
if there are exactly two leaves x and y in T **then**
  Add the edge $\{x, y\}$ and return the biconnected graph
else
  Compute equivalence classes
  $X_{ec} = \{X | X$ is an equivalence class with associated vertex $w(X)\}$
  if $|X_{ec}| > 1$ **then**
    `/* T is not a star-like tree */`
    Y_{ec}=non-star-augment(X_{ec}) `/* Call to non-star-augment( ) of Section 2, returns` Y_{ec}`*/`
    `bc-star-augment(`Y_{ec}`) /*` Y_{ec} `has exactly one equivalence class. The associated tree is a star */`
  else
    `/*`T `is a star-like tree */`
    bc-star-augment(X_{ec})
  end if
end if
For each edge $\{x, y\}$ added into T, add $\{u, v\}$ to G such that $u \in label(x)$ and $v \in label(y)$ and u and v are non-cut vertices in G

Algorithm 5 Biconnectivity Augmentation in stars: *bc-star-augment*(*List-of-eclass* X_{ec})

Let $X = \{x_1, \ldots, x_l\}$ be the set of leaves in T such that $X \in X_{ec}$
if T is a c-star **then**
  `/*` T `is a star-like tree with a central vertex corresponding to a cut-vertex in` G `*/`
  Add $|X| - 1$ edges to T, i.e., add $\{\{x_i, x_{i+1}\} \mid 1 \leq i \leq |X| - 1, x_i \in X\}$.
else
  `/*` T `is a star-like tree with a central vertex corresponding to a biconnected-component in` G `*/`
  if l is even **then**
    Add $\{\{x_i, x_{l-i+1}\} \mid 1 \leq i \leq \frac{l}{2}\}$ to T
  else
    Add $\{\{x_i, x_{(l-1)-i+1}\} \mid 1 \leq i \leq \frac{l-1}{2}\} \cup \{x_1, x_l\}$ to T
  end if
end if

Proof. The algorithm guarantees that at the end of each iteration we always have a tree. Further, we know from Lemma 4 that if $|X_{ec}| \geq 2$, then $M_1 = M_0 - 1$. Therefore, let us assume that the *while-loop* of the function *non-star-augment* of

Section 2 is called p times, after which the function *bc-star-augment* is called once. Then $M_p = M_0 - p$. In other words, in the tree obtained after p calls to the *while-loop* of *non-star-augment* there is a single equivalence class. If that equivalence class is associated with a cut-vertex of G, then the resulting tree obtained from *non-star-augment* is a c-star. We know that for c-star $\Delta_c^p = l_p$, and by definition $M_p = \Delta_c^p - 1$. The function *bc-star-augment* biconnects this tree using M_p edges. Therefore, the total number of edges added is $M_p + p = M_0 - p + p = M_0$. If that equivalence class is associated with a biconnected component of G then the resulting tree obtained from *non-star-augment* is a b-star. The total number of edges added in this case is $\left\lceil \frac{l-l_p}{2} \right\rceil + \left\lceil \frac{l_p}{2} \right\rceil = \left\lceil \frac{l}{2} \right\rceil$. It is also easy to see by induction on p that the set of edges added is a biconnectivity augmentation set. The base case is when $p = 0$, and clearly, *bc-star-augment* biconnects the graph G. After the first iteration, the resulting tree goes through a strictly smaller number of iterations, and we assume by induction on the number of iterations that the algorithm returns a biconnectivity augmentation set using $M_0 - 1$ edges. The first iteration counts one more edge that ensures an additional path between the unaccounted vertices of degree at most two, thus guaranteeing biconnectivity. Note that the unaccounted vertices are two paths in the tree each originating at a distinct leaf from two distinct equivalence classes, namely X_1 and X_2. □

The overall time complexity of our algorithm is given as follows: the Depth First Search(DFS) on the given graph can be used to compute the biconnected component tree in $O(n + m)$ time [13]. The number of nodes in the biconnected component tree is $O(n)$. We know from the previous section that *non-star-augment* can be implemented in $O(n)$ time. Hence, the total time complexity of our algorithm is $O(n + m)$, linear in the input size.

