A Novel Data Structure for Biconnectivity, Triconnectivity, and k-tree Augmentation

N.S.Narayanaswamy **N.Sadagopan**

Department of Computer Science and Engineering, Indian Institute of Technology Madras, Chennai-600036, India.
Email: {swamy,sadagopu}@cse.iitm.ac.in

Abstract

For a connected graph, a subset of vertices of least size whose deletion increases the number of connected components is the *vertex connectivity* of the graph. A graph with 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 of connectivity augmentation in 1-connected graphs, 2-connected graphs, and k-trees. For a graph, our data structure maintains the set of equivalence classes based on an equivalence relation on the set of leaves of an associated 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 optimum connectivity augmentation. While this is the first attempt on the study of k-tree augmentation, the study on other two augmentations is reported in the literature. Compared to other augmentations reported in the literature, we avoid recomputation of the associated tree by maintaining the data structure under edge additions.
Keywords: Connectivity, connectivity augmentation, 1-connected graphs, 2-connected graphs, and k-trees.

1 Introduction

Finding optimum 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-connected graph G, find a minimum 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 (Vegh 2010) 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. This study was initiated by Eswaran and Tarjan (Eswaran et al. 1976) and in (Eswaran et al. 1976) they presented biconnectivity augmentation of 1-connected graphs. A different algorithm for the same problem is given in (Hsu et al. 1976, Rosenthal et al. 1977). Watanabe and Nakamura (Watanabe et al. 1988) presented an algorithm for finding a smallest augmentation to triconnect a graph. A linear time algorithm for the same problem is proposed in (Hsu et al. 1991). Hsu (Hsu 2000) presented an algorithm for four connecting a 3-connected graph. For fixed k, Jackson and Jordan presented in (Jackson et al. 2005) a polynomial time algorithm to k-vertex connect a given graph and the run time of the algorithm is $O(n^6)$. For a graph G with arbitrary connectivity, they presented a *min-max* formula involving the number of edges to be augmented to G so that G is k-vertex connected. In what follows, the combinatorial analysis and the complexity of the vertex connectivity augmentation in general are 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 is still open. Recently, Vegh in (Vegh 2010) presented a polynomial time algorithm for optimum $(k+1)$-connectivity augmentation of k-connected graphs. Analogous to vertex connectivity augmentation, edge connectivity augmentation is another well studied problem in the literature. The general k-edge connectivity augmentation problem was solved by Watanabe and Nakamura (Watanabe et al. 1987). The same problem was solved by Cai and Sun (Cai et al. 1989). The other basic augmentation problems namely to make a digraph k-vertex connected (Frank et al. 1995) and a digraph k-edge connected (Frank 1992) are shown to be polynomial time solvable. Frank's survey (Frank 1994) is an excellent source for more results on connectivity augmentation. Interestingly, these problems are also of practical interest in network reliability and fault-tolerant computing (Steiglitz et al. 1969, Frank et al. 1970, Jain et al. 1986).

Our Contribution: Given a graph, our framework first finds the associated tree by understanding the structural decomposition of graphs with respect to vertex separators. For example, given a 1-connected graph G, we construct the associated biconnected component tree where each node is a cut vertex or a biconnected component of G. The tree obtained by this approach is unique. In all three augmentations reported in this paper, we focus our combinatorial analysis on the associated tree to decide an optimum connectivity augmentation set. Our new data structure maintains a partition of the set of leaves of the tree and this partition determines the edge to be augmented to the graph at each iteration. In short, this partition guarantees a recursive subproblem 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 our paper.

To exemplify our framework, we present three case studies. The three problems 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-trees using minimum vertex separator trees. For each of the three augmentations reported here, we first discuss lower bound results for optimum connectivity augmentation, and based on the new framework we provide a new proof of tightness and an elegant linear time algorithm. Further, this paper is the first attempt in studying k-tree augmentation, for any k. While (Hsu et al. 1976, Hsu 2000) presented a linear time algorithm for the other two case studies, our novelty is in the data structure that yields an elegant linear time algorithm and an elegant combinatorial analysis.

