A Primal-Dual based Distributed Approximation Algorithm for Prize Collecting Steiner Tree

Parikshit Saikia, Sushanta Karmakar
s.parikshit@iitg.ernet.in, sushantak@iitg.ernet.in
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati, India, 781039

October 20, 2017

Abstract

Constructing a steiner tree of a graph is a fundamental problem in many applications. Prize collecting steiner tree (PCST) is a special variant of the steiner tree problem and has applications in network design, content distribution etc. There are a few centralized approximation algorithms [12, 21, 7] for solving the PCST problem. However no distributed algorithm is known that solves the PCST problem with non-trivial approximation factor. In this work we present a distributed algorithm that constructs a prize collecting steiner tree for a given connected undirected graph with non-negative weight for each edge and non-negative prize value for each node. Initially each node knows its own prize value and weight of each incident edge. Our algorithm is based on primal-dual method and it achieves an approximation factor of \((2 - \frac{1}{n})\) of the optimal. The total number of messages required by our distributed algorithm to construct the PCST for a graph with \(|V|\) nodes and \(|E|\) edges is \(O(|V|^2 + |E||V|)\). The algorithm is spontaneously initiated at a special node called the root node and when the algorithm terminates each node knows whether it is in the prize part or in the steiner tree of the PCST.

Keywords: steiner tree, prize collecting steiner tree, distributed approximation, primal-dual
1 Introduction

Minimum spanning tree problem is a fundamental problem in graph algorithm design. Given a connected graph $G = (V, E)$ and a function $w : E \rightarrow \mathbb{R}^+$, the goal of the MST problem is to find a subgraph $H = (V, E')$ of $G$ such that $\sum_{e \in E'} w(e)$ is minimized. There are many centralized [31, 37] and distributed algorithms [20] for MST construction. Steiner tree problem is a generalization of the MST problem. The definition of steiner tree is as follows: given a connected graph $G = (V, E)$ and a function $w : E \rightarrow \mathbb{R}^+$, and a set of vertices $Z \subseteq V$ known as the set of terminals, the goal of the steiner tree problem is to find a subgraph $H = (V', E')$ of $G$ such that $\sum_{e \in E'} w(e)$ is minimized subject to the condition that $Z \subseteq V' \subseteq V$. Both problems have many applications such as VLSI layout design [32], communication network [18], transportation network [35], perfect phylogeny in bioinformatics [10] etc. It is known that MST problem can be solved in polynomial time, however steiner tree problem is an NP-hard optimization problem [29]. Therefore many polynomial time approximation algorithms have been proposed for the steiner tree problem [3, 11, 4, 39, 36, 40] with various approximation factors and other performance guarantees. Byrka et al. [13] proposed a polynomial time approximation algorithm for the steiner tree problem for a general graph which has the best known approximation factor of $\ln 4 + \epsilon \approx 1.386 + \epsilon$, for $\epsilon > 0$. However it is a centralized algorithm and uses the technique of iterative randomized rounding of LPs. It is also known that the steiner tree problem for general graph cannot be solved in polynomial time with an approximation factor $\leq \frac{96}{95}$ [16].

There are many variations of the steiner tree problem. Some of them are directed steiner tree problem [47, 1, 15], metric steiner tree problem [41, 36], euclidian steiner tree problem [8, 45], rectilinear steiner tree problem [24, 27, 3, 11, 36], steiner forest problem [2, 33, 23, 46], and so on. The website [26] gives a continuously updated state of the art results for many variants of the problem. Out of the many variants, we focus on a special variant named “prize collecting steiner tree problem”.

Prize Collecting Steiner Tree

Definition 1.1 Given a connected weighted graph $G = (V, E, p, w)$ where $V$ is the set of vertices, $E$ is the set of edges, $p : V \rightarrow \mathbb{R}^+$ is a non-negative prize function and $w : E \rightarrow \mathbb{R}^+$ is a non-negative weight function, the goal is to find a tree $T' = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E$ that minimizes the following function:

$$GW(T') = \sum_{e \in E'} w(e) + \sum_{v \in V'} p(v)$$

Prize collecting steiner tree problem (PCST) involves situations in which various demand points (nodes) need to form a structure with minimum total connection cost. Each node has some non-negative prize associated with it. If some of the demand points are too expensive to connect then it is better not to include them in the structure and instead pay a penalty. Suppose a company wants to build an optical fibre network to provide broadband internet service to various customers.
In this case the network can be modeled as a graph considering street segments as the set of edges, and street segment endpoints as the set of vertices (aka customer locations). The cost of the edges are the installation costs of the cables. An optimal design of this network needs to take care of two objectives, (i) connect to a set of customers that is as large as possible and (ii) connection cost is minimum. In doing so some customers may be excluded from the structure as they may incur more connection cost. For such customers, the company pays a penalty which is proportional to the prize of the node. Therefore, the overall goal is to decide a subset of customers that should be connected so that the sum of the connection cost and the total penalty (for not connecting some customers) is minimized. Similarly many other practical problems like protein protein interaction network [17, 42], leakage detection system [38], image segmentation [43] etc. can be modeled as a PCST problem. It is clear that PCST problem is a further generalization of the well known steiner tree problem and therefore it is also an NP-hard optimization problem.

In this work we propose a distributed algorithm for constructing a PCST for a given graph with vertex and edge weights. It can have application in content distribution network formation or structuring an unstructured network. Any utility distribution company (video, gas, or electricity etc.) will find applicability of PCST in cost optimization. Unlike a centralized algorithm, the entire graph information is not available to any node in a distributed computing setting. Here each node is a computing entity and can communicate with its neighbors only. Each node locally decides whether it will belong to the steiner tree component or the prize components. Note that any node must belong to one of the two mentioned components in a mutually exclusive manner. Our distributed algorithm uses primal-dual technique to construct a PCST with an approximation factor of $(2 - \frac{1}{n-1})$ where $n$ is the number of nodes. Also it incurs $O(|V|^2 + |E||V|)$ message complexity. The correctness of the algorithm is proved and the goodness of the algorithm is justified by analyzing the approximation factor. We believe that our algorithm is the first distributed approximation algorithm for PCST resulting non-trivial approximation guarantee.

The rest of the work is organized as follows. Section 2 contains the works related to the PCST problem. In section 3 we introduce the system model. In section 4 the PCST problem is formulated using integer programming (IP) and linear programming (LP) and we briefly describe the centralized PCST algorithm proposed by Goemans and Williamson [21]. High level description of our distributed PCST algorithm and an example is given in section 5. In section 6 the proof of correctness of our proposed distributed PCST algorithm is explained and pseudo-code of the algorithm is provided in the appendix. Section 7 concludes the paper with future directions.

2 Related Work

The first centralized approximation algorithm for PCST was given by Bienstock et al. [12] in 1993, although a related problem named prize collecting travelling salesmen problem (PCTSP) was introduced earlier by Balas [9]. Bienstock et al. achieved an approximation factor of 3 by using linear programming (LP) relaxation technique. Two years later, based on the work of Agrawal, Klein and Ravi [2], Goemans and Williamson [21, 22] proposed a primal-dual algorithm using the LP
relaxation which runs in $O(n^3 \log n)$ time. The algorithm proposed by Goemans and Williamson consists of two phases namely growing phase and pruning phase and yields a solution of factor $(2 - \frac{1}{n-1})$ of optimality. This algorithm is often denoted as GW-algorithm.

Johnson et al. [28] proposed an improved version of the GW-algorithm that also contains two phases namely growth phase and strong pruning phase. The running time is improved to $O(n^2 \log n)$ maintaining the same approximation factor $(2 - \frac{1}{n-1})$ as of the GW-algorithm. This is achieved by enhancing the pruning phase of GW-algorithm which is termed as strong pruning. Johnson et al. also presented a review of different PCST related problems. The original growth phase of GW-algorithm is modified so that the algorithm works without a root node.

However result of Johnson et al. [28] was shown to be incorrect by Feofiloff, Fernandes, Ferreira and De Pina [19]. They proved it by a counter example that the algorithm proposed by Johnson et al. returns an approximation factor of 2 instead of $(2 - \frac{1}{n-1})$. They introduced a new algorithm for PCST based on the GW-algorithm having a different LP formulation. They achieved a solution of $(2 - \frac{2}{n})$ approximation factor for the unrooted version of the PCST whose running time is $O(n^2 \log n)$. Archer et al. [7] provided a $(2 - \epsilon)$-approximation algorithm for the PCST problem. Specifically the approximation ratio of this algorithm for PCST is below 1.9672. They achieved this by using the improved steiner tree algorithm of Byrka et al. [13] as a black box in their algorithm.

The “quota” version of PCST problem was studied by Haouari et al. [25] for tree graphs in which the goal is to find a subtree that includes the root node and collects a total prize not smaller than the specified quota, while minimizing the cost of the PCST. Although the problem remains NP-hard for the tree-graph, the authors managed to give a pseudo-polynomial time algorithm for the problem. A polynomial time algorithm for PCST was given by Miranda et al. [5] for a special network called 2-tree graph where prizes (node weights) and edge weights belong in a given interval. This result is based on the work of Wald and Colbourn [44] who proved that steiner tree problem is polynomial time solvable on 2-tree graphs. An algorithm for robust prize collecting steiner tree problem was proposed by Miranda et al. [6]. There are other approaches for solving the PCST problem. Canuto et al. [14] gave a multi-start local search based algorithm for the PCST problem. Klau et al. [30] provided an evolutionary algorithm for the same problem. All of these are centralized algorithms for PCST. To the best of our knowledge there exists no published work on solving the PCST problem under distributed setting. Our work in this paper provides a primal-dual based distributed approximation algorithm for the PCST problem resulting non-trivial approximation factor.

3 System Model

We model the distributed PCST problem on a connected network as a graph $G = (V, E, p, w)$, where vertex set $V$ and edge set $E$ represent the set of nodes and set of communication links of the network respectively. Each edge $e \in E$ has a non-negative cost denoted by $w(e)$. Each vertex $v \in V$ has an unique identification number and a non-negative prize value denoted by
We assume that each node in the network knows its own prize value and cost of each of its incident links. We consider the network topology to be static. Hence addition or removal of nodes or communication links are not allowed during the course of execution of the algorithm. Each node performs the same local algorithm and communicates and coordinates their actions with their neighbors by passing messages only. We consider that communication links are reliable. A message sent by a sender is eventually received by a receiver. However no upper bound of message delay is assumed. A special node of the network designated as root (r) initiates the algorithm. In this work we assume that nodes and links do not fail.

4 IP and LP formulation of PCST

The prize-collecting steiner tree can be formulated as the following integer program (IP).

\[
\begin{align*}
\text{Min} & \quad \sum_{e \in E} w_e x_e + \sum_{U \subseteq V : r \notin U} z_U \left( \sum_{v \in U} p_v \right) \\
\text{Subject to:} & \quad x(\delta(S)) + \sum_{U \supseteq S} z_U \geq 1 \quad S \subseteq V : r \notin S \\
 & \quad \sum_{U \supseteq V : r \notin U} z_U \leq 1 \\
 & \quad x_e \in \{0, 1\} \quad e \in E \\
 & \quad z_U \in \{0, 1\} \quad U \subseteq V : r \notin U
\end{align*}
\]

For each edge \( e \in E \) there is a variable \( x_e \) that takes a value in \{0, 1\}. Here \( \delta(S) \) denotes the set of edges having exactly one endpoint in \( S \) and \( x(F) = \sum_{e \in F} x_e \). For every possible \( U \subseteq V : r \notin U \), there is a variable \( z_U \) that takes values from \{0, 1\}. The first integral constraint says that any \( S \subseteq V : r \notin S \) is connected to a tree \( T \) rooted at special node \( r \) if there exists at least one \( e \in \delta(S) \) such that \( x_e = 1 \) or it is not connected to a tree \( T \) rooted at \( r \) if \( S \subseteq U \subseteq V : r \notin U \), \( \forall e \in \delta(S), x_e = 0 \) and \( z_U = 1 \). That means either \( S \) has at least one edge \( e \) whose one endpoint is a vertex \( v \in T \) or none of the vertices of \( S \) is spanned by \( T \). Also the second constraint of the IP implies that there can be at most one such \( U \) for which \( z_U = 1 \). Note that we can set \( p_r = \infty \) since every feasible tree is required to include the root node \( r \).

Since finding the exact solution of an IP is NP-hard, we generally go for its LP-relaxation and find an approximate solution to the problem. The LP-relaxation of the above IP is as follows (taken
from [7]):

\[
\text{Min } \sum_{e \in E} w_e x_e + \sum_{U \subseteq V} z_U \left( \sum_{v \in U} p_v \right)
\]

Subject to:

\[
\sum_{e \in \delta(S)} x_e + \sum_{U \supseteq S} z_U \geq 1 \quad \forall S \subseteq V - \{r\}
\]

\[
x_e \geq 0 \quad \forall e \in E
\]

\[
z_U \geq 0 \quad \forall U \subseteq V
\]

The above LP-relaxation has two types of basic variables namely \(x_e\) and \(z_U\) and exponential number of constraints. If it is converted into its dual then there will be one type of basic variables and two types of constraints, which is computationally advantageous. Also by weak LP-duality every feasible solution to the dual LP gives a lower bound on the optimal value of the primal LP.