3.2. *Triconnectivity augmentation*

We present an elegant linear-time algorithm that finds a minimum set of edges whose augmentation to a 2-connected graph makes it 3-connected. We work with the standard 3-block tree of a 2-connected graph [7].

The Associated Tree Description: Given a 2-connected graph G, the vertex set of a 3-block tree T, $V(T) = \{x \mid label(x)$ is a triconnected component (a maximal 3-connected subgraph) in G or a 2-sized vertex separator or a polygon or a vertex of degree 2$\}$. For convenience, we use the following notation. $V(T)$ consists of four kinds of vertices called σ vertices, π vertices, α vertices, and β vertices. We create a σ vertex for each 2-sized vertex separator such that components separated by the vertex separator is either a triconnected component or a polygon, a π vertex for each polygon, a α vertex for each vertex of degree 2, and a β vertex for each triconnected component. A vertex of degree 2 (α vertex) is the only trivial triconnected component. Note that the set of α vertices is a subset of the set of β vertices. The

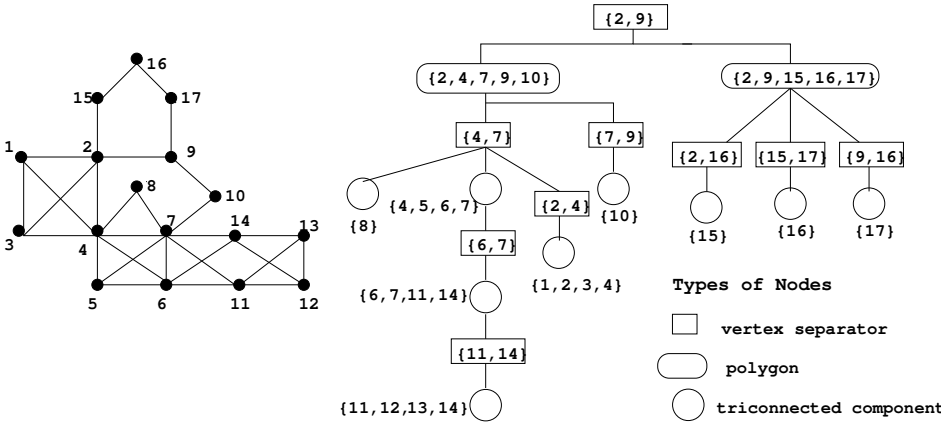


Fig. 4. A 2-connected graph and the associated 3-block tree.

adjacency between a pair of vertices in $V(T)$ is defined as follows: for $x, y \in V(T)$, $\{x, y\} \in E(T)$, if one of the following is true (see Fig. 4 for an illustration)

- $x \in \sigma$ and $y \in \beta$ and $label(x) \subset label(y)$.
- $x \in \sigma$ and $y \in \pi$ and $label(x) \subset label(y)$.
- $x \in \alpha$ and $y \in \sigma$ such that $label(y) = N_G(x)$.

Note that we do not create a σ vertex for each pair of non adjacent vertices in a polygon (π vertex), we create a σ vertex for a pair in the polygon if it is involved in the Tutte split. More information about Tutte split and merge can be found in [14]. Let $\Delta_\sigma(T) = \max_{x \in \sigma} deg(x)$.

The Lower Bound on Triconnectivity Augmentation: Given a 2-connected graph G and a 3-block tree T of G, the number of edges to triconnect G is at least $\max\{\lceil \frac{l}{2} \rceil, \Delta_\sigma(T) - 1\}$. This lower bound is indeed tight as shown in [6, 7]. We omit the proof here and a proof similar to Section 3.1 can be given.

