Replica Placement on Directed Acyclic Graphs

Sonika Arora[1], Venkatesan T. Chakaravarthy[2], Kanika Gupta[1], Neelima Gupta[1], and Yogish Sabharwal[2]

1 **Department of Computer Science, University of Delhi, India**
sonika.ta@gmail.com,kanika.g.mcs.du.2012@gmail.com,ngupta@cs.du.ac.in

2 **IBM India Research Lab, New Delhi, India**
{vechakra,ysabharwal}@in.ibm.com

Abstract

The replica placement problem has been well studied on trees. In this paper, we study this problem on directed acyclic graphs. The replica placement problem on general DAGs generalizes the set cover problem. We present a constant factor approximation algorithm for the special case of DAGs having bounded degree and bounded tree-width (BDBT-DAGs). We also present a constant factor approximation algorithm for DAGs composed of local BDBT-DAGs connected in a tree like manner (TBDBT-DAGs). The latter class of DAGs generalizes trees as well; we improve upon the previously best known approximation ratio for the problem on trees. Our algorithms are based on the LP rounding technique; the core component of our algorithm exploits the structural properties of tree-decompositions to massage the LP solution into an integral solution.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Approximation Algorithms, LP Rounding

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2014.213

1 Introduction

The replica placement problem is an important problem that finds applications in a variety of domains such as internet and video on demand service delivery (see [8, 10, 5]). We refer to [13] for additional applications. This problem is concerned with the optimal placement of copies (replicas) of a database on the nodes of a network in order to serve periodic requests from a set of clients under the setting wherein each replica can serve a limited number of requests and a client can only be served by a replica within a specified distance (QOS requirement). Prior work has studied the problem for the case of tree networks [5, 13, 3, 9, 2, 1]. The goal of this paper is to address the problem on more general DAG networks.

Replica Placement Problem. The input consists of a DAG $G = (V, E)$. Each leaf node (having no in-edges) represents a *client*. Let A be the set of all the clients and let $|A| = m$. The input specifies a *request* $r(a)$ for each client $a \in A$. The input also includes a *capacity* W. For each edge (u, v) in E, the input specifies a distance $d(u, v)$. For a node u and a client a such that there is a path from a to u in the graph, let $d(a, u)$ be the shortest distance from a to u. Each client a is associated with a quantity $d_{\max}(a)$, the maximum distance it can travel.

A feasible solution selects a subset of nodes and places replicas on them in order to service the requests of the clients. The solution must assign the request of each client a to some (unique) replica u such that there is a path from a to u in the graph and $d(a, u) \le d_{\max}(a)$; we call this latter condition the *distance constraint*. Furthermore, the total requests assigned

© Sonika Arora, Venkatesan T. Chakaravarty, Kanika Gupta, Neelima Gupta, and Yogish Sabharwal; licensed under Creative Commons License CC-BY

to any replica must not exceed the capacity W. A client can be serviced by opening a replica at the client node itself. Our goal is to minimize the number of replicas placed.

We assume that the capacity W and the requests $r(\cdot)$ are integral and that W is polynomially bounded in the number of nodes. Furthermore, without loss of generality, we assume that $r(a) \leq W$ for all clients $a \in A$.

Prior Work. The above problem and its variants have been well-studied for tree networks in the existing literature [5, 13, 3, 9, 2, 1], from both practical and algorithmic perspectives.

Benoit et al. [3] studied the the replica placement problem on trees. They showed that the problem is NP-hard to approximate within a factor of 3/2 even without distance constraints (i. e., $d_{\max}(a) = \infty$, for all clients a) and when the network is a binary tree. They obtained the above result by showing that the above problem generalizes the bin-packing problem. Benoit et al. [2] presented a 2-approximation for the replica placement problem without distances. For the case with distances, they designed a greedy algorithm with an approximation ratio of $(1 + \Delta)$, where Δ is the maximum number of children of any node.

Arora et al.[1] presented an algorithm for tree networks with a constant approximation ratio (independent of Δ). Their result also applies to the partial cover version, wherein the input additionally specifies a number K and only K clients need to be serviced by a solution.

The replica placement problem can easily be seen to be a special case of the capacitated set cover problem. The latter problem admits an algorithm with an approximation ratio of $O(\log n)$ [12]; up to constant factors, the ratio is the best possible, unless NP = P [6]. For the case of vertex cover, Chuzhoy and Naor [4] and Gandhi et al. [7] presented algorithms for the capacitated vertex cover problem with approximation ratio of 3 and 2, respectively. However, their algorithms can handle only the case of simple graphs. Saha and Khuller [11] presented a 34-approximation algorithm for the more general case of multi-graphs.

The capacitated vertex cover problem plays an important role in the constant factor approximation algorithm for the replica placement problem on tree networks, due to Arora et al. [1], mentioned earlier. The above algorithm is based on the LP rounding technique. It works by reducing the issue of rounding an LP solution for the replica placement problem to an issue of rounding LP solutions of a suitably construed capacitated vertex cover instance. The algorithms presented in the current paper also make use of the above strategy.

Our Results. Prior work has primarily dealt with the replica placement problem on tree networks. In this paper, we study the problem on directed acyclic graphs. On DAGs, the replica placement problem is as hard as the capacitated set cover problem. We therefore focus on special classes of DAGs and provide constant factor approximation algorithms.

The first class of DAGs that we address are rooted DAGs that have bounded degree and bounded tree-width; we call these BDBT-DAGs. Tree-width is a notion traditionally associated with undirected graphs. A graph with bounded tree-width can be decomposed into disconnected pieces by removing a small number of nodes. By tree-width of a DAG, we shall mean the tree-width of the graph obtained by ignoring the direction on the edges.

Our first result is a constant factor approximation algorithm for BDBT-DAGs.

- There exists a polynomial time algorithm for the replica placement problem on BDBT-DAGs having an approximation ratio of $2 \cdot (d + t + 2)$, where d and t are respectively the degree bound and tree-width bound. of the input BDBT-DAG.