The dual of the above LP-relaxation is as follows:

\[
\text{Max } \sum_{S \subseteq V - \{r\}} y_S
\]

Subject to:

\[
\sum_{S : e \in \delta(S)} y_S \leq w_e \quad \forall e \in E
\]

\[
\sum_{S \subseteq U} y_S \leq \sum_{v \in U} p_v \quad \forall U \subseteq V
\]

\[
y_S \geq 0 \quad \forall S \subseteq V
\]

Here the variable \(y_S\) corresponds to the primal constraint \(\sum_{e \in \delta(S)} x_e + \sum_{U \supseteq S} z_U \geq 1\). The dual objective function indicates that for all \(S \subseteq V - \{r\}\), the variable \(y_S\) can be increased as large as possible without violating the two dual constraints \(\sum_{S : e \in \delta(S)} y_S \leq w_e\) and \(\sum_{S \subseteq U} y_S \leq \sum_{v \in U} p_v\).

The constraint \(\sum_{S : e \in \delta(S)} y_S \leq w_e\) is known as edge packing constraint which is corresponding to the primal variable \(x_e\) of LP relaxation. It says that for all \(S \subseteq V - \{r\}\) such that \(e \in \delta(S)\), \(y_S\) can be increased as large as possible until the edge packing constraint becomes tight, i.e. \(\sum_{S : e \in \delta(S)} y_S = w_e\). This equality implies the exact situation where the primal variable \(x_e = 1\) for the corresponding edge \(e\) and \(w_e\) is added to the objective function of the primal. The dual constraint \(\sum_{S \subseteq U} y_S \leq \sum_{v \in U} p_v\) is known as penalty packing constraints which is corresponding to the primal variable \(z_U\) of the LP relaxation. For all \(S \subseteq U\) variable, \(y_S\) can be increased as large as possible until the penalty packing constraint becomes tight i.e. \(\sum_{S \subseteq U} y_S = \sum_{v \in U} p_v\). Any positive value of \(y_S\) can be considered feasible provided it does not lead to the violation of any of the two dual packing constraints. If we set \(y_S = 0\) then it gives a trivial feasible solution to the dual LP since it satisfies both the packing constraints.
**Goemans-Williamson algorithm:** Since our algorithm is inspired by the centralized PCST algorithm of Goemans and Williamson [21][22], here we briefly describe the algorithms and highlight the difficulties associated with its distributed formulation. GW-algorithm consists of two phases namely growth phase and pruning phase. The growth phase maintains a forest $F$ which contains a set of candidate edges to be added in the objective function. Initially each node is unmarked and each single node is considered as a component. Growth phase also maintains a set of components whose possible states can be either active or inactive. If a component $C$ is active then current state of $C$, i.e. $CS(C) = 1$, otherwise $CS(C) = 0$. The state of a component containing $r$ is always inactive. Initially, except the root component, all other components are in active state. Associated with each component $C$, there is a dual variable $y_C$. Each $y_C$ is initialized to 0. The Algorithm also maintains a deficit value $d_i$ for each vertex $i \in V$ and weight of a component $WC(C)$, for each $C$. In each iteration, the algorithm finds a global minimum $\epsilon = \min(\epsilon_1, \epsilon_2)$ where $\epsilon_1 = \frac{w_C - d_i - d_j}{CS(C)_{p} + CS(C)_{q}}$ for two distinct components $C_p$ and $C_q$, and $\epsilon_2 = \sum_{v \in C} p_v - WC(C)$ for any active component $C$. Depending on the value of $\epsilon$, the algorithm may decide to do any one of the two operations: (i) if $\epsilon = \epsilon_1$ then it merges two distinct components $C_p$ and $C_q$ using the edge $e$ (that gave the min $\epsilon$) and adds $e$ to $F$. (ii) if $\epsilon = \epsilon_2$ then the corresponding component $C$ is deactivated. Note that for every decided value of $\epsilon$, for each component $C$ where $CS(C) = 1$, the value of $y_C$ and the value of each $d_i : i \in C$ is increased by the value of $\epsilon$. In the case of merging, if the resulting component contains the root $r$ then it becomes inactive; otherwise it is active. In the other case i.e. deactivation of component $C$, the algorithm labels each $v \in C$ with the name of the component $C$. Since in each iteration of the algorithm, sum of the total number of components or the number of active components decreases therefore after at most $2n - 1$ iterations all components become inactive. In pruning phase the algorithm removes as many edges as possible from $F$ without violating the two properties: (i) all unmarked vertices must be connected to the root, as these vertices were never in any deactivated components (ii) if a vertex with label $C$ is connected to the root then every vertex labelled with $C' \supseteq C$ should be connected to the root. The GW-algorithm achieves an approximation ratio of $(2 - \frac{1}{n-1})$ and running time of $O(n^3 \log n)$ for a graph of $n$ vertices.

The GW-algorithm is a centralized algorithm. In each iteration it scans all outgoing edges of each active component, and all vertices of each active component in order to calculate the global minimum $\epsilon$. It is assumed that the whole graph information and all related data structures are available in a single computing node. On the other hand in many networks such as peer to peer content distribution network or telecommunication network where PCST is applied [24], a single node may not have the information of the whole network. In such cases it is natural that each node possesses information about its neighbors. For such networks, a distributed variant of the PCST problem might be useful since the input is distributed to many nodes. Many well known problems like minimum spanning tree, steiner tree have been already solved for distributed setting and similarly distributed variant of PCST algorithm is quite interesting for the community interested in distributed optimization. To the best of our knowledge, currently no distributed PCST algorithm is known to exist. Even GW-algorithm can not be directly adapted to solve PCST in a distributed setting as the input is stored in a distributed way in this case. Taking these facts into considera-
5 Distributed Algorithm for PCST (D-PCST)
Initially each node $v \in V$ knows its own prize value $p_v$ and weight $w_e$ of each edge $e \in Adj(v)$. Here $Adj(v)$ denotes the set of edges incident on $v$. When the distributed algorithm terminates, each node $v \in V$ knows whether it is in the prize part or in the steiner part. A node $v$ belongs to the prize part if its local variable $\text{prize\_flag}$ is set to $\text{TRUE}$. Otherwise it belongs to the steiner part. In addition, if a node $v$ belongs to the steiner part then at least one $e \in Adj(v)$ must be assigned as a $\text{branch}$ edge of the steiner tree. At each node $v$ the state of an edge $e \in Adj(v)$ is $\text{branch}$ if its local variable $\text{SE}(e) = \text{branch}$. So the pair $(\text{prize\_flag}, Z)$ at each node $v$ clearly defines the distributed output of the algorithm. Here $Z \subseteq Adj(v)$. If $\text{prize\_flag} = \text{TRUE}$ then $Z = \phi$. Otherwise for each $e \in Z$, $\text{SE}(e) = \text{branch}$. Apart from $\text{branch}$, the state of an edge can also be $\text{rejected}$ or $\text{basic}$. Initially each edge is a $\text{basic}$ edge. As components grow, the status of some edges are changed to $\text{branch}$ or $\text{rejected}$. The edges of the steiner tree of a component $C$ are the $\text{branch}$ edges. Any edge inside a component (between two nodes $u, v \in C$) which is not $\text{branch}$ is stated as $\text{rejected}$. An edge $e$ which is neither $\text{branch}$ nor $\text{rejected}$ has state named $\text{basic}$. Note that a set of nodes $C$ such that $C \subseteq V$ and is connected by a set of edges is termed as a $\text{component}$. The edges of a component can be either $\text{branch}$ or $\text{rejected}$.

Our algorithm consists of two phases namely $\text{growth phase}$ and $\text{pruning phase}$. At any instant of time the algorithm maintains a set of components. Each component has a state which can be $\text{sleeping}$ or $\text{active}$ or $\text{inactive}$. At the beginning of the algorithm each component comprises of a single node. Initially each component except the root component (containing the special node $r$, the root) is in $\text{sleeping}$ state. The initial state of the root component is $\text{inactive}$. Each component has a leader node. Each node $v$ in a component $C$ locally knows the current state of $C$, denoted as $CS(C)$. Each component $C$ has a weight denoted by $W(C)$ which is known to each $v \in C$. A vertex $v \in C$ locally knows the weight of the component $C$, denoted as $W_v(C)$. Also each node $v$ has a $\text{deficit}$ value denoted by $d_v$. Initially $d_v = 0$ and $W_v(C) = 0$ for each $v \in V (v \in C)$. In addition, the following symbols are used in the description of our algorithm.

1. $\epsilon_e$: a value that is calculated for an edge $e$.
2. $\epsilon_1(v) = \min_{e \in Adj(v) \cap e \in \delta(C)} \{\epsilon_e\}$
3. $\epsilon_1(C) = \min_{v \in C} \{\epsilon_1(v)\}$
4. $\epsilon_2(C) = \sum_{v \in C} p_v - W(C)$
5. MOE (minimum outgoing edge): the edge $e \in \delta(C)$ that gives the $\epsilon_1(C)$.
6. $d_h(C) = \max_{v \in C} \{d_v\}$: it denotes the highest $\text{deficit}$ value of a component $C$. 


At any point in time only one component $C$ tries to calculate $\epsilon(C)$. The leader of $C$ computes $\epsilon(C) = \min(\epsilon_1(C), \epsilon_2(C))$ using message passing. Depending on the value of $\epsilon(C)$, the leader of $C$ proceeds with any one of the following actions.

(i) If $CS(C) = \text{active}$ then it may decide to merge with one of its neighboring component $C'$ or it may decide to become inactive.

(ii) If $CS(C) = \text{inactive}$ then it asks one of its neighboring components, say $C'$, to proceed further. The choice of $C'$ depends on the value $\epsilon_1(C)$ computed at $C$. Note that an inactive component $C$ never computes $\epsilon_2(C)$ and its $\epsilon(C)$ is equal to $\epsilon_1(C)$.

To compute $\epsilon_1(C)$, the leader of $C$ broadcasts $\langle \text{initiate} \rangle$ (a message $M$ is denoted by $\langle M \rangle$) asking each frontier node $v \in C$ to finds its $\epsilon_1(v)$. A node $v \in C$ is called a frontier node if it has at least one edge $e : e \in \text{Adj}(v) \land e \in \delta(C)$. Upon receiving $\langle \text{initiate} \rangle$, each frontier node $v \in C$ calculates $\epsilon_e$ for each edge $e : e \in \text{Adj}(v) \land e \in \delta(C)$. Let $C'$ be a neighboring component of $C$ such that $e \in \delta(C')$ and $e \in \text{Adj}(u)$ and $u \in C'$. Now the following cases may happen.

- $CS(C) = \text{active}$ and $CS(C') = \text{active}$ : in this case $\epsilon_e = \frac{w_v - d_v - d_u}{2}$.
- $CS(C) = \text{active}$ and $CS(C') = \text{inactive}$ : in this case $\epsilon_e = w_v - d_v - d_u$.
- $CS(C) = \text{active}$ and $CS(C') = \text{sleeping}$ : in this case $\epsilon_e = \frac{w_v - d_v - d_u}{2}$. Here the state of the component $C'$ is sleeping and therefore the deficit value $d_u$ of the node $u \in C'$ is considered to be equal to the $d_h$ of $C$.
- $CS(C) = \text{inactive}$ and $CS(C') = \text{sleeping}$ : in this case $\epsilon_e = w_v - d_v - d_u$. Similar to the previous case the deficit value $d_u$ of the node $u \in C'$ is considered to be equal to the $d_h$ of $C$.
- $CS(C) = \text{inactive}$ and $CS(C') = \text{inactive}$ : in this case the value of $\epsilon_e$ for an edge $e \in \text{Adj}(v)$ calculated by $v$ depends on state of a local boolean variable $RFE(e)$. If $RFE(e) = \text{TRUE}$ then $\epsilon_e = w_v - d_v - d_u$. Otherwise, $\epsilon_e = \infty$.

Note that the following cases are not possible.

- $CS(C) = \text{inactive}$ and $CS(C') = \text{active}$ : when an inactive component $C$ is in the state of computing its $\epsilon_1(C)$ then there cannot exist any component $C'$ in the neighborhood of $C$ such that $CS(C') = \text{active}$.

- A component $C$ never computes its $\epsilon_1(C)$ (or $\epsilon_2(C)$) while it is in the sleeping state.

Following these conditions a frontier node $v$ calculates the value of $\epsilon_e$ for each of its outgoing edge $e : e \in \text{Adj}(v) \land e \in \delta(C)$ and the $\epsilon_1(v)$ is locally selected for reporting to the leader of $C$. In this way each frontier node $v \in C$ locally calculates $\epsilon_1(v)$ and reports it to the leader using convergecast technique using a tree rooted at the leader. During the convergecast process the
overall $\epsilon_1(C)$ survives and eventually reaches the leader node of $C$. Also during the convergecast each node $v \in C$ reports the total prize value of all the nodes in the subtree rooted at $v$. So eventually the total prize ($TP$) of the component $C$ is also known to the leader. Now the leader calculates $\epsilon_2(C) = \sum_{v \in C} p_v - W(C) = TP - W(C)$.

The leader of $C$ now computes $\epsilon(C) = \min(\epsilon_1(C), \epsilon_2(C))$. If $\epsilon(C) = \epsilon_1(C)$ then $C$ decides to deactivate itself. This indicates that the dual penalty packing constraint $\sum_{S \in C} y_S \leq \sum_{v \in C} p_v$ becomes tight for the component $C$. On the other hand, if $\epsilon(C) = \epsilon_2(C)$ then it indicates that for component $C$ the dual edge packing constraint $\sum_{S \in \delta(S)} y_S \leq w_e$ becomes tight for one of the edges $e : e \in \delta(C) \land e \in \delta(C')$. In this case $C$ sends $\langle$connect$(v, W(C), d_v, d_h)\rangle$ to $C'$ to merge with it. Note here that $v$ is the frontier node which resulted the $\epsilon(C)$. Actually the leader sends the message to $w$ which in turn send it to the adjoining node $u \in C'$. When the node $u \in C'$ receives $\langle$connect$(v, W(C), d_v, d_h)\rangle$ on its edge $e$ then depending on the state of $C'$ following actions are taken.