Note the similarity between this lower bound and the one presented in Section 2. $\Delta_\sigma(T)$ appears here instead of $\Delta(T)$. Because of a similarity between the two lower bounds and the associated representation is a tree, all combinatorial analysis presented in Section 2 naturally extends in the study of triconnectivity augmentation of 2-connected graphs. Hence, we get a new proof of tightness which leads to a new linear-time algorithm for finding triconnectivity augmentation sets in 2-connected graphs. This approach is fundamentally different from the results presented in [6, 7].

The Sketch of Algorithms 6 and 7: Compute the set X_{ec} of equivalence classes of a 3-block tree T. If $|X_{ec}| \geq 2$, then find an augmenting set using the subroutine *non-star-augment* of Section 2. The tail condition is when there is exactly one

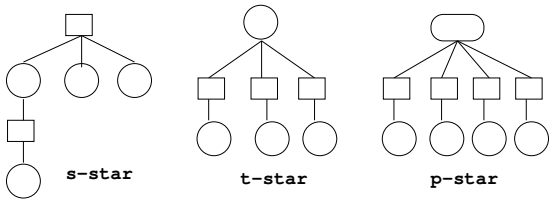


Fig. 5. The three special star-like trees, a s-star (if root is a separator), a t-star (if root is a triconnected component), and a p-star (if root is a polygon).

equivalence class and such a tree is a star shaped tree. In this case, we get three special trees depending on the label of the representative vertex z. σ vertex (a vertex separator of size 2) or z can be a β vertex (a triconnected component) or π vertex (a polygon) of T and accordingly we call T as a *s-star*, *t-star*, and *p-star*, respectively. An example is illustrated in Fig. 5. An Augmentation for tail condition is done separately. With this new approach, we simplify our triconnectivity augmentation algorithm by calling the subroutine *non-star-augment* with 3-block tree as the input. A proof similar to Theorem 6 can be given to prove that the number of edges augmented by our algorithm is $\max\{\lceil \frac{l}{2} \rceil, \Delta_\sigma(T) - 1\}$ and the resulting graph is 3-connected.

Algorithm 6 Augmentation in 2-connected graphs: 2-*connect-augment*(*Graph G*)

Compute the 3-block tree T of G
if there are exactly two leaves x and y in T **then**
  Add the edge $\{x, y\}$ and return the triconnected graph
else
  Compute equivalence classes
  $X_{ec} = \{X \mid X$ is an equivalence class with associated vertex $w(X)\}$
  if $|X_{ec}| > 1$ **then**
    `/* T is not a star-like tree */`
    Y_{ec}=non-star-augment(X_{ec}) `/* Call to non-star-augment() of Section 2, returns` Y_{ec}`*/`
    tail-star-augment(Y_{ec}) `/* Associated tree is a star with a single equivalence class */`
  else
    `/*`T `is a star-like tree */`
    tail-star-augment(X_{ec})
  end if
end if
For each edge $\{x, y\}$ added into T, add $\{u, v\}$ to G such that $u \in label(x)$ and $v \in label(y)$ and u and v are elements of a β vertex and not elements of any σ vertex in G

Algorithm 7 Triconnectivity Augmentation in stars: *tail-star-augment*(*List-of-eclass* X_{ec})

Let $X = \{x_1, \ldots, x_l\}$ be the set of leaves in T such that $X \in X_{ec}$
if T is a s-star **then**
  `/* T is a star-like tree with a central vertex corresponding to a σ vertex */`
  Add $|X| - 1$ edges to T, i.e., add $\{\{x_i, x_{i+1}\} \mid 1 \leq i \leq |X| - 1, x_i \in X\}$.
else
  `/* T is a star-like tree with a central vertex corresponding to a π vertex or β vertex */`
  if l is even **then**
    Add $\{\{x_i, x_{l-i+1}\} \mid 1 \leq i \leq \frac{l}{2}\}$ to T
  else
    Add $\{\{x_i, x_{(l-1)-i+1}\} \mid 1 \leq i \leq \frac{l-1}{2}\} \cup \{x_1, x_l\}$ to T
  end if
end if