Our second result deals with a generalization of BDBT-DAGs, which we call TBDBT-DAGs (tree of BDBT-DAGs). Intuitively, a TBDBT-DAG is composed of BDBT-DAGs connected in a tree-like manner. A TBDBT-DAG is constructed by starting with a skeletal

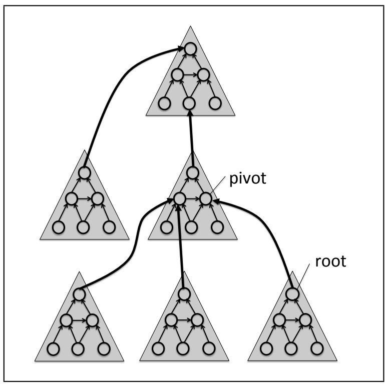

Figure 1 Example of TBDBT-DAG. The figure also shows the root of one of the component BDBT-DAGs and the pivot that it connects to in another BDBT-DAG.

tree $\mathcal{T}$, whose vertices are referred to as proxies. Then, each proxy is substituted with a BDBT-DAG. For each proxy q in $\mathcal{T}$, the root vertex of the associated BDBT-DAG is connected to some vertex (called pivot) in the BDBT-DAG associated with the parent proxy of q (see Figure 1).

The overall DAG has bounded tree-width, but may not have bounded degree. Our main result is a constant factor approximation algorithm for TBDBT-DAGs.

- There exists a polynomial time algorithm for the replica placement problem on TBDBT-DAGs having an approximation ratio of $O(d + t)$, where d and t are respectively the maximum degree bound and maximum tree-width bound of any component BDBT-DAG of the input TBDBT-DAG.

The class of TBDBT-DAGs clearly generalizes trees (wherein each component BDBT-DAG consists of a single vertex). Therefore the above result generalizes the constant factor approximation algorithm for the case of trees given in the prior work [1]. In fact, our analysis is more refined and leads to an improved constant factor.

Discussion. Tree networks, considered in prior work, have a simpler structure wherein each node has only one out-neighbor (i. e., parent). If we ignore the distance constraints, replica placement on such networks reduces to a capacitated set cover scenario over a set system consisting of a laminar family of sets. The classical set cover problem on such laminar familes can easily be handled. Thus, the distance and the capacity constraints are the only critical issue in the case of tree networks. On the the other hand, networks considered by us are DAGs, wherein a node can have multiple (but, bounded) number of out-neighbors leading to more complex set systems. In fact, any arbitrary set system can be encoded as an instance of the replica placement problem if we allow general DAGs, or even bounded degree DAGs. One of the important technical contributions of this paper is to show that the issue can be addressed, if the DAGs additionally have bounded tree-width (namely, BDBT-DAGs). However, the family of BDBT-DAGs do not encompass trees. Our main result deals with the larger class of TBDBT-DAGs, which generalizes both BDBT-DAGs and trees.

2 Preliminaries

In this section we describe a natural LP for our problem and setup terminology that we will use. We then formally define the specific class of DAGs that we address in this paper.

LP Formulation. We consider a natural LP formulation. We say that a solution *opens* node u, if it places a replica on it. We say that a client a is *attachable* to a node u, if there exists a path from a to u and $d(a,u) \leq d_{\max}(a)$. For a client a, let $\mathtt{Att}(a)$ denote the set of all nodes to which client a can be attached. For a node u, let $\mathtt{Att}(u)$ denote the set of all clients that can be attached to node u. For a set of nodes U, let $\mathtt{Att}(U) = \cup_{u\in U}\mathtt{Att}(u)$.

For each node $u \in V$, we introduce a variable $y(u)$ that specifies the extent to which u is open. For each client $a \in A$ and each node $u \in \mathtt{Att}(a)$, we introduce a variable $x(a,u)$ that specifies the extent to which a is assigned to u. A node u is said to *service* a client a, if $x(a,u) > 0$; we also say that a is *assigned* to u if u services a.

$$\min \quad \sum_{u\in V} y(u)$$

$$\sum_{a\in\mathtt{Att}(u)} x(a,u)\cdot r(a) \quad \leq \quad y(u)\cdot W \qquad (\forall u \in V) \tag{1}$$

$$x(a,u) \quad \leq \quad y(u) \qquad (\forall a \in A, u \in \mathtt{Att}(a)) \tag{2}$$

$$\sum_{u\in\mathtt{Att}(a)} x(a,u) \quad = \quad 1 \qquad (\forall a \in A) \tag{3}$$

$$y(u) \quad \leq \quad 1 \qquad (\forall u \in V) \tag{4}$$

Further, we add non-negativity constraints for all the variables. Constraint (1) (called the *capacity constraint*) enforces that at any node the total request assigned does not exceed the capacity W. Constraint (2) ensures that a client can be assigned to a node only to an extent to which the node is open; without this constraint, it can be shown that the LP has an unbounded integrality gap. Constraint (3) enforces that every client is serviced to an extent of exactly one, i. e. every client is fully served. Constraint (4) requires that a node can be opened to an extent of at most one.

For an LP solution σ, we shall denote the variables of the solution by x_σ and y_σ unless stated otherwise. We shall also use x and y to denote the variables when the solution is clear from the context of the discussion. Consider an LP solution σ. The cost of an LP solution is given by the objective function: $\mathtt{cost}(\sigma) = \sum_{u\in V} y_\sigma(u)$. For a set $X \subseteq V$, the cost of the set X in the solution σ is given by $\mathtt{cost}_\sigma(X) = \sum_{u\in X} y_\sigma(u)$.