Preliminaries: We follow standard graph theoretic definitions and notation, see (West 2003, Golumbic 1980). Let $G = (V, E)$ be an undirected non weighted graph where $V(G)$ is the set of vertices and $E(G) \subseteq \{\{u, v\} \mid u, v \in V(G),\ u \neq v\}$. The *neighborhood* of a vertex v in a graph G is the set $\{u \mid \{u, v\} \in E(G)\}$ and is denoted by $N_G(v)$. The degree of a vertex v, denoted as $deg_G(v)$ is the size of $N_G(v)$. $\delta(G)$ denotes minimum degree in G. $\Delta(G)$ denotes maximum degree 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}. Note that a path on 2 vertices is just an edge. For a connected graph G, a vertex separator of G is a set $S \subseteq V(G)$ such that the induced subgraph, denoted by $G - S$, on the vertex set $V(G) \setminus S$ has more than one connected component. The vertex connectivity of a graph G, written $\kappa(G)$, is the minimum cardinality of a vertex set S (minimum vertex separator) such that $G - S$ is disconnected or has only one vertex. A graph is k-connected if its connectivity is 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, this section presents a new approach to find an optimum augmenting set which makes the tree biconnected. We first discuss the lower bound on the size of 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 new framework also guarantees a tree at each iteration and hence we obtain a recursive sub-problem efficiently. It is now natural to extend tree augmentation approach to augmentation in graphs which have tree like structure. Since trees are a subclass of singly connected graphs, a natural extension is to study augmentation in singly connected graphs by representing them by the associated trees. The standard approach in the literature for biconnectivity augmentation of singly connected graphs is to maintain a tree that captures the biconnected components, cut vertices, and bridges of a singly 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. Our framework avoids recomputation of the associated tree at each iteration and this new approach is fundmentally different from the results reported in (Eswaran et al. 1976, Hsu et al. 1976). 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. In particular our methods naturally extend to connectivity augmentation in k-trees, and this does not seem to be easily feasible using the approach of Tarjan et. al. (Eswaran et al. 1976) and Hsu et. al. (Hsu et al. 1976). We also present a new algorithm for triconnectivity augmentation of biconnected graphs using our new data structure.

Lower Bound on Biconnectivity Augmentation in trees: Given a tree T we now present the lower bound on 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 lowerbound on the optimum biconnectivity augmenting set. 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 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 aug-

menting 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 (Eswaran et al. 1976, Hsu et al. 1976).
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 lesser. The main contribution here is the identification of the equivalence relation which consequently guarantees easy construction of the recursive sub-problem. The equivalence relation and therefore the set of equivalence classes yields an elegant algorithm to compute $E_{min}(G)$, an optimal biconnectivity augmenting set. This approach is fundamentally different from the results presented in (Hsu et al. 1976, Hsu 2002). We present an algorithm by focussing 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 Equivalence Classes: Definition and Structural observations

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 $deg(z') \geq 3$. Proof of Lemma 2.1 is omitted in this paper.

Lemma 2.1 *R is an equivalence relation.*

Since R is an equivalence relation, R induces a set X_{ec} of equivalence classes on the set $L(T)$ of leaves in T. 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 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.

2.2 Augmentation using Equivalence Classes

In this section, we present our linear time algorithm to find an optimum biconnectivity augmenting set 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 (Hsu et al. 1976) and this key observation leads to an optimum algorithm reported in (Hsu et al. 1976). We present our key observation in Lemma 2.3 and this observation yields an elegant combinatorial analysis for finding optimum connectivity augmentation.

Lemma 2.2 *(Hsu et al. 1976) 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 $\Delta(T)$.*

From the above lemma we observe the following result and this observation is used in the proof of Lemma 2.4

Lemma 2.3 *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 $w(X) = z$.