- **$CS(C') = inactive$**: in this case the node $u \in C'$ sends $\langle accept \rangle$ to $v \in C$. This confirms the merging of two components $C'$ and $C$.

- **$CS(C') = sleeping$**: in this case it is obvious that $C'$ is a single node component $\{u\}$. The state of $C'$ becomes $active$ and each of its local variables $d_u, W_u(C')$ and $d_h(C')$ is initialized to $d_h(C)$ (received in the $connect$ message). After that the leader of the component $C'$ ($u$ itself since it is a single node component) computes $\epsilon_e$ for edge $e : e \in \delta(C') \land e \in \delta(C)$ and $\epsilon_2(C')$. If $\epsilon_e < \epsilon_2(C')$ then the component $C'$ sends $\langle accept \rangle$ to the component $C$ which confirms the merging of two components $C'$ and $C$. On the other hand if $\epsilon_2(C') \leq \epsilon_e$ then $C'$ decides to deactivate itself and sends $\langle$refind$\epsilon$\rangle$ to the component $C$.

Whenever a node $v \in C$ receives $\langle$refind$\epsilon$\rangle$ in response to a $connect$ message on an edge $e : e \in \text{Adj}(v)$ then it sets its local variable $RF_E(e) = \text{TRUE}$ for edge $e$. Node $v$ also reports the $\langle$refind$\epsilon$\rangle$ to the leader of $C$. Upon receiving $\langle$refind$\epsilon$\rangle$ the leader node of $C$ proceeds to calculate its $\epsilon$ once again.

Whenever a component $C$ decides to merge or deactivate (i.e. $CS(C) = active$) then each node $v \in C$ increases each of $d_v$ and $W_v(C)$ by $\epsilon(C)$ and $d_h(C)$ is also updated. Note that for each component $C$ of the whole graph there is an implicit dual variable $y_C$ which we want to maximize subject to the dual constraints. Whenever the local variables of a component $C$ are updated by $\epsilon(C)$, $y_C$ is also implicitly updated. Therefore if $C$ is $active$ or $inactive$ then it knows its $y_C$ automatically. However if $C$ is $sleeping$ then it gets its $y_C$ through $d_h(C)$ when it receives $\langle$connect$(u, W(C'), d_u, d_h(C'))\rangle$ or $\langle$proceed$(d_h(C'))\rangle$ from a neighboring component $C'$.

If two components $C$ and $C'$ decide to merge through an edge $e = (v, u) : v \in C \land u \in C'$ then the dual edge packing constraint, $\sum_{S \in \delta(S)} y_S \leq w_e$ becomes tight for the edge $e$. Both nodes $v$ and $u$ set their local variables $SE(e) = \text{branch}$ for edge $e$. The weight of the resulting component $C \cup C'$ is the sum of the weights of $C$ and $C'$, i.e. $W(C \cup C') = W(C) + W(C')$. If $C \cup C'$ contains the root node $r$ then it becomes inactive (root component is always inactive) and $r$ remains the leader of the new component $C \cup C'$. In addition, whenever a component $C'$ merges with the
root component then each \( v \in C' \) sets its local variable \( \text{prize\_flag} = \text{FALSE} \) and there exists at least one edge \( e \in \text{Adj}(v) \) such that \( SE(e) = \text{branch} \). This indicates that each node \( v \in C' \) contributes to the steiner part of the PCST. On the other hand if none of the merging components \( C \) or \( C' \) is the root component then the resulting component \( C \cup C' \) becomes active. In this case the node with the higher ID between the two adjacent nodes of the merging edge becomes the new leader of the component \( C \cup C' \) and for each node \( v \in C \cup C' \) the boolean variable \( \text{prize\_flag} \) remains \( \text{TRUE} \).

In case of deactivation of a component \( C \), each node \( v \in C \) labels itself by the name of the component \( C \). Whenever an active component \( C \) becomes inactive and there exists no active component in its neighborhood then the leader of \( C \) may decide to send \( \langle \text{proceed}(d_h(C)) \rangle \) or \( \langle \text{back} \rangle \) to one of its neighboring component \( C' \). For this, first of all the leader of \( C : SE(C) = \text{inactive} \) computes its \( \epsilon_1(C) \). Note that the value of \( \epsilon_1(C) \) may be some finite real number or \( \infty \). If \( C \) has at least one neighboring component \( C' \) such that \( CS(C') = \text{sleeping} \) or if \( CS(C') = \text{inactive} \) for each neighboring component \( C' \) of \( C \) and \( \exists e, v : e \in \delta(C) \wedge v \in C \wedge e \in \text{Adj}(v) \wedge \text{REF}(e) = \text{TRUE} \) then the value of \( \epsilon_1(C) \) is guaranteed to be a finite real number. Otherwise the value of \( \epsilon_1(C) = \infty \). If \( \epsilon_1(C) \) corresponding to the edge \( e : e \in \delta(C) \wedge \delta(C') \) is a finite real number then the component \( C \) sends \( \langle \text{proceed}(d_h(C)) \rangle \) to the component \( C' \) through edge \( e \). Upon receiving \( \langle \text{proceed}(d_h) \rangle \) the component \( C' \) initializes its local variables and starts computing the \( \epsilon(C') \) for taking further actions. If the value of \( \epsilon_1(C) = \infty \) then the leader of the component \( C \) sends \( \langle \text{back} \rangle \) to a neighboring component \( C'' \) from which it received \( \langle \text{proceed}(d_h(C'')) \rangle \) in some early stages of the algorithm. Eventually when the leader of the root component \( C_r \) finds minimum \( \epsilon_1(C_r) = \infty \) then all components in the whole graph become \( \text{inactive} \). This ensures the termination of the \textit{growth} phase of the algorithm. After the termination of growth phase, the root node initiates the \textit{pruning} phase of the algorithm. In pruning phase following operations are performed.

- Each node \( v \in C \) where \( C \) is a non root inactive component, sets \( SE(e) = \text{basic} \) for each edge \( e : e \in \text{Adj}(v) \wedge SE(e) = \text{branch} \).

- In the root component \( C_r \) pruning starts parallelly at each leaf node \( v \) of the steiner tree rooted at the root node \( v \) and repeatedly applied at every leaf node at any stage as long as the following two conditions hold.
  1. The node \( v \) is labeled with the name of some component \( C \neq C_r \).
  2. There exists exactly one edge \( e : e \in \text{Adj}(v) \wedge SE(e) = \text{branch} \)

Each pruned node \( v \in C_r \) sets its \( \text{prize\_flag} = \text{TRUE} \) and status of one or more \( e \in \text{Adj}(v) \) are also updated accordingly \( (SE(e) = \text{basic}) \). Finally for each of the non-pruned and unmarked node \( u \in C_r \), \( \text{prize\_flag} = \text{FALSE} \) and there exists at least one edge \( e \in \text{Adj}(u) : SE(e) = \text{branch} \).
Figure 1: A case of merging operation. (a) state before merging of components \( \{v_2, v_5\} \) and \( \{v_1\} \) and (b) state after merging.

Figure 2: A case of deactivation. (a) state before the deactivation of the active component \( \{v_7, v_{11}\} \). (b) state after the deactivation.

Figure 3: A case of Proceed operation. (a) is the state of sending \( \langle \text{proceed}(15) \rangle \) by the inactive component \( \{v_3\} \). (b) is the state of after the component \( \{v_4\} \) receives \( \langle \text{proceed}(15) \rangle \).
In this section we explain the working of the proposed D-PCST algorithm with an example. Due to space constraints, we illustrate only the major operations.

In Figure 4 merging of two components is illustrated. Each node has a prize value that is labeled just outside the node. For example, the prize of node $v_1$ is 10. Similarly each edge is labeled with a weight. Figure 4(a) shows the graph before the merging of two neighboring components $C = \{v_2, v_3\}$ and $C' = \{v_1\}$. ($v_2, v_1$) is the MOE for the active component $C$ which gives $\epsilon_1(C) = -1$. The leader node $v_2$ also computes $\epsilon_2(C) = 6$. Hence $\epsilon(C) = min(\epsilon_1(C), \epsilon_2(C)) = \epsilon_1(C)$. So $v_2$ sends $\langle$connect$(v_2, 14, 7, 7)\rangle$ on the MOE to merge with $C'$. Now $C'$ becomes active and finds $\epsilon(C') = \epsilon_1(C') = -1$ and $(v_1, v_2)$ is the MOE. Therefore it decides to merge with $C$. The new active component $\{v_1, v_2, v_3\}$ is shown in Figure 4(b). The rectangular box below the graph shows the value of local variables $d_i$ and $W_i$ for each $v_i \in V$.

Figure 2 shows the deactivation of an active component $C = \{v_7, v_{11}\}$. In Figure 2(a) the leader of $C$ finds that its MOE is $(v_7, v_{11})$ which gives $\epsilon_1(C) = 7.5$. $C$ also computes its $\epsilon_2(C)$ which is equal to 3. Since $\epsilon(C) = min(\epsilon_1(C), \epsilon_2(C)) = \epsilon_2(C)$, therefore the component $C$ deactivates itself. Each node of $C$ labels itself by the name of the component $C$. The graph after deactivation of $C$ is shown in Figure 2(b).

Figure 3 shows the action of proceed operation performed by an inactive component $C = \{v_3\}$. 

5.1 An Example
In Figure 3(a), the MOE of $C$ is $(v_3, v_4)$ which gives $\epsilon_1(C) = 10$. The component $C$ sends $\langle proceed(d_h(C)) \rangle$ (denoted by $P(15)$ in the graph) on its MOE to the component $C' = \{v_4\}$. Upon receiving $P(15)$, the sleeping component $C'$ becomes active and initializes its local variables $d_4$ and $W_4$ to 15. Actually $d_h(C) = 15$ which is received through the proceed message. The MOE of component $C'$ is $(v_4, v_3)$ which gives $\epsilon_1(C') = 10$. The value of $\epsilon_2(C') = 25$. Since $\epsilon(C') = \epsilon_1(C')$, $C'$ sends a connection request to $C$ which is shown in Figure 3(b).

Figure 4 shows the case of pruning operation performed in each component of the graph. In Figure 4(a) the state of each component is inactive which indicates the termination of the growth phase. The root node broadcasts $\langle prune \rangle$ in the root component and sends the same to all non-root inactive components. In a non-root inactive component the state of each branch edge changes its state to basic. In the root component $C_r$, a subcomponent $C = \{v_7, v_{11}\} \subset C_r$ and its corresponding branch edges are pruned. The component $C$ was deactivated at some point of the growth phase of the algorithm. The node $v_{11}$ is pruned since $v_{11} \in C$, there is only one adjacent branch edge of node $v_{11}$. At $v_{11}$ the local variable $prize\_flag$ is set to TRUE and $SE((v_{11}, v_7))$ is set to basic. Similarly the node $v_7$ and its corresponding adjacent branch edges are also pruned from the root component. The Figure 4(b) shows the state of the graph after the pruning phase.

Figure 5(a) shows the initial state of the graph where each component comprises of single node and except the root all other components are in sleeping state. Initially the state of the root component is inactive. Figure 5(b) shows the final PCST of the graph after the application of distributed PCST algorithm.

6 Proof of Correctness

A component $C'$ is called a neighbor of a component $C$ if $\delta(C) \cap \delta(C') \neq \phi$. A round of $proc\_initiate()$ in a component $C$ means the time from the beginning of the execution of $proc\_initiate()$ till the completion of finding $\epsilon(C)$. By an action of an event $M$ we mean the start of event $M$.

Claim 6.1 When an inactive component $C$ is in the state of computing its $\epsilon_1(C)$ then there cannot exist any component $C'$ in the neighborhood of $C$ such that $CS(C') = active$.

6.1 Termination

Lemma 6.2 A round of $proc\_initiate()$ generates at most $7|V| + 2|E| - 8$ messages.

Proof. In each round of $proc\_initiate()$ the following messages are possibly generated: $\langle initiate \rangle$, $\langle test \rangle$, $\langle status \rangle$, $\langle report \rangle$, $\langle merge \rangle$, $\langle connect \rangle$, $\langle update\_info \rangle$, $\langle back \rangle$, $\langle proceed \rangle$, $\langle disable\_proceed \rangle$, $\langle accept \rangle$ and $\langle refind\_epsilon \rangle$. Since maximum number of branch edges in a component is at most $(|V| - 1)$, therefore at most $(|V| - 1)$ number of messages are exchanged for each kind of $\langle initiate \rangle$, $\langle report \rangle$, $\langle proceed \rangle$ and $\langle update\_info \rangle$. Similarly in each round of $proc\_initiate()$, at most $|E|$ number of $\langle test \rangle$ are sent and in response at most $|E|$ number of messages $\langle status \rangle$ or $\langle reject \rangle$ are generated. For $\langle merge \rangle$, $\langle disable\_proceed \rangle$ and $\langle back \rangle$ exactly two components are
required between which these messages are communicated. Therefore in each round of \textit{proc\_initiate()} number of messages exchanged for each kind of (merge), (disable \textit{proceed}) and (back) is at most \(|V| - 2\). For each (connect) message, either an (accept) or a (refind \textit{epsilon}) is generated. Therefore in each round of \textit{proc\_initiate()}, at most any two of the combinations of \{ (connect), (accept), (refind \textit{epsilon}) \} are generated. Therefore number of messages exchanged in each round of \textit{proc\_initiate()} is at most 4(|V| - 1) + 3(|V| - 2) + 2 + 2|E| = 7|V| + 2|E| - 8. \qed

\textbf{Lemma 6.3} If the leader of a component \(C\) finds \(\epsilon_1(C) = \infty\) and \(CS(C) = \text{inactive}\) then for each neighboring component \(C_k\) of \(C\), \(CS(C_k) = \text{inactive}\).