▶ **Definition 1** (Fully-open, fully-closed and partially-open nodes)**.** We call a node u *fully-open*, if $y_\sigma(u) = 1$; *fully-closed*, if $y_\sigma(u) = 0$; and *partially open*, if $0 < y_\sigma(u) < 1$. ◀

▶ **Definition 2** (Load and Fully Loaded nodes)**.** For a set of clients $B \subseteq A$ and set of nodes $U \subseteq V$, by $load_\sigma(B,U)$ we mean the total assignments of B to nodes in U, i. e., $load_\sigma(B,U) = \sum_{a\in B}\sum_{v\in U} x_\sigma(a,v)\cdot r(a)$.

For a node $u \in V$, By $load_\sigma(u)$ we mean the total assignments to node u, i. e., $load_\sigma(u) = \sum_{a\in\mathtt{Att}(u)} x_\sigma(a,u)\cdot r(a)$. A node u is said to *fully loaded* if $load_\sigma(u) = W$. ◀

▶ **Definition 3** (Integrally Open and Integral Solutions)**.** A solution σ is said to be *integrally open*, if every node is either fully-open or fully-closed and an integrally open solution is said to be an *integral solution* if every client is serviced by exactly one node. ◀

DAGs of interest. In this paper we shall address two types of DAGs; these DAGs have bounded tree-width. We recollect the concept of tree-width and then formally define the DAGs that we address in this paper.

▶ **Definition 4** (Tree-decomposition and tree-width)**.** A *tree-decomposition* of graph $G = (V,E)$ is a pair $\langle\{V_i | i \in I\}, T\rangle$ where $V_1,\ldots,V_h$ are subsets of V called pieces, $I = [1,h]$, and T is a tree with elements of I as nodes. A tree-decomposition must satisfy the following properties:

1. *Node coverage*: every node of G belongs to at least one piece V_i, i. e., $\cup_{i \in I} V_i = V$
2. *Edge coverage*: for every edge $e = (u, v) \in E$, there is some piece V_i containing both ends of e, i. e., $\exists i \in I$ such that $\{u, v\} \subseteq V_i$
3. *Coherence*: for every graph vertex v , all pieces containing v form a connected component, i. e.,$\forall i, j, k \in I$, if j lies on the path between i and k in T, then $V_i \cap V_k \subseteq V_j$

The width of $\langle \{V_i | i \in I\}, T\rangle$ equals $max\{|V_i| : i \in I\} - 1$. The *tree-width* of G is the minimum k such that G has a tree-decomposition of width k. ◀

We will use the following property of tree-decompositions for designing our algorithm.

▶ **Property 1** (Separation Property). *Let p be any piece of T. Suppose that $T - p$ has components $T_1, T_2 ..... T_d$. Then the subgraphs $G[T_1 - V_p], G[T_2 - V_p], \ldots, G[T_d - V_p]$ have no nodes in common, and there are no edges between them.*

The first type of DAGs that we address have bounded degree and tree-width.

▶ **Definition 5** (Bounded Degree Bounded Tree-width DAG *(BDBT-DAG)*)**.** We say that a rooted DAG is a *bounded degree bounded tree-width DAG* if the undirected graph obtained by ignoring directions on the edges has bounded degree and bounded tree-width.

We denote the root node of a BDBT-DAG, G, by $\texttt{Rt(G)}$. ◀

The second type of DAGs that we consider generalize BDBT-DAGs as well as trees.

▶ **Definition 6** (Tree of BDBT-DAGs *(TBDBT-DAG)*)**.** A TBDBT-DAG G is a pair $\langle \{D_j | j \in J\}, \mathcal{T}\rangle$ where $D_1, D_2, \ldots, D_h$ are BDBT-DAGs, $J = [1, h]$ and $\mathcal{T}$ is a tree with the elements of J as nodes and labeled edges satisfying the following properties:

- The vertices of the BDBT-DAGs are disjoint, i. e., $V(D_i) \cap V(D_j) = \phi \ \forall \ i, j \in J, \ i \neq j$.
- The vertex set of G is the union of the vertices of BDBT-DAGs, i. e., $V(G) = \cup_{j \in J} V(D_j)$.
- The edges of all the BDBT-DAGs are contained in G, i. e., $E(D_j) \subseteq E(G)$ for all $j \in J$.
- For every edge $e = (D_i, D_j)$ in $\mathcal{T}$, there is an edge $(\texttt{Rt(D}_\texttt{i}\texttt{)}, \ell(\texttt{e}))$ in G that links the two BDBT-DAGs by connecting the root of D_i to a node of $V(D_j)$ determined by the label $\ell(e)$ on the edge; $\ell(e) \in V(D_j)$ is said to be a *pivot node*.

Let $\texttt{Roots(G)}$ be the roots of all BDBT-DAGs of G, i. e., $\texttt{Roots(G)} = \{\texttt{Rt(D}_\texttt{j}\texttt{)} : \texttt{j} \in \texttt{J}\}$. ◀

We make the following observation regarding the tree-width of TBDBT-DAGs.

▶ **Observation 1.** *If the maximum tree-width of any component BDBT-DAG is t, then the TBDBT-DAG has tree-width* $\max\{t, 1\}$.

3 Algorithm for BDBT-DAGs via Stable Solutions

For the ease of exposition, we first consider the simpler case of BDBT-DAGs and present a constant factor approximation algorithm. The components developed as part of the algorithm will also be useful in handling the more generic TBDBT-DAGs.

The algorithm is based on the LP rounding technique. Let σ_{in} be an optimal solution to the LP formulation. The algorithm works by applying a sequence of transformations until an integral solution is obtained, wherein each transformation increases the cost by at most a constant factor. The notion of *stable solutions* plays a key role in the above process.

▶ **Definition 7** (Stable solution)**.** A solution σ is said to be *stable* if the nodes can be partitioned into two sets R and P called *rich* and *poor* nodes respectively such that the following properties are satisfied:

1. The rich nodes, R, are fully open.
2. For any poor node $u \in P$, the total extent to which the poor nodes service the clients attachable to u is less than W, i. e., $load_\sigma(\texttt{Att}(u), P) < W$.
3. Every client is either serviced by only nodes in R or only nodes in P but not both. ◀

Intuitively, a stable solution segregates the input instance into two parts, the first part comprising of the rich nodes and the clients serviced by them, and the second part comprising of the poor nodes and the clients serviced by them. It is easy to handle the first part, since all the nodes in the instance are fully open. The second part has the useful feature that it is uncapacitated in essence; meaning, no matter how we assign the clients, the capacity W at a node can never be exceeded and hence, the capacity constraints can safely be ignored.

