Computing optimal total vertex covers for trees

Pak Ching Li

Department of Computer Science
University of Manitoba
Winnipeg, Manitoba
Canada R3T 2N2

Abstract. Let $G=(V,E)$ be a simple, undirected, connected graph with at least two vertices and let $w : V \rightarrow \mathbb{Z}^+$ be a weight function on the vertices of $V$. A total vertex cover (TVC) of $G$ is a vertex cover of $G$, with the additional property that each connected component of the subgraph of $G$ that is induced by the vertex cover contains at least two vertices. The total vertex cover problem is to find a total vertex cover of $G$ of minimum weight. We show that a minimum-weight TVC can be computed in polynomial time for weighted trees using a dynamic programming approach. Additionally, we give a greedy algorithm for finding a minimum-cardinality total vertex cover in an unweighted tree.

1 Introduction

Let $G=(V,E)$ be a simple, undirected, connected graph with at least two vertices and let $w : V \rightarrow \mathbb{Z}^+$ be a weight function on the vertices of $V$. A vertex cover (VC) of $G$ is a subset $C$ of $V$, such that for each edge $e = uv$ in $E$, $C \cap \{u,v\} \neq \emptyset$. The vertex cover problem is to find a vertex cover $C$, such that $w(C) = \sum_{v \in C} w(v)$ is minimized. The vertex cover problem is NP-Hard.

For $C \subseteq V$, let $G[C]$ denote the subgraph of $G$ induced by $C$. A total vertex cover (TVC) of $G$ is a vertex cover $C$, such that each of the connected components of $G[C]$ contains at least two vertices. Clearly, every total vertex cover is a vertex cover, but not every vertex cover is a total vertex cover. The (weighted) total vertex cover problem is to find a total vertex cover of $G$ of minimum weight. A minimum-weight total vertex cover of $G$ is called an optimal total vertex cover of $G$. If we ignore the weight function $w$, then the (unweighted) total vertex cover problem is to find a total vertex cover of $G$ of minimum cardinality.

It is known that the problem of computing a minimum-cardinality total vertex cover in a graph is NP-Hard [5]. On the other hand, it can easily be shown that for a cycle of $n$ vertices, a minimum-cardinality TVC has size $\lceil \frac{2}{3}n \rceil$ and an optimal solution can be constructed in polynomial time.

In this paper, we show that computing an optimal TVC for trees can be done in polynomial time using dynamic programming. Additionally, we give a simple greedy algorithm for finding a minimum-cardinality TVC in an unweighted tree. To the best of our knowledge, this is the first time weighted total vertex covers have been investigated.
1.1 Related Work

In 2006, (unweighted) total vertex coverings were investigated by Fernau and Manlove [5]. They attributed the concept of total vertex coverings to a personal communication from Jean Blair in 2001. Additionally, they showed that the TVC problem is \( \text{NP-Hard} \) and there is no \( \alpha \)-approximation algorithm with \( \alpha < 10\sqrt{5} - 21 \), unless \( \text{P} = \text{NP} \). They also showed that the TVC problem is also \( \text{NP-Hard} \) for planar bipartite graphs with maximum degree 3 and gave an exact \( \mathcal{FPT} \) algorithm for the TVC problem with time complexity \( O^*(2.3655^k) \).

In 2010, Klostermeyer [7] gave upper bounds in terms of vertex covers. Dutton [1] related bounds for TVCs to those for vertex covers, connected vertex covers, dominating sets, total dominating sets and edge dominating sets. Also in 2010, Fernau et al. [4] showed that the \( \text{TVC} \) problem has no polynomial kernel unless the Polynomial Hierarchy collapses to the third level.

In 2012, Fernau [3] gave an exact \( \mathcal{FPT} \) algorithm with time complexity \( O^*(1.151^k) \), where \( k \) is some bound on an optimal solution. This algorithm computes an approximate solution within a factor of 1.5 of an optimal solution.

