Approximation Algorithms for Data Placement Problems

A thesis submitted to University of Delhi

for the award of the degree of

Doctor of Philosophy

by

Sonika Thakral

Department of Computer Science

University of Delhi

Delhi-110007, India

December, 2017

Approximation Algorithms for Data Placement Problems

Abstract of thesis submitted to University of Delhi

for the award of the degree of

Doctor of Philosophy

by

Sonika Thakral

Department of Computer Science

University of Delhi

Delhi-110007, India

December, 2017

Dedicated to my son, Parth.

Declaration

The thesis entitled "*Approximation Algorithms for Data Placement Problems*", which is being submitted for the award of the degree of Doctor of Philosophy is a record of original and bona fide research work carried out by me in the Department of Computer Science, University of Delhi, Delhi, India.

The work presented in this thesis has not been submitted to any other university or institution for any academic award.

Sonika Thakral
Department of Computer Science,
University of Delhi,
Delhi, India.

Certificate

This is to certify that the thesis entitled "*Approximation Algorithms for Data Placement Problems*" being submitted by Sonika Thakral in the Department of Computer Science, University of Delhi, Delhi, for the award of degree of Doctor of Philosophy is a record of original research work carried out by her under the supervision of Prof. Neelima Gupta and Dr. Yogish Sabharwal.

The thesis or any part thereof has not been submitted to any other University or institution for any academic award.

Supervisor
Neelima Gupta
Department of Computer Science
University of Delhi
Delhi, India.

Head
Department of Computer Science
University of Delhi
Delhi, India.

Supervisor
Yogish Sabharwal
IBM Research Lab.
Delhi, India

Department of Computer Science
University of Delhi
Delhi-110007, India

Date:

Certificate of Originality

The research work embodied in this thesis entitled "*Approximation Algorithms for Data Placement Problems*" has been carried out by me at the Department of Computer Science, University of Delhi, Delhi, India. The manuscript has been subjected to plagiarism check by Turnitin software. The work submitted for consideration of award of Ph.D. is original.

Name and Signature of the Candidate

Student Approval Form

Name of the Author	Sonika Thakral
Department	Computer Science
Degree	Ph.D.
University	University of Delhi
Supervisors	Prof. Neelima Gupta, Dr. Yogish Sabharwal
Thesis Title	Approximation Algorithms for Data Placement Problems
Year of Submission	2017

Agreement

1. I hereby certify that, if appropriate, I have obtained and attached hereto a written permission/statement from the owner(s) of each third party copyrighted matter to be included in my thesis/dissertation, allowing distribution as specified below.

2. I hereby grant to the university and its agents the non-exclusive license to archive and make accessible, under the conditions specified below, my thesis/dissertation, in whole or in part in all forms of media, now or hereafter known. I retain all other ownership rights to the copyright of the thesis/dissertation. I also retain the right to use in future works(such as articles or books) all or part of this thesis/dissertation or project report.

Conditions:

1. Release the entire work for access worldwide	
2. Release the entire work for 'My University' only for 1 Year 2 Years 3 Years and after this time release the work for access worldwide	
3. Release the entire work for 'My University' only while at the same time releasing the following parts of the work(e.g. because other parts relate to publications) for worldwide access a) Bibliographic details and Synopsis only. b) Bibliographic details, Synopsis and the following chapters only. c) Preview/Table of Contents/24 page only	
4. View Only(No Downloads)(worldwide)	

Signature of the Candidate **Signature and seal of Supervisors**

Place:

Date:

Acknowledgment

First of all, I thank the Almighty for giving me the strength to undertake this work.

I take this opportunity to express my sincere gratitude towards my supervisors, Prof. Neelima Gupta and Dr. Yogish Sabharwal for their continued support and guidance throughout this work. I wish to thank Prof. Neelima Gupta for believing in me and making me equipped to enter the world of research. If it was not for her, I would not have been exposed to this wonderful learning experience. I have always found her meticulous ways of analysing techniques and verifying results amazing. She has been a constant support at both academic and personal fronts. Her appreciative ways not only instilled confidence in me, but also gave me the courage to carry on even in the toughest of times. I cannot thank Dr. Yogish Sabharwal enough for his patience and ever-ready helping hand. He has always been accessible and willing to extend help and guidance. I shall always be grateful for his efforts towards developing in me the confidence to approach problems. I have always been inspired by his unique ways of arriving at solutions himself and pushing me towards a solution through discussions. I wish to thank him for letting me know the essence of 'giving' and motivating me to make my contribution towards the society.

I would like to extend very special thanks to Prof. Samir Khuller and Dr. Venkatesan Chakaravarthy. Prof. Samir Khuller's helping attitude towards students is overwhelming and has deeply influenced me. I shall always be grateful for his efforts and support in grooming me as a researcher. Dr. Venkatesan Chakaravarhty's simplicity and passion for research are exemplary. I have learnt from him not only techniques related to my Ph.D. work, but also effective teaching techniques. I hope to have adopted some fraction of his prowess over the few years of working with him.

I thank Prof. Naveen Kumar, Prof. Vasudha Bhatnagar, Dr. Rajiv Raman and Dr. Venkatesan Chakaravarthy for accepting to be a part of my advisory com-

mittee. I would also like to thank my co-authors in different papers, Swati Singhal, Dr. Koyel Mukherjee, Kanika Gupta, Anshul Agarwal and Sachin Sharma. I express sincere gratitude to Prof. S.K. Muttoo and Prof. Vasudha Bhatnagar for creating a conducive environment in the department of Computer Science during their respective periods of headship. I thank Dr. Poonam Verma, Principal, Shaheed Sukhdev College of Business Studies, for extending moral and administrative support during the tenure of my Ph.D. My heartfelt thanks to Dr. Manisha Bansal and Geeta Aggarwal, my co-research scholars, for their encouragement and emotional support at both personal and academic fronts. I thank all my colleagues in S.R.C.C., Kalindi College, Hansraj College, S.S.C.B.S. and the Department of Computer Science for accommodating in difficult times.

I am forever indebted to my mom and my son, Parth, for their unshakable support, patience and encouragement. I owe this accomplishment to my mom's never-ending and unconditional succour. Without her, I would not have been able to do justice to personal and professional lives. I extend sincere apology to my son for all those times when I gave my work priority over him. I thank him for accommodating and cooperating in demanding times. My heartfelt gratitude to my late father for dreaming big for me. I would like to thank both my brothers, Praveen and Naveen, for taking the position of our father for me. I particularly thank Naveen for inculcating in me the confidence and courage to never give up. A very special thanks to my friend and sister-in-law, Seema, for being a constant source of inspiration and being there for me in times of need. Her calmness and simplicity, in both thought and lifestyle, has always moved me. This acknowledgement would be incomplete without mentioning the name of my aunt, Smt. Neera, I have always been the subject of whose prayers and wishes. I thank her for loving me so much. Last but not the least, I wish my son Parth, my neice Nitika, and my nephews Anshuman, Kushagra and Tanmay, good luck for all their endeavours in life.

Sonika Thakral

Abstract

Data placement problems deal with cost effective placement of data on servers in order to serve a given set of clients requiring access to the data. Each client is assigned to some server on which data is placed for getting its request served. The cost function may include different parameters, such as the cost of placing data on different servers or the total sum of "distances" between the clients and the servers they are assigned to. In this work we address two types of data placement problems under capacity constraints - the Replica Placement problem and the *typed* Data Placement problem. The two problems differ in the notion of capacity. Whereas in the Replica Placement problem capacity defines the number of clients that can be served by one server, in the *typed* Data Placement problem capacity indicates the maximum number of services that any server may offer. We study variants of these problems that are NP hard and present LP rounding based constant factor approximation algorithms for them.

We study the following variants of the replica placement problem:

- Replica Placement on tree graphs with unit-length edges; we present a polynomial time $O(1)$ approximation algorithm for this variant. We extend our techniques for graphs having bounded tree-width and present an $O(t)$ approximation algorithm where t is the tree-width of the graph.

- Replica Placement on graphs with arbitrary edge lengths, having bounded degree and bounded tree-width (called BDBT graphs); we present polynomial time $O(d + t)$ approximation algorithm for this variant where d and t respectively denote the degree and tree-width of the graph.

- Replica Placement on a generalization of BDBT graphs wherein BDBT graphs are connected in a tree-like manner; we call such graphs Trees of Bounded Degree Bounded Tree-width graphs (TBDBT graphs). We present $O(d+t)$ approximation algorithm for this variant where d and t respectively denote the degree and tree-width of any component BDBT graph.

For the *typed* Data Placement problem (which closely resembles the Facility Location problem), we study the variant having two different service types and no facility opening costs; we present a polynomial time 4-approximation algorithm for this variant.

Contents

List of Figures

Chapter 1

Introduction

Data placement problems are concerned with the placement of data on servers in a cost effective manner so as to serve a set of clients that require access to the data. Different measures for determining the cost include cost of placing data at different locations and the total sum of "distances" between the clients and the data they access. We study two types of data placement problems under capacity constraints that differ in the notion of capacity. The first is the Replica Placement problem in which the capacity refers to the number of clients that can be served by one server. The second problem is called the *typed* Data Placement problem wherein there are different types of data (or services) and the capacity refers to the number of services that can be offered by one server. We describe the replica placement problem in Section 1.1 and the *typed* data placement problem in Section 1.2.

1.1 Replica Placement problem

In the replica placement problem, we have a network of nodes represented as a (connected) graph and a set of clients. Each client is connected (via an edge) to a specific node of the network and requires access to a shared database. In order to serve the clients, we wish to place a minimum number of servers (or replicas) of the database at selected network nodes and assign each client to a server. In the absence of any additional constraints, a simple solution would be to place

one replica at any node. Since the underlying network graph is connected, this replica would clearly be accessible to every client. However, the problem becomes interesting and useful if every server has an upper limit on the number of client requests it can serve. With this constraint the problem is NP-hard. This can be shown by a simple reduction from the bin packing problem. The problem also becomes more involved and meaningful if every client has a limit on the distance it can travel to get its request served. It can be shown that this constraint makes the problem $\lg n$-hard to approximate by a simple reduction from the set cover problem. In our setup we have both the above constraints. However, the upper limit on the number of requests that can be accommodated is assumed to be same for all the servers. In a nutshell, our setup is described as below. Every client has a quality of service (QoS) requirement that stipulates the maximum distance between the client and the server it is assigned to. The clients require periodic access to a shared database. The rate at which a client requests access is termed as its demand. Each server has limited capacity that stipulates the total demand it can accommodate. A server can also be placed directly at a client itself, but in this case it is not allowed to serve any other client and therefore we refer to it as a *dedicated replica*.

The replica placement problem finds applications in settings such as data distribution by Internet Service Providers (ISPs) and Video-on-Demand service delivery (see [CKS02, KDW01, LLW06, TX05]). We next define the problem formally and discuss its variants.

1.1.1 Problem Definition

The input consists of a graph $G = (V, E)$ and a set of clients A. Each edge $e \in E$ is associated with a length $w(e)$. Each client $a \in A$ is attached to a node in V, denoted by $\texttt{att}(a)$. For each client $a \in A$, the input specifies a request (or demand) $r(a)$ and an integer $d_{\max}(a)$, representing the maximum distance the request can travel. For a client $a \in A$ and a node $u \in V$, let $d(a, u)$ denote the length of the shortest path between u and $\texttt{att}(a)$. The length of a path, p, is the sum of lengths

of the edges on that path, i.e., $len(p) = \sum_{e \in p} w(e)$, and we take $d(a, u) = 0$, if $u = \texttt{att}(a)$. We say that a client $a \in A$ can *access* a node $u \in V$, if $d(a, u)$ is at most $d_{\max}(a)$ (we call this the distance constraint). We shall say that u is *accessible* to a, or alternatively, a is *attachable* to u. The input also includes a capacity parameter W. A feasible solution consists of two parts: (i) a subset of nodes $S \subseteq V$ where servers are opened; (ii) for each client $a \in A$, it either opens a dedicated server at a itself or assigns the request to the server opened at some node $u \in S$ accessible to a. The solution must satisfy the constraint that for each node $u \in S$, the sum of requests assigned to the server at u does not exceed W (we call this the capacity constraint). The cost of the solution is the number of servers opened, i.e., cardinality of S plus the number of dedicated servers opened at the clients. The goal is to compute a solution of minimum cost.

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

In case the input graph is directed, $d(a, u)$ denotes the length of the shortest directed path from $\texttt{att}(a)$ to u.

If the length of every edge is taken to be unity, i.e., $w(e) = 1 \ \forall e \in E$, the length of a path is measured by the number of edges on the path. We refer to this variant of the problem as **Replica Placement with hop counts**.

In another variant of the problem, there is no upper limit on the distance a client can travel to have its request served. This variant is referred to as **Replica Placement without distance constraints**.

We shall study the Replica Placement problem and its variants on different graphs.

1.1.2 Integer Program Formulation

For each $u \in V$, a variable $y(u)$ is introduced to represent whether or not a replica is placed at u, for each client $a \in A$, a variable $y(a)$ is added to represent whether a dedicated replica is opened at a itself. A variable $x(a, u)$ is used to represent

whether or not client a is assigned to u, where u is a node accessible to a. For a client $a \in A$ and a node $u \in V$, we use the shorthand "$a \sim u$" to mean that a can access u.

$$\begin{aligned}
& & \textit{Minimize} \quad & \sum_{a \in A} y(a) + \sum_{u \in V} y(u) & & \\
\text{s.t.} \quad & y(a) + \sum_{u \in V \,:\, a \sim u} x(a,u) & \geq \; & 1 & \forall \quad a \in A & \quad (1.1) \\
& \sum_{a \in A \,:\, a \sim u} x(a,u) \cdot r(a) & \leq \; & y(u) \cdot W & \forall \quad u \in V & \quad (1.2) \\
& x(a,u) & \leq \; & y(u) & \forall \quad a \in A,\, u \in V\colon\, a \sim u & \quad (1.3) \\
& y(u), y(a) & \in \; & \{0,1\} & & \quad (1.4) \\
& x(a,u) & \in \; & \{0,1\} & & \quad (1.5)
\end{aligned}$$

Constraint (1.1) ensures that every client is served. Constraint (1.2) (called the *capacity constraint*) enforces that at any node the total demand assigned does not exceed the available capacity W. Constraint (1.3) stipulates that a client a cannot be serviced at a node u which is not open. The integrality constraints (1.4) and (1.5) can be relaxed as follows to yield the corresponding linear program:

$$0 \leq y(u), y(a) \leq 1$$

$$0 \leq x(a,u) \leq 1$$

1.1.3 Hardness

The hardness of the replica placement problem is dependent on the underlying graph. We show that for arbitrary graphs, the replica placement problem is $\lg n$ hard to approximate and for the case when the graph is a path, it is strongly NP-hard.

The following theorem captures the hardness result for arbitrary graphs.

Theorem 1.1. *For arbitrary graphs, the replica placement problem is* $\lg n$ *hard to approximate, even when the requests are all unit* ($r(a) = 1 \; \forall a \in A$) *and there is no capacity constraint.*

Proof. We consider the following reduction from the set cover problem. Consider an arbitrary instance of set cover, where we are given a set U of elements, $U = \{e_1, e_2, ..., e_n\}$ and a set of subsets $S = \{S_1, S_2, ..., S_r\}$ such that $S_j \subseteq U \; \forall j$. From the given instance of set cover we construct an instance of the replica placement problem on a bipartite graph $G = (X \cup Y, E)$ as follows. For every set $S_i \in S$, introduce a node u_i in Y. Corresponding to an element $e_j \in U$, introduce a node a_j in X. For every set S_i and every element e_j in S_i, introduce an edge of length 1 between u_i and a_j. This completes the construction of the underlying network graph for the replica placement instance. Finally, for each element $e_j \in U$, we introduce a client $\hat{a}_j$ and attach it to node a_j. Set $d_{\max}(\hat{a})$=1 for every client $\hat{a}$. It is easy to see that a solution $\mathcal{O}$ to the set cover instance can be transformed into a solution $\mathcal{O}^*$ (of the same cost) to the replica placement instance in polynomial time. For every set $S_i \in \mathcal{O}$, add u_i to $\mathcal{O}^*$; assign every client $\hat{a}_j$ to some $u_i \in \mathcal{O}^*$ such that a_j is adjacent to u_i. Similarly, a solution $\mathcal{O}^*$ to the replica placement instance can be transformed into a solution $\mathcal{O}$ (of cost no more than the cost of $\mathcal{O}^*$) to the set cover instance. For every node $u_i \in \mathcal{O}^*$, add S_i to $\mathcal{O}$; for every node $a_j \in \mathcal{O}^*$ or client $\hat{a}_j \in \mathcal{O}^*$, identify any node u_k adjacent to a_j and add S_k to $\mathcal{O}$ (if no such node is already in $\mathcal{O}$). □

The following theorem captures the hardness result for the case when the graph is a path.