Lemma 2.4 *Let T be a tree such that T is not a star. Then $M_1 = M_0 - 1$.*

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 1(b) Biconnectivity Augmentation in star: *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 1(c) 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 star */`

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 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 2.2 we know that there can be at most two vertices $v_1, v_2 \in V(T)$ of degree Δ^0. Further, from Lemma 2.3 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 2.2), both are associated representatives of two equivalence classes (by Lemma 2.3), 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 2.5 *Let T be a tree. The minimum biconnectivity augmentation of T uses $M_0 = \max\{\lceil \frac{l}{2} \rceil, \Delta - 1\}$ edges.*

Proof The algorithm guarantees that at the end of each iteration we always have a tree. Further, we know from lemma 2.4 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 otherwords, 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 Linear Time Implementation of *non-star-augment()* using a Novel Data Structure

The important steps to be analyzed in our algorithm are computing the set of equivalence classes, finding two equivalence classes such that degree of its representatives are maximum and second maximum, and updating of equivalence classes after edge addition.

We below mention possible situations that may arise on execution of line (3) and line (4) of Algorithm 1(c) 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 *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.

We propose an abstract data type(ADT) to *maintain* the set of equivalence classes. We call the ADT as *Equivalence-Class-List* and using the operations listed in Table 1, we present a linear time implementation of the above reorganization steps. 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 line (2) and line (5), respectively of Algorithm 1(c). These subroutines make use 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[ ]*.

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 size of all equivalence classes is at most the sum of degrees in the tree, the time spent in identifying the augmentation set is $O(n)$. When the tree is 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 Application to Biconnectivity, Triconnectivity and k-tree Augmentations

In this section, we report our study of augmentation in three graph classes using the equivalence classes from the associated tree. We first, generalize tree augmentation results and present an augmentation in 1-connected graphs. Second, we discuss triconnectivity augmentation results in biconnected graphs. Finally, we report augmentation in k-trees which are a natural extension of trees.

3.1 Biconnectivity Augmentation using Equivalence Classes

As mentioned earlier, the approach is to represent a 1-connected graph by an appropriate tree and find an optimum augmentation set of 1-connected graphs by using the results presented in Section 2. 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 $label(x) = S$,

Table 1: Operations Defined on *Equivalence-Class-List* 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

$S \subseteq V(G)$ 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

- $x \in B$ 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 $label(y) = \{v\}$.

An example of a biconnected component tree is shown in Figure 1. **Lower Bound on Biconnectivity Augmentation in 1-connected Graphs:** We use the 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 $\lceil \frac{|X|}{2} \rceil$ 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 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 (Hsu et al. 1976, Eswaran et al. 1976).

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 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 biconnectivity augmentation set in 1-connected graphs. The equivalence relation and therefore the set of equivalence classes yields an elegant algorithm (Algorithm 2) that avoids completely the recomputation of the associated tree at each iteration, unlike, the results presented in (Hsu et al. 1976, Rosenthal et al. 1977). 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 2.2,2.3, and 2.4 are true in biconnected component tree T. We only present a proof of Theorem 3.1.

Theorem 3.1 *Let T be a biconnected component tree. The minimum biconnectivity augmentation of G uses $M_0 = \max\{\lceil \frac{l}{2} \rceil, \Delta_c - 1\}$ edges.*

Proof The algorithm guarantees that at the end of each iteration we always have a tree. Further, we know from lemma 2.4 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* of Section 2 is called p times, after which *bc-star-augment* is called once. Then $M_p = M_0 - p$. In otherwords, 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 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 $M_p = \Delta_c^p - 1$. *bc-star-augment* function biconnects this tree using M_p edges. Therefore, the total number of edges added is $M_p + p = M_0 - p + p = M_0$. 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 $\lceil \frac{l - l_p}{2} \rceil + \lceil \frac{l_p}{2} \rceil = \lceil \frac{l}{2} \rceil$. 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: Depth First Search(DFS) on the given graph can be used to compute the biconnected component tree in $O(n+m)$ time (Tarjan 1972). The number of nodes in the biconnected component tree is $O(n)$. We know from 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 using Equivalence Classes

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 (Hsu et al. 1991). Given a 2-connected graph G, the vertex set of 3-block tree T, $V(T) = \{x \mid label(x)$ is a triconnected component 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 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.