We next present two procedures. The first procedure works on any bounded degree DAG and transforms an arbitrary LP solution σ_{in} into a stable solution σ_s with only a $(d+2)$ factor increase in cost, where d is the degree bound. The second procedure works on any bounded tree-width DAG and transforms any stable solution σ_s into an integrally open solution σ_{io} with only a $(t+1)$ factor increase in cost, where t is the tree-width bound. Combining the two procedures, we can handle any BDBT-DAG and transform an arbitrary LP solution σ_{in} into an integrally open solution σ_{io} with only a constant factor increase in cost. Finally, the integrally open solution can be transformed into an integral solution using a cycle cancellation based method proposed in prior work [1].

The above procedures make use of a subroutine called *pulling procedure*, described next. This subroutine is also employed by other algorithms in the paper to reassign the clients.

Pulling Procedure. Given a node u and a set of nodes X such that $u \notin X$, by performing the *pulling* procedure from X on to u, we mean reassigning the clients that are attachable to u from X on to u while it has remaining capacity.

Formally, let σ_{in} be the input solution. We process each client-node pair $\langle a, v\rangle$ iteratively where $a \in Att(u)$ and $v \in X$. Let π_{old} be the LP solution at the start of the current iteration. We construct a new solution π_{new} as follows. We set $x_{\pi_{new}}(a,u) = x_{\pi_{old}}(a,u) + \delta$ and $x_{\pi_{new}}(a,v) = x_{\pi_{old}}(a,v) - \delta$ where

$$\delta = \min\left\{x_{\pi_{old}}(a,v), \frac{W - load_{\pi_{old}}(u)}{r(a)}\right\}.$$

All other values of $x_{\pi_{new}}(.,.)$ and $y_{\pi_{new}}(.)$ are retained as in π_{old}. π_{new} is taken as the input solution π_{old} for the next iteration. At the end of processing all client-node pairs, the solution π_{new} of the last iteration is taken as the final solution σ_{out} output by this procedure.

Note that if $load_{\sigma_{in}}(u) + load_{\sigma_{in}}(\texttt{Att}(u), X) \le W$ (before the pulling procedure), then $load_{\sigma_{out}}(\texttt{Att}(u), X) = 0$, otherwise $load_{\sigma_{out}}(u) = W$ (after the pulling procedure).

3.1 Constructing Stable Solutions for Bounded Degree DAGs

In this section, we show how to transform any feasible solution σ_{in} for a bounded degree DAG into a stable solution σ_s as stated in the following Lemma.

▶ **Lemma 8.** *Any LP solution σ_{in} for a bounded degree DAG can be converted into a stable solution σ_s such that $\texttt{cost}_{\sigma_s}(R) \le (d+1)\cdot \texttt{cost}(\sigma_{in})$ and $\texttt{cost}_{\sigma_s}(P) \le \texttt{cost}(\sigma_{in})$ where R and P are respectively the rich and poor nodes of the stable solution σ_s.*

Proof Sketch. The transformation is divided into the *reddening phase* and *browning phase.*

Reddening Phase. In this phase, the algorithm tries to make as many nodes fully loaded as possible. We shall color all fully loaded nodes red. For this, the algorithm processes the nodes iteratively in an arbitrary order. For every node, it checks if the node can become fully loaded by performing the pulling procedure from non-red nodes. If so, we actually perform the pulling procedure, open this node and color it red. The reddening phase completes when all the nodes are processed. Hereafter, no node becomes fully loaded (red).

Browning Phase. In this phase, we process all the neighbors of red nodes that are not already red. For each such node, u, we open the node, perform the *pulling procedure* on u from non-red nodes and color it brown. Note that the total load on u, even after the reassignments, will be less than W (otherwise u would have become red in the first step). This completes the processing of the browning phase (and stage 1).

It can be shown that the solution at this stage is a stable solution taking the set of red and brown nodes to be rich and the remaining nodes to be poor. The first two properties are easy to check. The third property follows from the fact that if a client is attachable to a rich node and is assigned to a poor node, then it must be attachable to a brown node; but then the brown node would have pulled the assignments of this client from the nodes in P.

We now analyse the cost. Note that the cost of any feasible LP solution must be at least $\lfloor \sum_{a \in A} r(a)/W \rfloor$. Moreover, each of the red nodes is fully loaded. Therefore the number of red nodes is no more than $\texttt{cost}(\sigma_{in})$. There are at most d neighbours of a red node; thus the total number of brown nodes is at most $d \cdot \texttt{cost}(\sigma_{in})$. This implies that $\texttt{cost}_{\sigma_s}(R) \leq (d+1) \cdot \texttt{cost}(\sigma_{in})$. As the extent of openness of the remaining nodes is untouched, it follows that $\texttt{cost}_{\sigma_s}(P) \leq \texttt{cost}(\sigma_{in})$. ◀

3.2 Bounded Tree-width DAGs: Stable to Integrally Open Solutions

In this section, we show how to transform any stable solution σ_s for a bounded tree-width DAG into an integrally open solution σ_{io} as captured in the following Lemma.