Theorem 1.2. *The replica placement problem is strongly NP-hard for the case where the graph is a path even when there are no distance constraints.*

Proof. This can be shown by the following reduction from the bin-packing problem. Given a bin-packing instance over m bins and n items, we construct a path having m nodes corresponding to the bins and for each item, we create a client having demand same as the size of the item. We attach all the clients to one end of the path. The distance limit of every client is set to ∞ and the capacity W for each node u is taken to be the bin capacity. □

Note that the above reduction rules out the possibility of designing exact algorithm running in time $n^{O(t)}$ (say via dynamic programming) or parametrized algorithms with t as the parameter for graphs having bounded tree-width t.

1.1.4 Related Work

The replica placement problem and its variants have been well-studied for tree networks in the existing literature [CKS02], [WLL08], [BRSR08], [KL09], [BLRG12], from both practical and algorithmic perspectives. Benoit et al. [BLRG12] presented a 2-approximation algorithm for the replica placement problem without distance constraints. 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.

The replica placement problem falls under the framework of the capacitated set cover problem, the generalization of the classical set cover problem wherein each set is associated with a capacity specifying the number of elements it can cover. Two versions of capacitated set cover problem and its special cases have been considered: soft capacity and hard capacity settings. In the former case, a solution can pick the same set an unbounded number of times, whereas in the latter case, a set can be picked at most a bounded number of times. Our work falls under the more challenging hard capacity setting where a set can be picked at most once.

For the capacitated set cover problem under hard capacities, the work of Wolsey [Wol82] yields an $O(\log \Delta)$-approximation algorithm, where Δ is the maximum set cardinality. Chuzhoy and Naor [CN06] presented a simpler proof of the above result. These algorithms apply to the replica placement problem and provide $O(\log n)$-approximations.

The capacitated version of the vertex cover problem has been addressed in prior work. For the soft-capacity setting, Guha et al.[GHKO03] designed a 3-approximation algorithm, and the approximation ratio was subsequently improved to 2 by Gandhi et al. [GHK$^+$06]. The case of hard-capacities was handled

by Chuzhoy and Naor [CN06], who devised a 3-approximation algorithm. They also showed that the weighted version of the problem (wherein the vertices have costs/weights) is as hard as the set cover problem. Saha and Khuller [SK12] considered the more difficult case of multi-graphs and presented a constant factor approximation. Their result applies to capacitated set cover problem as well and yields an $O(f)$-approximation algorithm, where f is the maximum number of sets in which an element appears. Recently, Cheung et al. [CGW14] improved these approximation factors to $(1 + 2\sqrt{3})$ and $2f$, respectively.

The replica placement problem is also related to the capacitated facility location framework (e.g., [LSS12]) However, a crucial difference is that replica placement problem restricts the access of a facility to some clients whereas in facility location problem every facility is accessible to all the clients; also, the cost model of replica placement does not include the distance between clients and facilities.

1.1.5 Our Results

As the problem is *lgn* hard on general graphs, we study it on specific underlying graphs and present LP rounding based approximation algorithms for each of them.

We begin by studying the problem with hop counts on tree graphs. From Theorem 1.2 it follows that the problem is strongly NP-hard. Our main result for this problem is to show that it admits a constant factor approximation algorithm. This is captured by the following theorem.

Theorem 1.3. *The replica placement problem with hop counts on tree graphs admits a polynomial time $O(1)$ approximation algorithm.*

The proof for this theorem is presented in Chapter 2; this also serves as an exposition of some of the basic ideas used in algorithms presented in subsequent chapters. We shall also show that the transformation involved in this theorem takes time polynomial in the input size.

We next consider generalizations of this variant along two dimensions; considering more general graphs and allowing for arbitrary lengths on the edges. We

first consider a variant allowing for more general graphs. We show that the replica placement problem with hop counts on bounded tree-width graphs admits a constant factor approximation algorithm. Tree-width is a measure of how tree-like the graph is; the smaller the tree-width, the closer the graph is to being a tree. A formal definition of tree-width is given in Chapter 3. Intuitively, a graph with bounded tree-width can be decomposed into disconnected components by removing a small number of nodes. This enables the graph to inherit many decomposition properties of trees thereby giving way to simpler algorithms for many important problems. Our result for the above variant of the problem is captured by the following theorem; the proof is presented in Chapter 3. We shall also show that the transformation involved in this theorem takes time polynomial in the input size and parameter t.

Theorem 1.4. *The replica placement problem with hop counts on bounded tree-width graphs admits a polynomial time $O(t)$ approximation algorithm, where t is the tree-width of the underlying network graph.*

We next consider variants of the problem allowing for arbitrary edge lengths. We start with a simple network graph, the *bounded degree bounded tree-width* (BDBT) graph and show that the problem admits a constant factor approximation algorithm for this network graph. A formal definition of BDBT graphs follows.

Definition 1. *Bounded Degree Bounded Tree-width graph (BDBT graph).* We say that a graph G is a *bounded degree bounded tree-width graph* if it has bounded degree and bounded tree-width. Moreover, it has a designated node that we call the root of the graph (denoted by $\texttt{Rt}(\texttt{G})$). □

Our main result for the problem on BDBT graphs is captured by the following theorem; the proof is presented in Chapter 4. We shall also show that the transformation involved in this theorem takes time polynomial in the input size, and parameters t and d.

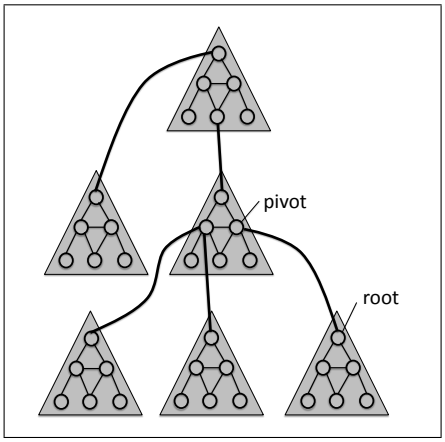


Figure 1.1: Example of TBDBT graph. The figure also shows the root of one of the component BDBT graphs and the pivot that it connects to in another BDBT graph.

Theorem 1.5. *The replica placement problem on BDBT graphs admits a polynomial time $O(d+t)$ approximation algorithm, where d and t are the degree and tree-width of the graph respectively.*

Note that the BDBT graphs do not include trees as trees do not have bounded degree. In order to generalize the above result to include tree graphs, we consider a more general class of graphs that we call TBDBT graphs. Intuitively, a TBDBT graph is composed of BDBT graphs connected in a tree-like manner. A TBDBT graph is constructed by starting with a skeletal rooted tree $\mathcal{T}$, whose vertices are referred to as proxies. Then, each proxy is substituted with a BDBT graph. For each proxy q in $\mathcal{T}$, the root vertex of the associated BDBT graph is connected to some vertex (called pivot) in the BDBT graph associated with the parent proxy of q (see Figure 1.1). The overall graph has bounded tree-width, but may not have bounded degree. A formal definition of TBDBT graphs follows.

Definition 2. *Tree of BDBT graphs (TBDBT graph). A TBDBT graph G is a pair $\langle \{G_j | j \in J\}, \mathcal{T} \rangle$ where $G_j(V(G_j), E(G_j))$ is a BDBT graph, $J = [1, h]$ and $\mathcal{T}$ is a (skeletal) tree with the elements of J as nodes and labeled edges. We consider an arbitrary element of J to be the root of $\mathcal{T}$. This imposes ancestor-descendant relationships between the elements of J. A TBDBT graph satisfies the following*

properties:

- *The vertices of the BDBT graphs are disjoint, i.e.,* $V(G_i) \cap V(G_j) = \phi \ \forall$ $i, j \in J$, $i \neq j$.
- *The vertex set of* G *is the union of the vertices of BDBT graphs, i.e.,* $V(G) = \cup_{j \in J} V(G_j)$.
- *The edges of all the BDBT graphs are contained in* G*, i.e.,* $E(G_j) \subseteq E(G)$ *for all* $j \in J$.
- *For every edge* $e = (i, j)$ *in* $\mathcal{T}$ *where* j *is ancestor of* i*, the label* $\ell(e)$ *on* e *represents a vertex in* $V(G_j)$ *and is said to be a pivot node.*
- *For every edge* $e = (i, j)$ *in* $\mathcal{T}$ *where* j *is ancestor of* i*, there is an edge* $(\mathtt{Rt}(\mathtt{G_i}), \ell(\mathtt{e}))$ *in* G*. We refer to* $\ell(e)$ *as the pivot of* $\mathtt{Rt}(\mathtt{G_i})$ *and denote it as* $pivot(\mathtt{Rt}(\mathtt{G_i}))$.

$\mathtt{Roots}(\mathtt{G})$ *denotes the roots of all BDBT graphs of* G*, i.e.,* $\mathtt{Roots}(\mathtt{G}) = \{\mathtt{Rt}(\mathtt{G_j}) : \mathtt{j} \in \mathtt{J}\}$. □

The class of TBDBT graphs clearly generalizes trees (wherein each component BDBT graph consists of a single vertex). Our main result for the replica placement problem on TBDBT graphs is captured by the following theorem; the proof is presented in Chapter 5. We shall also show that the transformation involved in this theorem takes time polynomial in the input size and parameter t and d.

Theorem 1.6. *The replica placement problem on TBDBT graphs admits a polynomial time* $O(d + t)$ *approximation algorithm, where* d *and* t *are respectively the degree and tree-width of any component BDBT graph.*

1.2 Typed Data Placement problem

In the *typed* data placement problem, we are concerned with the placement of different types of data on servers in order to serve a set of clients, wherein the cost

depends on the distance between the clients and the servers serving them. It is thus more closely related to the facility location problem.

Facility location is a widely studied problem in operations research concerned with the optimal placement of facilities providing certain services or commodities to clients. In the Uncapacitated Facility Location (UFL) version of the problem, we are given as input a set $\mathcal{F}$ of facilities and a set $\mathcal{D}$ of clients in a metric space. The goal is to open facilities S ($\subseteq \mathcal{F}$) and assign every client to an open facility so as to minimize the cost $\sum_{i \in S} f_i + \sum_{j \in \mathcal{D}} d_j \cdot c_{\sigma(j)j}$ where f_i is the cost of opening facility i, d_j is the demand associated with client j, c_{ij} is the distance between i and j, and $\sigma(j)$ denotes the facility that client j is assigned to.

Consider a business offering two types of services (commodities) to the clients that wants to open outlets in every town; it wants to determine which business to setup at each outlet given that a limited number of sites are available in each town. For example:

- a financial institution providing Banking and Non-Banking services. It wants to setup a branch for at least one of its services in every town of a given region of interest.

- a car manufacturer wants to setup a showroom and/or a service station in every town of a given region of interest.

Motivated by these scenarios, we study the *typed* data placement problem defined next. We study the problem with two types of services.

1.2.1 Problem Definition

We are given as input, facilities $\mathcal{F}$, and clients $\mathcal{D}$, as before. We assume that clients are a subset of the facilities, i.e., $\mathcal{D} \subseteq \mathcal{F}$. We also assume that there are no facility opening costs, i.e., $f_i = 0 \ \forall i \in \mathcal{F}$. There are two types of objects, o_1 and o_2; let O denote the set $\{o_1, o_2\}$. For every client j, its demand for o_1 (o_2 respectively) is specified by $d_j^{o_1}$ ($d_j^{o_2}$ respectively); a demand of 0 indicates absence of demand

for the corresponding object-type. Moreover, each facility has a storage capacity specifying the number of objects (1 or 2) that can be placed on it. A facility that has a storage capacity of 2 can be treated as two co-located facilities each with storage capacity 1. Therefore, we can assume that each facility can accommodate only a single object-type. A feasible solution comprises of (i) a subset $S \subseteq \mathcal{F}$ that can be partitioned into sets S^{o_1} and S^{o_2} corresponding to facilities to be opened for o_1 and o_2 object-types respectively, and (ii) an assignment of demands to the set of open facilities, $\sigma : \mathcal{D} \times O \rightarrow S$, wherein an o_1 (o_2 respectively) demand is assigned to a facility in S^{o_1} (S^{o_2} respectively).

1.2.2 Integer Program Formulation

The IP for the *typed* data placement problem is as follows:

$$\begin{aligned}
Minimize \quad & \sum_{j \in \mathcal{D}} \sum_{o \in O} \sum_{i \in \mathcal{F}} d_j^o \cdot c_{ij} \cdot x_{ij}^o \\
\text{s.t.} \quad \sum_{i \in \mathcal{F}} x_{ij}^o &\geq 1 && \forall\ j \in \mathcal{D},\ o \in O && (1.6)\\
x_{ij}^o &\leq y_i^o && \forall\ i \in \mathcal{F},\ j \in \mathcal{D},\ o \in O && (1.7)\\
\sum_{o \in O} y_i^o &\leq 1 && \forall\ i \in \mathcal{F} && (1.8)\\
x_{ij}^o &\in \{0,1\} && \forall\ i \in \mathcal{F},\ j \in \mathcal{D},\ o \in O && (1.9)\\
y_i^o &\in \{0,1\} && \forall\ i \in \mathcal{F},\ j \in \mathcal{D},\ o \in O && (1.10)
\end{aligned}$$

where $x_{ij}^o = 1$ iff client j is assigned to facility i for object o and $y_i^o = 1$ iff facility i stores object o. Constraint (1.6) ensures that the demand of every client for both the object types is served. Constraint (1.7) stipulates that the demand of a client j for an object type o cannot be served at a facility i which is not open for the same object type. Constraint (1.8) is the storage capacity constraint on the facilities which enforces that at most one object type can be placed on any facility. The relaxed LP is obtained by modifying the integrality constraints (1.9) and (1.10) to:

$$x_{ij}^o,\ y_i^o \ \geq\ 0 \qquad \forall\ i \in \mathcal{F},\ j \in \mathcal{D},\ o \in O$$

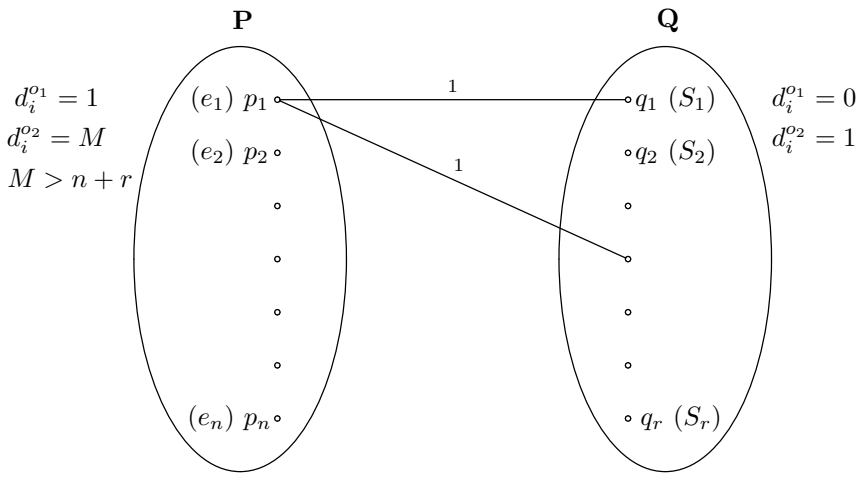


Figure 1.2: NP-Hardness Construction

1.2.3 NP Hardness

The following theorem establishes the NP hardness of the *typed* data placement problem. The proof of the theorem goes via a simple reduction from the set cover problem.

Theorem 1.7. *The typed data placement problem is NP hard.*

Proof. Consider an arbitrary instance of set cover, where we are given a set U of elements, $U = \{e_1, e_2, ..., e_n\}$ and a set of subsets $S = \{S_1, S_2, ..., S_r\}$ such that $S_j \subseteq U\ \forall j$. We construct an instance of the *typed* data placement problem from the above instance as follows. We define two sets of clients P and Q such that each client in P has demand for both o_1 and o_2, and each client in Q has demand for o_2 only (see Figure 1.2).