\textbf{Proof.} Suppose by contradiction the leader of \(C\) finds \(\epsilon_1(C) = \infty\) and there exists a neighboring component \(C_k\) of \(C\) such that \(CS(C_k) \neq \text{inactive}\). Therefore \(CS(C_k) = \text{sleeping}\) or \(CS(C_k) = \text{active}\). Let us first assume that \(CS(C_k) = \text{sleeping}\). To find \(\epsilon_1(C)\) the leader starts the procedure \textit{proc\_initiate()} (as per pseudocode given in the appendix) which in turn sends (initiate) on each of its branch edges in the component \(C\). This message is forwarded on the branch edges by each node \(v \in C\). Upon receiving (initiate), each frontier node \(v \in C\) sends (test) on each edge \(e : e \in \text{Adj}(v) \land SE(e) \neq \text{branch} \land SE(e) \neq \text{rejected}\). In response to each (test), \(v\) either receives (status(\(CS(C_k), d_u\))) on an edge \(e\) from a node \(u \in C \neq C\) if \(e \in \text{Adj}(v) \land e \in \text{Adj}(u) \land e \in \delta(C)\) or \(v\) receives (reject) if \(e = (v, u) : v, u \in C\). The (reject) is simply discarded by the node \(v\). In case of (status(\(CS(C_k), d_u\))), the node \(v\) calculates the \(\epsilon_e\) for edge \(e\). Since \(CS(C_k) = \text{sleeping}\) therefore the frontier node \(v\) computes \(\epsilon_e = w_e - d_v - d_u\). In this case the computed value \(\epsilon_e\) is a finite real number, since \(w_e, d_v\) and \(d_u\) all are finite real numbers. Eventually the value of each \(\epsilon_1(v)\) computed by each frontier node \(v \in C\) reaches to the leader of \(C\). Eventually the leader of \(C\) finds the value of \(\epsilon_1(C)\) to be a finite real number, a contradiction to the fact that \(\epsilon_1(C) = \infty\).

Since it is given that the leader of the component \(C\) is in a state of finding its \(\epsilon_1(C)\) therefore by Claim 6.1, it is ensured that \(CS(C_k) \neq \text{active}\) for each neighboring component \(C_k\) of \(C\). This completes the proof. \qed

\textbf{Claim 6.4} Each component \(C\) generates (back) at most once.

\textbf{Proof.} A non-root component \(C\) decides to take the action of (back) only if \(CS(C) = \text{inactive}\) and it finds its \(\epsilon(C) = \infty\). In this case, for each neighboring component \(C_k\) of \(C\), \(CS(C_k) = \text{inactive}\) as shown by Lemma 6.3 \text{and} \(RF\text{E}(e) = \text{FALSE}\) for each \(e \in \delta(C)\). These facts in the D-PCST algorithm ensure that component \(C\) never receives (proceed) in future. And without receiving (proceed), a component can not proceed with any other actions of the algorithm. These observation concludes that each component \(C\) decides to take the action of (back) at most once. \qed

\textbf{Lemma 6.5} D-PCST algorithm generates the action of (back) at most \(|V| - 1\) times and the action of pruning phase exactly once.
Claim 6.6 If none of the four consecutive rounds of proc\textunderscore initiate() initiates the action of $\langle \text{back} \rangle$ then any one of the following events is guaranteed to happen: (i) sum of the number of components decreases (ii) one of the sleeping or active components decreases.

Proof. In the D-PCST algorithm the leader of a component $C$ starts finding its $\epsilon(C)$ by executing the procedure proc\textunderscore initiate(). Depending on $\epsilon(C)$, the leader of the component $C$ decides to take any one of the following actions: (i) merging (ii) deactivation (iii) sending $\langle \text{proceed} \rangle$, (iv) sending $\langle \text{back} \rangle$ or (v) start of the pruning phase. If the action of sending $\langle \text{back} \rangle$ is not taken by any one of the four consecutive rounds of proc\textunderscore initiate() then within these four rounds of the proc\textunderscore initiate() any one of the remaining actions is guaranteed to happen.

First consider the case of merging. Suppose merging is initiated at some component $C$. First the leader computes its $\epsilon(C)$ in one round of proc\textunderscore initiate(). After that it sends a $\langle \text{merge} \rangle$ to the corresponding component say $C'$. If $CS(C') = \text{inactive}$ then it immediately merges and in this case merge happens in one round of proc\textunderscore initiate() and as a result one component decreases in the graph. If $CS(C') = \text{sleeping}$ then $C'$ goes through one round of proc\textunderscore initiate() to decide whether to merge or deactivate itself. If it decides to merge with $C$ then one component decreases in the graph. On the other hand if it decides to deactivate itself then one sleeping component decreases in the graph. Therefore action of merging takes place in at most two rounds of proc\textunderscore initiate().

We know that only an active component can decide to deactivate itself. To be deactivated a component $C$ finds its $\epsilon(C) = \epsilon_2(C)$ in exactly one round of proc\textunderscore initiate(). As a result one active component decreases in the graph.

In case of the action of sending $\langle \text{proceed} \rangle$, to guarantee any one of the events to take place at most four rounds of proc\textunderscore initiate() are required. A component $C$ initiates this action only when it is in inactive state. To decide to send $\langle \text{proceed} \rangle$ to a neighboring component $C'$ it first computes it $\epsilon(C)$ which takes one round of proc\textunderscore initiate(). Upon receiving $\langle \text{proceed} \rangle$ from $C$, $C'$ does the following:

- $CS(C') = \text{sleeping}$: $C'$ starts finding its $\epsilon(C')$ to decide either merging or (ii) deactivation. In case of merging it takes at most another two rounds of proc\textunderscore initiate() and as a result one of the component decreases, i.e. from the point of action of sending $\langle \text{proceed} \rangle$ at $C$ up to the merging of the component $C'$ with some other component it takes at most three rounds of proc\textunderscore initiate(). In case of deactivation of $C'$ it takes exactly one round of proc\textunderscore initiate() for which one sleeping component decreases and this takes two rounds of proc\textunderscore initiate().

\hfill \Box
- $CS(C') = \text{inactive}$: In this case $C'$ receives $\langle\text{proceed}\rangle$ because one of its edge $e$ must be in the state of $REF(e) = \text{TRUE}$: $e \in \delta(C) \land e \in \delta(C')$. And there should be at least one component $C''$ where $CS(C'') = \text{sleeping}$ in the neighborhood of $C'$. Otherwise $C'$ has to take the action of sending $\langle\text{back}\rangle$ which is not possible according to our assumption. Since $CS(C') = \text{inactive}$ therefore $C'$ takes one round of $\langle\text{initiate}\rangle$ to compute its $\epsilon(C')$ to take the action of sending $\langle\text{proceed}\rangle$ to any component $C''$ such that $CS(C'') = \text{sleeping}$. After that $C''$ follows at most two rounds of $\langle\text{initiate}\rangle$ to decide any one of the events either merging or (ii) deactivation which guarantees to happen any one of the mentioned events. Therefore from the the action of sending $\langle\text{proceed}\rangle$ from component $C'$ upto any one of the events to be happened takes at most four rounds of $\langle\text{initiate}\rangle$.

- $CS(C') = \text{active}$: By Claim 6.1 this condition is not possible.

In case of pruning the root component $C_r$ takes exactly one round of $\langle\text{initiate}\rangle$ to computes its $\epsilon(C_r) = \infty$. Therefore we claim that If none of the four consecutive rounds of $\langle\text{initiate}\rangle$ initiates the action of sending $\langle\text{back}\rangle$ then any one of the following events is guaranteed to happen: (i) sum of the number of components decreases (ii) one of the sleeping or active components decreases.

\[
\text{Lemma 6.7} \quad \text{The growth phase of the D-PCST algorithm terminates after at most } 9|V| - 7 \text{ rounds of } \langle\text{initiate}\rangle.\
\]

\textbf{Proof.} Initially the state of the root component $C_r$ is $\text{inactive}$ and it takes one round of $\langle\text{initiate}\rangle$ to compute its $\epsilon(C_r)$ and send $\langle\text{proceed}\rangle$ to a neighboring component to take further actions. By Claim 6.6 it is ensured that in the worst case at most $4(|V| - 1)$ round of $\langle\text{initiate}\rangle$ is required to decrease the sum of the number of components and becomes one or at most $4(|V| - 1)$ rounds of $\langle\text{initiate}\rangle$ is required to change the state of each sleeping/active component to the state of $\text{inactive}$. By Claim 6.4 it is ensured that the action of $\langle\text{back}\rangle$ is at most $|V| - 1$ times and the action of pruning phase is exactly once. Summing for all the cases we get the number of rounds of $\langle\text{initiate}\rangle$ is equal to $1 + 4(|V| - 1) + 4(|V| - 1) + (|V| - 1) + 1 = 9|V| - 7$. Therefore after at most $9|V| - 7$ rounds of $\langle\text{initiate}\rangle$ it is ensured that initiation of round of $\langle\text{initiate}\rangle$ stops. Furthermore no initiation of round $\langle\text{initiate}\rangle$ implies that no message related to growth phase is exchanged in the graph. This ensured that growth phase eventually terminates after at most $9|V| - 7$ rounds of $\langle\text{initiate}\rangle$.

\[
\text{Lemma 6.8} \quad \text{Pruning phase of D-PCST eventually terminates.}\
\]

\textbf{Proof.} After the termination of the growth phase the root $r$ initiates the pruning phase by sending $\langle\text{prune}\rangle$ on an edge $e \in \text{Adj}(r)$ if $SE(e) = \text{branch}$ or $ETP(e) = \text{TRUE}$. Here at node $v \in C$ for any component $C$, $ETP(e)$ is a local variable for each edge $e \in \text{Adj}(v)$ and by default $ETP(e) = \text{FALSE}$ for each $e \in \text{Adj}(v)$. A frontier node $v \in C$ sets $ETP(e) = \text{TRUE}$ when it sends a $\langle\text{proceed}\rangle$ to some other components $C' \neq C$ through the outgoing edge $e$. When
a node \( v \) receives \(<\text{prune}\rangle\) on an edge \( e \) it forwards \(<\text{prune}\rangle\) on edge \( e' \land e' \in \text{Adj}(v) : e' \neq e \) if \( SE(e') = \text{branch} \) or \( \text{ETP}(e') = \text{TRUE} \). In a non-root inactive component \( C_k \) when a node \( v \in C_k \) receives \(<\text{prune}\rangle\) on some edge \( e \in \text{Adj}(v) \) then except on edge \( e \) the node \( v \) forwards \(<\text{prune}\rangle\) on all other branch edges and after that sets \( SE(e) = \text{basic} \) for each edge \( e \) such that \( e \in \text{Adj}(v) \land SE(e) = \text{branch} \). In the root component \( C_r \) pruning starts at leaf nodes of the tree rooted at \( r \). Whenever \(<\text{prune}\rangle\) arrives at a leaf node \( v \in C_r \) over an edge \( e \) then \( v \) prunes itself if \( \text{labelled\_flag} = \text{TRUE} \) and either \( \forall e' \in \text{Adj}(v) : e' \neq e \land SE(e') = \text{basic} \) or \( \text{prune\_msg\_count} = 0 \). In this case, \( v \) sets its local variable \( \text{prize\_flag} \) to \( \text{TRUE} \) and \( SE(\text{in\_bound}) \) to \( \text{basic} \) indicating that it is contributing to the prize part of the algorithm and non of its incident edges belongs to the steiner part of PCST. Then \(<\text{backward\_prune}\rangle\) is sent back on its \( \text{in\_bound} \) edge to its parent. Upon receiving \(<\text{backward\_prune}\rangle\) on an edge say \( e \), a node first sets \( SE(e) = \text{basic} \) and checks same conditions as we mentioned above to prune it. If a node \( v \) fails to prune then no further message is sent on any of its incident edges. In this way all nodes in the \( C_r \) stops sending any messages on their incident edges and thus eventually pruning phase terminates.

**Theorem 6.9** The distributed PCST (D-PCST) algorithm eventually terminates.

**Proof.** Lemma 6.7 and Lemma 6.8 prove that the growth phase and the pruning phase of the D-PCST algorithm terminate respectively. Together Lemma 6.7 and Lemma 6.8 prove that the D-PCST algorithm terminates.

### 6.2 Optimality of the algorithm

**Lemma 6.10** If \( CS(C_l) = \text{sleeping} \) and it receives \(<\text{connect}(v, W(C_k), d_v, d_h(C_k))\rangle \) or \(<\text{proceed}(d_h(C_k))\rangle \) over an edge \( e \) from a node \( v \in C_k \) where \( C_k \) is a neighboring component of \( C_l \) then \( C_l \) correctly computes each of its local variables without violating any of the dual constraints.