- $x \in \sigma$ 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 (Tutte 1966). Let $\Delta_\sigma(T) = \max_{x \in \sigma} deg(x)$ and l denote the number of leaves in T. A 3-block tree is illustrated in Figure 2. **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 (Watanabe et al. 1988, Hsu et al. 1991). 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 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 (Algorithm 3) for finding triconnectivity augmentation set in 2-connected graphs. This approach is fundamentally different from the results presented in (Watanabe et al. 1988, Hsu 2000).
Sketch of the Algorithm: Compute the set X_{ec} of equivalence classes of 3-block tree T. If $|X_{ec}| \geq 2$ then find 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 three special trees depending on the label of the representative vertex z. z can be β (a triconnected component) or σ (a vertex separator of size 2) or π (a polygon) vertex of T and accordingly we call T as *t-star*, *s-star*, and *p-star*, respectively. Augmentation for tail condition is done separately. With this new approach we simplify our triconnectivity augmentation algorithm by calling *non-star-augment* subroutine with 3-block tree as the input. A proof similar to Theorem 3.1 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. From (Hopcroft et al. 1973) we know that 3-block tree can be constructed in linear time and from Section 2, we know that triconnectivity augmentation set can be obtained in linear time as well. Therefore, overall time complexity to triconnect a biconnected graph is linear in the input size.

3.3 Augmentation in k-trees using Equivalence Classes

Given a k-tree G, we find a minimum set of edges whose augmentation to G makes it $(k+1)$-connected. We represent the given k-tree using a minimum vertex separator tree. For a given k-tree, minimum vertex separator tree is unique and it is different from the standard tree decomposition tree. 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 a k-tree. A vertex $v \in V(G)$ is *simplicial* if $N_G(v)$ induces a clique. Note that in every k-tree, every minimum vertex separator(MVS) is a k-clique and hence k-trees are k-connected. Also every maximal clique is of size $k+1$. For a given k-tree G we construct a Minimum Vertex Separator tree(MVS tree) T associated with G as follows: $M(G) = \{S \subset V(G) \mid S$ is a minimum vertex separator in $G\}$ and $K(G) = \{K \subset V(G) \mid K$ is a $k+1$ clique 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 $x, y \in V(T)$, 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 $label(x) \subset label(y)$.

3.3.1 A lower bound on k-tree augmentation

Let l denote the number of leaves in T and s denote the number of simplicial vertices in G. By our construction, it is easy to see that $l = s$. The degree of a minimum vertex separator $S \in M(G)$ is the number of $k+1$ cliques in G 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 $deg(x) = S^{max}$. A proof of Lemma 3.2 and Lemma 3.3 is omitted in this paper.

Lemma 3.2 *For any optimum solution $E_{min}(G)$ of G, $|E_{min}(G)| \geq \max\{\lceil \frac{s}{2} \rceil, S^{max} - 1\}$.*

For the MVS tree T the above lower bound translates into $\max\{\lceil \frac{l}{2} \rceil, \Delta_k(T) - 1\}$ edges. We observe that the MVS tree is very similar to the biconnected component tree. A node $x \in V_M$ corresponds to an element in B' and $x \in V_K$ corresponds to an element in B. With this observation we simplify our k-tree augmentation algorithm (Algorithm 4) by calling *1-connect-augment* with MVS tree as the input. Since MVS tree can be constructed in linear time from the construction order of k-trees, *ktree-augment()* runs in linear time. Let $A = \{\{x, y\} \mid x, y$ are leaves in $T\}$ be the set of edges returned by our algorithm. Let $E_a(G)$ 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 3.3 *Let G_{aug} be the graph obtained from G by augmenting $E_a(G)$. G_{aug} is $(k+1)$-connected.*