The set of clients will be defined by $\mathcal{D} = P \cup Q$. For each $e_i \in U$, we introduce a client p_i in P and for each $S_j \in S$, we introduce a client q_j in Q. We associate demands $d_i^{o_1} = 1$ and $d_i^{o_2} = M$ with each $p_i \in P$, where $M > n + r$. We also associate demands $d_j^{o_1} = 0$ and $d_j^{o_2} = 1$ with each $q_j \in Q$. For each $S_j \in S$ and $e_i \in S_j$, we set the distance $c_{p_i q_j} = 1$. For all the remaining pairs of demands, we set the distance to be the length of the shortest path between them (observe that this cost would be at least 3). The set of facilities $\mathcal{F}$ is taken to be the same as $\mathcal{D}$.

We next show that given a feasible solution to the set cover instance of cost $\mathcal{C}$, there exists a feasible solution to the *typed* data placement problem instance having cost $\mathcal{C}+n$. Let SC_{opt} be a solution to the set cover instance having cost $\mathcal{C}$. We construct a solution to the instance of our problem as follows.

- $\forall q_j \in Q$ do the following - If $S_j \in SC_{opt}$, place o_1 on q_j, else place o_2 on it.
- Place o_2 on $p_i \forall i$.

The feasibility of the above solution can be shown as follows. Assign each client $p_i \in P$ to itself for its o_2 demand, and to some $q_j \in Q$ for its o_1 demand such that $e_i \in S_j$ and $S_j \in SC_{opt}$ (so that q_j has been opened for o_1). Note that such an S_j must exist since SC_{opt} is a set cover. Also, for each q_j such that $S_j \notin SC_{opt}$, assign q_j to itself for its o_2 demand, and for each $q_{j'}$ such that $S_{j'} \in SC_{opt}$, assign $q_{j'}$ to some $p_{i'}$ such that $e_{i'} \in S_{j'}$. The cost of this solution can be easily determined as follows:

1. Each p_i traverses a unit cost edge for its o_1 demand thereby contributing an amount n to the total cost.
2. Each q_j for which $S_j \in SC_{opt}$ traverses a unit cost edge for its o_2 demand thereby contributing an amount of $|SC_{opt}|$ to the cost.

Therefore total cost of the solution $= |SC_{opt}| + n = \mathcal{C} + n$.

Also, given a feasible solution to the *typed* data placement problem instance of cost $\mathcal{C}$, there exists a feasible solution to the set cover problem instance having cost at most $\mathcal{C}-n$. This can be shown as follows. Suppose that we have a solution $\hat{S}$ of cost $\mathcal{C}$ to the reduced instance. Let $\hat{S}_1 \subseteq \hat{S}$ be the facilities that host o_1. We first make the following claim: The solution $\hat{S}$ can be moulded into another solution $\hat{S}'$ of cost lesser than $\mathcal{C}$, such that all facilities which host o_1 belong to Q. Let $j \in \hat{S}_1$ be a facility in P. Clearly, $\mathcal{C} \geq M$. We modify this solution to obtain the new solution $\hat{S}'$ by doing the following: Place o_2 on all facilities in P and o_1 on all facilities in Q. The o_1 demands of all such facilities as j can now be satisfied

by facilities in Q since each client in P is connected to at least one facility in Q by a unit-cost edge. The o_2 demands in Q can similarly be satisfied by facilities in P since each client in Q is connected to at least one facility in P through a unit edge (This is true under the assumption that there are no empty subsets in S. In fact, any empty subset can safely be removed without affecting the solution). It is easy to see that $cost(\hat{S}') = n + r < M \leq \mathcal{C}$. We now begin with solution $\hat{S}$ of cost $\mathcal{C}$ to our problem and construct a solution of size at most $\mathcal{C} - n$ to the Set Cover Problem. On the basis of above claim, it can be assumed that all facilities in $\hat{S}$ which host o_1 belong to Q.

- Let $\hat{S_1} \subseteq Q$ be the clients in Q that have been allocated o_1.
- Let $\hat{S_2} \subseteq Q$ be the clients in Q that have been allocated o_2.
- Let $C \subseteq P$ be the clients which are connected to some facility in $\hat{S_1}$ by an edge.
- Let $NC \subseteq P$ be the clients which are not connected to any facility in $\hat{S_1}$ by an edge.

Clearly, the components of $\mathcal{C}$ would be:

- $|\hat{S_1}|$ contributed by clients of $\hat{S_1}$ for their o_2 demand.
- $|C|$ contributed by clients of C for their o_1 demand.
- at least $3|NC|$ contributed by the clients of NC for their o_1 demand. (This is because each element of set $|NC|$ traverses at least 3 edges to satisfy its o_1 demand)

Thus,

$$\begin{aligned} \mathcal{C} &\geq |\hat{S_1}| + |C| + 3|NC| \\ &= |\hat{S_1}| + 2|NC| + (|NC| + |C|) \\ &= |\hat{S_1}| + 2|NC| + n \qquad \text{(because } |C| + |NC| = n\text{)} \end{aligned}$$

The solution for Set Cover instance can be constructed as follows:

1. Include all sets corresponding to elements in $\hat{S}_1$ in the set cover. These sets cover the elements in C.

2. For each element in NC, pick an element from $\hat{S}_2$ that covers it.

Clearly, cost of the above solution is at most $(|\hat{S}_1| + |NC|)$. Also it is easy to see that

$$\begin{aligned} |\hat{S}_1| + |NC| &< |\hat{S}_1| + 2|NC| \\ &< \mathcal{C} - n \end{aligned}$$

□

1.2.4 Related Work

The Uncapacitated Facility Location (UFL) problem has been widely studied. The first constant factor approximation algorithm for this problem was given by Shmoys et al.[STA97]. Since then many approximation algorithms have been proposed for this problem. Mahdian et al.[MYZ06] gave a 1.52 approximation algorithm that combines the greedy algorithm of Jain et al.[JMS02, JMM$^+$03] with the idea of cost scaling, and is analyzed using a factor-revealing LP. Subsequently, Byrka and Aardal[BA10] gave a 1.5-approximation algorithm and recently Li[Li11] gave a 1.488-approximation algorithm for this problem. The best LP rounding based algorithm with an approximation guarantee of 1.58 is due to Sviridenko[Svi02]. Guha and Khuller[GK99] proved that it is impossible to get an approximation guarantee of 1.463 for the UFL problem, unless $NP \subseteq DTIME[n^{O(loglogn)}]$.

A closely related problem to the UFL problem is the k-median problem; in this problem there are no facility opening costs, but there is a limit, k, on the number of facilities that can be opened. Prior work ([HKK10, RKN$^+$11]) has studied a variant of the k-median problem that has types associated with the facilities. For the case of two types of facilities, Hajiaghayi et al.[HKK10] proposed a constant factor approximation algorithm using local search techniques.

Krishnaswamy et al.[RKN+11] used LP rounding techniques to provide a constant factor approximation algorithm for this problem for arbitrary number of types. We note that the notion of types in their problem is different from our problem. In their problem, facilities are partitioned based on types, while the clients have no types associated with them and hence can be served by any open facility. On the contrary, in our problem, the notion of types is associated only with clients, while the facilities can be opened for any object-type; however, a demand can only be served by a facility open for the same type. This fundamental difference rules out the possibility of solving our problem by reducing it to their problem.

Another closely related line of work is that of data placement that has applications in the context of multimedia systems ([AAG+10]). This is a generalization of our problem, where there is an arbitrary number of object-types and there is a facility cost associated with each <facility, object-type> pair. Baev and Rajaraman[BR01] gave a 20.5-approximation algorithm for this problem. This was later improved to a 10-approximation algorithm by Baev et al.[BRS08]. Both these algorithms are based on LP-rounding.

1.2.5 Our Result

Several new algorithmic ideas have bridged the gap between upper and lower bounds on approximation ratio for a single object type (i.e., the UFL problem). However, there is no improved lower bound for multiple object types. The 20.5 approximation factor of Baev and Rajaraman[BR01] was reduced to 10 by Baev et al[BRS08] using several new and elegant ideas borrowed from the K-median work [CGTS99]. However, a fundamental question remains – why do we lose so much when we go to multiple object types?

In an attempt to understand the cause of huge loss incurred while attacking the problem with multiple object types, we consider a simple scenario involving two object types with no facility opening cost, and show that we can get a significant reduction in the approximation guarantee and obtain a bound of 4 for this interesting special case. It is worth noting that the algorithm due to Baev

et al[BRS08] does not lead to any improvements in the approximation bounds for these special cases. This is a first step towards closing the gap between the existing lower bounds and approximation guarantees.

Our algorithm for this problem is based on LP-rounding and the result is formally captured by the following theorem; the proof is presented in Chapter 6. We shall also show that the transformation involved in this theorem takes time polynomial in the input size.

Theorem 1.8. *There exists a polynomial time $O(1)$-approximation algorithm for the typed data placement problem.*

This result improves upon [BRS08] when applied to our problem.

Chapter 2

Replica Placement on Tree graphs

2.1 Introduction

In this chapter we study the replica placement problem with hop counts on tree graphs. This also serves as an exposition of some of the basic ideas used in subsequent algorithms. Some procedures introduced here shall also be used in subsequent algorithms.

As an illustration of this problem, consider the instance in Figure 2.1. In this example, the tree graph is defined by the set of vertices $V = \{v_1, v_2, v_3, v_4, v_5\}$. The set of clients is $A = \{a_1, a_2, a_3, a_4, a_5, a_6\}$. For each client, the request, $r(.)$, and the maximum distance it can travel, $d_{\max}(.)$, are mentioned next to it in the figure. Based on the LP formulation for the replica placement problem described in Section 1.1.2, an LP solution is also shown in the figure; the $y(.)$ values are mentioned against the corresponding nodes and the non-zero $x(.,.)$ assignments of the clients to the replicas are indicated against the associated edges (shown with dotted lines).

Our Result. We present a constant-factor approximation algorithm for the replica placement problem with hop counts on tree graphs so as to establish Theorem 1.3. This result is captured by the following theorem and the rest of the chapter is dedicated to proving the theorem.

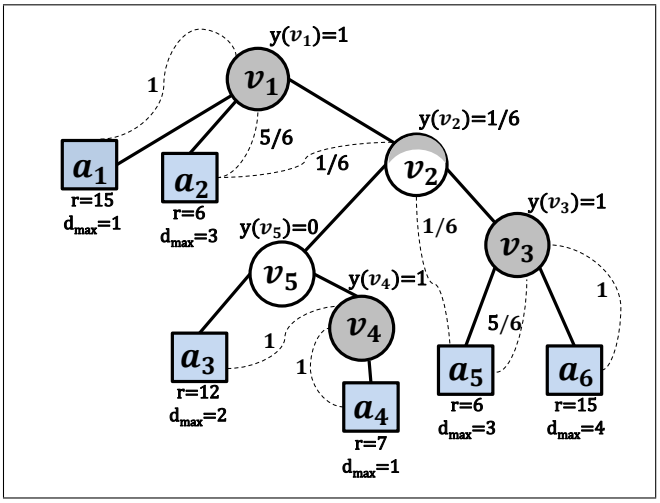


Figure 2.1: An instance on tree graph with unit edge length: W=20

Theorem 2.1. *The replica placement problem with hop counts on tree graphs admits a polynomial time approximation algorithm having cost* $320 \cdot \texttt{OPT} + 28$*, where* OPT *is the cost of the optimal solution.*

2.2 Overview of the Algorithm

In this section, we present an outline of our constant-factor approximation algorithm highlighting its main features, deferring a detailed description to subsequent sections. The algorithm is based on rounding solutions to the natural LP formulation presented in Section 1.1.2. The following simple notations will be useful in our discussion. With respect to an LP solution $\sigma = \langle x, y \rangle$, we classify the nodes into three categories based on the extent to which they are open. A node u is said to be *fully-open*, if $y(u) = 1$; *partially-open*, if $0 < y(u) < 1$; and *fully-closed*, if $y(u) = 0$. A client a is said to be *assigned* to a node u, if $x(a, u) > 0$ (note that a client may be assigned to multiple nodes). For a set of nodes U, let $y(U)$ denote the extent to which the vertices in U are open, i.e., $y(U) = \sum_{u \in U} y(u)$.

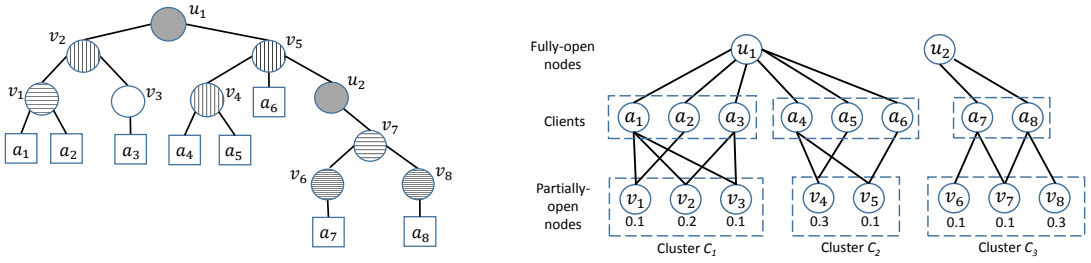
Outline. The major part of the rounding procedure involves transforming a given LP solution $\sigma_{in} = \langle x_{in}, y_{in} \rangle$ into an *integrally open solution*: a solution in which each node $u \in V$ is either fully open or closed. Such a solution differs from an

integral solution in two minor aspects. Firstly, a client $a \in A$ may be partly served by a dedicated replica ($0 < y(a) < 1$) and partly by the network nodes ($1 - y(a)$). Secondly, a client $a \in A$ may be assigned to multiple network nodes. Once an integrally open solution is obtained, we derive an integral solution by applying a simple post-processing step based on a cycle cancellation strategy. While the first step (of obtaining an integrally open solution) exploits the structural properties of tree graphs, the second step (of obtaining an integral solution) is generic and can be applied to other capacitated set cover settings, which could be of independent interest.

The procedure for obtaining an integrally open solution works in two stages. First it transforms the input solution into a clustered solution, which is then transformed into an integrally open solution. The notion of clustered solution lies at the heart of the rounding algorithm. Intuitively, in a clustered solution, the set of partially open nodes are partitioned into a collection of clusters $\mathcal{C}$ and the clients can be partitioned into a set of corresponding groups satisfying three useful properties. Firstly, the assignments from clients to the partially-open nodes is localized: any group of clients is assigned only to the partially-open nodes from the corresponding cluster. Secondly, the assignments from the clients to fully-open nodes are restricted: any group of clients (put together) is assigned to a bounded number of fully-open nodes. Thirdly, the clusters are tiny: for any cluster C, the extent to which it is open, $y(C)$, is bounded by a small constant. The concept is formally defined next.

Let $\sigma = \langle x, y \rangle$ be an LP solution. It will be convenient to express the three properties using the notion of linkage: we say that a node u is *linked* to a node v, if there exists a client a assigned to both u and v. For constants α and ℓ, the solution σ is said to be (α, ℓ)-*clustered*, if the set of partially-open nodes can be partitioned into a collection of clusters, $\mathcal{C} = \{C_1, C_2, \ldots, C_k\}$ (for some k), such that the following properties are true:

- *Localization:* Two partially-open nodes are linked only if they belong to the same cluster.



(a) (b)

Figure 2.2: (a) Illustration of a problem instance showing the underlying tree graph and set of clients $A = \{a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8\}$. The nodes filled solidly ($\{u_1\}$ and $\{u_2\}$) represent fully-open nodes. The solution contains three clusters $C_1 = \{v_1, v_2, v_3\}$, $C_2 = \{v_4, v_5\}$, $C_3 = \{v_6, v_7, v_8\}$ shown by checkered, vertical and horizontal patterns respectively. (b) Solution corresponding to these three clusters C_1, C_2 and C_3, open to an extent of 0.4, 0.4 and 0.5 respectively; no two nodes in different clusters are linked; the nodes in clusters C_1 and C_2 are linked to the fully-open node u_1 and nodes in C_3 are linked to u_2. The solution is 0.5 clustered.

- *Distributivity:* For any C_j, there are at most ℓ fully-open nodes associated with C_j, to which the nodes in C_j may be linked. We refer to ℓ as the distributivity parameter.

- *Bounded opening:* The total extent to which any cluster is open is less than α, i.e., $y(C_j) < \alpha$.