**Proof.** Since \( CS(C_l) = \text{sleeping} \) therefore \( C_l \) is a single node component. Let it be \( \{u\} \). If \( u \) receives \(<\text{connect}(v, W(C_k), d_v, d_h(C_k))\rangle \) from a node \( v \in C_k \) over the edge \( e \) then first \( C_l \) becomes active and then \( u \) initializes its \( d_u = d_h(C_k) \), \( W_u(C_l) = d_h(C_k) \) and \( d_h(C_l) = d_h(C_k) \). After that \( u \) computes \( \epsilon(C_l) \) as follows:

\[
\epsilon_e = \frac{w_e - d_u - d_v}{2}
\]

\[
\epsilon_2(C_l) = TP(C_l) - W(C_l) = p_u - W_u(C_l)
\]

and \( \epsilon(C_l) = \min(\epsilon_e, \epsilon_2(C_l)) \). Therefore it is clear that \( d_h(C_k) \) is used by \( C_l \) to compute its \( \epsilon(C_l) \) in case \( CS(C_l) = \text{sleeping} \). Now there are four possible cases:

**Case 1:** \( \epsilon(C_l) = \epsilon_e \) and \( \epsilon(C_l) < 0 \)

\( \epsilon(C_l) < 0 \) indicates that the dual edge packing constraint \( \sum_{S: e \in \delta(S)} y_S \leq w_e \) is violated on the edge \( e \) when each dual variable \( y_S : S \subset V \land e \in \delta(S) \) is increased by a value \( \epsilon(C_l) \).
specifically dual variables \( y_{C_l} \) and \( y_{C_k} \) are increased by \( \epsilon(C_l) \). To ensure that the dual constraint is not violated, the excess value \( \epsilon(C_l) \) must be deducted from each of the dual variables \( y_{C_l} \) and \( y_{C_k} \). After the deduction, both components \( C_l \) and \( C_k \) merge and form a new component \( C_l \cup C_k \) without violating the dual constraints. Note that every node \( v \in C_l \cup C_k \) also updates its local variables \( d_v = d_v - \epsilon(C_l) \) and \( W_v = W_v - \epsilon(C_l) \). In addition, corresponding \( d_h(C_l \cup C_k) \) is also updated accordingly.

**Case 2:** \( \epsilon(C_l) = \epsilon_o \) and \( \epsilon(C_l) \geq 0 \)

This ensures that at most \( \epsilon(C_l) \) can be added to both \( y_{C_l} \) and \( y_{C_k} \) without violating the dual edge packing constraint \( \sum_{S \in \delta(S)} y_S \leq w_e \) for edge \( e \). Therefore the components \( C_l \) and \( C_k \) merge through the edge \( e \) and forms a bigger component \( C_l \cup C_k \). Each node \( v \in C_l \cup C_k \) also updates its local variables \( d_v = d_v + \epsilon(C_l) \) and \( W_v = W_v + \epsilon(C_l) \). In addition, corresponding \( d_h(C_l \cup C_k) \) is also updated accordingly.

**Case 3:** \( \epsilon(C_l) = \epsilon_2(C_l) \) and \( \epsilon(C_l) < 0 \)

In this case the dual variable \( y_{C_l} \) for the component \( C_l \) is increased by \( \epsilon(C_l) \) and this indicates that the dual penalty packing constraint \( \sum_{S \subseteq C_l} y_S \leq \sum_{v \in C_l} p_v \) is violated at \( C_l \). Therefore \( y_{C_l} = y_{C_l} - \epsilon_2 \) and it ensures that the dual penalty packing constraint for the component \( C_l \) is not violated and becomes tight. Node \( u \in C_l \) updates its local variables \( d_u = d_u - \epsilon(C_l) \) and \( W_u = W_u - \epsilon(C_l) \). In addition, corresponding \( d_h(C_l) \) is also updated accordingly.

**Case 4:** \( \epsilon(C_l) = \epsilon_2(C_l) \) and \( \epsilon(C_l) \geq 0 \)

This indicates that at most \( \epsilon(C_l) \) can be added to the dual variable \( y_{C_l} \) in component \( C_l \) without violating the dual penalty packing constraint \( \sum_{S \subseteq C_l} y_S \leq \sum_{v \in C_l} p_v \). Since after the addition of \( \epsilon(C_l) \) to the dual variable \( y_{C_l} \), the dual penalty packing constraint \( \sum_{S \subseteq C_l} y_S \leq \sum_{v \in C_l} p_v \) becomes tight therefore the component \( C_l \) decides to deactivate itself. The node \( u \in C_l \) updates its local variables \( d_u = d_u + \epsilon(C_l) \) and \( W_u = W_u + \epsilon(C_l) \). In addition, corresponding \( d_h(C_l) \) is also updated accordingly.

Therefore after receiving a connect message \( C_l \) correctly computes each of its local variable without violating any of the dual constraints.

Similarly if \( u \) receives \( \langle \text{proceed}(d_h(C_k)) \rangle \) from a node \( v \in C_k \) over the edge \( e \) then first \( C_l \) becomes active and then node \( u \) initializes its \( d_u = d_h(C_k) \), \( W_u(C_l) = d_h(C_k) \) and \( d_h(C_l) = d_h(C_k) \). Note that if a component receives a proceed message then the state of each component in its neighborhood is either sleeping or inactive. This is because a proceed message is generated only if the currently existing active component becomes inactive. Let \( e' \in \delta(C_l) \) be the MOE of \( C_l \) which connects to a node \( w \in C_p \neq C_l \). Then \( C_l \) computes its \( \epsilon(C_l) \) as follows:

\[
\epsilon_1(C_l) = \begin{cases} 
\frac{w_{e'} - d_u - d_h(C_l)}{2}, & \text{if } CS(C_p) = \text{sleeping} \\
-w_{e'} - d_u - d_w, & \text{if } CS(C_p) = \text{inactive}
\end{cases}
\]

\[
\epsilon_2(C_l) = TP(C_l) - W(C_l) = p_u - W_u(C_l)
\]

and \( \epsilon(C_l) = \min(\epsilon_1(C_l), \epsilon_2(C_l)) \). Therefore it is clear that \( d_h(C_k) \) is used by \( C_l \) to compute
its $\epsilon(C_1)$ value in case $CS(C_1) = sleeping$. Now in the same way we have shown for the case of receiving connect message it can be shown that upon receiving $\langle proceed(d_h(C_j)) \rangle$, $C_i$ correctly computes each of its local variable without violating any of the dual constraints.

We claim that the approximation factor achieved by our proposed distributed PCST algorithm for the PCST problem on a graph of $n$ nodes is $(2 - \frac{1}{n-1})$ of the optimal (OPT). This can be proved straightforwardly from the facts that $d_v = \sum_{S:\forall S} y_S$ for each node $v \in V$ and $W(C) = \sum_{S \subseteq C} y_S$ for each component $C$. Let $OPT_{LP}$ and $OPT_{IP}$ be the optimal solutions to (LP) and (IP) of PCST problem respectively. Then it is obvious that $\sum_{S \subseteq V} y_S \leq OPT_{LP} \leq OPT_{IP}$.

**Theorem 6.11** Distributed PCST algorithm selects a set of edges $F'$ and a set of vertices $X$ such that

$$\sum_{e \in F'} w_e + \sum_{v \in X} p_v \leq \left(2 - \frac{1}{n-1}\right) \sum_{S \subseteq V} y_S \leq \left(2 - \frac{1}{n-1}\right) OPT_{IP}$$

(1)

Hence the D-PCST algorithm is a $(2 - \frac{1}{n-1})$ approximation algorithm for PCST problem.

**Proof.** In the construction of $F'$ if a node $v \in V$ is not covered by $F'$ then $v$ must be belong to some component deactivated at some point of execution of the algorithm. Let $X = \{C_1, C_2, \ldots, C_z\}$ is the set of deactivated components whose nodes are not covered in $F'$. Therefore $X$ can be considered as a disjoin set of nodes and each set is some $C_j$ for $j : 1 \leq j \leq z$. Since each $C_j$ is a deactivated component therefore it follows the fact that $\sum_{S \subseteq C_j} y_S = \sum_{v \in C_j} p_v$. For each edge $e \in F'$ it also follows that $\sum_{S: e \in \delta(S)} y_S = w_e$ and this implies $\sum_{e \in F'} w_e = \sum_{e \in F'} \sum_{S: e \in \delta(S)} y_S$. Putting these in the inequality (1) we get

$$\sum_{e \in F'} \sum_{S: e \in \delta(S)} y_S + \sum_{j=1}^{z} \sum_{S \subseteq C_j} y_S \leq \left(2 - \frac{1}{n-1}\right) \sum_{S \subseteq V} y_S$$

(2)

Now it can be shown by the method of induction that for each $\epsilon(C) > 0$ computed by a component $C$, the inequality (2) always holds. Note that if $\epsilon(C) \leq 0$ then component $C$ use it to adjust its dual variables in such a way that dual constraints are not violated. This is ensured by the Lemma 6.10.

At the beginning of the algorithm the inequality (2) holds since $F' = \phi$, the component containing the root node $r$ is the only trivial single node tree and $y_C = 0$ for each single node component $C$. Let $C$ is the set of components in the graph when a component $C$ computes its $\epsilon(C)$. The set of components of $C$ are categorized into two types of components namely type $A$ and type $I$ as follows:

- A component $C' \in C$ is denoted as type $A$ if $CS(C') = active$ or $CS(C') = sleeping$ and $TP(C') > d_h(C) + (0, \epsilon(C))$

- A component $C' \in C$ is denoted as type $I$ if $CS(C') = inactive$ or $CS(C') = sleeping$ and $TP(C') \leq d_h(C)$
The type of a component \( C \) is denoted as \( \text{type}(C) \). To show that the inequality (2) always holds first we construct a special graph termed as \( H = (V', E') \). The set of components of \( C \) are considered as the set of vertices \( V' \) of the graph \( H \). \( V' \) contains two types of vertices namely type \( A \) and type \( I \). The set of edges \( E' = \{ e : e \in \delta(C' \cap F') \wedge \text{type}(C') = A \} \). All isolated vertices of type \( I \) are discarded from the graph \( H \). Let \( N_A \) denotes the set of vertices of type \( A \), \( N_I \) denotes the set of vertices of type \( I \), \( N_D \) denotes the set of vertices of type \( A \) that corresponds to some \( C_j \) for \( j : 1 \leq j \leq z \) and \( d_v \) denotes the degree of a vertex \( v \) in graph \( H \). Note that degree of each vertex \( v \in N_D \) is zero, i.e. \( N_D = \{ v \in N_A : d_v = 0 \} \). For each \( \epsilon(C') > 0 \), maximum increment in the left hand side of the inequality (2) is \( \sum_{v \in N_A} \epsilon_v d_v + \sum_{v \in N_I} \epsilon_v \), where \( \epsilon_v \in (0, \epsilon(C')] \) for each vertex \( v \in V \) (note that here \( \epsilon_v \) is the actual adjusted value for a vertex \( v \) in \( H \) which is corresponding to the component \( C_v \) and this correct adjustment of \( \epsilon_v \) to the dual variable \( y_{C_v} \) is ensured by the Lemma 6.10). On the other hand maximum increment in the right hand side of the inequality is \( (2 - \frac{1}{n-1}) \sum_{v \in N_A} \epsilon_v \). Therefore we can write

\[
\sum_{v \in N_A - N_D} \epsilon_v d_v + \sum_{v \in N_I} \epsilon_v \leq (2 - \frac{1}{n-1}) \sum_{v \in N_A} \epsilon_v
\]

Writing the above inequality in details we get

\[
\sum_{v \in N_A - N_D} \epsilon_v d_v \leq (2 - \frac{1}{n-1}) \sum_{v \in N_A - N_D} \epsilon_v + (2 - \frac{1}{n-1}) \sum_{v \in N_I} \epsilon_v - \sum_{v \in N_D} \epsilon_v
\]

Since degree of each vertex in \( N_D \) is zero therefore the coefficient \( (2 - \frac{1}{n-1}) \) of the term \( (2 - \frac{1}{n-1}) \sum_{v \in N_D} \epsilon_v \) is actually equal to 1. This implies the following inequality

\[
\sum_{v \in N_A - N_D} \epsilon_v d_v \leq (2 - \frac{1}{n-1}) \sum_{v \in N_A - N_D} \epsilon_v
\]

Rewriting the left hand side of the above inequality in terms of set \( N_A, N_I \) and \( N_D \) we get

\[
\sum_{v \in N_A - N_D} \epsilon_v d_v \leq \sum_{v \in (N_A - N_D) \cup N_I} \epsilon_v d_v - \sum_{v \in N_I} \epsilon_v d_v \tag{3}
\]

Before continuing with the proof we show that in graph \( H \) there can be at most one leaf vertex of type \( I \) which is corresponding to the component containing \( r \). Suppose by contradiction \( v \in V' \) is a leaf vertex of type \( I \) in graph \( H \) which is not the root vertex containing \( r \) and is adjacent to the edge \( e : SE(e) = \text{branch} \). Let \( C_v \) be the inactive component corresponding to the vertex \( v \). Since \( C_v \) is the leaf of \( H \) therefore the edge \( e \in F' \). Note that \( F' \) is the set of \( \text{branch} \) edges selected for the construction of the PCST while pruning phase of the algorithm terminates. Since the state of \( C_v \) is inactive and it does not contain \( r \), therefore it is deactivated at some point of execution of the algorithm and each \( u \in C_v \) is labelled. Furthermore, since \( C_v \) is a leaf component, therefore no node \( u \in C_v \) can be an intermediate node on the path of \( \text{branch} \) edges between the root vertex
and a vertex which is unlabelled. Since each node \( u \in C_v \) is labelled and \( C_v \) is an inactive leaf component therefore by the pruning phase of the algorithm each node \( u \in C_v \) is pruned and \( SE(e') = \text{basic} \) for each edge \( e' : e' \in \text{Adj}(u) \land SE(e') = \text{branch} \). In this case one of the \( e' \) must be \( e : SE(e) = \text{basic}, \) a contradiction to the fact that \( SE(e) = \text{branch} \). Therefore except the root vertex, all other vertex of type \( I \) are non leaf vertex in graph \( H \).