2 An algorithm for unweighted trees

In this section, we deal with the special case where the nodes of a tree are unweighted. The TVC problem becomes one of finding a TVC of minimum cardinality. We give a simple greedy algorithm for solving this problem for unweighted trees.

Let \( T = (V, E) \) be a tree rooted at a leaf node of \( T \). We may assume without loss of generality that \( T \) is a rooted tree whose root node has degree one, and the height of the rooted tree is at least two. The second condition eliminates the need to deal with trivial scenario where \( T \) consists of a single edge. The basic idea of the algorithm is to perform a depth-first traversal on \( T \) and as vertices are visited, a greedy criteria is applied to determine if a node should be placed in the solution being constructed. The algorithm is given by Algorithm 1. To compute an optimal TVC using Algorithm 1, the subroutine \( \text{TVC-Tree} \) is to be invoked with input tree \( T \) and a designated root node \( r \), which is a leaf node of \( T \).

The algorithm starts at the root node of \( T \) and performs a depth-first traversal. When it returns from visiting a leaf node, it puts the current node into the cover \( S \) (line 4-5). Since the height of \( T \) is at least two, we can ignore leaf nodes as candidates for being in an optimal TVC.

Consider some non-leaf node \( x \). After returning from processing a child \( c \) of \( x \), the algorithm checks to see if \( c \) is in the cover \( S \), but is not adjacent to another vertex in \( S \). If this is the case, \( x \) is placed into \( S \) (line 8-10). This ensures that \( S \) will not have any isolated connected components. In addition, the algorithm checks to see if neither \( c \) nor \( x \) is in \( S \). If this is the case, the algorithm places \( x \) in \( S \) (lines 11-13). This ensures that the edge \( \{x, c\} \) is covered.

When the subroutine call to the \( \text{Traverse} \) routine on line 19 returns, there is a possibility that the root node \( r \) is in \( S \), but its sole child \( d \) is not in returned
Algorithm 1 Compute a minimum-cardinality TVC for input tree $T$

\[\downarrow \text{assume that } T \text{ is a tree rooted at a leaf node } r\]

1: \textbf{procedure} Traverse$(T, x, S)$
2: \hspace{1em} $C \leftarrow$ the set of children of $x$, in the tree $T$
3: \hspace{1em} \textbf{for all } $c \in C$ \textbf{do}
4: \hspace{2em} \text{if } $c$ \text{ is a leaf node} \text{ then}
5: \hspace{3em} $S \leftarrow S \cup \{x\}$
6: \hspace{2em} \text{else}
7: \hspace{3em} \text{Traverse}$(T, c, S)$
8: \hspace{2em} \text{if } $c \in S$ and $\Gamma_T(c) \cap S = \emptyset$ \text{ and } $x \notin S$ \textbf{then}
9: \hspace{3em} $S \leftarrow S \cup \{x\}$  \hspace{1em} $\triangleright$ Ensure $S$ will be a TVC
10: \hspace{2em} \text{end if}
11: \hspace{2em} \text{if } $c \notin S$ \text{ and } $x \notin S$ \textbf{then}
12: \hspace{3em} $S \leftarrow S \cup \{x\}$ \hspace{1em} $\triangleright$ Ensure $S$ will be a vertex cover
13: \hspace{2em} \text{end if}
14: \hspace{2em} \text{end if}
15: \hspace{1em} \textbf{end for}
16: \textbf{end procedure}

17: \textbf{procedure} TVC – Tree$(T, r)$
18: \hspace{1em} $S \leftarrow \emptyset$  \hspace{1em} $\triangleright$ S will contain the TVC
19: \hspace{1em} \text{Traverse}$(T, r, S)$
20: \hspace{1em} Let $d$ denote the sole child of $r$
21: \hspace{1em} \text{if } $r \in S$ \text{ and } $d \notin S$ \textbf{then} \hspace{1em} $\{r\}$ is a connected component in $T[S]$
22: \hspace{1em} $S \leftarrow S \setminus \{r\} \cup \{d\}$
23: \hspace{1em} \text{end if}
24: \hspace{1em} \textbf{return} $S$
25: \textbf{end procedure}

solution $S$. In that case, lines 21-23 modify $S$ so that it will be a total vertex cover. Figure 1 shows the result of running Algorithm 1 on a sample instance. The total vertex cover that is constructed is the vertices in white.