For a tree graph, we aim for distributivity parameter 1, i.e., we attain an $(\alpha, 1)$-*clustered* solution. For simplicity, we shall refer to such a solution as α-*clustered*. Figure 2.2 provides an illustration. In the first stage of the rounding algorithm, we transform the input solution σ_{in} into an α-clustered solution with the additional guarantee that the number of clusters is at most a constant factor of $\text{cost}(\sigma_{in})$, where $\alpha \in [0, \frac{1}{2}]$ is a tunable parameter. The lemma below specifies the transformation performed by the first stage.

Lemma 2.2. *Fix any constant $\alpha \leq 1/2$. Any LP solution σ can be transformed into an α-clustered solution σ' such that $\text{cost}(\sigma')$ is at most $2 + 6 \cdot \text{cost}(\sigma)/\alpha$. Furthermore, the number of clusters is at most $3 + 8 \cdot \text{cost}(\sigma)/\alpha$.*

At a high level, the lemma is proved by performing a bottom-up traversal of the tree graph and identifying a suitable set of nodes, that we call boundary nodes. We use these boundary nodes to split the tree into a set of disjoint clusters. We then fully open the boundary nodes and transfer some assignments from the remaining partially open nodes to these fully-opened boundary nodes so as to form clusters. Note that boundary nodes are themselves not part of the clusters. The transfer of assignments is performed in such a manner that clusters get localized and have distributivity of 1. By carefully selecting the boundary nodes, we shall enforce that each cluster is open to an extent of at most α and that the number of clusters is also bounded. We fix the parameter $\alpha = 1/4$ and use the above lemma to obtain a solution that is 1/4-clustered with $\mathcal{C}$ being the collection of clusters. The proof is discussed in Section 2.3.

The goal of the second stage is to transform the clustered solution into an integrally open solution. The following lemma captures the transformation performed by this stage.

Lemma 2.3. *Let $\sigma = \langle x, y \rangle$ be a 1/4-clustered solution with $\mathcal{C}$ being the collection of clusters. The solution can be transformed into an integrally open solution $\sigma' = \langle x', y' \rangle$ such that* $\text{cost}(\sigma') \leq 2 \cdot \text{cost}(\sigma) + |\mathcal{C}|$.

At a high level, the localization property allows us to independently process each cluster $C \in \mathcal{C}$ and its corresponding group of clients A_c. The clients in A_c are assigned to exactly one fully-open node, say u. We identify a suitable node $v \in C$ called the "consort" of u in C and fully open v. Then the idea is to transfer assignments from the non-consort nodes in C to u and its consort in such a manner that at the end, no client is assigned to the non-consort nodes. This allows us to fully close the non-consort nodes. The localization and bounded opening properties facilitate the above manoeuvre. On the other hand, the distributivity property ensures that we fully open at most 1 consort per cluster. Thus, the overall increase in cost is at most $|\mathcal{C}|$. Since $|\mathcal{C}|$ is bounded, we get a constant approximation factor. The proof is discussed in Section 2.4.

Once we obtain an integrally open solution, it can easily be transformed into an integral solution by applying a cycle cancellation strategy, as given by the following lemma. It is proved in Section 2.5.

Lemma 2.4. *Any integrally open solution $\sigma = \langle x, y \rangle$ can be transformed into an integral solution $\sigma' = \langle x', y' \rangle$ such that* $\text{cost}(\sigma') \leq 4 \cdot \text{cost}(\sigma)$.

We can transform any input LP solution σ_{in} into an integral solution σ_{out} by applying the above three transformations. We fix $\alpha = 1/4$ and apply Lemma 2.2 to obtain a solution σ_1, which is 1/4-clustered via a collection of clusters $\mathcal{C}$. It is guaranteed that $\text{cost}(\sigma_1) \leq 2 + 24 \cdot \text{cost}(\sigma_{in})$ and $|\mathcal{C}| \leq 3 + 32 \cdot \text{cost}(\sigma_{in})$. We next apply Lemma 2.3 on the solution σ_1 and obtain an integrally open solution σ_2 such that $\text{cost}(\sigma_2) \leq 2 \cdot \text{cost}(\sigma_1) + |\mathcal{C}|$. Finally, we transform σ_2 into integral solution σ_{out} using Lemma 2.4 such that $\text{cost}(\sigma_{out}) \leq 4 \cdot \text{cost}(\sigma_2)$. It follows that $\text{cost}(\sigma_{out})$ is at most $28 + 320 \cdot \text{cost}(\sigma_{in})$. This proves Theorem 2.1. Thus, the overall approximation ratio is constant and Theorem 1.3 is established. In Section 2.3 we show that the transformation involved in obtaining a clustered solution (Lemma 2.2) takes time polynomial in the input size. In Section 2.4 we show that the transformation involved in obtaining an integrally open solution (Lemma 2.3) runs in polynomial time. In Section 2.5 we argue that the transformation involved in obtaining an integral solution (Lemma 2.4) runs in polynomial time. Thus, the overall transformation involved in Theorem 2.1 (and hence Theorem 1.3) takes time polynomial in the input size. The rest of the chapter is devoted to proving Lemmas 2.2, 2.3 and 2.4.

2.3 Clustered Solutions: Proof of Lemma 2.2

The goal is to transform a given solution into an α-clustered solution with the properties claimed in the lemma. The idea is to select a set of partially-open or closed nodes and open them fully, and then transfer assignments from the other partially-open nodes to them in such a manner that the partially-open nodes get

For each partially-open node $v \neq u$ (considered in an arbitrary order)
  For each client a that can access both u and v (considered in an arbitrary order)
    Compute capacity available at u: $\mathtt{cap}(u) = W - \sum_{b \in A \,:\, b \sim u} x(b,u) \cdot r(b)$
    If $\mathtt{cap}(u) = 0$ **exit**
    $\delta = \min\left\{x(a,v), \frac{\mathtt{cap}(u)}{r(a)}\right\}$
    Increment $x(a,u)$ by δ and decrement $x(a,v)$ by δ.

Figure 2.3: Pulling procedure for a given partially-open or closed node u.

partitioned into clusters satisfying the three properties of clustered solutions. An issue in executing the above plan is that the capacity at a newly opened node may be exceeded during the transfer. We circumvent the issue by first performing a pre-processing step called de-capacitation.

2.3.1 De-capacitation

Consider an LP solution $\sigma = \langle x, y \rangle$ and let u be a partially-open or closed node. The clients that can access u might have been assigned to other partially-open nodes in σ. We call the node u *de-capacitated*, if even when all the above assignments are transferred to u, the capacity utilization at u is less than W; meaning,

$$\sum_{a \sim u} \sum_{v:\ a \sim v \ \wedge \ v \in \mathsf{PO}} x(a,v) \ < \ W,$$

where PO is the set of partially-open nodes under σ (including u). The solution σ is said to be *de-capacitated*, if all the partially-open and the closed nodes are de-capacitated.

The preprocessing step transforms the input solution into a de-capacitated solution by performing a pulling procedure (given in Figure 2.3) on the partially-open and closed nodes. It is easy to see that the pulling procedure takes time polynomial in the number of nodes of the input graph. The transformation into a de-capacitated solution is captured by the following lemma.

Lemma 2.5. *Any LP solution $\sigma = \langle x, y \rangle$ can be transformed into a de-capacitated solution $\sigma' = \langle x', y' \rangle$ such that* $\mathrm{cost}(\sigma') \leq 2 \cdot \mathrm{cost}(\sigma)$.

Proof. We consider the partially-open and closed nodes, and process them in an arbitrary order, as follows. Let u be a partially-open or closed node. Hypothetically, consider applying the pulling procedure on u. The procedure may terminate in one of two ways: (i) it reaches its capacity limit of W: there may or may not be more assignments that could have been pulled; (ii) all the assignments are pulled and the total capacity utilization at u is still less than W. In the former case, we fully open u and perform the pulling procedure on u. In the latter case, the node u is de-capacitated and so, we leave it as partially-open or closed, without performing the pulling procedure. Thus, every node is either fully-open or de-capacitated. Hence, the above method produces a de-capacitated solution σ'. We next analyze the cost of σ'. Let s be the number of partially-open or closed nodes converted to be fully-open. Apart from these conversions, the method does not alter the cost and so, $\text{cost}(\sigma')$ is at most $s + \text{cost}(\sigma)$. Let the total amount of requests be $r_{\text{tot}} = \sum_{a \in A} r(a)$. The extra cost s is at most $\lfloor r_{\text{tot}}/W \rfloor$, since any newly opened node is filled to its capacity. Due to the capacity constraints, the input solution σ must also incur a cost of at least $\lfloor r_{\text{tot}}/W \rfloor$. It follows that $\text{cost}(\sigma')$ is at most $2 \cdot \text{cost}(\sigma)$. Clearly, the time taken by the de-capacitation procedure is polynomial in the number of nodes of the input graph as it involves applying the pulling procedure to the partially open and closed nodes. □

2.3.2 Clustering

Given Lemma 2.5, assume that we have a de-capacitated solution $\sigma = \langle x, y \rangle$. We next discuss how to transform σ into an α-clustered solution. The transformation would perform a bottom-up traversal of the tree graph and identify a set of partially-open or closed nodes. It would then fully open them and perform the pulling procedure on these nodes. The advantage is that the above nodes are de-capacitated and so, the pulling procedure would run to its entirety (without having to exit mid-way because of reaching capacity limits). As a consequence, the linkage between the nodes gets restricted, leading to a clustered solution. Below we first describe the transformation and then present an analysis.

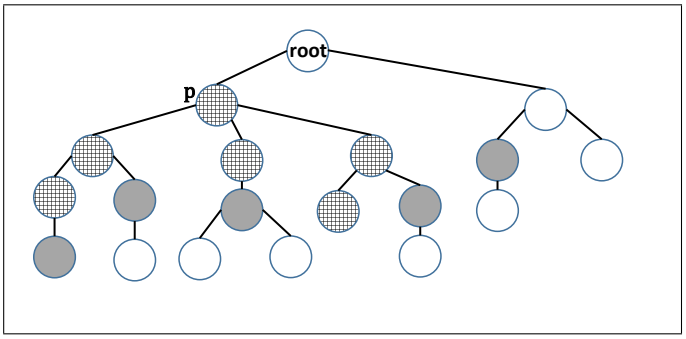


Figure 2.4: Illustration of active(p). The solidly shaded nodes represent the identified boundary nodes; all the checkered nodes are active at p.

Transformation. Consider the given tree G. We select an arbitrary node of G and make it the root. A node p is said to be an *ancestor* of a node q, if p lies on the path connecting q and the root; in this case, q is called a *descendant* of p. We consider p to be both an ancestor and descendant of itself.

In transforming σ into a clustered solution, we shall encounter three types of nodes and it will be convenient to color them as red, blue and brown. To start with, all the fully-open nodes are colored red and the remaining nodes (partially-open nodes and closed nodes) are colored blue. The idea is to carefully select a set of blue nodes, fully-open them and perform the pulling procedure on these nodes; these nodes are then colored brown. Thus, while the blue nodes are partially-open or closed, the red and the brown nodes are fully-open, with the brown and the blue nodes being de-capacitated. A node u is said to be an *anchor* if either it is itself red or it is the parent of a red node.

The transformation identifies two kinds of nodes to be colored brown, *helpers* and *boundary nodes*. We say that a red node $u \in V$ is *proper*, if it has at least one neighbor $v \in V$ which is a blue node. For each such proper red node u, we arbitrarily select one such blue neighbor v and declare it to be the helper of u. Once the helpers have been identified, we color them all brown. To identify the set $\mathcal{B}$ of boundary nodes, we arrange the nodes in G in any bottom-up order (i.e., a node gets listed only after all its children are listed) and then iteratively process each node p as per the above order.

A blue node u is said to be *active at* p, if it is a descendant of p but not a descendant of any other node already in $\mathcal{B}$. See Figure 2.4 for an illustration. Let $\texttt{active}(p)$ denote the set of nodes active at p. We declare p to be a boundary node and add it to $\mathcal{B}$ under three scenarios.

- The node p is the root node
- The node p is an anchor node (i.e., a red node or the parent of a red node)
- If the extent to which the nodes in $\texttt{active}(p)$ are open is at least α, i.e., $\sum_{u \in \texttt{active}(p)} y(u) \geq \alpha$.

If p is identified as a boundary node and it is not red or brown, then we change its color to brown. Note that a node may change its color from blue to brown in the above process, and the new color is to be considered while determining the active sets thereafter. Once the bottom-up traversal is completed, we have a set of brown nodes (helpers and boundary nodes). We consider these nodes in any arbitrary order, open them fully, and perform the pulling procedure on them. We take σ' to be the solution obtained by the above process. This completes the construction of σ'. A pseudocode is presented in Figure 2.5.

Analysis. We now show that σ' is an α-clustered solution. To start with, we had a set of red nodes that were fully-open and a set of blue nodes that were either partially-open or closed under σ. The red nodes do not change color during the transformation. Also, the transformation partitions the set of blue nodes into a set of brown nodes and a set of nodes that stay blue. In the following discussion, we shall use the term 'blue' to refer to the nodes that stay blue. With respect to the solution σ', the red and brown nodes are fully-open, whereas the blue nodes are partially-open or closed.

The transformation outputs a set of boundary nodes $\mathcal{B}$; let $\overline{\mathcal{B}}$ denote the set of non-boundary nodes. If we treat the nodes in $\mathcal{B}$ as cut-vertices and delete them from G, the tree splits into a collection $\mathcal{R}$ of disjoint regions. Alternatively, these regions can be identified in the following manner. For each node $p \in \mathcal{B}$

Input: De-capacitated solution $\sigma = \langle x, y \rangle$

Output: α-clustered solution $\sigma' = \langle x', y' \rangle$

$\texttt{Red} \leftarrow \{u \ : \ u \text{ is fully-open under } \sigma\}$

$\texttt{Blue} \leftarrow \{u \ : \ u \text{ is partially-open or closed under } \sigma\}$

$\texttt{Brown} \leftarrow \emptyset$

// Helpers

Set helpers $H \leftarrow \emptyset$

For each node $u \in \texttt{Red}$

  if u has some neighbor $v \in \texttt{Blue}$, then $H \leftarrow H \cup \{v\}$

Make helpers brown: $\texttt{Blue} \leftarrow \texttt{Blue} \setminus H$ and $\texttt{Brown} \leftarrow \texttt{Brown} \cup H$

// Boundaries: Bottom-up traversal

set $\mathcal{B} \leftarrow \emptyset$

Arrange the nodes in a bottom-up order

For each node p in the above order

  if p is the root, add p to $\mathcal{B}$

  if p is an anchor node, then add p to $\mathcal{B}$

  $\texttt{active}(p) \leftarrow \{q \in \texttt{Blue} : \ q \text{ is a descendant of } p, \text{ but not a descendant of any node in } \mathcal{B} \ \}$

  if $\left(\sum_{u \in \texttt{active}(p)} y(u) \geq \alpha\right)$, add p to $\mathcal{B}$

  if p was added to $\mathcal{B}$ and p is not red or brown

    Delete p from $\texttt{Blue}$ and add to $\texttt{Brown}$

//Pulling

Arrange the nodes in $\texttt{Brown}$ in an arbitrary order

For each node u in the above order

  Fully open u and perform the pulling procedure on it.

Output σ' as the solution obtained above.

Figure 2.5: Pseudocode for clustering

and each of its non-boundary child $q \in \overline{\mathcal{B}}$, define the region headed by q, denoted as R_q to be the set $\texttt{active}(q)$. We call p to be the node defining the region R_q. Note that p may define more than one regions. Let the collection derived be $\mathcal{R} = \{R_1, R_2, \ldots, R_k\}$. It is easy to see that $\mathcal{R}$ partitions $\overline{\mathcal{B}}$ and that the regions in $\mathcal{R}$ are pairwise disconnected (not connected by edges of the tree).

We are now ready to show that σ' is an α-clustered solution. Towards that goal, let us suitably partition the partially open nodes into a collection of clusters $\mathcal{C}$. For each region R_j, let C_j be the set of partially open nodes that occur in R_j. We take $\mathcal{C}$ to be the collection $\{C_1, C_2, \ldots, C_k\}$. We prove some claims below that will be useful in showing that σ' is α-clustered.

Recall that two nodes u and v are linked, if there is a client a assigned to both u and v. In order to prove the properties of α-clustering, we need to analyze the linkage information for the blue nodes. We first show that the blue nodes cannot be linked to brown nodes, by proving the following stronger observation.