▶ **Lemma 9.** *Let R and P respectively be the rich and poor nodes of a stable solution σ_s of a DAG, G, of constant tree-width t. Then σ_s can be converted to an integrally open solution σ_{io} such that the nodes of R remain untouched, i. e., $y_{\sigma_{io}}(u) = y_{\sigma_s}(u)$ for all $u \in R$ and $x_{\sigma_{io}}(a,u) = x_{\sigma_s}(a,u)$ for all clients $a \in A$ and nodes $u \in R$. Thus, $\texttt{cost}_{\sigma_{io}}(R) = \texttt{cost}_{\sigma_s}(R)$. Moreover, if Z_1 is the set of of fully-opened nodes of P in σ_{io}, then $\texttt{cost}_{\sigma_{io}}(Z_1) \leq (t+1) \cdot \texttt{cost}_{\sigma_s}(P)$. The remaining nodes of P are fully closed.*

Proof Sketch. Our procedure shall color poor nodes yellow and white during its course of execution; the yellow nodes will be opened and the white nodes will be closed. At the end of the procedure all the poor nodes will be colored yellow or white thereby obtaining an integrally open solution. We shall call a node *resolved* if it is either rich or colored (yellow or white) and *unresolved* otherwise. Recall that rich nodes are already open. We shall maintain two sets, `Res` and `Unres` of the *resolved* and *unresolved* nodes respectively. Every node of the graph will either be in `Res` or in `Unres`. Initially we set `Res` $= R$ and `Unres` $= P$. These sets will be modified as we color the nodes and resolve them. We shall also say that a client is *resolved* if it is only assigned to resolved nodes; we shall call it unresolved otherwise.

Consider a tree decomposition, $\langle \{V_i | i \in I\}, T \rangle$, of the DAG G having tree-width t. Fix some piece p_r containing $Rt(G)$ to be the root of the tree decomposition and assume all edges to be directed towards p_r. We say that a client a is *critical* at piece $p \in I$ if p is the highest piece along the path to p_r such that a can be assigned to some node of V_p. We call a piece *critical* if it is critical for some client.

We process the pieces of the tree decomposition in any topological order (bottom-up); let p be the piece being processed in the current iteration. We check if there is any client, a, that is critical at p. If not, we do nothing. Otherwise, consider the partitioning of the set `Unres` based on whether a node appears in p or not; let $Y = \texttt{Unres} \cap V_p$ and $\overline{Y} = \texttt{Unres} \setminus V_p$. Further, let X be the nodes of $\overline{Y}$ that appear in some piece q below p in the tree decomposition, i. e., $X = \{u \in \overline{Y} : u \in V_q \text{ and } \exists \text{ a path from } q \text{ to } p \text{ in the tree decomposition}\}$. We can show that (i) the extent to which the nodes in X and Y are open collectively is at least 1, i. e., $\sum_{v \in X \cup Y} y_s(v) \geq 1$; and (ii) all the clients assigned to nodes in X are attachable to some node of Y. We open up the nodes of Y and color them yellow. We account for the cost of fully opening these nodes of Y by charging them to the extent to which the nodes of $X \cup Y$ are open in the solution σ_s. We perform the pulling procedure on all the nodes of Y by pulling all the attachable clients from all nodes in $\overline{Y}$. Note that the pulling on nodes of Y will ensure that no clients remain assigned to the nodes in X any more. We therefore close all these nodes in X and color them white. We incur a factor $(t + 1)$ loss in the process as the tree-width is t. Now, the nodes of Y are opened and the nodes of X are closed - we therefore move them from the set `Unres` to `Res`. Note that any poor node is charged at most once as it must be in `Unres` to be charged and it is moved to the set `Res` immediately after being charged. This completes the processing of the piece p. We now outline why (i) and (ii) must hold. For (i), since client a critical at p is unresolved, it can only be assigned to nodes in $X \cup Y$. Moreover $\sum_{v \in V} x_{\pi_{old}}(a, v) = 1$ and $x_{\pi_{old}}(a, v) \leq y_{\pi_{old}}(v)$ on any node v. Hence $\sum_{v \in X \cup Y} y_{\pi_{old}}(v) \geq 1$. For (ii), consider any client, b, serviced by some $u \in X$. Now b could not have become critical at any piece below p otherwise it would have been pulled to the node it became critical on. Thus b must be attachable to some node of Y in p.

We close any unresolved nodes left at the end as no clients can be assigned to them (every client has to be critical on some node). Hence we obtain an integrally open solution. ◀

3.3 BDBT-DAGs: Constant Factor Approximation Algorithm

In this section, we consider BDBT-DAGs and present a constant factor approximation algorithm. Combining Lemma 8 and 9, we can convert the optimal LP solution σ_{in} into an integrally open solution σ_{io}. The only issue with σ_{io} is that the request $r(a)$ of a client a may be split and assigned to multiple nodes. On the other hand, our problem definition requires that the request must be wholly assigned to a single node. We can address the issue using a cycle cancellation procedure described in prior work[1].

▶ **Lemma 10.** *Any integrally open solution σ_{io} can be converted into an integral solution σ_{out} such that $\texttt{cost}(\sigma_{out}) \leq 2 \cdot \texttt{cost}(\sigma_{io})$.*

Using the cost analysis as stated in the lemmas, we see that $\texttt{cost}(\sigma_{out}) \leq 2 \cdot \texttt{cost}(\sigma_{io}) \leq 2 \cdot (\texttt{cost}_{\sigma_s}(R) + (t+1) \cdot \texttt{cost}_{\sigma_s}(P)) \leq 2 \cdot ((d+1) \cdot \texttt{cost}(\sigma_{in}) + (t+1) \cdot \texttt{cost}(\sigma_{in})) = 2 \cdot (d+t+2) \cdot \texttt{cost}(\sigma_{in})$.

We have thus established the following theorem.