**Theorem 1.** Algorithm 1 returns an optimal total vertex cover. That is, it returns a minimum-cardinality total vertex cover.

**Proof.** It is easy to see that $S$ is a total vertex cover, since lines 11-13 ensure that $S$ is a vertex cover and lines 8-10 ensure that $T[S]$ does not contain isolated connected components, except for possibly the root node of $T$. In the case that the root node is isolated in $T[S]$, lines 20-23 remove the root node from $S$ and place its sole neighbour $d$ into $S$. Therefore, $S$ is a total vertex cover by line 24 of the algorithm.

Let $OPT$ denote an optimal total vertex cover of the $T$. Our goal is to show $|S| \leq |OPT|$, where $S$ is the set of vertices returned by the the traverse routine (line 19). We accomplish this by defining an injective function $f : S \rightarrow OPT$. Clearly, if such a function exists, then the set $S$ (at line 24 of the algorithm) is an optimal total vertex cover.
By the way the algorithm works, a vertex \( v \) is selected to be in \( S \) if and only if one of the following two scenarios occur:

1. to cover an edge that is not covered, or
2. to ensure the \( T[S] \) does not have isolated connected components.

Consider an arbitrary node \( v \in S \). If \( v \in \text{OPT} \), define \( f(v) = v \). Otherwise, we have that \( v \notin \text{OPT} \). Then we consider the reason why \( v \) was selected to be in \( S \) by the algorithm.

1. Suppose \( v \) was selected to cover an uncovered edge. Then there must exist a child \( u \) of \( v \) that was visited by the algorithm previously, but was not selected to be in \( S \). Since \( v \notin \text{OPT} \) and \( \text{OPT} \) is a vertex cover, we see that \( u \in \text{OPT} \). In this case, we define \( f(v) = u \).

2. Suppose \( v \) was selected to ensure that \( T[S] \) does not have isolated connected components. Then there is a child \( w \) of \( v \) that has previously been selected by the algorithm to be in \( S \) at the time the algorithm selects \( v \) to be in \( S \). Note that \( v \in S \setminus \text{OPT} \) and \( w \in S \cap \text{OPT} \) (the vertex \( w \in \text{OPT} \), because the edge \( vw \) must be covered by \( \text{OPT} \)). Since \( \text{OPT} \) is a TVC and \( w \) is isolated in \( T[S] \) at the time \( w \) was selected to be in \( S \), there is a child \( u \) of \( w \) such that \( u \in \text{OPT} \setminus S \). Therefore we define \( f(v) = u \).

We now show that \( f \) is injective. By definition of \( f \), \( f(v) \in \text{OPT} \setminus S \) if and only if \( v \in S \setminus \text{OPT} \). Since \( f(v) = v \) for \( v \in \text{OPT} \cap S \), we only need to consider the scenario where \( f(v) = u \), where \( v \in S \setminus \text{OPT} \) and \( u \in \text{OPT} \setminus S \).

If \( u \) is a child of \( v \), then the only other possible member of \( S \) that could possibly be mapped to \( u \) is the parent, \( p(v) \), of \( v \). For this to happen, we must have \( p(v) \in S \setminus \text{OPT} \) and \( v \in S \cap \text{OPT} \). But this contradicts the assumption that \( v \notin \text{OPT} \). Therefore \( v \) is the only node that maps to \( u \) in this case.

Otherwise, \( v \) is mapped to a grand-child \( u \). Then there is a \( w \in S \cap \text{OPT} \) that is a child of \( v \) and the parent of \( u \). The only node in \( S \), other than \( v \), that
could possibly be mapped to \( u \) by \( f \) is \( w \). But since \( w \in S \cap OPT \), \( f(w) = w \) and therefore \( v \) is the only node that maps to \( u \).