Claim 2.6. *If a client $a \in A$ is assigned to a blue node u under σ', then a cannot access any brown node v.*

Proof. As part of the transformation, we perform the pulling procedure on the brown node v. Since σ is de-capacitated, the node v is de-capacitated under σ. As a result, the pulling procedure on v would run to its entirety (without having to exit mid-way because of reaching the capacity limit). This means that the assignment $x(a, u)$ would get transferred to v. Thus, under σ', the client a cannot remain assigned to u, contradicting the assumption in the lemma statement. □

Next, in the following claim we show that clients cannot pass via red nodes to be assigned to blue nodes.

Claim 2.7. *Consider a client a that is assigned to a blue node v. Then, the path from $\texttt{att}(a)$ to v cannot contain a red node.*

Proof. Suppose the path from $\texttt{att}(a)$ to v contains a red node. Let u be the last red node along this path (closest to v) and let w be the node succeeding u along

this path (note that w may be v itself). By Claim 2.6, w cannot be brown and hence it must be blue. This implies that u is a proper red node and hence it has a helper z that is brown. Moreover, z is accessible to a (since v is also accessible to a). This contradicts Claim 2.6. □

The following claim uses Claims 2.6 and 2.7 to give a structure to the linkage between blue nodes.

Claim 2.8. *Consider a client a.*

- *If $\texttt{att}(a) \in R_i$, then the partially open nodes that a is assigned to must be in C_i.*
- *If $\texttt{att}(a)$ does not belong to any region, then a is not assigned to any partially open node.*

Proof. First consider the case where $\texttt{att}(a)$ belongs to some R_i. Suppose a is assigned to a partially open node $u \in C_j$ such that $C_j \neq C_i$. Let p and p' be the nodes defining R_i and R_j respectively. Then clearly, the path from $\texttt{att}(a)$ to u must go through either p or p' or both. Recall that p and p' can be either red or brown. By Claim 2.6 the path from a to u (and hence $\texttt{att}(a)$ to u) cannot contain a brown node and by Claim 2.7 the path from $\texttt{att}(a)$ to u cannot contain a red node. Thus, we arrive at a contradiction, and so a cannot be assigned to a partially open node outside C_i.

Next consider the case where $\texttt{att}(a)$ does not belong to any region. Then either $\texttt{att}(a)$ is brown or it is red. Suppose a is assigned to a partially open node u. Since the path from a to u has to pass through $\texttt{att}(a)$, $\texttt{att}(a)$ cannot be brown or red by Claim 2.6 and Claim 2.7 respectively. Thus, we arrive at a contradiction to the assumption that a is assigned to u. □

The following claim shows that the blue nodes can only be linked to a single red node.

Claim 2.9. *If a client a is assigned to one or more partially open nodes in C_i and a red node p, then p must be the node defining R_i.*

Proof. Suppose that p is not the node defining R_i. Let the node defining R_i be q. Then, either the path from $\texttt{att}(a)$ to p contains q, or p is a descendant of q.

First consider the case where the path from $\texttt{att}(a)$ to p contains q. Recall that q being the node defining R_i can either be brown or red. Claim 2.6 rules out the possibility of q being brown and hence it must be red. Let w be the first red node along the path from $\texttt{att}(a)$ to p (w may be q itself). Note that w cannot be $\texttt{att}(a)$: if $\texttt{att}(a)$ is red, then it does not belong to any region and hence by Claim 2.8, a cannot be assigned to a partially open node. Hence it must be that $\texttt{att}(a)$ is not red. Let u be the node preceding w along this path (u may be $\texttt{att}(a)$ itself). Then, u must be blue and hence w is a proper red node. Thus, w has a helper z that is brown. Moreover, z is accessible to a (since p is also accessible to a and w cannot be p). This contradicts Claim 2.6.

Next, consider the case where p is a descendant of q. Note that p cannot lie on the path from $\texttt{att}(a)$ to q for otherwise p would have been the node defining R_i. Now, consider the path from $\texttt{att}(a)$ to p. Let w be the first red node along this path (w can be p itself). Let u be the node preceding w on this path (u could be $\texttt{att}(a)$ itself); note that u must be the parent of w and hence an anchor node. Thus, u must have been colored brown. This contradicts Claim 2.6. □

We next argue that $\mathcal{C}$ satisfies the three properties of localization, distributivity and bounded opening. However, the number of clusters in the collection may exceed the bound claimed in Lemma 2.2. Later, we show that the issue can be easily rectified by suitably merging the clusters.

Lemma 2.10. *The solution σ' is α-clustered.*

Proof. We prove that the collection $\mathcal{C}$ satisfies the three properties. **Localization** and **Distributivity** trivially follow from Claims 2.8 and 2.9 respectively. We next show that the **Bounded opening** property holds. First of all observe that $y(C_j) \leq y(R_j)$ since $C_j \subseteq R_j$ and the nodes in $R_j \setminus C_j$ are fully closed. We claim that each cluster C_j is open to an extent of less than α, i.e., $y(C_j) < \alpha$. If possible, let $y(C_j) \geq \alpha$, which implies $y(R_j) \geq \alpha$. Consider the blue node q heading region

R_j. Notice that if $y(R_j) \geq \alpha$, the transformation would have made q itself a boundary node, but any region in the collection $\mathcal{R}$ contains only non-boundary nodes. Thus, $y(C_j) \leq y(R_j) < \alpha$ □

We next analyze the cost of the solution $\sigma' = \langle x', y' \rangle$. Let `Red`, `Blue` and `Brown` denote the sets of red, blue and brown nodes respectively. Then, $\text{cost}(\sigma')$ is given by $|\texttt{Red}| + |\texttt{Brown}| + y'(\texttt{Blue}) + y'(A)$, where $y'(A)$ represents the extent to which dedicated replicas are opened, i.e., $y'(A) = \sum_{a \in A} y'(a)$. The red nodes do not change their color, the extent to which any blue node is open also does not change and similarly, for any client a, $y(a)$ does not change. Thus, $|\texttt{Red}| + y'(\texttt{Blue}) + y'(A) \leq \text{cost}(\sigma)$ and hence, $\text{cost}(\sigma') \leq \text{cost}(\sigma) + |\texttt{Brown}|$. We create a brown helper node for each proper red node. Furthermore, we convert each boundary node $p \in \mathcal{B}$ to be brown if it is not red. Thus, $|\texttt{Brown}| \leq |\texttt{Red}| + |\mathcal{B}(blue)|$, where $\mathcal{B}(blue)$ is the set of boundary nodes that were blue at the beginning. A node p is made boundary node under one of the three scenarios. (i) p is the root node; (ii) p is an anchor node; (iii) the total extent to which the nodes in $\texttt{active}(p)$ are open is at least α. The number of boundary nodes that were blue at the beginning, and are of the first two types, are $1 + |\texttt{Red}|$. Regarding the third scenario, the total extent to which the originally blue nodes are open is at most $\text{cost}(\sigma)$. Thus, the number of boundary nodes of the third type is at most $\lceil \text{cost}(\sigma)/\alpha \rceil$. Therefore,

$$|\mathcal{B}(blue)| \leq 1 + |\texttt{Red}| + \lceil \text{cost}(\sigma)/\alpha \rceil \leq 2 + |\texttt{Red}| + \text{cost}(\sigma)/\alpha.$$

It follows that $\text{cost}(\sigma')$ is at most $\text{cost}(\sigma) + |\texttt{Red}| + (2 + |\texttt{Red}| + \text{cost}(\sigma)/\alpha)$, i.e., $\text{cost}(\sigma') \leq 2 + 3 \cdot \text{cost}(\sigma) + \text{cost}(\sigma)/\alpha$ (since $|\texttt{Red}| \leq \text{cost}(\sigma)$). A simple arithmetic shows that $\text{cost}(\sigma')$ is at most $2 + 3\text{cost}(\sigma)/\alpha$ (since α is at most $1/2$). By Lemma 2.5, the preprocessing step of de-capacitation incurs a 2-factor increase in cost. Taking this into account, we get the cost bound claimed in the statement of Lemma 2.2. The clustering step involves identification of helpers and boundary nodes, and applying the pulling procedure on them. Clearly, identifying helpers takes polynomial time as it requires examining the neighbors of some of the nodes of the graph. Identifying boundary nodes takes polynomial time as it requires

performing a simple bottom up traversal of the input graph. Also, the pulling procedure runs in polynomial time, as argued earlier. Thus, the clustering step takes time polynomial in the number of nodes of the input graph.

As mentioned earlier, an issue with the collection $\mathcal{C}$ is that it may have more clusters than the bound claimed in Lemma 2.2. We reduce the number of clusters by suitably merging the clusters. Consider each boundary node p. All the non-boundary children of p have a corresponding cluster in $\mathcal{C}$ and let $\mathcal{C}_p$ denote the collection of these clusters. We start with the collection $\mathcal{C}_p$ and repeatedly perform the following merging operation. Select any two clusters C and C' from $\mathcal{C}_p$ such that $y(C) < \alpha/2$ and $y(C') < \alpha/2$ and merge the two into a single cluster. The process is stopped when we cannot find two such clusters. This way we get a set of new clusters all (except one) of which are open to an extent of at least $\alpha/2$; we refer to these as *normal clusters* and the exceptional one as *abnormal*. We perform this processing for all the boundary nodes and obtain a new collection $\mathcal{C}'$. Note that the solution obtained after merging is also an α-clustered solution with $\mathcal{C}'$ being the set of clusters. The localization property trivially follows; distributivity follows since for any two merged clusters, the only fully open node, to which the partially-open nodes in the clusters can be linked, is the parent boundary node - p in this case (as shown in Claim 2.9); and bounded opening holds since any cluster in $\mathcal{C}'$ is open to an extent of less than α. The number of abnormal clusters is at most $|\mathcal{B}| = |\texttt{Red}| + |\mathcal{B}(blue)|$. The collection $\mathcal{C}'$ is a partitioning of $\texttt{Blue}$ and each normal cluster is open to an extent of at least $\alpha/2$. Thus, the number of normal clusters can be at most

$$\left\lceil \frac{y'(\texttt{Blue})}{\alpha/2} \right\rceil \leq \left\lceil \frac{2\text{cost}(\sigma)}{\alpha} \right\rceil .$$

Hence, the total number of clusters in $\mathcal{C}'$ is at most $3 + 5\text{cost}(\sigma)/\alpha$. The pre-processing step of de-capacitation incurs a 2-factor increase in cost. Taking this into account, we get the bound on number of clusters claimed in the statement of Lemma 2.2. The number of merging operations performed for each boundary node p is bounded by the number of clusters defined by p, and the number of boundary

nodes is bounded by the number of nodes in the graph. Hence, the merging of clusters takes time polynomial in the number of nodes of the input graph.

2.4 Integrally Open Solution: Proof of Lemma 2.3

Our goal is to transform a given 1/4-clustered solution $\sigma = \langle x, y \rangle$ into an integrally open solution σ'. We classify each client as *small* or *large* based on the extent to which it is served by itself (i.e., the dedicated replica opened at it): a client $a \in A$ is said to be *small*, if $y(a) < 1/2$, and it is said to be *large* otherwise. Let A_s and A_l denote the set of small and large clients, respectively.

We pre-process the solution σ by opening a dedicated replica at each large client a and removing its assignments to the nodes (set $y(a) = 1$ and for all nodes u accessible to a, set $x(a, u) = 0$). The transformation at most doubles the cost.

We assume that the solution σ is pre-processed. The solution remains 1/4-clustered via a collection of disjoint clusters $\mathcal{C}$ that partitions the set of partially-open nodes. We shall operate on each cluster independently. For each cluster $C \in \mathcal{C}$, we shall fully open a selected node and fully close the rest of the nodes in the cluster. We now describe the processing for a cluster $C \in \mathcal{C}$. Set $A_c = A_c \setminus A_l$. By the localization property, the clients in A_c cannot be assigned to nodes in any other cluster and by the distributivity property, they are assigned to at most 1 fully-open node, say u.

For a node v and a client a, let $\texttt{load}(a, v)$ denote the amount of load imposed by a on v towards the capacity: $\texttt{load}(a, v) = x(a, v)r(a)$. It will be convenient to define the notion over sets of clients and nodes. For a set of clients B and a set of nodes U, let $\texttt{load}(B, U)$ denote the load imposed by the clients in B on the nodes U: $\texttt{load}(B, U) = \sum_{a \in B, v \in U : a \sim v} x(a, v)r(a)$; when the sets are singletons, we shall omit the curly braces.

The intuition behind the remaining transformation is as follows. We would

have liked to identify a suitable node v in C, fully open it and transfer the assignments of other partially open nodes to it. This would have enabled us to close all the other partially open nodes in C. However, the following issue prevents us from executing this - v may not be accessible to all the clients. Hence, we instead transfer these assignments to u. (We can argue that u is accessible to all these clients: the bounded opening property ensures that $y(C) < 1/4$; also, any client in A_c being *small* is open at most to an extent of $1/2$, therefore these clients are assigned to u at least to an extent of $1/4$). However doing so might result in violation of capacity at u. To address this issue we identify a suitable node v from C, called the *consort* of u in C, fully open it and transfer some assignments from u to v. Consider the non-consort nodes $C' = C - \{v\}$ and the load $\texttt{load}(A_c, C')$ of A_c on C'. The amount of load that we need to assign to u is $\texttt{load}(A_c, C')$ and this is the load we may have to transfer from u to v. Note that $y(C) < 1/4$ implies $y(C') < 1/4$, which further implies $\texttt{load}(A_c, C') < W/4$. Also, note that for every node v in C, we have $y(v) < 1/4$ and hence $\texttt{load}(A_c, v) < W/4$, which in turn implies that if we fully open v, we get an additional space of at least $(3/4)W$. Thus every node in C is a feasible candidate to become consort of u as far as capacity is concerned. However, this capacity can be exploited only if the clients that can access v impose a load of at least $\texttt{load}(A_c, C')$ on u. Note that this requirement may not be satisfied by all the nodes in C. We establish that a node satisfying this requirement exists and identify the node too. Towards this purpose, we define the notion of *pushable load*. For a node u and a node $v \in C$, let $\texttt{pushable}(u, v) = \sum_{a \in A_c : a \sim v} x(a, u) r(a)$. We next show how to identify a suitable consort such that the pushable load is more than the load that we wish to transfer.

Lemma 2.11. *We can find a node v such that* $\texttt{pushable}(u, v) > \texttt{load}(A_c, C')$.

Proof. For a node $v \in C$, let $\texttt{acc}(v)$ denote the total requests of the clients from A_c that can access v: $\texttt{acc}(v) = \sum_{a \in A_c : a \sim v} r(a)$. Select $\hat{v} = \text{argmax}_{v \in C} \texttt{acc}(v)$; see Figure 2.6.

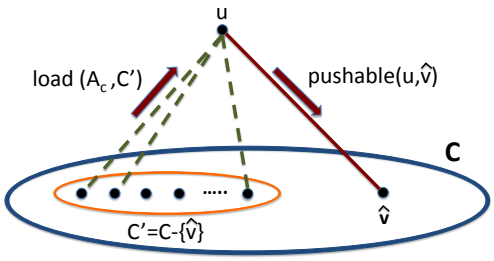


Figure 2.6: Illustration of notion of pushing.

Let $C' = C - \{\hat{v}\}$. Now derive a bound on $\texttt{load}(A_c, C')$:

$$\begin{aligned}
\texttt{load}(A_c, C') &= \sum_{v \in C'} \sum_{a \in A_c : a \sim v} x(a,v) r(a) \\
&\leq \sum_{v \in C'} y(v) \sum_{a \in A_c : a \sim v} r(a) \\
&\leq \sum_{v \in C'} y(v) \texttt{acc}(v) \\
&\leq \texttt{acc}(\hat{v}) \sum_{v \in C'} y(v) \quad < \quad (1/4)\texttt{acc}(\hat{v})
\end{aligned}$$

The second statement follows from the LP constraint (1.3), whereas the third statement is by the definition of $\texttt{acc}(v)$. The fourth statement follows from the construction and the last statement follows from the bounded opening property.