Theorem 7. *Let T be a 3-block tree. The minimum triconnectivity augmentation of G uses*
$M_0 = \max\{\lceil \frac{l}{2} \rceil, \Delta_\sigma(T) - 1\}$ *edges.*

Proof. The algorithm guarantees that at the end of each iteration we always have a tree. Further, we know from Lemma 4, that if $|X_{ec}| \geq 2$, then $M_1 = M_0 - 1$. Therefore, let us assume that the *while-loop* of the function *non-star-augment* of Section 2 is called p times, after which *tail-star-augment* is called once. Then $M_p = M_0 - p$. In other words, in the tree obtained after p calls to *while-loop* of *non-star-augment* there is a single equivalence class. If that equivalence class is associated with a σ vertex of G then the resulting tree obtained from *non-star-augment* is a *s-star*. We know that for *s-star* $\Delta_\sigma^p = l_p$, and by definition $M_p = \Delta_\sigma^p - 1$. The function *tail-star-augment* triconnects this tree using M_p edges. Therefore, the total number of edges added is $M_p + p = M_0 - p + p = M_0$. If that equivalence class is associated with a triconnected component or a polygon of G, then the resulting tree obtained from *non-star-augment* is a *t-star* or a *p-star*. The total number of edges added in this case is $\left\lceil \frac{l-l_p}{2} \right\rceil + \left\lceil \frac{l_p}{2} \right\rceil = \lceil \frac{l}{2} \rceil$. It is also easy to see by induction on p that the set of edges added is a triconnectivity augmentation set. The base case is when $p = 0$, and clearly, *tail-star-augment* triconnects the graph G. After the first iteration, the resulting tree goes through a strictly smaller number of iterations, and we assume by induction on the number of iterations that the algorithm returns a triconnectivity augmentation set using $M_0 - 1$ edges. The first iteration counts one more edge that ensures an additional path between the unaccounted vertices of degree at most two, thus guaranteeing triconnectivity. Note that the unaccounted

vertices are two paths in the tree each originating at a distinct leaf from two distinct equivalence classes, namely X_1 and X_2. □

From [15], we know that the associated 3-block tree of a 2-connected graph can be constructed in linear time. From Section 2, we know that a triconnectivity augmentation set can be obtained in linear time as well. Therefore, the overall time complexity to triconnect a biconnected graph is linear in the input size.

3.3. *Connectivity augmentation in k-connected chordal graphs*

A graph is chordal if it contains no induced cycle of length at least 4. Chordal graphs are very important subclass of perfect graphs and chordal graphs are a super class of k-separator chordal graphs. The highlights of chordal graphs are every minimal vertex separator is a clique and there exists a perfect elimination ordering of vertices [12]. As with other augmentations reported in this paper, we represent a k-connected chordal graph using a tree, namely, a *clique separator tree*, the label of whose vertices denote either a minimum vertex separator of size k or a maximal clique of size at least $k+1$ or a $(k+1)$-connected component (a maximal $(k+1)$-connected subgraph) of the graph. To the best of our knowledge such a tree does not exist in the literature. Some of the well-known representations of chordal graphs are clique graphs, clique trees, and clique separator graphs [17]. From the following observation, it may be noted that such a clique separator tree exists and it is unique.

Lemma 8. *Let G be a k-connected chordal graph and S be a minimum vertex separator in G. Let $\{C_1, \ldots, C_r\}$ denote the set of connected components in $G \setminus S$. Every other minimum vertex separator S' in G is either contained in some C_i or contained in $C_i \cup S$. In other words, there is no S' that spans across the components.*

Proof. Suppose there exists S' such that $C_i \cap S' \neq \phi$ and $C_j \cap S' \neq \phi, i \neq j$. Since G is chordal, every minimal vertex separator is a clique. In particular, S' is a clique and this implies $\{u, v\} \in E(G)$. This implies that there is a path between C_i and C_j avoiding the vertices of S. However, we know that such a path does not exist as C_i and C_j are connected components of $G \setminus S$. A contradiction. Hence the lemma. □