Thus we have shown that \( f \) is injective. As for the running time, the algorithm does a depth-first traversal of the tree. Let \( n \) denote the number of nodes in the tree. There are \( O(n) \) set operations each taking \( O(n) \) steps. Therefore, the algorithm takes \( O(n^2) \) steps.

3 A dynamic programming algorithm for weighted trees

In this section, we give an efficient dynamic programming algorithm for finding the minimum-cost TVC of a weighted tree. Given a weighted tree \( T = (V,E) \) with weight function \( w : V \to \mathbb{Z}^+ \), we say that a node \( v \in V \) is a support node if \( v \) is adjacent to one or more leaf nodes of \( T \). Let us assume without loss of generality that \( T \) is a rooted tree whose root node is denoted by \( r \). From a high level, the algorithm consists of several steps:

1. For each support node \( x \), we remove all but one of the leaf nodes adjacent to \( x \). We keep the leaf node with the smallest weight. Note that the non-leaf children of \( x \) are untouched.
2. Relabel the vertices of the tree using a function \( \text{label} : V \to \{1, 2, ..., n\} \) so that if \( \text{depth}(u) > \text{depth}(v) \) then \( \text{label}(u) < \text{label}(v) \). The depth of a vertex \( v \) in the tree \( T \) is denoted by \( \text{depth}(v) \).
3. Apply dynamic programming to find an optimal TVC in the tree \( T \).

In the first step of the algorithm, we prune all but one of the leaf nodes adjacent to \( x \). We keep the leaf node with the smallest weight. Note that the non-leaf children of \( x \) are untouched.

In the second step of the algorithm, we relabel the nodes of the rooted tree so that if \( u \) has depth \( i \) and \( v \) has depth \( j \) and \( i < j \), then \( \text{label}(u) > \text{label}(v) \). This can be accomplished by performing a breadth-first search starting at the root. Each node that is visited is labeled with a value that is one less than the value assigned to the previous node. The value assigned to the root node is \( |V| \).

The purpose of the first step is to ensure we do not consider vertices that will never be in an optimal solution. The purpose of the second step is to ensure that we have a way of building up solutions to problems from solutions to sub-problems. From a high level, it makes sense to process subtrees rooted at nodes in decreasing order of depth. The labelling of the vertices allows us to do this and to build the tables used in the dynamic programming algorithm in a consistent manner. The dynamic programming algorithm is given in the following section.
Fig. 2. The figure on the left is the original tree. The figure on the right is the result of pruning leaf nodes from support nodes.

3.1 Dynamic programming

We now give our dynamic programming algorithm for computing the weight of an optimal TVC. For each non-leaf node $v$, let $c(v)$ denote the set of children of $v$. For a leaf node $v$, set $c(v) = \emptyset$. For $S \subseteq V$, let $w(S) = \sum_{v \in V} w(v)$. Let $T_v$ denote the subtree of $T$ rooted at node $v$. The key observation is that a minimum-cost TVC of $T$ will either contain the root or will not contain the root. For each node $v$ in $T$, we define three sets $S_v^1$, $S_v^2$, and $S_v^3$ as follows:

1. $S_v^1$ is a minimum-weight subset of vertices of $T_v$ such that $v \in S_v^1$ and $S_v^1 \setminus \{v\}$ is a TVC of $T_v$.
2. If $x$ is a non-leaf node of $T$, $S_v^2$ is a minimum-weight subset of vertices of $T_v$ such that $v \in S_v^2$ and $S_v^2$ is a TVC of $T_v$. If $x$ is a leaf-node of $T$, then let $S_v^2 = \{v\}$, and
3. $S_v^3$ is a minimum-weight subset of vertices of $T_v$ such that $v \notin S_v^3$ and $S_v^3$ is a TVC of $T_v$.