For any client $a \in A_c$, the solution has opened a dedicated replica to an extent of $y(a)$ and the remaining assignment of $1 - y(a)$ is made to the nodes. Our construction has ensured that a is a small client and so $y(a) < 1/2$. This means that the client a is assigned to an extent of at least $1/2$ to the nodes in the cluster. Furthermore, the only nodes to which the client is assigned are u and the nodes in the cluster C. Since $y(C) < 1/4$, the total extent to which the client a is assigned to the nodes in C is less than $1/4$. This implies that $x(a,u) \geq 1/4$. Therefore,

$$\texttt{pushable}(u, \hat{v}) \quad = \quad \sum_{a \in A_c : a \sim \hat{v}} x(a,u) r(a) \quad \geq \quad (1/4) \sum_{a \in A_c : a \sim \hat{v}} r(a) \quad = \quad (1/4)\texttt{acc}(\hat{v}).$$

We have proved the lemma. □

We have shown that each node u has a load of more than $\texttt{load}(A_c, C')$ which can be pushed to its consort $\hat{v}$. As observed earlier $\texttt{load}(A_c, C') < W/4$ and

$\texttt{load}(A_c, \hat{v}) < W/4$. Hence, when we fully open the consort, we get an additional space of at least $(3/4)W$, which is sufficient to receive the load from u. The pseudo-code for processing a cluster C is shown in Figure 2.7.

Let A_c be the set of clients assigned to nodes in C
Let u be the fully-open node linked to nodes in C

/* Selection of consort */
For each node $v \in C$: let $\texttt{acc}(v) = \sum_{a \in A_c : a \sim v} r(a)$.
Let $\hat{v} \leftarrow \text{argmax}_{v \in C} \texttt{acc}(v)$.
Let $C' \leftarrow C - \hat{v}$

/* Push from u to $\hat{v}$
Let load to push: $\texttt{pushable}(u, v) = \sum_{a \in A_c : a \sim v} x(a, u) r(a)$
Let remaining load: $\texttt{rem} \leftarrow \texttt{pushable}(u, \hat{v})$
For each $a \in A_c$ such that $a \sim \hat{v}$ (considered in an arbitrary order)
  Let $\texttt{amnt} \leftarrow \min\{\texttt{rem}, x(a, u) r(a)\}$
  Let $\delta \leftarrow \texttt{amnt}/r(a)$
  $x'(a, \hat{v}) \leftarrow x(a, \hat{v}) + \delta$ and $x'(a, u) \leftarrow x(a, u) - \delta$
  $\texttt{rem} \leftarrow \texttt{rem} - \texttt{amnt}$
  If $\texttt{rem} == 0$ exit loop.

/* Transfer load from C' to u */
For each node $v \in C'$
  For each client $a \in A_c$ and $a \sim v$
    $x'(a, u) \leftarrow x(a, u) + x(a, v)$ and $x'(a, v) \leftarrow 0$

Figure 2.7: Processing for a Cluster C

Given the above discussion, we iteratively consider each cluster $C_j \in \mathcal{C}$ and perform the above transformation. This results in one consort from C_j being fully-opened and all the other nodes in C_j being fully closed. At the end of processing all the clusters, we get a solution in which each node is either fully open or fully closed. For each cluster C_j, we incur an extra cost of at most 1 for opening the

consort. Thus, the cost increases by at most $|\mathcal{C}|$. It is easy to see that identifying the consorts and modifying the assignments takes time polynomial in the size of the input. Hence, this procedure runs in polynomial time.

2.5 Integral Solutions: Proof of Lemma 2.4

An integrally open solution falls short from being an integral solution in two aspects: (i) a client may be assigned to more than one nodes; (ii) a client may be served partly by a dedicated replica and partly by the network nodes. We address the first issue by appealing to the following proposition.

Proposition 2.12. *Given a solution $\sigma = \langle x, y \rangle$, a set of fully-open nodes F and a set of clients $\mathcal{A}$, we can obtain a solution $\sigma' = \langle x', y' \rangle$ such that each client $a \in \mathcal{A}$ is assigned to at most one node from F. Furthermore, the transformation does not alter the other assignments, i.e., for any node $u \in V$ and any client $a \in A$, if $u \notin F$ or $a \notin \mathcal{A}$, then $x'(a, u) = x(a, u)$. The transformation incurs an additional cost of at most F:* $\text{cost}(\sigma') \leq \text{cost}(\sigma) + |F|$.

Proof. Construct an edge-weighted bipartite graph with nodes in F on one side and the clients in $\mathcal{A}$ on the other side. For a pair of nodes $u \in F$ and $a \in \mathcal{A}$, add an edge between the two, if a is assigned to u under σ. In this case, we imagine that a imposes a load of $x(a, u)r(a)$ on the node u and represent the above quantity as the weight on the edge. The plan is to employ a standard cycle-cancellation strategy and make the graph acyclic. Towards that goal, consider any cycle in the graph. Since the graph is bipartite, the cycle must be of even length. Partition the edges of the cycles into two groups, odd and even, by alternating on the cycle. Let $e = (a, u)$ be the edge having the least weight and let $w_{\min} = x(a, u)r(a)$. Assume without loss of generality that e is an odd edge. The idea is to decrease the load on all the odd edges by an amount $w_{\min}$ and increase the load on all the even edges by the same amount. This can be accomplished by adjusting the assignments as follows. For each edge $e' = (a', u')$, compute $\delta = w_{\min}/r(a')$. If e' is an odd edge,

decrease $x(a', u')$ by an amount δ, and otherwise, increase $x(a', u')$ by an amount δ. The edge weights are recomputed accordingly. The above process makes the assignment $x(a, u)$ to be zero and so, we can delete the edge, thereby breaking the cycle. We repeat the process until the bipartite graph becomes acyclic, i.e., a forest.

Consider the resultant LP solution. The forest provides us information on the nodes that the clients are assigned to: a client $a \in \mathcal{A}$ is assigned to a node $u \in F$, if u is a neighbor of a in the forest. Thus, any client $a \in \mathcal{A}$ appearing as a leaf (vertex of degree one) is assigned to only a single node from F. These clients satisfy the property claimed in the proposition. This leaves us with having to deal with clients having multiple neighbors – let $\mathcal{A}'$ denote the set of such clients. We handle these clients simply by opening a dedicated replica at the client node itself. The process produces a solution σ' wherein each client $a \in \mathcal{A}$ is assigned to at most one node $u \in F$.

The above process incurs an extra cost of one unit per dedicated replica and so, the total increase in cost is $|\mathcal{A}'|$. It is not difficult to argue that $|\mathcal{A}'| \leq |F|$. To prove this, we shall produce a one-to-one mapping from $\mathcal{A}'$ to F. Consider each tree in the forest and root it at an arbitrary node from F. Since the graph is bipartite, the nodes from F and the clients from $\mathcal{A}$ appear in alternate levels of the tree. Thus, for any client $a \in \mathcal{A}'$, all its children are from F. For each client $a \in \mathcal{A}'$, pick one of its children $u \in F$ and map a to u. This is a one-to-one mapping and so, $|\mathcal{A}'| \leq |F|$. We have shown that $\text{cost}(\sigma') \leq \text{cost}(\sigma) + |F|$. Clearly, identification and handling of a cycle take time polynomial in the input size. After every cycle cancellation, at least one of the variables becomes integral. So the procedure terminates in polynomial number of steps. Hence, this procedure runs in polynomial time. □

We take F to be the set of all fully open nodes and $\mathcal{A}$ to be the problematic clients and invoke the above lemma. In the resultant solution each client is assigned to at most one fully open node and the cost can increase by a factor of at most two. The second issue is addressed by the following proposition.

Proposition 2.13. *Let $\sigma = \langle x, y \rangle$ be an integrally open solution in which each client is assigned to at most one node. It can be transformed into an integral solution σ' such that* $\text{cost}(\sigma') \leq 2 \cdot \text{cost}(\sigma)$.

Proof. We iteratively consider each fully open node u. Let $A_u \subseteq A$ denote the clients assigned to u. Each client $a \in A$ is served partly by its own dedicated replica to an extent of $y(a)$, while the remaining assignment of $1 - y(a)$ is going to u. We wish to obtain a solution wherein at most one client a with $y(a) > 0$ is assigned to u. Suppose multiple such clients are assigned to u. Choose any two such clients a and b. Without loss of generality, assume that $r(a) \geq r(b)$. Let $\delta = \min\{x(a, u), y(b)\}$. Decrease $x(a, u)$ and $y(b)$ by δ, and increase $y(a)$ and $x(b, u)$ by δ. The assumption that $r(a) \geq r(b)$ ensures that the above transfers do not violate the capacity constraint at the node u. The transfer results in a getting fully served by a dedicated replica and no longer being assigned to u. By repeating the process, we can derive a solution wherein at most one client assigned to u is also served by its dedicated replica. We then open a dedicated replica at this client (say a) and remove its assignment to u (set $y(a) = 1$ and $x(a, u) = 0$). The procedure is repeated for all fully open nodes, leading to an integral solution σ'. The cost increases by at most one unit for each full open node. Thus, $\text{cost}(\sigma') \leq 2 \cdot \text{cost}(\sigma)$. It is easy to see that the processing time for any fully open node is polynomial in the number of clients assigned to it. Hence, the total processing time is polynomial in the size of the input. □

We convert the input integrally open solution σ into an integral solution σ' by applying the above two steps. Each step incurs a 2-factor increase in cost and thus, $\text{cost}(\sigma')$ is at most $4 \cdot \text{cost}(\sigma)$.

Chapter 3

Replica Placement on Bounded Tree-width Graphs

3.1 Introduction

In this chapter we study the replica placement problem with hop counts on graphs having bounded tree-width. Recall that tree-width is a measure of how tree-like the graph is; the smaller the tree-width, the closer the graph is to being a tree. Since bounded tree-width graphs are tree-like, most of the ideas presented in Chapter 2 carry forward here.

Formally, tree decomposition of a graph $G = (V, E)$ is a pair $(X = \{X_j : j \in J\}, T = (J, K))$, where T is a tree over the nodes J and each node $j \in J$ is associated with a subset of vertices (called a bag) $X_j \subseteq V$ such that the following three conditions are satisfied:

- each vertex belongs to at least one bag, i.e., $\bigcup_{j \in J} X_j = V$;
- for every edge $(u, v) \in E$, there is a bag containing both u and v; and
- for all vertices $v \in V$, the set of nodes $\{j \in J : v \in X_j\}$ induces a subtree of T.

The *width* of a tree decomposition is defined to be $\max_{j \in J} (|X_j| - 1)$. The tree-

width t of a graph G is the minimum width over all tree decompositions of G. It is NP-hard to find the tree decomposition of minimum width, but fixed parameter tractable algorithms are known. [DJGT99] provides an algorithm that finds a tree decomposition whose width is at most 4 times the tree-width of the input graph and runs in time $O(f(t).mn)$, where t is the tree-width of the graph, m and n are respectively the number of edges and nodes of the graph, and $f(.)$ is a function that depends only on t.

Our Result. We assume that the input includes a decomposition $\mathcal{T}$ of tree-width t of the input network graph $G = (V, E)$ and present an $O(t)$ approximation algorithm for the problem. This result is captured by the following theorem and the rest of the chapter is dedicated to proving the theorem.

Theorem 3.1. *The replica placement problem with hop counts on bounded tree-width graphs admits a polynomial time* $448(t+1)$*-approximation algorithm with an additional additive factor of* $16 + 24(t + 1)$.

3.2 Overview of the Algorithm

In this section, we present an outline of the algorithm highlighting its main features, deferring a detailed description to subsequent sections. The algorithm is based on rounding solutions to the natural LP formulation presented in Section 1.1.2.

Outline. The rounding procedure is similar to that used for tree graphs. The major part of the procedure involves transforming a given LP solution $\sigma_{in} = \langle x_{in}, y_{in} \rangle$ into an integrally open solution, which is then converted into an integral solution by the same cycle cancellation strategy that was used for tree graphs.

As in case of tree graphs, the procedure for obtaining an integrally open solution works in two stages - the input LP solution is first transformed into a clustered solution, which is further transformed into an integrally open solution. The notion of a clustered solution is as defined in Chapter 2. We recall it here for completeness. A solution σ is said to be (α, ℓ)-*clustered*, if the set of partially-open nodes can be partitioned into a collection of clusters, $\mathcal{C} = \{C_1, C_2, \ldots, C_k\}$ (for

some k), such that the the following properties are true:

- *Localization:* Two partially-open nodes are linked only if they belong to the same cluster.

- *Distributivity:* For any C_j, there are at most ℓ fully-open nodes associated with C_j, to which the nodes in C_j may be linked. We refer to ℓ as the distributivity parameter.

- *Bounded opening:* The total extent to which any cluster is open is less than α, i.e., $y(C_j) < \alpha$.

For bounded tree-width graphs, we cannot aim for distributivity of 1 (unlike tree graphs), i.e., the clients assigned to nodes of a cluster may be assigned to more than one fully open nodes. The distributivity parameter ℓ attains value $t + 1$, where t is the tree-width of the graph; this is to say that the clients assigned to the nodes of a cluster may be assigned to $t+1$ fully open nodes. Figure 3.1 provides an illustration. In the first stage of the rounding algorithm, we transform the input solution σ_{in} into an $(\alpha, t+1)$-clustered solution with the additional guarantee that the number of clusters is at most a constant factor of $\text{cost}(\sigma_{in})$, where $\alpha \in [0, \frac{1}{2}]$ is a tunable parameter. The lemma below specifies the transformation performed by the first stage.

Lemma 3.2. *Fix any constant $\alpha \leq 1/2$. Any LP solution σ can be transformed into an $(\alpha, t+1)$-clustered solution σ' such that $\text{cost}(\sigma')$ is at most $2 + 6(t+1)\text{cost}(\sigma)/\alpha$. Furthermore, the number of clusters is at most $3 + 8 \cdot \text{cost}(\sigma)/\alpha$.*

At a high level, the lemma is proved by considering the tree decomposition $\mathcal{T}$ of the input graph $G = (V, E)$ and performing a bottom-up traversal that identifies a suitable set of boundary bags. We use these boundary bags to split the tree into a set of disjoint regions and create one cluster per region. We then fully open the nodes in the boundary bags and transfer some assignments from the nodes that stay partially-open to these fully-open nodes. The transfer of assignments is performed in such a manner that clusters get localized and have distributivity of

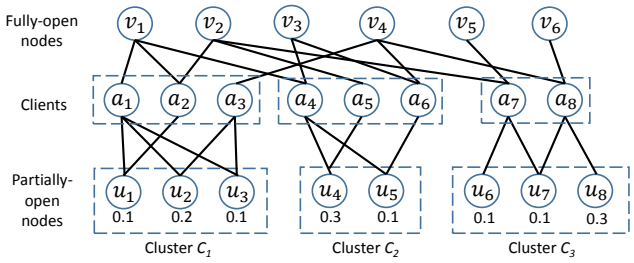


Figure 3.1: Illustration of a clustered solution showing the fully open and partially open network nodes and set of clients $A = \{a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8\}$. Three clusters are shown C_1, C_2 and C_3, open to an extent of 0.4, 0.4 and 0.5, respectively; the clusters are linked to the sets of fully-open nodes $\{v_1, v_2, v_4\}$, $\{v_1, v_2, v_3, v_4\}$, and $\{v_2, v_4, v_5, v_6\}$, respectively. The solution is $(0.5, 4)$-clustered.

$(t+1)$. By carefully selecting the boundary bags, we shall enforce that each cluster is open to an extent of less than α and that the number of clusters is also bounded. We fix the parameter $\alpha = 1/4$ and use the above lemma to obtain a solution that is $(1/4, t+1)$-clustered with $\mathcal{C}$ being the collection of clusters. The proof of the lemma is discussed in Section 3.3.

The goal of the second stage is to transform the clustered solution into an integrally open solution. As discussed in case of tree graphs, the localization property allows us to independently process each cluster $C \in \mathcal{C}$ and its corresponding group of clients A_c. The clients in A_c are assigned to a set of fully-open nodes, say L. For each node $u \in L$, we identify a suitable node $v \in C$ called the "consort" of u and fully open v. Then the idea is to transfer assignments from the non-consort nodes to the nodes in L and their consorts in such a manner that at the end, no client is assigned to the non-consort nodes. This allows us to fully close the non-consort nodes. The localization and bounded opening properties facilitate the above maneuver. On the other hand, the distributivity property ensures that $|L|$ is at most $(t+1)$. This means that we fully open at most $(t+1)$ consorts per cluster. Thus, overall increase in cost is at most $(t+1)|\mathcal{C}|$. Since $|\mathcal{C}|$ is guaranteed to be bounded, we get an $O(t)$ approximation factor. The above transformation is

captured by the following lemma. It is proved in Section 3.4.

Lemma 3.3. *Let $\sigma = \langle x, y \rangle$ be a $(1/4, t+1)$-clustered solution via a collection of clusters $\mathcal{C}$. The solution can be transformed into an integrally open solution $\sigma' = \langle x', y' \rangle$ such that* $\text{cost}(\sigma') \leq 2 \cdot \text{cost}(\sigma) + 2(t+1)|\mathcal{C}|$.