Using the above fact and since \( \epsilon_v \in (0, \epsilon(C)) \), therefore replacing each \( \epsilon_v \) by \( \epsilon(C) \) and we get the inequality (3) as follows

\[
\sum_{v \in N_A - N_D} \epsilon(C)d_v \leq 2\epsilon(C)((|N_A - N_D|) + 2|N_I| - 2 - 2|N_I| + 1
\]

(In the above inequality we use the fact that sum of degrees of all vertices is \( 2m \) where \( m \) is the total number of edges in the graph.)

Since \( (N_A - N_D) \) and \( N_I \) are disjoint therefore \(|(N_A - N_D) \cap N_I| = |\phi| = 0 \) then

\[
\sum_{v \in N_A - N_D} d_v = 2(|N_A - N_D|) + 2|N_I| - 2 - 2|N_I| + 1
\]

\[
= 2(|N_A - N_D|) - 1
\]

\[
= (2 - \frac{1}{|N_A - N_D|})|N_A - N_D|
\]

\[
\leq (2 - \frac{1}{n - 1})|N_A - N_D|
\]

The last inequality holds since the number of type \( A \) components is at most \( n - 1 \) for \( n \) node graph, i.e. \(|N_A - N_D| \leq (n - 1) \). Therefore the inequality (1) always holds for every computed value \( \epsilon(C) \) by a component \( C \) in the graph. \( \square \)

**Message complexity**

We determine here the upper bound on the number of messages exchanged during the execution of the distributed algorithm. In the algorithm the leader of a component \( C \) starts finding its \( \epsilon(C) \) by executing the procedure \( \text{proc\_initiate}() \) which in turn generates different messages and procedures until it finds its \( \epsilon(C) \). Depending on \( \epsilon(C) \), the leader of the component \( C \) decides to take any one of the following actions: (i) merging (ii) deactivation (iii) sending \( \langle \text{proceed} \rangle \), (iv) sending \( \langle \text{back} \rangle \) or (v) start of the pruning phase (only root component initiates the pruning phase). Lemma 6.5 ensures that at most \(|V| - 1 \) rounds of \( \text{proc\_initiate}() \) are required for the action of sending \( \langle \text{back} \rangle \) and exactly one round of action of pruning phase is required till the termination of the growth phase. We observe that if none of the four consecutive rounds of \( \text{proc\_initiate}() \) initiates the action of sending \( \langle \text{back} \rangle \) then any one of the following events is guaranteed to happen during these four rounds: (i) sum of the number of components decreases (ii) one of the sleeping or active components decreases in the graph. Initially the state of the root component \( C_r \) is inactive and it executes \( \text{proc\_initiate}() \) only once to compute is \( \epsilon(C_r) \) and send \( \langle \text{proceed} \rangle \) to a neighboring component to
take further actions. In the worst case at most $4(|V| - 1)$ rounds of $\text{proc}_{\text{initiate}}()$ are required to decrease the sum of the number of components and becomes one or at most $4(|V| - 1)$ rounds of $\text{proc}_{\text{initiate}}()$ are required to change the state of each sleeping/active component to the state of inactive. Therefore in the worst case at most $8(|V| - 1) + (|V| - 1) + 1 + 1 = 9|V| - 7$ rounds of $\text{proc}_{\text{initiate}}()$ are required until the termination of the growth phase.

By Lemma 6.2 it is shown that each round of $\text{proc}_{\text{initiate}}()$ can generates at most $(7|V| + 2|E| - 8)$ number of messages. Therefore number of messages exchanged until the termination of the growth phase are at most $(9|V| - 7)(7|V| + 2|E| - 8)$. In the pruning phase at most $2|E|$ number of messages are exchanged since at most $|E|$ number $\langle\text{prune}\rangle$ and $|E|$ number $\langle\text{backward\_prune}\rangle$ are sent in the graph. Therefore total number of messages exchanged until the termination of the distributed PCST are at most $(9|V| - 7)(7|V| + 4|E| - 8)$. By simplifying the message complexity of the D-PCST algorithm is $O(|V|^2 + |E||V|)$.

**Deadlock Issue**

We show here that deadlock does not exist. First we claim that deadlock can not occur when a component $C$ is in the phase of finding its $\epsilon(C)$. To find $\epsilon(C)$ the $\langle\text{initiate}\rangle$ is broadcast over the branch edges inside $C$. Upon receiving $\langle\text{initiate}\rangle$, each frontier node $v \in C$ sends $\langle\text{test}\rangle$ on each edge $e : e \in Adj(v) \land SE(e) \neq \text{branch} \land SE(e) \neq \text{rejected}$. Upon receiving $\langle\text{test}\rangle$ message each node $u$ immediately response with $\langle\text{status}\rangle$ or $\langle\text{reject}\rangle$ and does not wait for any event to occur since it has no dependency on any other events to response to a $\langle\text{test}\rangle$. Therefore a frontier node does not have to wait infinitely to finds its $\epsilon_1(v)$ and each frontier node $v \in C$ reports its $\epsilon_1(v)$ to the leader and it is guaranteed that leader of a component $C$ finds its $\epsilon(C)$ in finite amount of time. Therefore a component $C$ never experience the situation of deadlock while it computes its $\epsilon(C)$. Now we show that at any point of time during the execution of the algorithm deadlock can not occur among the components too.

We already mention that depending on the value of $\epsilon(C)$, the leader of the component $C$ decides to take any one of the following actions: (i) merging (ii) deactivation (iii) sending $\langle\text{proceed}\rangle$, (iv) sending $\langle\text{back}\rangle$ or (v) starts of the pruning phase. Deactivation deals with a single component and hence no deadlock issue. In case of sending $\langle\text{proceed}\rangle$ or sending $\langle\text{back}\rangle$ each receiving node forwards these messages in the appropriate path without any delay as forwarding events does not depends on any other events of the node. Therefore these two cases does not lead to any deadlock issue. Similarly in the pruning phase there is no issue of deadlock since for pruning a component does not depend on any other components. Now consider the action of merging. This is only the case when a component needs to wait for another component to proceed further. The component $C$ sends $\langle\text{connect}\rangle$ to merge with the component $C'$ where $C'$ is a neighboring component of $C$. Upon receiving $\langle\text{connect}\rangle$, $C'$ responds to $C$ in finite amount of time. If $CS(C') = \text{inactive}$ then $C'$ immediately sends $\langle\text{accept}\rangle$. If $CS(C') = \text{sleeping}$, then first it changes $CS(C') = \text{active}$ and then finds its $\epsilon(C')$ and depending on $\epsilon(C')$ it sends $\langle\text{accept}\rangle$ or $\langle\text{refind\_epsilon}\rangle$ to $C$. Since $C'$ does not depends on any event that delays the process of findings its $\epsilon(C')$ and response to $C$ therefore there is no possibility of deadlock. In our proposed D-PCST algorithm, component grows
by sequential merging and no concurrent merging is allowed to happen. All of the observations ensure that deadlocks do not exist.

7 Conclusion and future directions

In this paper we propose a distributed PCST algorithm with an approximation factor of \( (2 - \frac{1}{n-1}) \) of the optimal. This approximation factor matches with the same of GW-algorithm. However we believe that our algorithm is the first distributed algorithm for PCST that achieves this approximation factor. We also analyze the correctness of the proposed algorithm. Detailed proof of correctness is provided in the appendix. It will be a challenging problem to reduce the approximation factor of our algorithm further. Moreover in this algorithm a single merging (of two distinct components) happens at a time. It will be interesting to investigate a correct distributed algorithm where concurrent merging happens. Tolerating different types of changes in the underlying graph during the execution of the distributed algorithm will lead to a number of research issues to be addressed.

References

[1] AbdI, A., Feldmann, A. E., Guenin, B., Konemann, J., and Sanita, L. Lehman’s theorem and the directed steiner tree problem. *SIAM Journal on Discrete Mathematics* 30, 1 (2016), 141–153.

[2] Agrawal, A., Klein, P., and Ravi, R. When trees collide: An approximation algorithm for the generalized steiner problem on networks. *SIAM Journal of Computing* 24, 3 (1995), 440–456.

[3] Alexander, Z. An 11/6-approximation algorithm for the network steiner problem. *Algorithmica* 9, 5 (1993), 463–470.

[4] Alexander, Z. Better approximation bounds for the network and euclidean steiner tree problems. Tech. rep., 1996.

[5] Alvarez-Miranda, E., Candida, A., Chen, X., Hu, X., and Li, B. Efficient algorithms for the prize collecting steiner tree problems with interval data. *AAIM* (2010), 13–24.

[6] Alvarez-Miranda, E., and and Paolo Toth, I. L. Exact approaches for solving robust prize-collecting steiner tree problems. *European Journal of Operational Research* 229 (2013), 599–612.

[7] Archer, A., Bateni, M., and Hajiaghayi, M. Improved approximation algorithms for prize-collecting steiner tree and tsp. *SIAM Journal on Computing* 40, 2 (2011), 309–332.

[8] Arora, S. Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems. *J. ACM* 45, 5 (1998), 753–782.
[9] BALAS, E. The prize collecting traveling salesman problem. *Networks* 19 (1989), 621–636.

[10] BAUER, M., KLAU, G. W., AND REINERT, K. Accurate multiple sequence-structure alignment of rna sequences using combinatorial optimization. *BMC Bioinformatics* 8, 1 (2007).

[11] BERMAN, P., AND RAMAIYER, V. Improved approximations for the steiner tree problem. *Journal of Algorithms* 17, 3 (1994), 381 – 408.

[12] BIENSTOCK, D., GOEMANS, M. X., SIMCHILEVI, D., AND WILLIAMSON, D. A note on the prize collecting traveling salesman problem. *Mathematical Programming* 59 (1993), 413–420.

[13] BYRKA, J., GRANDONI, F., ROTHVOSS, T., AND SANITÀ, L. An improved lp-based approximation for steiner tree. In *Proceedings of the Forty-second ACM Symposium on Theory of Computing* (New York, NY, USA, 2010), STOC ’10, ACM, pp. 583–592.

[14] CANUTO, S., RESENDE, M., AND RIBEIRO, C. Local search with perturbations for the prize-collecting steiner tree problem in graphs. *Networks* 38, 1 (2001), 50–58.

[15] CHARIKAR, M., CHEKURI, C., CHEUNG, T.-Y., DAI, Z., GOEL, A., GUHA, S., AND LI, M. Approximation algorithms for directed steiner problems. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms* (1998), SODA ’98, pp. 192–200.

[16] CHLEBÍK, M., AND CHLEBÍKOVÁ, J. The steiner tree problem on graphs: Inapproximability results. *Theoretical Computer Science* 406, 3 (2008), 207–214.

[17] DITTRICH, M. T., KLAU, G. W., ROSENWALD, A., DANDEKAR, T., AND MILLER, T. Identifying functional modules in protein-protein interaction networks: an integrated exact approach. *Intelligence System for Molecular Biology (ISMB)* 24, 13 (2008), 119–141.

[18] DU, D., AND HU, X. *Steiner Tree Problems In Computer Communication Networks*. World Scientific Publishing Co., Inc., 2008.

[19] FEOFILOFF, P., FERNANDES, C. G., FERREIRA, C. E., AND DE PINA, J. C. Primal-dual approximation algorithms for the prize-collecting steiner tree problem. *Information Processing Letter* 103, 5 (2007), 195–202.

[20] GALLAGER, R. G., HUMBLET, P. A., AND SPIRA, P. M. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems* 1 (1983), 66–77.

[21] GOEMANS, M. X., AND WILLIAMSON, D. E. A general approximation technique for constrained forest problems. *SIAM Journal on Applied Mathematics* (April 1995) 24, 2 (1995), 296–317.
[22] GOEMANS, M. X., AND WILLIAMSON, D. E. The primal-dual method for approximation algorithms and its application to network design problems. In D S Hochbaum, editor, *Approximation problem for NP-hard problems*, PSW publishing company, Boston (1997), 144–191.

[23] GOEMANS, M. X., AND WILLIAMSON, D. P. A general approximation technique for constrained forest problems. *SIAM J. Comput.* 24, 2 (1995), 296–317.

[24] HANAN, M. On steiner’s problem with rectilinear distance. *SIAM Journal on Applied Mathematics* (March 1966) 14, 2 (1966), 255–265.

[25] HAOUARI, M., LAYEB, S. B., AND SHERALI, H. D. Algorithmic expedients for the prize collecting steiner tree problem. *Discrete Optimization* 7 (2010), 32–47.

[26] HAUPTMANN, M., AND KARPINSKI, M. A compendium on steiner tree problems. [http://theory.cs.uni-bonn.de/info5/steinerkompendium/netcompendium.html](http://theory.cs.uni-bonn.de/info5/steinerkompendium/netcompendium.html) (Retrieved April 2015).

[27] HWANG, F. K. On steiner minimal trees with rectilinear distance. *SIAM Journal on Applied Mathematics*, (January 1976) 30, 1 (1976), 104–114.