Note that if $v$ is a support node of $T$, then $S_v^3$ is not defined, and we denote this by setting $S_v^3 = \infty$ and setting the weight of this solution to $\infty$. Also, note that $S_v^1$ is not necessarily a TVC. However, we need to compute $S_v^1$, because it is possible that $S_v^1 \cup \{\text{parent}(v)\}$ is a minimum-cost TVC for $T_{\text{parent}(v)}$, where $\text{parent}(v)$ denotes the parent node of $v$. For each leaf node $v$, we set $S_v^1 = \{v\}$,
Fig. 3. A relabelling of the nodes of the right tree from Figure 2.

$S_v^2 = \{v\}$ and $S_v^3 = \emptyset$. Clearly setting each of the three sets in this manner satisfies the conditions of these sets.

From a high level, the dynamic programming algorithm will compute $S_v^1, S_v^2$ and $S_v^3$ from the bottom of the tree up to the root of the tree. By definition of $S_r^2$ and $S_r^3$, a minimum-weight TVC of $T = T_r$ will be the smaller of $S_r^2$ and $S_r^3$.

We can now show how $S_v^1, S_v^2$ and $S_v^3$ can be formed, based on $S_u^1, S_u^2$ and $S_u^3$, where $u \in c(v)$.

**Forming $S_v^3$**: First, consider a node $v \in V(T)$ that is not a support node. A minimum-weight TVC $S$ of $T_v$ with the property that $v \notin S$ satisfies the following conditions:

1. For each child $u \in c(v)$, $S$ must contain a minimum-weight TVC of the subtree $T_u$ and
2. $u \in S \cap V(T_u)$.

The first condition must hold, for otherwise we can replace $S \cap V(T_u)$ with a TVC of $T_u$ of smaller weight, therefore contradicting the optimality of $S$. The second condition must hold, for otherwise the edge $\{u, v\}$ will not be covered by $S$. The smallest TVC of $T_u$ that contains the node $u$ is, by definition, $S_u^2$. Therefore, we see that we can set $S_v^3 = \cup_{u \in c(v)} S_u^2$ in order for $w(S) = w(S_v^3)$.

Now, consider the case where $v$ is a support node of $T$, then we can set $S_v^3 = \infty$, as there is no TVC of $T_v$ that does not contain $v$ in this case.

**Forming $S_v^1$**: Consider a node $v \in V(T)$ and let $S$ denote a minimum-weight set, such that $v \in S$ and $S \setminus \{v\}$ is a TVC of $T_v \setminus \{v\}$. For each child $u \in c(v)$, $S$ contains a TVC of $T_u$ and either $u \in S$ or $u \notin S$. Since $S$ has minimum weight, then
for each $v \in c(v)$, we must have that the TVC of $T_v$ has to have minimum weight. Therefore, $T_v$ contains a TVC whose weight is the smaller of $w(S_2^v)$ and $w(S_3^v)$. For each $u \in c(v)$, let $i_u \in \{2, 3\}$ be such that $w(S_{i_u}^v) = \min\{w(S_2^v), w(S_3^v)\}$. Then we see that we can set $S_1^v = \bigcup_{u \in c(v)} S_{i_u}^v \cup \{v\}$ in order for $w(S) = w(S_1^v)$. Note that if $u$ is a leaf node of $T$. Then the TVC of $T_v$ selected is $S_3^u = \emptyset$.