Theorem 3.4 *Let G be a k-tree. An optimum $(k+1)$-connectivity augmentation of G uses $|E_a(G)| = \max\{\lceil \frac{s}{2} \rceil, S^{max} - 1\}$ edges.*

Proof We know that our algorithm augments $|E_a(G)|$ edges. Therefore from Lemma 3.3 it follows that G augmented with $E_a(G)$ is $(k+1)$-connected. Hence the theorem.

References

L.A.Vegh (2010), Augmentating undirected node connectivity by one, *in* 'Proceedings of the 42nd ACM symposium on Theory of computing', pp. 563-572

K.P.Eswaran &R.E.Tarjan (1976), Augmentation problems, SIAM Journal of Computing, 5, 653-665.

T.S.Hsu &V.Ramachandran (1993), On finding a smallest augmentation to biconnect a graph, SIAM Journal of Computing, 22, 889-912.

A.Rosenthal &A.Goldner (1977), Smallest augmentation to biconnect a graph, SIAM Journal of Computing, 6, 55-66.

T.Watanabe &A.Nakamura (1988), 3-connectivity augmentation problems, In Proc. of IEEE Int. Symp. on Circuits and Systems, pp. 1847-1850.

T.S.Hsu &V.Ramachandran (1991), A linear time algorithm for triconnectivity augmentation, In Proc. of 32nd Annual IEEE Symp. on Foundations of Comp. Sci. pp.548-559.

T.S.Hsu (2000), On four connecting a triconnected graph, Journal of Algorithms, 35, 202-234.

B.Jackson &T.Jordan (2005), Independence free graphs and vertex connectivity augmentation, Journal of Combinatorial Theory Series B, 94, 31-77.

T.Watanabe &A.Nakamura (1987), Edge-connectivity augmentation problems, Computer System Sciences, 35(1), 96-144.

G.R.Cai &Y.G.Sun (1989), The minimum augmentation of any graph to k-edge connected graph, Networks, 19, 151-172.

A.Frank &T.Jordan (1995), Minimal edge-coverings of pairs of sets, Journal of Combinatorial Theory Series B, 65, 73-110.

A.Frank (1992), Augmenting graphs to meet edge-connectivity requirements, SIAM J. Dis.Math., 5(1), 22-53.

A.Frank (1994), Connectivity augmentation problems in network design, Mathematical Programming: State of the Art, 34-63.

K.Steiglitz, P.Weiner, &D.J.Klietman (1969), The design of minimum-cost survivable networks, IEEE Trans. on Circuit Theory, CT-16, 455-460.

H.Frank &W.Chou (1970), Connectivity considerations in the design of survivable networks, IEEE Trans. on Circuit Theory, CT-17, 486-490.

S.P.Jain &K.Gopal (1986), On network augmentation, IEEE Trans. on Reliability, R-35, 541-543.

D.B.West (2003), Introduction to graph theory, Prentice Hall of India.

M.C.Golumbic (1980), Algorithmic graph theory and perfect graphs, Academic Press.

T.S.Hsu (2002), Simpler and faster biconnectivity augmentation, Journal of Algorithms, 45, 55-71.

R.Tarjan (1972), Depth first search and linear graph algorithms, SIAM Journal of Computing, 1(2), 146-160.

T.S.Hsu &V.Ramachandran (1991), A linear time algorithm for triconnectivity augmentation, In Proc. of 32nd IEEE symp. on Foundations of Computer Science(FOCS), pp.548-559.

W.T.Tutte (1966), Connectivity in graphs. University of Toronto Press.

J.E.Hopcroft &R.E.Tarjan (1973), Dividing a graph into triconnected components, SIAM Journal of Computing, 2, 135-158.