[28] JOHNSON, D. S., MINKOFF, M., AND PHILLIPS, S. The prize collecting steiner tree problem: Theory and practice. In *Proceeding of ACM-SIAM Symposium on Discrete Mathematics* (2000), 760–769.

[29] KARP, R. M. Reducibility among combinatorial problems. In *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York.* (1972), pp. 85–103.

[30] KLAU, G. W., LJUBIČ, I., MOSER, A., MUTZEL, P., NEUNER, P., PFERSCHY, U., RAIDL, G., AND WEISKIRCHER, R. Combining a Memetic Algorithm with Integer Programming to Solve the Prize-Collecting Steiner Tree Problem. 2004, pp. 1304–1315.

[31] KRUSKAL, J. B. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proceedings of the American Mathematical Society* 7, 1 (1956), 48–50.

[32] LENGAUER, T. *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[33] LENZEN, C., AND PATT-SHAMIR, B. Improved distributed steiner forest construction. *In Proceedings of the 33th Annual ACM Symposium on Principles of Distributed Computing (PODC)* (2014).

[34] LJUBIC, I., WEISKIRCHER, R., PFERSCHY, U., KLAU, G. W., MUTZEL, P., AND FISCHETTI, M. An algorithmic framework for the exact solution of the prize-collecting steiner tree problem. *Mathematical Programming Series B*, 105 (2006), 427–449.
[35] MAGNANTI, T., AND WONG, R. Network design and transportation planning: models and algorithms. *Transportation Science* 18 (1984), 1–55.

[36] MAREK, K., AND ALEXANDER, Z. New approximation algorithms for the steiner tree problems. *Journal of Combinatorial Optimization* 1, 1 (1997), 47–65.

[37] PRIM, R. C. Shortest connection networks and some generalizations. *Bell System Technical Journal* 36, 6 (1957), 1389–1401.

[38] PRODON, A., DENEGRE, S., AND LIEBLING, T. M. Locating leak detecting sensors in a water distribution network by solving prize-collecting steiner arborescence problems. *Mathematical Programming* 124, 1 (2010), 119–141.

[39] PRMEL, H. J., AND STEGER, A. A new approximation algorithm for the steiner tree problem with performance ratio 5/3. *Journal of Algorithms* 36, 1 (2000), 89 – 101.

[40] ROBINS, G., AND ZELIKOVSKY, A. Improved steiner tree approximation in graphs. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms* (2000), SODA ’00, pp. 770–779.

[41] ROBINS, G., AND ZELIKOVSKY, A. Tighter bounds for graph steiner tree approximation. *SIAM J. Discret. Math.* 19, 1 (2005), 122–134.

[42] SHAN CAROL HUANG, S. A constraint optimization framework for discovery of cellular signaling and regulatory networks. Phd thesis, Massachusetts Institute of Technology (2011), 882–887.

[43] VIJAYANARASIMHAN, S., AND GRAUMAN, K. Efficient region search for object detection. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR) 1* (2011), 1401–1408.

[44] WALD, J. A., AND COLBOURN, C. J. Steiner trees, partial 2trees, and minimum ifi networks. *Networks* 13 (1983), 159–167.

[45] WATEL, D., WEISSER, M.-A., BENTZ, C., AND BARTH, D. Steiner problems with limited number of branching nodes. In *Revised Selected Papers of the 20th International Colloquium on Structural Information and Communication Complexity - Volume 8179* (2013), SIROCCO 2013, pp. 310–321.

[46] WILLIAMSON, D. P., GOEMANS, M. X., MIHAIL, M., AND VAZIRANI, V. V. A primal-dual approximation algorithm for generalized steiner network problems. *Combinatorica* 15, 3 (1995), 435–454.

[47] ZOSIN, L., AND KHULLER, S. On directed steiner trees. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (2002), SODA ’02, pp. 59–63.
8 Appendix
Algorithm 1: The D-PCST algorithm: pseudocode for node $v$

1: Upon receiving no message
2: execute procedure `initialization()`
3: if $v = r$ then \[\triangleright \text{spontaneous awaken of the root node}\]
4: \(CS \leftarrow \text{inactive};\)
5: \(\text{root\_flag} \leftarrow \text{TRUE};\)
6: \(\text{prize\_flag} \leftarrow \text{FALSE};\)
7: execute procedure `proc\_initiate()`
8: else
9: \(CS \leftarrow \text{sleeping};\)
10: \(\text{root\_flag} \leftarrow \text{FALSE};\)
11: \(\text{prize\_flag} \leftarrow \text{TRUE};\)
12: end if

13: procedure `initialization()`
14: for all $e \in \text{Adj}(v)$ do
15: \(SE(e) \leftarrow \text{basic};\)
16: \(\text{RFE}(e) \leftarrow \text{FALSE};\) \[\triangleright\text{RFE: Refind epsilon}\]
17: \(\text{ETP}(e) \leftarrow \text{FALSE};\) \[\triangleright\text{ETP: Edge for termination path}\]
18: end for
19: \(d_h \leftarrow 0; d_v \leftarrow 0; WC \leftarrow 0; \text{labelled\_flag} \leftarrow \text{FALSE}; \text{prune\_msg\_count} \leftarrow 0; \text{proceed\_in\_edge} \leftarrow \phi; \text{proceed\_flag} \leftarrow \text{FALSE}; \text{leader\_flag} \leftarrow \text{FALSE};\)
20: end procedure

21: procedure `proc\_initiate()`
22: \(SN \leftarrow \text{find}; \text{find\_count} \leftarrow 0; \text{best\_epsilon} \leftarrow \infty; \text{best\_edge} \leftarrow \phi; LC \leftarrow v; TP \leftarrow 0; PF \leftarrow \text{FALSE}; \text{proceed\_edge} \leftarrow \phi;\)
23: for all $e \in \text{Adj}(v)$ do
24: if $SE(e) = \text{branch}$ then
25: \hline
26: send \((\text{initiate}(LC, SN))\) on $e$
27: \hline
28: \(\text{find\_count} \leftarrow \text{find\_count} + 1;\) \[\triangleright\text{count the number of } \langle \text{initiate} \rangle \text{ that are sent}\]
29: end if
30: end for
31: execute procedure `proc\_test()`
32: end procedure
Upon receiving $\langle \text{initiate}(L, S) \rangle$ on edge $e$

$SN \leftarrow S; \text{find\_count} \leftarrow 0; \text{best\_epsilon} \leftarrow \infty; \text{best\_edge} \leftarrow \phi; \text{LC} \leftarrow L; TP \leftarrow 0; PF \leftarrow \text{FALSE}; \text{proceed\_edge} \leftarrow \phi; \text{in\_branch} \leftarrow e$

for all $e' \in \text{Adj}(v): e' \neq e$

if $SE(e') = \text{branch}$ then

send $\langle \text{initiate}(L, S) \rangle$ on $e'$

$\text{find\_count} \leftarrow \text{find\_count} + 1; \triangleright$ count the number of $\langle \text{initiate} \rangle$ that are sent

end if

end for

if $S = \text{find}$ then

execute procedure $\text{proc\_test}()$

end if

\begin{verbatim}
42: \textbf{procedure} $\text{proc\_test}()$
43: \hfill $\text{test\_count} \leftarrow 0;$
44: \hfill for all $e \in \text{Adj}(v)$ do
45: \hfill if $SE(e) = \text{basic}$ then
46: \hfill \hfill send $\langle \text{test}(\text{LC}) \rangle$ on $e$
47: \hfill \hfill $\text{test\_count} \leftarrow \text{test\_count} + 1; \triangleright$ count the number of $\langle \text{test} \rangle$ that are sent
48: \hfill \hfill end if
49: \hfill end for
50: \hfill end procedure
\end{verbatim}

Upon receiving $\langle \text{test}(L) \rangle$ on edge $e$

if $LC = L$ then

send $\langle \text{reject} \rangle$ on $e$

else

send $\langle \text{status}(CS, d_v) \rangle$ on $e$.

end if

Upon receiving $\langle \text{status}(NS, d_u) \rangle$ on edge $e$

$\text{test\_count} \leftarrow \text{test\_count} - 1;$

if $NS = \text{sleeping and } CS = \text{active}$ then

$\epsilon_1 \leftarrow \frac{w_e - d_v - d_h}{2}$;

else if $NS = \text{inactive and } CS = \text{active}$ then

$\epsilon_1 \leftarrow w_e - d_v - d_u$;

else if $NS = \text{sleeping and } CS = \text{inactive}$ then

$\epsilon_1 \leftarrow w_e - d_u - d_h$;

end if
else if $NS = \text{inactive}$ and $CS = \text{inactive}$ then
  if $RFE(e) = \text{TRUE}$ then
    $\epsilon_1 \leftarrow w_e - d_v - d_u$;
  else
    $\epsilon_1 \leftarrow \infty$;
  end if
end if

if $\epsilon_1 < \text{best}_\epsilon$ then
  $\text{best}_\epsilon \leftarrow \epsilon_1$;
  $\text{best}\_edge \leftarrow e$;
end if

execute procedure proc\_report()

upon receiving \langle reject \rangle on edge $e$
  $\text{test\_count} \leftarrow \text{test\_count} - 1$;
if $SE(e) = \text{basic}$ then
  $SE(e) \leftarrow \text{rejected}$;
end if

if $\text{proceed\_in\_edge} = e$ then
  $\text{proceed\_flag} \leftarrow \text{FALSE}$;
  $\text{proceed\_in\_edge} \leftarrow \phi$;
end if
execute procedure proc\_report()

procedure proc\_report()
  if $\text{find\_count} = 0$ and $\text{test\_count} = 0$ then \(\triangleright\) Receives responses for each \langle initiate\rangle and \langle test\rangle
    $SN \leftarrow \text{found}$;
  if $d_h < d_v$ then
    $d_h \leftarrow d_v$;
  end if
  if $CS = \text{active}$ then
    $TP \leftarrow TP + p_v$ \(\triangleright\) $TP$ (Total Prize) of the subtree rooted at $v$
  end if
  if $\text{proceed\_flag} = \text{TRUE}$ then
    $PF \leftarrow \text{TRUE}$;
  end if
end if
if in_branch ≠ φ then
   send ⟨report(bst_epsilon, d_h, TP, PF)⟩ on in_branch
else
   execute procedure proc_merge_or_deactivate_or_proceed()
end if
end if

Upon receiving ⟨report(ε₁, d_k, T, P)⟩ on edge e
find_count = find_count - 1;
if P = TRUE then
   proceed_edge ← e;
   PF ← TRUE;
end if
if CS = active then
   TP ← TP + T;
end if
if d_h < d_k then
   d_h ← d_k;
end if
if ε₁ < bst_epsilon then
   bst_edge ← e;
end if
execute procedure proc_report()

procedure proc_merge_or_deactivate_or_proceed()
   ε₁ ← bst_epsilon
   if root_flag = FALSE and CS = active then
      ε₂ ← TP − WC;
      if ε₁ < ε₂ then
         bst_epsilon ← ε₁
      else
         bst_epsilon ← ε₂
      end if
   end if
   if bst_epsilon = ε₂ then
      CS ← inactive;
      d_v ← d_v + ε₂;
      WC ← WC + ε₂;
   end if
136: \[ \text{root\_flag} \leftarrow \text{FALSE}; \]
137: \[ \text{d}_h \leftarrow \text{d}_h + \epsilon_2; \]
138: \[ \text{labelled\_flag} \leftarrow \text{TRUE}; \]
139: \[ \text{deactivate\_flag} \leftarrow \text{TRUE}; \] \hspace{1cm} \triangleright \text{deactivate\_flag is a temporary variable}
140: \[ \text{send} \langle \text{update\_info}(\epsilon_2, \text{root\_flag}, \text{deactivate\_flag}, WC, d_h) \rangle \text{ on all } e \in \text{Adj}(v) \] such that \( SE(e) = \text{branch} \)
141: \[ \text{if } \epsilon_1 \neq \infty \text{ then} \quad \triangleright \text{Exists at least one sleeping neighbor component.} \]
142: \[ \text{execute procedure } \text{proc\_initiate}() \quad \triangleright \text{Computes } \epsilon_1 \text{ to send } \langle \text{proceed} \rangle \]
143: \[ \text{end if} \]
144: \[ \text{else} \hspace{1cm} \triangleright \text{Start the merge procedure at the leader node} \]
145: \[ \text{if } SE(\text{best\_edge}) = \text{branch} \text{ then} \]
146: \[ \text{send} \langle \text{merge}(\text{best\_epsilon}, \text{d}_h) \rangle \text{ on best\_edge} \]
147: \[ \text{else} \]
148: \[ \text{send} \langle \text{connect}(v, WC, d_v, d_h) \rangle \text{ on best\_edge}; \]
149: \[ \text{if } RFE(\text{best\_edge}) = \text{TRUE} \text{ then} \]
150: \[ RFE(\text{best\_edge}) \leftarrow \text{FALSE}; \]
151: \[ \text{end if} \]
152: \[ \text{end if} \]
153: \[ \text{end if} \]
154: \[ \text{else if } \text{CS} = \text{inactive} \text{ then} \]
155: \[ \text{if } (\epsilon_1 = \infty) \text{ then} \]
156: \[ \text{if } v = r \text{ then} \quad \triangleright \text{Starts of pruning phase at the root node } r. \]
157: \[ \text{for all } e \in \text{Adj}(v) \text{ do} \]
158: \[ \text{if } SE(e) = \text{branch} \text{ or } \text{ETP}(e) = \text{TRUE} \text{ then} \]
159: \[ \text{send} \langle \text{prune} \rangle \text{ on } e \]
160: \[ \text{end if} \]
161: \[ \text{if } SE(e) = \text{branch} \text{ then} \]
162: \[ \text{prune\_msg\_count} \leftarrow \text{prune\_msg\_count} + 1; \]
163: \[ \text{end if} \]
164: \[ \text{end for} \]
165: \[ \text{else} \hspace{1cm} \triangleright \text{Start of sending } \langle \text{back} \rangle \]
166: \[ \text{if } \text{proceed\_edge} = \phi \text{ and } \text{proceed\_flag} = \text{TRUE} \text{ then} \]
167: \[ \text{proceed\_flag} \leftarrow \text{FALSE}; \]
168: \[ \text{send} \langle \text{back} \rangle \text{ on } \text{proceed\_in\_edge} \]
169: \[ \text{proceed\_in\_edge} \leftarrow \phi; \]
170: \[ \text{else} \]
171: \[ \text{send} \langle \text{disable\_proceed} \rangle \text{ on } \text{proceed\_edge} \]
172: \[ \text{end if} \]
173: \[ \text{end if} \]
else if ($\epsilon_1 \neq \infty$) then
    send $\langle$proceed($d_h$)$\rangle$ on best_edge
    if $SE$(best_edge) = basic then
        $ETP$(best_edge) ← TRUE;
    end if
    if $RFE$(best_edge) = TRUE then
        $RFE$(best_edge) ← FALSE;
    end if
end if
end if