▶ **Theorem 11.** *Any LP solution σ_{in} for a BDBT-DAG instance can be transformed into an integral solution σ_{out} such that $\texttt{cost}(\sigma_{out}) \leq 2 \cdot (d+t+2) \cdot \texttt{cost}(\sigma_{in})$, where d and t are respectively the degree bound and tree-width bound of the DAG.*

4 Replica Placement Problem on TBDBT-DAGs

The goal of this section is to design a constant factor approximation algorithm for the replica placement problem on TBDBT-DAGs. The algorithm builds on the procedure for handling

BDBT-DAGs presented earlier. Recall that the procedure for BDBT-DAGs works in two stages, wherein the first stage transforms an LP solution σ_{in} into a stable solution σ_s and the second stage transforms σ_s into an integrally open solution σ_{io}. Finally, the solution σ_{io} is converted into an integral solution using techniques from prior work. In the context of TBDBT-DAGs, it is difficult to obtain a stable solution, because these DAGs do not have bounded degree. Instead, our algorithm goes via a similar, but weaker notion of pseudo-stable solutions. The process of converting pseudo-stable solutions to integrally open solutions is also more involved and utilizes the concept of hierarchical solutions, which generalize integrally open solutions. It suffices to get hierarchical solutions, since prior work has shown how to transform such solutions to integral solutions. We next define pseudo-stable and hierarchical solutions, and outline our two transformations.

▶ **Definition 12** (Pseudo-stable solution)**.** An LP solution σ is said to be *pseudo-stable*, if the nodes can be partitioned into two sets R and P called *rich* and *poor* nodes respectively satisfying the following properties:

1. The rich nodes, R, are fully open.
2. For any poor node $u \in P$, the total extent to which the poor nodes service the clients attachable to u is less than W, i. e., $load_\sigma(\texttt{Att}(u), P) < W$.
3. Every client is either serviced by
   a. only nodes in R.
   b. only nodes in P.
   c. nodes in both R and P. Any client a in this category should be attachable to exactly one node in S, where S is the set of all poor roots having a rich pivot as their out-neighbor, i. e., $S = \{u \in \texttt{Roots(G)} \cap \texttt{P} : \text{out-neighbor of } u \text{ is a rich pivot}\}$. The lone node in S to which a is attachable is called the *special root* of a.

   The clients in the first two categories are said to be *settled*, whereas those in the third category are said to be *unsettled*. ◀

We transform any given LP solution to a pseudo-stable solution using a procedure similar to the one used for obtaining stable solutions in the context of bounded degree DAGs; the transformation is discussed in Section 4.1. The next (and more sophisticated) stage of the algorithm converts a pseudo-stable solution into a hierarchical solution, defined below.

▶ **Definition 13** (Hierarchical solutions)**.** An LP solution σ is said to be *hierarchical* if every client is assigned to at most one partially-open node. ◀

Hierarchical solutions generalize the notion of integrally open solutions. In the context of designing constant factor approximation algorithm for the case of trees, prior work presented a procedure for transforming an LP solution into a hierarchical solution. For the case of TBDBT-DAGs, we present a procedure for transforming pseudo-stable solutions into hierarchical solutions. Given a pseudo-stable solution σ_{ps}, our procedure works by segregating σ_{ps} into two parts σ_1 and σ_2 with the following properties:

- σ_1 is a stable solution for a subset of clients. We can transform this partial solution into an integrally open solution σ_1' using the procedure in Lemma 9, since TBDBT-DAGs have bounded tree-width.
- σ_2 is a feasible solution for the remaining set of clients and has certain nice properties that allow us to transform it into a hierarchical solution σ_2'. This transformation is based on the intuition that a TBDBT-DAG consists of a skeletal tree $\mathcal{T}$, where each node is in turn a BDBT-DAG. The skeletal tree structure allows us to use ideas from the prior work[1] in obtaining the hierarchical solution σ_2'.

We finally merge σ_1' and σ_2' into a single hierarchical solution σ_h servicing all the clients. The rest of the section is devoted to describing the different transformations.

4.1 Obtaining a Pseudo-stable Solution

In this section, we show how to transform any feasible solution σ_{in} for a TBDBT-DAG into a pseudo-stable solution σ_{ps}. The following Lemma formally captures the transformation.

▶ **Lemma 14.** *Any LP solution σ_{in} can be converted into a pseudo-stable solution σ_{ps} such that $\texttt{cost}_{\sigma_{ps}}(R) \leq (d+2) \cdot \texttt{cost}(\sigma_{in})$ and $\texttt{cost}_{\sigma_{ps}}(P) \leq \texttt{cost}(\sigma_{in})$ where R and P are respectively the rich and poor nodes of the stable solution σ_{ps}.*

Proof Sketch. This transformation is a minor modification from that used in Lemma 8 for BDBT-DAGs. The transformation is divided into the *reddening* and *browning* phases.

The reddening phase is same as before. We make as many nodes fully loaded (red) as possible, and open them. We perform the browning phase with slight modifications. For every red node we color all its non-red neighbors brown except for the in-neighbors that are in $\texttt{Roots(G)}$. Note that every node has bounded degree ignoring the in-neighbors in $\texttt{Roots(G)}$; a node may have an arbitrary number of in-neighbors from $\texttt{Roots(G)}$.

We now show that the solution is pseudo-stable. We take the set of all red and brown nodes as R and the remaining nodes as P. Consider any client, a, assigned to a node v_1 in R as well as a node v_2 in P; we need to show that a is attachable to exactly one node in S. We first observe that a cannot be attachable to any brown node since then the brown node must have pulled the assignments of a from nodes in P and a would not be assigned to v_2. This also implies that v_1 is red. We now show that a is attachable to at least one node in S. Consider the path from a to v_1. The non-red node closest to v_1 on this path, say v_3, must either be brown or in $\texttt{Roots(G)} \cap \texttt{P}$. But since a is not attachable to any brown node, $v_3 \in \texttt{Roots(G)} \cap \texttt{P}$. Moreover, the out-neighbor of v_3 is a red (and hence rich) pivot and therefore $v_3 \in S$. Next we show that a cannot be attachable to two nodes in S. Suppose a is attachable to two nodes, $u_1, u_2 \in S$. Then there must be a path between them; w.l.o.g. let there be a path from u_1 to u_2. The out-neighbor of u_1 is a rich pivot, say v. Then, either v is itself brown or there exists a brown node on the path from v to u_2 (as v is red, $u_2 \in P$ and we color all non-red out-neighbors of red nodes brown). This contradicts our inference that a cannot be attachable to a brown node. Hence the solution is pseudo-stable.