**Forming $S_2^v$:** Consider a node $v \in V(T)$ and let $S$ denote a minimum-weight set, such that $v \in S$ and $S$ is a TVC of $T_v$. Since $S$ is a TVC of $T_v$, $S$ must contain at least one of $v$'s children and for each child $u$ of $v$ contained in $S$, it is possible that none of $u$'s children are in $S$. For each $u \in c(v)$, consider $S \cap T_u$. Since $S$ is a minimum-weight TVC of $T_v$ containing $v$ and by the previous statement, it must be that $w(S \cap T_u)$ is at least $\min\{w(S_1^v), w(S_2^v), w(S_3^v)\}$. For each $u \in c(v)$, let $S_{i_u}^v$ be a member of $\{S_1^v, S_2^v, S_3^v\}$ having minimum weight. Suppose there exists at least one $u$, such that $S_{i_u}^v \in \{S_1^v, S_2^v\}$. Since $S$ has minimum-weight, then it is easy to show that we must have that $w(S) = w(\bigcup_{u \in c(v)} S_{i_u}^v \cup \{v\})$. In this scenario, we can set $S_2^v = \bigcup_{u \in c(v)} S_{i_u}^v \cup \{v\}$ in order for $w(S) = w(S_2^v)$. Otherwise, we have the condition that $S_{i_u}^v = S_3^v$, for every $u \in c(v)$. In this scenario, let $u' \in c(v)$, such that the value $\min\{w(S_1^{v'}), w(S_2^{v'})\} - w(S_3^{v'})$ is minimized. Let $S'$ be equal to $S_1^{u'}$ or $S_2^{u'}$, whichever has smaller weight. Since $S$ has minimum-weight, then it is easy to show that we must have that $w(S) = w(\bigcup_{u \in c(v) \setminus \{u'\}} S_3^u \cup S' \cup \{v\})$. Therefore, we can set $S_2^v = \bigcup_{u \in c(v) \setminus \{u'\}} S_3^u \cup S' \cup \{v\}$ in order for $w(S) = w(S_1^v)$.

By definition of $S_1^v$, $S_2^v$ and $S_3^v$, an optimal weighted total vertex cover of the subtree $T_v$ is given by either $S_2^v$ or $S_3^v$, whichever set has smaller weight. The dynamic-programming solution is given by Algorithm 2. To give an upper bound on the algorithm’s run time, we note that the loop in line 5 of the algorithm executes $n$ times, where $n$ is the number of nodes in the tree and each of the loops in lines 16 and 20 requires at most $O(n)$ steps. The actual construction of the table members (lines 14, 19, 24, 28) can be done in $O(n)$ steps. Therefore, the algorithm takes $O(n^2)$ steps. We summarize these results in the following theorem.

**Theorem 2.** Algorithm 2 computes a minimum-weight TVC for a tree in $O(n^2)$ steps.

Consider the TVC instance given in Figure 3. Associated with each node is two numbers $v/w$, where $v$ is the label of the node and $w$ is the weight of the node. Table 3.1 shows the values of the sets $S_1^v, S_2^v$. Since both $S_2^{10}$ and $S_2^{11}$ has weight 18, then either set gives an optimal TVC. Note that $S_3^{10}$ is a feasible TVC, but not an optimal TVC. In general, $S_1^v$ may not be a TVC and if it is a TVC, its weight is at least as great as that of $S_2^v$. 
4 Conclusions

In this paper, we gave two polynomial-time algorithms for finding optimal total vertex covers in trees. A greedy algorithm was given that works for unweighted trees and a dynamic programming algorithm was given that works for weighted trees.

There remains much to be investigated in regards to the TVC problem. In fact, not much is known about total vertex covers. For example, it is not known whether an efficient algorithm for computing a minimum cardinality TVC in a chordal graph exists. Gavril [6] showed that computing a minimum vertex cover for chordal graphs can be done in polynomial time. A vertex cover is connected if the subgraph induced by the vertex cover is connected. Escoffier, et al. [2] showed that computing a minimum connected vertex cover for chordal graphs can also be done in polynomial time. Since a total vertex cover is more restrictive than a vertex cover but less restrictive than a connected vertex cover, we believe that computing a minimum total vertex cover for chordal graphs can be done in polynomial time.