Upon receiving $\langle$merge($\epsilon, d_k$)$\rangle$ on edge $e$
if $SE$(best_edge) = branch then
    $\triangleright$ receiving node is an intermediate node
    send $\langle$merge($\epsilon, d_k$)$\rangle$ on best_edge
else
    $\triangleright$ receiving node is a frontier node
    if $CS$ = active then
        send $\langle$connect($v, WC, d_v, d_k$)$\rangle$ on best_edge;
        if $RFE$(best_edge) = TRUE then
            $RFE$(best_edge) ← FALSE;
        end if
    end if
    else if $CS$ = inactive then
        send $\langle$proceed($d_k$)$\rangle$ on best_edge
    end if
end if

Upon receiving $\langle$disable_proceed$\rangle$ on edge $e$
if $proceed$-edge = $\phi$ and $proceed$-flag = TRUE then
    $proceed$-flag ← FALSE;
    send $\langle$back$\rangle$ on $proceed$-in_edge
    $proceed$-in_edge ← $\phi$;
else
    send $\langle$disable_proceed$\rangle$ on $proceed$-edge
end if

Upon receiving $\langle$back$\rangle$ on edge $e$
if $in$-branch $\neq \phi$ then
    send $\langle$back$\rangle$ on $in$-branch
209: else
210:   execute procedure proc_initiate()
211: end if

212: Upon receiving \{proceed(d_k)\} on edge e
213: if $SE(e) = \text{branch}$ and $in_{branch} = e$ then
214:   send \{proceed(d_k)\} on best_edge
215:   if $SE(best\_edge) = \text{basic}$ then
216:     $ETP(best\_edge) \leftarrow \text{TRUE};$
217:   end if
218:   if $RFE(best\_edge) = \text{TRUE}$ then
219:     $RFE(best\_edge) \leftarrow \text{FALSE};$
220: end if
221: else if $SE(e) = \text{basic}$ then \hspace{1em} $\triangleright$ Edge e is a non branch edge that means a \textit{basic} edge
222:   proceed\_flag $\leftarrow \text{TRUE};$
223:   proceed\_in\_edge $\leftarrow e;$
224:   if $CS = \text{sleeping}$ then
225:     execute procedure wakeup(d_k)
226:   else if $CS = \text{inactive}$ then
227:     if $in_{branch} \neq \phi$ then
228:       send \{proceed(d_k)\} on in\_branch
229:     else
230:       execute procedure proc_initiate() \hspace{1em} $\triangleright$ Receiving node is the leader node
231:     end if
232: end if
233: else if $SE(e) = \text{branch}$ and $in_{branch} \neq e$ then
234:   if $in\_branch \neq \phi$ then
235:     send \{proceed(d_k)\} on in\_branch
236:   else
237:     execute procedure proc_initiate() \hspace{1em} $\triangleright$ Receiving node is the leader node
238:   end if
239: end if

240: procedure wakeup(d_k)
241:   $CS \leftarrow \text{active};$
242:   $d_v \leftarrow d_k;$
243:   $WC \leftarrow d_k;$
244:   if $d_k > d_h$ then
245:     $d_h \leftarrow d_k;$
246: end if
execute procedure proc_initiate()
end procedure

upon receiving ⟨connect(NID, WNC, d_u, d_k)⟩ on edge e \(\triangleright \) WNC (Weight of Neighbor Component)

if CS = sleeping then
    CS ← active;
    \(d_h \leftarrow d_k\);
    \(d_v \leftarrow d_k\);
    WC \(\leftarrow d_k\);
    \(\epsilon_1 \leftarrow \frac{w_k - d_h - d_v}{2}\);
    \(\epsilon_2 \leftarrow p_v - WC\);
    if \(\epsilon_1 < \epsilon_2\) then
        if v > NID then
            leader_flag \(\leftarrow TRUE\);
        else
            leader_flag \(\leftarrow FALSE\);
        end if
        \(d_h \leftarrow d_h + \epsilon_1\);
        \(d_v \leftarrow d_v + \epsilon_1\);
        WC \(\leftarrow WC + WNC + 2 \times \epsilon_1\);
        SE(e) \(\leftarrow branch\);
        send ⟨accept(leader_flag, root_flag, WC, d_h)⟩ on e
        if leader_flag \(\leftarrow TRUE\) then
            execute procedure proc_initiate()
        end if
        \(\triangleright \epsilon_1 \geq \epsilon_2\)
    else
        CS \(\leftarrow inactive\);
        WC \(\leftarrow WC + \epsilon_2\);
        \(d_v \leftarrow d_v + \epsilon_2\);
        \(d_h \leftarrow d_k + \epsilon_2\);
        labelled_flag \(\leftarrow TRUE\);
        send ⟨refind_\epsilon⟩ on e
    end if
else if CS = inactive then
    if root_flag = TRUE then
        leader_flag \(\leftarrow TRUE\);
    else
        CS \(\leftarrow active\);
if \( v > NID \) then

\[
\text{leader\_flag} \leftarrow \text{TRUE};
\]

else

\[
\text{leader\_flag} \leftarrow \text{FALSE};
\]

end if

\[
\epsilon_1 = w_e - d_v - d_u;
\]

\[
WC \leftarrow WC + WNC + \epsilon_1;
\]

\[
d_t \leftarrow dk + \epsilon_1;
\]

\( d_h < d_t \) then

\[
d_h \leftarrow d_t;
\]

end if

\[
dt \text{ is a temporary variable}
\]

\[
deactivate\_flag = \text{FALSE};
\]

\( deactivate\_flag \) is a temporary variable

\[
\text{send} \langle \text{update\_info}(0, \text{root\_flag}, \text{deactivate\_flag}, WC, d_h) \rangle \text{ on all } e' \in \text{Adj}(v) : e' \neq e \\
\]

and \( SE(e') = \text{branch} \)

\[
\text{if} \ ETP(e) = \text{TRUE} \text{ then}
\]

\[
ETP(e) = \text{FALSE};
\]

\[
\text{end if}
\]

\[
\text{SE}(e) \leftarrow \text{branch};
\]

\[
\text{send} \langle \text{accept}(\text{leader\_flag}, \text{root\_flag}, WC, d_h) \rangle \text{ on } e;
\]

\[
\text{if} \ (\text{leader\_flag} = \text{TRUE}) \text{ then}
\]

\[
\text{execute procedure proc\_initiate()}
\]

\[
\text{end if}
\]

end if

Upon receiving \( \langle \text{refind\_epsilon} \rangle \) on edge \( e \)

\[
\text{if} \ SE(e) = \text{basic} \text{ then}
\]

\[
RFE(e) \leftarrow \text{TRUE};
\]

\[
\text{else}
\]

\[
\text{if} \ in\_branch \neq \phi \text{ then}
\]

\[
\text{send} \langle \text{refind\_epsilon} \rangle \text{ on } in\_branch
\]

\[
\text{else}
\]

\[
\text{execute procedure proc\_initiate()}
\]

\[
\text{end if}
\]

\[
\text{end if}
\]

Upon receiving \( \text{accept}(LF, RF, TW\_C, d_k) \) on edge \( e \)

\( TW\_C \) (Total Weight of the Component)

\[
SE(e) \leftarrow \text{branch};
\]
if RF = TRUE then
    CS ← inactive;
    prize_flag = FALSE;
else
    CS ← active;
end if
root_flag ← RF;
d_h ← d_k;
d_v ← d_v + best_epsilon;
W_C ← TWC;
if SE(proceed_in_edge) = branch and proceed_flag = TRUE then
    proceed_flag ← FALSE;
    proceed_in_edge ← φ;
end if
deactivate_flag = FALSE; ▶ deactivate_flag is a temporary variable
send ⟨update_info(best_epsilon, root_flag, deactivate_flag, TWC, d_h)⟩ on all e' ∈ Adj(v) : e' ≠ e and SE(e') = branch
if LF = FALSE then
    execute procedure proc_initiate()
end if

Upon receiving ⟨update_info(epsilon_value, RF, DF, TWC, d_h)⟩ on edge e
if RF = TRUE and DF = FALSE then
    CS ← inactive;
    prize_flag ← FALSE;
else if RF = FALSE and DF = TRUE then
    CS ← inactive;
else if RF = FALSE and DF = FALSE then
    CS ← active;
end if
root_flag ← RF;
d_h ← d_k;
d_v ← d_v + epsilon_value;
W_C ← TWC;
send ⟨update_info(epsilon_value, RF, DF, TWC, d_h)⟩ on all e' ∈ Adj(v) : e' ≠ e and SE(e') = branch
if v = r then ▶ r is the root node
    execute procedure proc_initiate()
end if
Upon receiving \(\langle\text{prune}\rangle\) on edge \(e\)

if \(\text{root\_flag} = \text{TRUE}\) and \(\text{SE}(e) = \text{branch}\) then \(\triangleright \) pruning inside the root component

\[
\begin{align*}
\text{if} & \ (\text{labelled\_flag} = \text{TRUE}) \ \text{and} \ ((\forall e' \in \text{Adj}(v) : e' \neq e \ \text{and} \ \text{SE}(e') = \text{basic}) \text{ or} \\
& \ (\text{prune\_msg\_count} = 0) \ \text{then} \\
& \ \text{prize\_flag} \leftarrow \text{TRUE}; \\
& \ \text{root\_flag} \leftarrow \text{FALSE}; \\
& \ \text{send} \ \langle\text{backward\_prune}\rangle \ \text{on \ in\_branch} \\
& \ \text{SE}(\text{in\_branch}) \leftarrow \text{basic}; \\
& \ \text{else} \\
& \ \text{for all} \ e' \in \text{Adj}(v) : e' \neq e \ \text{and} \ \text{SE}(e') = \text{branch} \ \text{do} \\
& \ \ \ \text{send} \ \langle\text{prune}\rangle \ \text{on} \ e' \\
& \ \ \ \text{prune\_msg\_count} \leftarrow \text{prune\_msg\_count} + 1; \\
& \ \text{end for} \\
& \ \text{end if} \\
& \ \text{for all} \ e' \in \text{Adj}(v) : e' \neq e \ \text{and} \ \text{ETP}(e') = \text{TRUE} \ \text{do} \\
& \ \ \ \text{send} \ \langle\text{prune}\rangle \ \text{on} \ e' \\
& \ \text{end for} \\
& \ \text{else if} \ \text{root\_flag} = \text{FALSE} \ \text{then} \ \triangleright \text{pruning inside non-root inactive component} \\
& \ \text{for all} \ e' \in \text{Adj}(v) : e' \neq e \ \text{do} \\
& \ \ \ \text{if} \ \text{ETP}(e') = \text{TRUE} \ \text{or} \ \text{SE}(e') = \text{branch} \ \text{then} \\
& \ \ \ \ \text{send} \ \langle\text{prune}\rangle \ \text{on} \ e' \\
& \ \ \ \ \text{end if} \\
& \ \ \ \ \text{if} \ \text{SE}(e') = \text{branch} \ \text{then} \\
& \ \ \ \ \text{SE}(e') \leftarrow \text{basic}; \\
& \ \ \ \ \text{end if} \\
& \ \text{end for} \\
& \ \text{end if}
\end{align*}
\]

Upon receiving \(\langle\text{backward\_prune}\rangle\) on edge \(e\)

\[
\begin{align*}
& \ \text{prune\_msg\_count} \leftarrow \text{prune\_msg\_count} - 1; \\
& \ \text{SE}(e) \leftarrow \text{basic}; \\
& \ \text{if} \ \text{labelled\_flag} = \text{TRUE} \ \text{and} \ \text{prune\_msg\_count} = 0 \ \text{then} \\
& \ \text{if} \ \text{in\_branch} \neq \phi \ \text{then} \\
& \ \ \ \text{prize\_flag} \leftarrow \text{TRUE}; \\
& \ \ \ \text{root\_flag} \leftarrow \text{FALSE}; \\
& \ \ \ \text{send} \ \langle\text{backward\_prune}\rangle \ \text{on \ in\_branch} \\
& \ \ \ \text{SE}(\text{in\_branch}) \leftarrow \text{basic}; \\
& \ \ \ \text{end if} \\
& \ \ \ \text{end if}
\end{align*}
\]