Once we obtain an integrally open solution, it can be transformed into an integral solution by applying Lemma 2.4.

We can transform any input LP solution σ_{in} into an integral solution σ_{out} by applying the above three transformations. We fix $\alpha = 1/4$ and apply Lemma 3.2 to obtain a solution σ_1, which is $(1/4, t+1)$-clustered via a collection of clusters $\mathcal{C}$. It is guaranteed that $\text{cost}(\sigma_1) \leq 2 + 24(t+1)\text{cost}(\sigma_{in})$ and $|\mathcal{C}| \leq 3 + 32 \cdot \text{cost}(\sigma_{in})$. We next apply Lemma 3.3 on the solution σ_1 and obtain an integrally open solution σ_2 such that $\text{cost}(\sigma_2) \leq 2 \cdot \text{cost}(\sigma_1) + 2(t+1)|\mathcal{C}|$. Finally, we transform σ_2 into integral solution σ_{out} using Lemma 2.4 such that $\text{cost}(\sigma_{out}) \leq 4 \cdot \text{cost}(\sigma_2)$. It follows that $\text{cost}(\sigma_{out})$ is at most $16 + 24(t+1) + 448(t+1)\text{cost}(\sigma_{in})$. This proves Theorem 3.1. Thus, the overall approximation ratio is $O(t)$. Hence, Theorem 1.4 is established. In Section 3.3 we show that the transformation involved in obtaining a clustered solution (Lemma 3.2) takes time polynomial in the input size and parameter t. In Section 3.4 we show that the transformation involved in obtaining an integrally open solution (Lemma 3.3) takes time polynomial in the input size and parameter t. Also, as argued in Section 2.5, the transformation involved in obtaining an integral solution (Lemma 2.4) runs in polynomial time. Thus, the overall transformation involved in Theorem Theorem 3.1 (and hence Theorem 1.4) takes time polynomial in the input size and parameter t. The rest of the chapter is devoted to proving Lemmas 3.2 and 3.3.

3.3 Clustered Solutions: Proof of Lemma 3.2

The goal is to transform a given solution into an $(\alpha, t+1)$-clustered solution with the properties claimed in the lemma. We begin by performing the de-capacitation

step as done in case of tree graphs. By Lemma 2.5, a factor 2 increase in cost is incurred in the process.

3.3.1 Clustering

Given Lemma 2.5, assume that we have a de-capacitated solution $\sigma = \langle x, y \rangle$. We next discuss how to transform σ into an $(\alpha, t+1)$-clustered solution. The transformation would perform a bottom-up traversal of the tree decomposition and identify a set of partially-open or closed nodes. It would then fully open them and perform the pulling procedure on these nodes. As before, de-capacitation ensures that the pulling procedure runs to its entirety without exceeding capacity limits. This restricts the linkage between the nodes, leading to a clustered solution. Below we first describe the transformation and then present an analysis.

Transformation. Consider the given tree decomposition $\mathcal{T}$. We select an arbitrary bag of $\mathcal{T}$ and make it the root. A bag X_p is said to be an *ancestor* of a bag X_q, if X_p lies on the path connecting X_q and the root; in this case, X_q is called a *descendant* of X_p. We consider X_p to be both an ancestor and descendant of itself. A node u may occur in multiple bags; its *anchor* denoted by $\texttt{anchor}(u)$ is the bag closest to the root among the bags containing it. A *region* in $\mathcal{T}$ refers to any set of contiguous bags (i.e., the set of bags in a region induce a connected sub-tree).

In transforming σ into a clustered solution, we shall encounter three types of nodes and we color them as red, blue and brown. To start with, all the fully-open nodes are colored red and the remaining nodes (partially-open nodes and closed nodes) are colored blue. The transformation identifies two kinds of blue nodes to be colored brown, *helpers* and *boundary nodes*. We say that a red node $u \in V$ is *proper*, if it has at least one neighbor $v \in V$ which is a blue node. Helpers are defined in the same way as done in case of tree graphs, i.e., for each proper red node u, we arbitrarily select one of its blue neighbors $v \in V$ and call it the helper of u. Multiple red nodes are allowed to share the same helper. Once

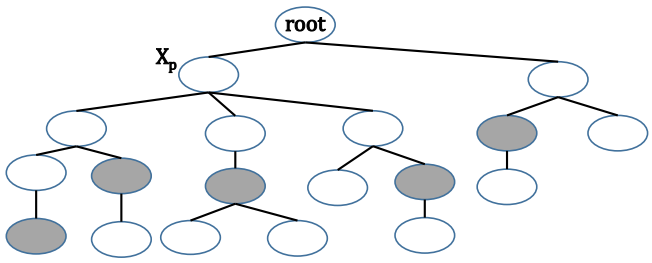


Figure 3.2: Illustration for regions. The figure shows an example tree decomposition. The bags filled solidly represent already identified boundary bags. All the checkered bags belong to the region headed by X_p.

identified, the helpers are colored brown. The boundary brown nodes are selected via a more involved bottom-up traversal of $\mathcal{T}$ that works by identifying a set $\mathcal{B}$ of bags, called the *boundary bags*. To start with, $\mathcal{B}$ is initialized to be the empty set. We arrange the bags in $\mathcal{T}$ in any bottom-up order (i.e., a bag gets listed only after all its children are listed) and then iteratively process each bag X_p as per the above order. Consider a bag X_p. We define the *region headed by* X_p, denoted $\texttt{Region}(X_p)$, to be the set of bags X_q such that X_q is a descendant of X_p, but not the descendant of any bag already in $\mathcal{B}$.

See Figure 3.2 for an illustration. A blue node u is said to be *active at* X_p, if it occurs in some bag included in $\texttt{Region}(X_p)$. Let $\texttt{active}(X_p)$ denote the set of blue nodes active at X_p. We declare a bag X_p to be a boundary bag and add it to $\mathcal{B}$ under three scenarios:

- X_p is the root bag.
- X_p is the anchor of some red node.
- the extent to which the nodes in $\texttt{active}(X_p)$ are open is at least α, i.e., $\sum_{u \in \texttt{active}(X_p)} y(u) \geq \alpha$.

If X_p is identified as a boundary bag, all the blue nodes appearing in X_p are selected as boundary nodes and are colored brown. The new color is to be considered while

determining the active sets thereafter. Once the bottom-up traversal is completed, we have a set of brown nodes (helpers and boundary nodes). We consider these nodes in any arbitrary order, open them fully, and perform the pulling procedure on them. We take σ' to be the solution obtained by the above process. This completes the construction of σ'. A pseudocode is presented in Figure 3.3.

Analysis. We now show that σ' is an $(\alpha, t+1)$-clustered solution. To start with, we had a set of red nodes that were fully-open and a set of blue nodes that were either partially-open or closed under σ. The red nodes do not change color during the transformation. On the other hand, each blue node u becomes active at some boundary bag X_p. If u occurs in the bag X_p, it changes its color to brown, otherwise it stays blue. Thus, the transformation partitions the set of originally blue nodes into a set of brown nodes and a set of nodes that stay blue. In the following discussion, we shall use the term 'blue' to refer to the nodes that stay blue. With respect to the solution σ', the red and brown nodes are fully-open, whereas the blue nodes are partially-open or closed.

Recall that two nodes u and v are linked if there is a client a assigned to both u and v. In order to prove the properties of $(\alpha, t+1)$-clustering, we need to analyze the linkage information for the blue nodes. We first observe that a client a assigned to a blue node u under σ' cannot access any brown node v, as was proved in Claim 2.6.

This observation rules out the possibility of a blue node u being linked to any brown node. Thus, u may be linked to a red node or another blue node. The following lemmas establish a crucial property on the connectivity in these two settings.

Lemma 3.4. *If two blue nodes u and v are linked under σ', then there must exist a path connecting u and v consisting of only blue nodes.*

Proof. Let a be any client that is assigned to both u and v. Consider any shortest path p_1 between u and $\texttt{att}(a)$ (the node to which the client a is attached in the network). The path cannot contain any brown node w, because in this case, $d(a, w)$

Input: De-capacitated solution $\sigma = \langle x, y \rangle$

Output: $(\alpha, t+1)$-clustered solution $\sigma' = \langle x', y' \rangle$

$\texttt{Red} \leftarrow \{u \ : \ u \text{ is fully-open under } \sigma\}$

$\texttt{Blue} \leftarrow \{u \ : \ u \text{ is partially-open or closed under } \sigma\}$

$\texttt{Brown} \leftarrow \emptyset$

// Helpers

Set helpers $H \leftarrow \emptyset$

For each node $u \in \texttt{Red}$

  if u has some neighbor belonging to `Blue` (i.e, a proper red node) then

    Let v be any neighbor of u belonging to `Blue`.

    Let $H \leftarrow H \cup \{v\}$

Make helpers brown: $\texttt{Blue} \leftarrow \texttt{Blue} - H$ and $\texttt{Brown} \leftarrow \texttt{Brown} \cup H$

// Boundaries: Bottom-up traversal

Set $\mathcal{B} \leftarrow \emptyset$

Arrange the bags in a bottom-up order

For each bag X_p in the above order

  if X_p is the root, add X_p to $\mathcal{B}$

  if X_p is the anchor of some node $u \in \texttt{Red}$, then add X_p to $\mathcal{B}$

  $\texttt{Region}(X_p) \leftarrow \{X_q \ : \ X_q \text{ is a descendant of } X_p, \text{ but not a descendant of any bag in } \mathcal{B}\}$

  $\texttt{active}(X_p) \leftarrow \{u \in \texttt{Blue} \ : \ u \text{ occurs in some bag } X_q \in \texttt{Region}(X_p)\}$

  if $\left(\sum_{u \in \texttt{active}(X_p)} y(u) \geq \alpha\right)$, add X_p to $\mathcal{B}$

  if X_p were added to $\mathcal{B}$

    For each node $u \in \texttt{Blue}$ occurring in X_p

      Delete u from `Blue` and add to `Brown`

//Pulling

Arrange the nodes in `Brown` in an arbitrary order

For each node u in the above order

  Fully open u and perform the pulling procedure on u.

Output σ' as the solution obtained above.

Figure 3.3: Pseudocode for clustering

would be smaller than $d(a, u)$, making w accessible to a. This would contradict Claim 2.6. In a similar vein, we claim that the path cannot contain any red node. For otherwise, traverse the path from $\texttt{att}(a)$ to u, and let w be the last red node encountered on the path. Let z be the node succeeding w (it may be the case that $z = u$). The node z is blue and is a neighbor of w in the graph. This means that w is a proper red node and must have a brown helper h. We have that $d(a, w) \leq d(a, u) - 1$ and $d(a, h) \leq d(a, w) + 1$, and hence $d(a, h) \leq d_{\max}(a)$. This means that a can access h, contradicting the above observation (Claim 2.6). We have shown that the path p_1 consists of only blue nodes.

The same argument also shows that any shortest path p_2 connecting $\texttt{att}(a)$ and v must also consist of only blue nodes. The path p_1 connects u and $\texttt{att}(a)$, and the path p_2 connects $\texttt{att}(a)$ and v. By combining the two, we can construct a path p' connecting u and v. The path p' may not be simple, but we can trim it to obtain a simple path p connecting u and v. The path p contains only blue nodes. □

The following lemma provides a similar result regarding linkage between blue and red nodes.

Lemma 3.5. *If a blue node u is linked to a red node v under σ', then there must exist a path p connecting u and v such that barring v, the path consists of only blue nodes.*

Proof. Let a be a client assigned to both u and v under σ'. Let p_1 and p_2 be any shortest paths connecting $\texttt{att}(a)$ with u and v, respectively. As argued in Claim 2.6, the two paths cannot contain any brown nodes and furthermore, p_1 must contain only blue nodes. This implies that the node to which a is attached, $\texttt{att}(a)$, must also be a blue node.

We claim that the path p_2 cannot contain any red nodes, barring v. By contradiction, suppose such a red node exists. Traverse the path from $\texttt{att}(a)$ to v. The first node on the path is $\texttt{att}(a)$, a blue node, and continuing further, let w be the first red node encountered (note that w is different from v). The node

preceding w is a blue node (the preceding node may be $\texttt{att}(a)$ itself). It follows that w is a proper red node and so, it must have a brown helper h. Furthermore, $d(a,w) \leq d(a,v) - 1$ and $d(a,h) \leq d(a,w) + 1$. Thus, $d(a,h) \leq d_{\max}(a)$, which implies that a can access the brown node h, contradicting Claim 2.6.

We have shown that barring v, the paths p_1 and p_2 consist of only blue nodes. By combining the two paths, we can obtain the path p claimed in the lemma. □

The transformation outputs a set of boundary bags $\mathcal{B}$; let $\overline{\mathcal{B}}$ denote the set of non-boundary bags. If we treat the bags in $\mathcal{B}$ as cut-vertices and delete them from $\mathcal{T}$, the tree splits into a collection $\mathcal{R}$ of disjoint regions. Alternatively, these regions can be identified in the following manner. For each bag $X_p \in \mathcal{B}$ and each of its non-boundary child $X_q \in \overline{\mathcal{B}}$, add the region headed by X_q ($\texttt{Region}(X_q)$) to the collection $\mathcal{R}$. Let the collection derived be $\mathcal{R} = \{R_1, R_2, \ldots, R_k\}$. It is easy to see that $\mathcal{R}$ partitions $\overline{\mathcal{B}}$ and that the regions in $\mathcal{R}$ are pairwise disconnected (not connected by edges of the tree decomposition). We next make two observations regarding connectivity among the regions, with the second one being a generalization of the first.

Proposition 3.6. *Consider any region $R_j \in \mathcal{R}$. Let u and v be two nodes such that u occurs only in the bags of R_j, whereas v does not occur in any bag of R_j. Then, any path p in G connecting u and v must pass through some boundary bag X_b, i.e., one of the nodes of p must occur in X_b.*

Proposition 3.7. *Consider any region $R_j \in \mathcal{R}$. Let X_q be the bag heading R_j and let $X_p \in \mathcal{B}$ be its parent bag. Let u and v be two nodes such that u occurs only in the bags of R_j, v does not occur in X_p and $\texttt{anchor}(v)$ does not belong to R_j. Then, any path p connecting u and v must include a node $w \neq v$ such that w occurs in X_p or $\texttt{anchor}(v)$.*

The two propositions can be proved by appealing to the properties of tree decompositions. The first follows as a direct consequence of these properties. We can prove the second by arguing two cases: (i) if $\texttt{anchor}(v)$ is a descendant of X_p,

then the path p must include a node $w \neq v$ occurring in $\texttt{anchor}(v)$; (ii) if $\texttt{anchor}(v)$ is not a descendant of X_p, the path must include a node $w \neq v$ occurring in X_p.

We are now ready to show that σ' is an $(\alpha, t+1)$-clustered solution. Towards that goal, let us suitably partition the set of partially open nodes into a collection of clusters $\mathcal{C}$. For each region R_j, let C_j be the set of partially open nodes that occur in some bag of R_j. We take $\mathcal{C}$ to be the collection $\{C_1, C_2, \ldots, C_k\}$.

Let us verify that the collection $\mathcal{C}$ constructed above is indeed a partitioning of the set of partially open nodes. Firstly, we can see that any partially open node u must belong to some cluster C_j: the node u cannot occur in any boundary bag (for otherwise, u would have turned brown) and so, it must occur in a non-boundary bag found in some region R_j and being partially-open would get included in C_j. Secondly, any blue node u cannot belong to two clusters C_i and C_j. For otherwise, u must occur in some bags $X_{q_1} \in R_i$ and $X_{q_2} \in R_j$. Since R_i and R_j are disconnected, the (unique) path connecting X_{q_1} and X_{q_2} in $\mathcal{T}$ must pass through some boundary bag X_p. By the properties of tree decomposition, the node u must also occur in X_p. In this case, u would have turned brown, contradicting the assumption that u is a blue node.

We next argue that $\mathcal{C}$ satisfies the three properties of localization, distributivity and bounded opening. However, the number of clusters in the collection may exceed the bound claimed in Lemma 3.2. Later, we show that the issue can be easily rectified by suitably merging the clusters.

Lemma 3.8. *The solution σ' is $(\alpha, t+1)$-clustered.*

Proof. We prove the collection $\mathcal{C}$ satisfies the three properties.