<table>
<thead>
<tr>
<th>v</th>
<th>$S_1^v$</th>
<th>$w(S_1^v)$</th>
<th>$S_2^v$</th>
<th>$w(S_2^v)$</th>
<th>$S_3^v$</th>
<th>$w(S_3^v)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{1}</td>
<td>2</td>
<td>{1}</td>
<td>2</td>
<td>∅</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>{2}</td>
<td>3</td>
<td>{1, 2}</td>
<td>5</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>3</td>
<td>{3}</td>
<td>4</td>
<td>{3}</td>
<td>4</td>
<td>∅</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>{1, 2, 4}</td>
<td>6</td>
<td>{2, 4}</td>
<td>4</td>
<td>{1, 2}</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>{5}</td>
<td>3</td>
<td>{3, 5}</td>
<td>7</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>6</td>
<td>{6}</td>
<td>1</td>
<td>{6}</td>
<td>1</td>
<td>∅</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>{1, 2, 7}</td>
<td>9</td>
<td>{2, 4, 7}</td>
<td>8</td>
<td>{2, 4}</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>{3, 5, 8}</td>
<td>15</td>
<td>{5, 8}</td>
<td>11</td>
<td>{3, 5}</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>{2, 4, 9}</td>
<td>6</td>
<td>{2, 4, 6, 9}</td>
<td>7</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>10</td>
<td>{2, 3, 4, 5, 6, 9, 10}</td>
<td>19</td>
<td>{2, 3, 4, 5, 9, 10}</td>
<td>18</td>
<td>{2, 4, 5, 6, 8, 9}</td>
<td>18</td>
</tr>
</tbody>
</table>

Table 1. Results of Algorithm 2 on input given by Figure 4.
References


Algorithm 2 Compute a minimum-weight TVC for tree $T$

▷ assume that $T$ is a tree rooted at a node $r$
1: procedure TVC – Weighted – Tree($T = (V, E)$, $w: V \to \mathbb{Z}^+$, $r$)
2: For each support on $r$, prune excessive leaf nodes.
3: Relabel the nodes using values $\{1, 2, \ldots, |V|\}$ so that if $u$ has depth $i$ and $v$ has depth $j$ in $T$ and $i < j$, then $\text{label}(u) > \text{label}(v)$.
4: $v \leftarrow 1$
5: while $v \leq |V|$ do
6: if $v$ is a leaf node then  ▷ Base Case
7: $S_1^v \leftarrow \{v\}$
8: $S_2^v \leftarrow \emptyset$
9: $S_3^v \leftarrow \emptyset$
10: else
11: if $v$ is a support node then  ▷ Compute $S_3^v$
12: $S_1^v \leftarrow \infty$
13: end if
14: for all $u \in c(v)$ do  ▷ Compute $S_1^v$
15: Let $i_u \in \{2, 3\}$ be such that $w(S_{i_u}^u) = \min\{w(S_2^u), w(S_3^u)\}$.
16: end for
17: $S_1^u \leftarrow \bigcup_{u \in c(v)} S_{i_u}^u \cup \{v\}$
18: for all $u \in c(v)$ do  ▷ Compute $S_2^v$
19: Let $S_{i_u}^u$ be a member of $\{S_1^u, S_2^u, S_3^u\}$ having minimum weight.
20: end for
21: if there exists at least one $u$ such that $S_{i_u}^u \in \{S_1^u, S_2^u\}$ then
22: $S_2^v \leftarrow \bigcup_{u \in c(v) \setminus \{u\}} S_{i_u}^u \cup \{v\}$
23: else
24: Let $u' \in c(v)$ such that the value $\min\{w(S_1^{u'}), w(S_2^{u'})\}$ is minimized.
25: $S_2^v \leftarrow \bigcup_{u \in c(v) \setminus \{u'\}} S_{i_u}^u \cup \{v\}$
26: end if
27: end if
28: $v \leftarrow v + 1$
29: end while
30: $w_1 \leftarrow w(S_1^r)$
31: $w_2 \leftarrow w(S_2^r)$
32: if $w_1 \leq w_2$ then
33: $S \leftarrow S_1^r$
34: else
35: $S \leftarrow S_2^r$
36: end if
37: return $S$
41: end procedure