The cost analysis is similar to that in Lemma 8. The change in factor arises because earlier we could color at most d neighbors brown, but now we may color brown one more neighbour – the out-neighbor of a red root (which is a pivot not in the BDBT-DAG). ◀

4.2 Obtaining a Hierarchical Solution

In this section, we show how transform a pseudo-stable solution σ_{ps} into a hierarchical solution σ_h and prove the following lemma.

▶ **Lemma 15.** *Let R and P respectively be the rich and poor nodes of a pseudo-stable solution σ_{ps} of a TBDBT-DAG, G. Let t be the maximum tree-width of any of its component BDBT-DAGs. Then σ_{ps} can be converted to a hierarchical solution σ_h such that the nodes of R remain untouched, i. e., $y_{\sigma_h}(u) = y_{\sigma_{ps}}(u)$ for all $u \in R$ and $x_{\sigma_h}(a,u) = x_{\sigma_{ps}}(a,u)$ for all clients $a \in A$ and nodes $u \in R$. Thus, $\texttt{cost}_{\sigma_h}(R) = \texttt{cost}_{\sigma_{ps}}(R)$. Moreover, $\texttt{cost}_{\sigma_h}(P) \leq (\max\{t,1\}+2) \cdot \texttt{cost}_{\sigma_s}(P)$.*

Proof. Let A_1 and A_2 be the set of settled and unsettled clients with respect to σ_{ps}. We split the original problem instance into two instances focusing on the settled and unsettled clients, respectively. This is achieved by taking two copies of the original DAG, denoted I_1 and I_2; we set $r(a) = 0$ for all unsettled clients in I_1 and $r(a) = 0$ for all settled clients in I_2. From σ_{ps}, we can get two feasible LP solutions σ_1 and σ_2 for the instances I_1 and I_2, respectively. The solutions σ_1 and σ_2 are copies of σ_{ps}, except that we set $x_{\sigma_1}(a, u) = 0$ for all unsettled clients a and $x_{\sigma_2}(a, u) = 0$ for all settled clients a, for all nodes $u \in V$.

Note that σ_1 is a stable solution for I_1. Hence, the solution σ_1 can be transformed into an integrally open solution σ_1' for the instance I_1, using the procedure given in Lemma 9 (since a TBDBT-DAG has tree width $\max\{t, 1\}$, where t is the maximum tree-width of any of its component BDBT-DAGs). Let Z_1 be the nodes of P opened by this algorithm.

We now focus on the second instance I_2, consisting of only unsettled clients, and transform the solution σ_2 into a hierarchical solution σ_2' using the following Lemma.

▶ **Lemma 16.** *Let R and P respectively be the rich and poor nodes of a pseudo-stable solution σ_2 of a TBDBT-DAG with constant tree-width t having only unsettled clients. Then σ_2 can be converted to a hierarchical solution σ_2' such that the nodes of R remain untouched, i. e., $y_{\sigma_2'}(u) = y_{\sigma_2}(u)$ for all $u \in R$ and $x_{\sigma_2'}(a, u) = x_{\sigma_2}(a, u)$ for all clients a and nodes $u \in R$. Thus, $\texttt{cost}_{\sigma_2'}(R) = \texttt{cost}_{\sigma_2}(R)$. Moreover if Z_2 is the set of fully or partially open nodes of P in σ_2', then $\texttt{cost}_{\sigma_2'}(Z_2) \leq \texttt{cost}_{\sigma_2}(P)$.*

Proof Sketch. Let U be the set of unsettled clients. We process the nodes that are special roots for some client iteratively in topological order (bottom-up) of the DAG. Let u be the node currently being processed. Let B be the set of clients for which u is a special root and let X_u be the set of nodes of P to which the clients of B are assigned (this may include u itself). We shall argue that all the clients assigned to X_u have the same special root, i. e., u. We shall perform the pulling procedure on u from $X_u \setminus \{u\}$. Note that all the clients in B will be reassigned to u and no clients will remain assigned to $X_u \setminus \{u\}$. We shall close down all the nodes of X_u and open u to the extent $\min\{\texttt{cost}_{\sigma_2}(X_u), 1\}$; thus the solution remains feasible. We shall account for the cost of opening u by charging the extent to which the nodes of X_u are open in the solution σ_2. This completes the processing of u. We now outline why all the clients assigned to X_u have u as their special root. Consider any client b assigned to a node v in X_u. By definition of X_u, there must be a client, c, assigned to v and having special root as u. It can be argued that clients attachable to the same node will have the same special root (this follows from the fact that the roots of the BDBT-DAGs are arranged in a tree-like manner in a TBDBT-DAG using a skeletal tree). Since c and b are both attachable to v, they must have the same special root; therefore the special root of b must also be u. Hence, all the clients assigned to X_u have u as the special root.

Note that the clients in B are not assigned to any poor node above u. Therefore u and B will not participate in any further processing (of other nodes that are special roots). Thus any node is charged at most once only. The special roots are taken as the set Z_2. ◀

We shall now combine the solutions σ_1' and σ_2' into a solution σ_h for the original input TBDBT-DAG as follows. Let Z_1 be the set of fully open nodes in σ_1' and Z_2 be the nodes of P that are fully or partially open in σ_2'. The nodes of R remain untouched in both the solutions. We construct σ_h by opening all the nodes of R and Z_1. Then, we open all the nodes in $Z_2 \setminus Z_1$ to the extent that they were open in σ_2'. The assignments are retained from both the solutions σ_1' and σ_2' (the client sets are disjoint). Formally: (i) set $y_{\sigma_h}(u) = y_{\sigma_1'}(u)$ (which is 1) for all $u \in R \cup Z_1$; (ii) set $y_{\sigma_h}(u) = y_{\sigma_2'}(u)$ for all $u \in Z_2 \setminus Z_1$; (iii) set $x_{\sigma_h}(a, u) = x_{\sigma_1'}(a, u)$ for all settled clients a and nodes u; (iv) set $x_{\sigma_h}(a, u) = x_{\sigma_2'}(a, u)$ for all unsettled clients a

and nodes u. Note that no node in $Z_1 \cup Z_2$ can become fully loaded, because they belong to the set P. Moreover, the nodes of $Z_2 \setminus Z_1$ are also sufficiently open to service the clients assigned to them as they are not assigned any clients in the solution of the instance I_1 and the solution to the instance I_2 is feasible. The solution σ_h is hierarchical as the unsettled clients are assigned to at most one partially-open node since σ'_2 is hierarchical.