The Associated Tree Construction: $M(G) = \{S \subset V(G) \mid S$ is a minimum (k size) vertex separator in $G\}$ and $K(G) = \{K \subset V(G) \mid K$ is a maximal clique of size $k+1$ in G or a $(k+1)$-connected component in $G\}$. For a vertex $x \in V(T)$, the associated label $label(x)$ is a subset of $V(G)$ such that $label(x) \in M(G)$ or $label(x) \in K(G)$. Let $V_M = \{x \mid label(x) \in M(G)\}$ and $V_K = \{x \mid label(x) \in K(G)\}$. $V(T) = V_M \cup V_K$. For $x, y \in V(T)$, the adjacency between x and y is defined as follows: $\{x, y\} \in E(T)$ if $x \in V_M$ and $y \in V_K$ such that $label(x) \subset label(y)$. An illustration is given in Fig. 6.

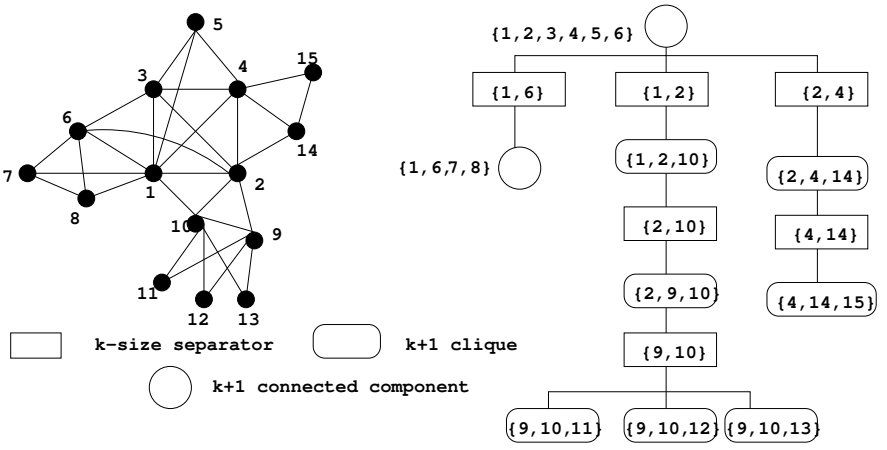


Fig. 6. A k-connected chordal graph and its clique separator tree.

The Lower Bound on k-Connected Chordal Graph Augmentation: Let X denote the set of maximal cliques of size $k+1$ containing *exactly one* k-clique separator in G and the set of $(k+1)$-connected components containing *exactly one* k-clique separator in G. Note that by our construction of T, each element of X corresponds to the *label* of a leaf in T. Clearly, to $(k+1)$-connect G we must augment at least $\left\lceil \frac{|X|}{2} \right\rceil$ edges and therefore we need at least $\left\lceil \frac{l}{2} \right\rceil$ edges, l denotes the number of leaves in T. Another lower bound is obtained due to the following observation. The degree of a minimum vertex separator $S \in M(G)$ is the number of maximal cliques of size at least $k+1$ in G that contain S plus the number of $(k+1)$-connected components that contain S and it is denoted by $deg(S)$. $deg(S) = |\{K \mid K \in K(G) \ \wedge \ S \subset K\}|$. Let $S^{max} = \max_{S \in M(G)} deg(S)$. Let $x \in V_M$ is such that $deg(x) = S^{max}$. Also $\Delta_k(T) = deg(x)$ such that $deg(x) = S^{max}$. Since S^{max} is a minimum vertex separator of size k, for each u and v in G at most k vertex disjoint paths contain elements from S^{max}. Therefore, to make G a $(k+1)$-connected graph, we must have at least one more path which does not contain elements from S^{max}. Since the removal of S^{max} from G creates S^{max} components, to get a path which avoids elements of S^{max}, we must to add at least $S^{max} - 1$ edges to G. Thus, we conclude that $|E_{min}(G)| \geq \max\{\left\lceil \frac{|X|}{2} \right\rceil, S^{max} - 1\}$. For the clique separator tree T, the above lower bound translates into $\max\{\left\lceil \frac{l}{2} \right\rceil, \Delta_k(T) - 1\}$ edges. We observe that a clique separator tree is very similar to a 3-block tree, a σ vertex , a π vertex, and a β vertex in a 3-block tree corresponds to a k-size separator, a maximal $(k+1)$-clique, and a $(k+1)$-connected component, respectively in a clique separator tree. With this observation, we simplify our k-connected chordal graph augmentation algorithm (Algorithm 8) by calling the function 2-*connect-augment* with clique separator tree as the input. Let $A = \{\{x, y\} \mid x, y$ are leaves in $T\}$ be the set of edges returned by