Algorithm 2 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 $X_{ec} = \{X \mid X$ is an equivalence class with associated vertex $w(X)\}$
  if $|X_{ec}| > 1$ **then**
    `/* T is not a star-like graph */`
    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 graph */`
    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

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 star-like 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 star-like 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

Algorithm 3 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 graph */`
    Y_{ec}=non-star-augment(X_{ec}) `/* Call to non-star-augment() of Section 2, returns` Y_{ec}`*/`
    tsp-star-augment(Y_{ec}) `/* Associated tree is a star with a single equivalence class */`
  else
    `/*T is a star-like graph */`
    tsp-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 β vertex and not elements of σ vertex in G

Triconnectivity Augmentation in stars: *tsp-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 star-like 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 star-like 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

Algorithm 4 k-tree connectivity Augmentation: *ktree-augment(Tree T)*

`/*` T `is the MVS-tree of a` k`-tree` G `*/`
Perform $(k+1)$-connectivity augmentation of G using *1-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

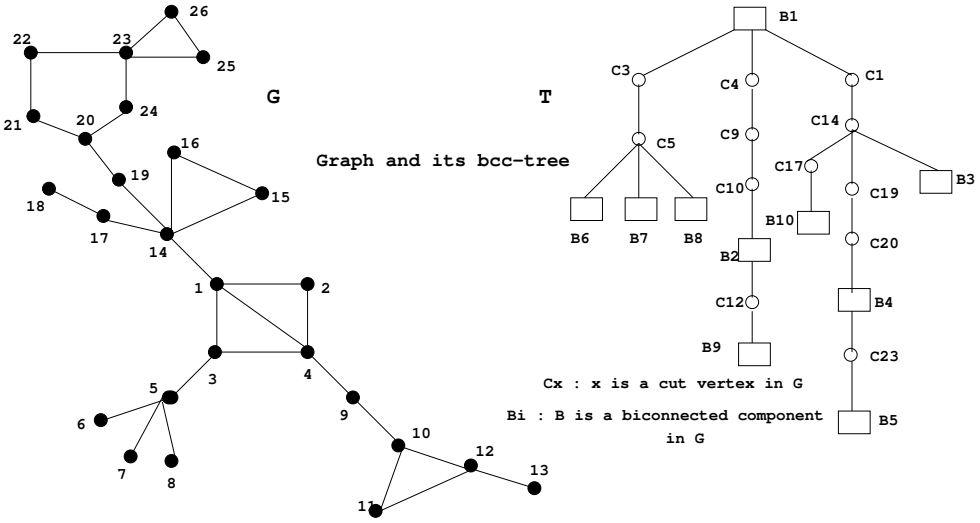

Figure 1: A 1-connected graph and its biconnected component tree

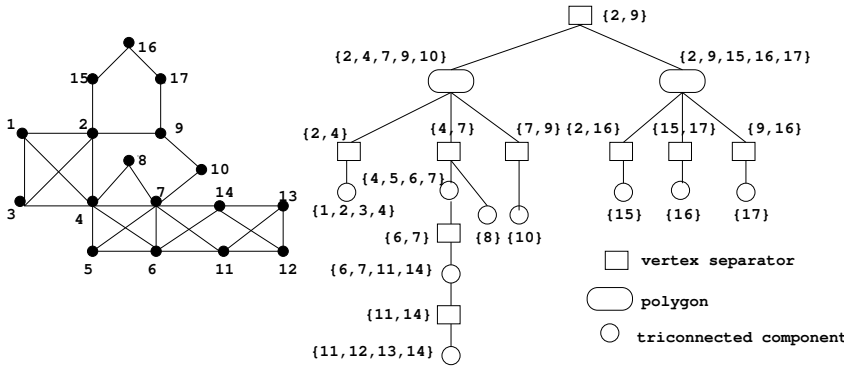

Figure 2: 2-connected graph and its 3-block tree