Using the cost analysis as stated in Lemmas 9 and 16, we see that $\texttt{cost}_{\sigma_h}(R) = \texttt{cost}_{\sigma_{ps}}(R)$ and $\texttt{cost}_{\sigma_h}(P) \leq \texttt{cost}_{\sigma'_1}(Z_1) + \texttt{cost}_{\sigma'_2}(Z_2) \leq (\max\{t,1\}+1) \cdot \texttt{cost}_{\sigma_1}(P) + \texttt{cost}_{\sigma_2}(P)$ $\leq (\max\{t,1\}+2) \cdot \texttt{cost}_{\sigma_{ps}}(P)$ This completes the proof of Lemma 15. ◀

4.3 TBDBT-DAGs: Constant Factor Approximation Algorithm

We now put together the different transformations and establish a constant factor approximation algorithm for TBDBT DAGs. Combining Lemma 14 and 15, we can convert the optimal LP solution σ_{in} into a hierarchical solution σ_h. A procedure for converting any hierarchical solution into an integral solution is implicit in prior work [1].

▶ **Lemma 17.** *Any hierarchical solution σ_h can be converted into an integral solution σ_{out} such that $\texttt{cost}(\sigma_{out}) \leq 136 \cdot \texttt{cost}(\sigma_h)$.*

The above procedure works by reducing the task to an issue of rounding LP solutions of a capacitated vertex cover instance, for which Saha and Khuller[11] present a 34-approximation. The proof is omitted.

Using the cost analysis as stated in the lemmas, we see that $\texttt{cost}(\sigma_{out}) \leq 136 \cdot \texttt{cost}(\sigma_h) = 136 \cdot (\texttt{cost}_{\sigma_h}(R) + \texttt{cost}_{\sigma_h}(P)) \leq 136 \cdot (\texttt{cost}_{\sigma_{ps}}(R) + (\max\{t,1\}+2) \cdot \texttt{cost}_{\sigma_{ps}}(P)) \leq 136 \cdot ((d+2) \cdot \texttt{cost}(\sigma_{in}) + (\max\{t,1\}+2) \cdot \texttt{cost}(\sigma_{in})) = 136 \cdot (d + \max\{t,1\} + 4) \cdot \texttt{cost}(\sigma_{in})$

We have thus established the following theorem.

▶ **Theorem 18.** *Any LP solution σ_{in} for a TBDBT-DAG instance can be transformed into an integral solution σ_{out} such that $\texttt{cost}(\sigma_{out}) \leq 136 \cdot (d + \max\{t,1\} + 4) \cdot \texttt{cost}(\sigma_{in})$, where d and t are respectively the maximum degree bound and the maximum tree-width bound of any component BDBT-DAG of the TBDBT-DAG.*

This yields a factor 680 approximation algorithm for the case of trees (substituting $d = 0$ and $t = 0$), improving upon the approximation ratio obtained in prior work[1].

References

1 S. Arora, V. T. Chakaravarthy, N. Gupta, K. Mukherjee, and Y. Sabharwal. Replica placement via capacitated vertex cover. *FSTTCS*, 2013.

2 A. Benoit, H. Larchevêque, and P. Renaud-Goud. Optimal algorithms and approximation algorithms for replica placement with distance constraints in tree networks. In *IPDPS*, pages 1022–1033, 2012.

3 A. Benoit, V. Rehn-Sonigo, and Y. Robert. Replica placement and access policies in tree networks. *IEEE Trans. on Parallel and Dist. Systems*, 19:1614–1627, 2008.

4 J. Chuzhoy and J. Naor. Covering problems with hard capacities. *SIAM Journal Computing*, 36(2):498–515, 2006.

5 I. Cidon, S. Kutten, and R. Soffer. Optimal allocation of electronic content. *Computer Networks*, 40:205–218, 2002.

6 U. Feige. A threshold of ln n for approximating set cover. *JACM*, 45(4):634–652, 1998.

7 R. Gandhi, E. Halperin, S. Khuller, G. Kortsarz, and A. Srinivasan. An improved approximation algorithm for vertex cover with hard capacities. *JCSS*, 72(1), 2006.

8 K. Kalpakis, K. Dasgupta, and O. Wolfson. Optimal placement of replicas in trees with read, write, and storage costs. *IEEE Trans. on Parallel and Dist. Sys.*, 12:628–637, 2001.

9 M. J. Kao and C. S. Liao. Capacitated domination problem. *Algorithmica*, pages 1–27, 2009.

10 Y. F. Lin, P. Liu, and J. J. Wu. Optimal placement of replicas in data grid environments with locality assurance. In *ICPADS*, 2006.

11 B. Saha and S. Khuller. Set cover revisited: Hypergraph cover with hard capacities. In *ICALP*, 2012.

12 L. Wolsey. An analysis of the greedy algorithm for the submodular set covering problem. *Combinatorica*, 2(4):385–393, 1982.

13 J. J. Wu, Y. F. Lin, and P. Liu. Optimal replica placement in hierarchical data grids with locality assurance. *J. of Parallel and Dist. Computing*, 68:1517–1538, 2008.