Algorithm 8 Augmentation in k-connected chordal graphs: *k-connected-chordal-augment*(*Tree* T)

`/* T is the clique separator tree of a k-connected chordal graph G */`

Perform $(k+1)$-connectivity augmentation of G using 2-*connect-augment*(T)
For each edge $\{x, y\}$ added into T, add $\{u, v\}$ to G such that $u \in label(x)$ and $v \in label(y)$ and u and v are simplicial vertices in G

our algorithm. Let $E_a(G)$ be the corresponding set of edges in G, $E_a(G) = \{\{u, v\} \mid \{x, y\} \in A$ and $u \in label(x)$ is a simplicial vertex in G and $v \in label(y)$ is a simplicial vertex in $G\}$.

Lemma 9. *Let G_{aug} be the graph obtained from G by augmenting $E_a(G)$. G_{aug} is $(k+1)$-connected.*

Proof. Suppose G_{aug} is not $(k+1)$-connected. This implies that there exists a k size minimal vertex separator S in G_{aug}. Since S is also a separator in G, it follows that S is a k-clique in G. An invariant maintained by our algorithm is that for each minimal vertex separator S', $deg(S') - 1$ edges are added to ensure that the $deg(S')$ components in $G \setminus S'$ are connected in G_{aug}. However, since the removal of S disconnects G_{aug}, it follows that the above invariant is not maintained. This is a contradiction. Hence there is no k-size vertex separator S in G_{aug}. Therefore, G_{aug} is $(k+1)$-connected. □

Theorem 10. *Let G be a k-connected chordal graph. An optimum $(k+1)$-connectivity augmentation of G uses $|E_a(G)| = \max\{\left\lceil \frac{|X|}{2} \right\rceil, S^{max} - 1\}$ edges.*

Proof. We know that our algorithm augments $|E_a(G)|$ edges. Therefore from Lemma 9, it follows that G augmented with $E_a(G)$ is $(k+1)$-connected. Hence the theorem. □

Given a clique separator tree, our algorithm yields a minimum augmentation set in linear time. The clique separator tree can be obtained from the *clique separator graph* of a chordal graph in linear time. The clique separator graph was introduced by Ibarra in [17]. The algorithm in [17] incurs $O(n^3)$ time to compute the clique separator graph of a chordal graph. Therefore, a minimum connectivity augmentation of chordal graphs can be computed in $O(n^3)$ time.

Remarks on Augmentation in k-Separator Chordal Graphs: Though chordal augmentation incurs cubic time effort, the augmentation for two important subclasses of chordal graphs, namely, k-trees and k-separator chordal graphs can be done in linear time. k-separator chordal graphs properly contain k-trees which are a generalization of trees (1-trees). A k-tree is defined recursively as follows: a

$k+1$ clique is a k-tree, if G' is a k-tree, the graph $G = G' \cup \{v\}$ such that $N_G(v)$ is a k-clique in G' is also a k-tree. Observe that the two important properties of k-trees are every minimal vertex separator is a k-clique and every maximal clique is of size $k+1$. It is now natural to characterize a super class of k-trees by relaxing the constraints on the maximal clique size, i.e. we allow maximal cliques of size at least $k+1$. Thus, we get k-separator chordal graphs with the property that every minimal vertex separator is exactly a k-clique and every maximal clique is of size at least $k+1$. Also, observe that k-separator chordal graphs are chordal graphs too. The notion of k-separator chordal graphs were introduced by Sreenivasa Kumar *et al.* in [16]. Observe that a clique separator tree of a k-separator chordal graph is a special tree in which the label of each node is either a k-clique separator or a maximal clique of size at least $k+1$ and such a tree can be obtained from the construction order in linear time. Therefore, connectivity augmentation in k-separator chordal graphs can be done in linear time.

4. Concluding Remarks

In this paper, we have presented a framework for finding an optimum connectivity augmentation set for three special graph classes. From the associated tree of the given graph, our framework maintains a partition of the set of leaves which inturn guides us in finding an optimum augmentation set. We have proposed a linear-time algorithm with an elegant combinatorial analysis for biconnectivity and triconnectivity augmentation and a cubic-time algorithm for connectivity augmentation in k-connected chordal graphs. A natural extension of this work is to study connectivity augmentation in graphs of treewidth k. Also, connectivity augmentation in general graphs using this framework is an interesting direction of research.

References

[1] L.A.Vegh: Augmenting undirected node connectivity by one. In Proceedings of the 42nd ACM symposium on Theory of computing(STOC), pp.563-572 (2010)
[2] K.P.Eswaran, R.E.Tarjan: Augmentation problems. SIAM Journal of Computing, 5, 653-665 (1976)
[3] A.Rosenthal, A.Goldner: Smallest augmentation to biconnect a graph. SIAM Journal of Computing, 6, 55-66 (1977)
[4] T.S.Hsu, V.Ramachandran: On finding a smallest augmentation to biconnect a graph. SIAM Journal of computing, 22, 889-912 (1993)
[5] T.S.Hsu: Simpler and faster biconnectivity augmentation. Journal of Algorithms, 45, 55-71 (2002)
[6] T.Watanabe, A.Nakamura: 3-connectivity augmentation problems. In Proc. of 1988 IEEE Int'l Symp. on Circuits and Systems, pp. 1847-1850 (1988)
[7] T.S.Hsu, V.Ramachandran: A linear-time algorithm for triconnectivity augmentation. In Proc. of 32nd Annual IEEE Symp. on Foundations of Comp. Sci.(FOCS), pp.548-559 (1991)
[8] T.S.Hsu: On four connecting a triconnected graph. Journal of Algorithms, 35, 202-234 (2000)

[9] T.Watanabe, A.Nakamura: Edge-connectivity augmentation problems. Computer System Sciences, 35(1), 96-144 (1987)
[10] K.Steiglitz, P.Weiner, D.J.Klietman: The design of minimum-cost survivable networks. IEEE Trans. on Circuit Theory, CT-16, 455-460 (1969)
[11] D.B.West: Introduction to graph theory. Prentice Hall of India (2003)
[12] M.C.Golumbic: Algorithmic graph theory and perfect graphs. Academic Press (1980)
[13] R.Tarjan: Depth first search and linear graph algorithms. SIAM Journal of Computing, 1(2), 146-160 (1972)
[14] W.T.Tutte: Connectivity in graphs. University of Toronto Press (1966)
[15] J.E.Hopcroft, R.E.Tarjan: Dividing a graph into triconnected components. SIAM Journal of Computing, 2, 135-158 (1973)
[16] P.Sreenivasa Kumar, C.E.Veni Madhavan: Clique tree generalization and new subclasses of chordal graphs. Discrete Applied Mathematics, 117, 109-131 (2002)
[17] L.Ibarra: Clique separator graph for chordal graphs. Discrete Applied Mathematics, 157 (8), 1737-1749 (2009)