Localization. We need to show that any two linked blue nodes u and v belong to the same cluster. By contradiction, suppose that there exist two blue nodes u and v belonging to two different clusters C_i and C_j such that a common client a is assigned to both of them under σ'. Lemma 3.4 shows that u and v are connected by a path p consisting only of blue nodes. Clearly, u and v occur only in the bags of the regions R_i and R_j, respectively. Thus, by Proposition 3.6, some

node w lying on the path p must occur in some boundary bag X_p. However, in this case, the transformation would have turned the blue node w into a brown node, contradicting the fact that w stayed blue.

Distributivity. Consider a cluster C_j and any blue node $u \in C_j$. Let X_q be the bag heading the corresponding region R_j and let X_p be the parent bag of X_q. We claim that any red node v linked to u must occur in X_p. By contradiction suppose v does not occur in X_p. By Lemma 3.5, there must exist a path p connecting u and v, which is made of all blue nodes, barring v. The bag $X_{\widehat{p}} = \texttt{anchor}(v)$ cannot belong to the region R_j as it is a boundary bag and the region R_j consists of only non-boundary bags. Thus, Proposition 3.7 implies that the path p must include a node $w \neq v$ such that w occurs in X_p or $\texttt{anchor}(v)$. This is a contradiction since both X_p and $\texttt{anchor}(v)$ are boundary bags and w is a blue node. The claim implies that all the red nodes that are linked to the blue nodes in C_j occur in the bag X_p. Since $\mathcal{T}$ is a decomposition of width t, X_p can contain at most $t+1$ elements. Thus, the blue nodes in C_j can be linked to at most $t+1$ red nodes. By Claim 2.6, the brown nodes cannot be linked to blue nodes. Thus the blue nodes in C_j can only be linked to the fully open nodes that are red. We have thus proved that the clustering has distributivity parameter $t+1$.

Bounded opening. We claim that each cluster C_j is open to an extent of less than α, i.e., $y(C_j) < \alpha$. For otherwise, consider the corresponding region R_j and the bag X_q heading R_j. Notice that if $y(C_j) \geq \alpha$, the transformation would have made X_q itself into a boundary bag, but any region in the collection $\mathcal{R}$ contains only non-boundary nodes. □

We next analyze the cost of the solution $\sigma' = \langle x', y' \rangle$ and prove the cost bound claimed in Lemma 3.2. Let $\texttt{Red}$, $\texttt{Blue}$ and $\texttt{Brown}$ denote the sets of red, blue and brown nodes respectively. Then, $\text{cost}(\sigma')$ is given by $|\texttt{Red}| + |\texttt{Brown}| + y'(\texttt{Blue}) + y'(A)$. The red nodes do not change their color, the extent to which any blue node is open also does not change and similarly, for any client a, $y(a)$ does not change. Thus, $|\texttt{Red}| + y'(\texttt{Blue}) + y'(A) \leq \text{cost}(\sigma)$ and hence, $\text{cost}(\sigma') \leq \text{cost}(\sigma) + |\texttt{Brown}|$. We create a brown helper node for each proper red node. Furthermore, for each

boundary bag $X_p \in \mathcal{B}$, we convert all the blue nodes in X_p to be brown, and the number of such brown nodes is at most $(t+1)$. Thus, $|\texttt{Brown}| \leq |\texttt{Red}| + (t+1)|\mathcal{B}|$. A bag X_p is made a boundary bag under one of the three scenarios. (i) X_p is the root bag; (ii) X_p is the anchor of some red node; (iii) the total extent to which the nodes in $\texttt{active}(X_p)$ are open is at least α. The number of boundary bags of the first two types are $1 + |\texttt{Red}|$. Regarding the third scenario, we show that each originally blue node u becomes active at exactly one boundary bag. Clearly, u becomes active at at-least one boundary bag; let X_p be the first such boundary bag processed. Then, either $u \in X_p$ or $u \notin X_p$. In the first case, u turns brown and therefore cannot become active at any boundary bag processed later. In the latter case, by the properties of tree decomposition u cannot occur in the region of any other boundary bag. Also, the total extent to which these originally blue nodes are open is at most $\text{cost}(\sigma)$. Thus, the number of boundary bags of the third type is at most $\lceil \text{cost}(\sigma)/\alpha \rceil$. Therefore,

$$|\mathcal{B}| \leq 1 + |\texttt{Red}| + \lceil \text{cost}(\sigma)/\alpha \rceil \leq 2 + |\texttt{Red}| + \text{cost}(\sigma)/\alpha.$$

It follows that $\text{cost}(\sigma')$ is at most $\text{cost}(\sigma) + |\texttt{Red}| + (t+1)(2 + |\texttt{Red}| + \text{cost}(\sigma)/\alpha)$. A simple arithmetic shows that $\text{cost}(\sigma')$ is at most $2 + 3(t+1)\text{cost}(\sigma)/\alpha$ (we use the fact that $|\texttt{Red}| \leq \text{cost}(\sigma)$ and our assumption that the parameter α is at most $1/2$). By Lemma 2.5, the preprocessing step of de-capacitation incurs a 2-factor increase in cost. Taking this into account, we get the cost bound claimed in the statement of Lemma 3.2. As argued in Section 2.3.1, the time taken by the de-capacitation step is polynomial in the number of nodes of the input graph. The clustering step involves identification of helpers and boundary bags, and applying the pulling procedure on the helpers and boundary nodes. Clearly, identifying helpers takes polynomial time as it requires examining the neighbors of some of the nodes of the graph. Identifying the boundary bags requires examining the nodes in each bag, and the number of bags is bounded by the number of nodes in the graph. Therefore the identification of boundary bags takes time polynomial in the input size and t. Also, the pulling procedure runs in polynomial time, as argued earlier. Thus, the

clustering step takes time polynomial in the size of the input and parameter t.

As mentioned earlier, an issue with the collection $\mathcal{C}$ is that it may have more clusters than the bound claimed in Lemma 3.2. We reduce the number of clusters by suitably merging the clusters. Consider each boundary bag X_p. All the non-boundary children of X_p have a corresponding cluster in $\mathcal{C}$ and let $\mathcal{C}_p$ denote the collection of these clusters. We start with the collection $\mathcal{C}_p$ and repeatedly perform the following merging operation. Select any two clusters C and C' from $\mathcal{C}_p$ such that $y(C) < \alpha/2$ and $y(C') < \alpha/2$ and merge the two into a single cluster. The process is stopped when we cannot find two such clusters. This way we get a set of new clusters all of which are open to an extent of at most α. Furthermore, except for perhaps a single cluster, all the others are open to an extent of at least $\alpha/2$; we refer to these as *normal clusters* and the exceptional one as *abnormal*. We perform this processing for all the boundary bags and obtain a new collection $\mathcal{C}'$.

Note that the solution obtained after merging is also an $(\alpha, t+1)$-clustered solution with $\mathcal{C}'$ being the set of clusters. The localization property trivially follows. The distributivity is not affected by the process of merging: as shown in the proof of Lemma 3.8, for any two merged clusters, the partially-open nodes in the clusters can only be linked to the fully-open nodes found in the parent boundary bag and the count of such fully-open nodes can be at most $(t+1)$. Bounded opening holds since any cluster in $\mathcal{C}'$ is open to an extent of at most α. The number of abnormal clusters is at most $|\mathcal{B}|$. The collection $\mathcal{C}'$ is a partitioning of `Blue` and each normal cluster is open to an extent of at least $\alpha/2$. Thus, the number of normal clusters can be at most $\lceil y'(\texttt{Blue})/(\alpha/2) \rceil$, which is at most $\lceil 2\text{cost}(\sigma)/\alpha \rceil$. Hence, the total number of clusters in $\mathcal{C}'$ is at most $3 + 4\text{cost}(\sigma)/\alpha$. The preprocessing step of de-capacitation incurs a 2-factor increase in cost. Taking this into account, we get the bound on number of clusters claimed in the statement of Lemma 3.2. The number of merging operations performed for each boundary bag X_p is bounded by the number of clusters defined by X_p, and the number of boundary bags is bounded by the number of nodes in the graph. Hence, the merging of clusters takes time polynomial in the number of nodes of the input graph.

3.4 Integrally Open Solution: Proof of Lemma 3.3

Our goal is to transform a given $(1/4, t+1)$-clustered solution $\sigma = \langle x, y \rangle$ into an integrally open solution σ'. As done for tree graphs, we classify the clients into two groups, *small* and *large*, based on the extent to which they are served by dedicated servers: a client $a \in A$ said to be *small*, if $y(a) < 1/2$, and it is said to be *large* otherwise. Let A_s and A_l denote the sets of small and large clients respectively.

We pre-process the solution σ by opening a dedicated server at each large client a and removing its assignments to the nodes (set $y(a) = 1$ and set $x(a, u) = 0$ for all nodes u accessible to a). We see that the transformation at most doubles the cost and the solution remains $(1/4, t+1)$-clustered.

Consider the pre-processed solution σ. Let $\mathcal{C}$ denote the set of clusters (of the partially-open nodes) under σ. For each cluster $C \in \mathcal{C}$, we shall fully open a selected set of at most $2(t+1)$ nodes and fully close the rest of the nodes in the cluster.

We now describe the processing for a cluster $C \in \mathcal{C}$. Set $A_c = A_c \setminus A_l$. By the localization property, the clients in A_c cannot be assigned to nodes in any other cluster and by the distributivity property, they are assigned to at most $(t+1)$ fully-open nodes, denoted $L = \{u_1, u_2, \ldots, u_{t+1}\}$. A client $a \in A_c$ may be assigned to multiple nodes from L. In our procedure, it would be convenient if each client is assigned to at most one node from L and we obtain such a structure by appealing to Proposition 2.12 with $F = L$ and $\mathcal{A} = A_c$.

The proposition does not alter the other assignments and so, its output solution is also $(1/4, t+1)$-clustered. Given Proposition 2.12 and the pre-processing, we can assume that $\sigma = \langle x, y \rangle$ is $(1/4, t+1)$-clustered wherein each client $a \in A_c$ is assigned to at most one node from L and that $y(a) < 1/2$. For each node $u_i \in L$, let $A_i \subseteq A_c$ denote the set of clients assigned to the node u_i. Proposition 2.12 guarantees that these sets are disjoint and hence form a partition of A_c; see Figure 3.4 for an illustration.

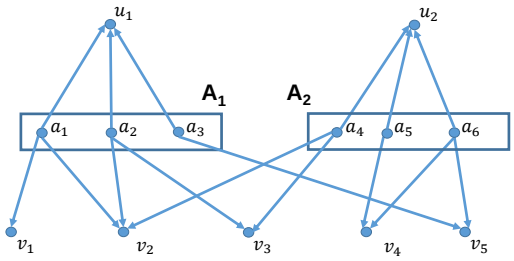


Figure 3.4: An example cluster of five nodes and two fully-open nodes. The clients are partitioned into two groups A_1 and A_2, one per fully-open node.

We next recall some notations used in Chapter 2. For a client a and node v, $\texttt{load}(a, v) = x(a, v)r(a)$; for a set of clients B and a set of nodes U, $\texttt{load}(B, U) = \sum_{a \in B, v \in U: a \sim v} x(a, v)r(a)$; for a subset $C' \subseteq C$, $\texttt{load}(C') = \sum_{v \in C'} \texttt{load}(v)$.

The intuition behind the remaining transformation is similar to that for tree graphs (see Section 2.4). The difference lies in the fact that for a bounded tree-width graph there are $t+1$ fully open nodes $L = \{u_1, u_2, \ldots, u_{t+1}\}$, associated to every cluster (instead of 1) and we identify $t+1$ consorts - one corresponding to each fully open node. We shall thus identify a suitable set of nodes $L' = \{v_1, v_2, \ldots, v_{t+1}\}$ from C, with v_i being called the *consort* of u_i, and fully open all these nodes. Then, we consider the non-consort nodes $C' = C - L'$ and for each $i \leq t+1$, we transfer the load $\texttt{load}(A_i, C')$ to the node u_i. Clearly, no clients are assigned to the non-consort nodes any more and so, they can be fully closed. As done for tree graphs, to facilitate the transfer, for each $i \leq t+1$, we create space in u_i by pushing a load equivalent to $\texttt{load}(A_i, C')$ from u_i to its (fully-opened) consort v_i. By the bounded opening property, $y(C) < 1/4$; clearly, $y(C') < y(C) < 1/4$ and thus, $\texttt{load}(A_i, C') < W/4$. Also, since $y(v_i) < 1/4$, therefore $\texttt{load}(A, v_i) < W/4$, which means that if we fully open the consort, we get an additional space of at least $(3/4)W$. Thus, v_i has enough space to receive the load. We next show how to identify a suitable set of consorts using the notion of *pushable* load. For a node $u_i \in L$ and a node $v \in C$, $\texttt{pushable}(u_i, v) = \sum_{a \in A_i: a \sim v} x(a, u_i)r(a)$.

Lemma 3.9. *We can find a set of nodes $L' = \{v_1, v_2, \ldots, v_{t+1}\}$ such that for all*

$i \leq t+1$, $\texttt{pushable}(u_i, v) > \texttt{load}(A_i, C')$.

Proof. For a set of clients B, let $r(B)$ denote the sum of requests of the clients in B. For a node v, let $r(B, v)$ denote the sum of requests of the clients in B that can access v, i.e., $r(B, v) = \sum_{a \in B: a \sim v} r(a)$.

We identify the required set via a greedy procedure. Initialize $L' = \emptyset$ and iterate over the nodes $u_1, u_2, \ldots, u_{t+1}$. For each node u_i, select $v_i = \text{argmax}_{v \in C - L'} r(A_i, v)$ and add v_i to L'.

Let L' be the set identified by the above procedure and let $C' = C - L'$. Fix any $i \leq t+1$. We derive a bound on $\texttt{load}(A_i, C')$:

$$\begin{aligned} \texttt{load}(A_i, C') &= \sum_{v \in C'} \sum_{a \in A_i: a \sim v} x(a, v) r(a) \quad \leq \quad \sum_{v \in C'} y(v) \sum_{a \in A_i: a \sim v} r(a) \\ &\leq \sum_{v \in C'} y(v) r(A_i, v) \quad \leq \quad r(A_i, v_i) \sum_{v \in C'} y(v) \quad < \quad (1/4) r(A_i, v_i) \end{aligned}$$

The second statement follows from the LP constraint (1.3), whereas the third statement is by the definition of $r(A_i, v)$. The fourth statement follows from the construction and the last statement follows from the bounded opening property.

For any client $a \in A_i$, the solution has opened a dedicated server to an extent of $y(a)$ and the remaining assignment $1 - y(a)$ is made to the nodes. Our construction has ensured that a is a small client and so $y(a) < 1/2$. This means that the client a is assigned to an extent of at least $1/2$ to the nodes in the cluster. Furthermore, the only nodes to which the client is assigned are u_i and the nodes in the cluster C. Since $y(C) < 1/4$, the total extent to which the client a is assigned to the nodes in C is less than $1/4$. This implies that $x(a, u_i) \geq 1/4$. Therefore,

$$\begin{aligned} \texttt{pushable}(u_i, v_i) = \sum_{a \in A_i: a \sim v_i} x(a, u_i) r(a) &\geq (1/4) \sum_{a \in A_i: a \sim v_i} r(a) \\ &= (1/4) r(A_i, v_i). \end{aligned}$$

We have proved the lemma. □

We have shown that each node u_i has a load of more than $\texttt{load}(A_i, C')$ which can be pushed to its consort v_i. The pseudo-code for processing a cluster C is shown in Figure 3.5.

Let A_c be the set of clients assigned to nodes in C

Let $L = \{u_1, u_2, \ldots, u_{t+1}\}$ be the set of fully-open nodes linked to nodes in C

Apply Proposition 2.12 to get a solution $\sigma' = \langle x', y' \rangle$

For $i \leq t+1$, let $A_i \subset A_c$ be the set of clients assigned to u_i.

/* Selection of consorts */

Let $L' \leftarrow \emptyset$

For i to 1 to $t+1$

  For each node $v \in C$: let $r(A_i, v) = \sum_{a \in A_i : a \sim v} r(a)$.

  Let $v_i \leftarrow \text{argmax}_{v \in C - L'} r(A_i, v)$ and add v_i to L'.

Let $C' \leftarrow C - L'$

/* Push from nodes in L to L'

For i from 1 to $t+1$

  Let load to push: $\texttt{pushable}(u_i, v_i) = \sum_{a \in A_i : a \sim v_i} x(a, u_i) r(a)$

  Let remaining load: $\texttt{rem} \leftarrow \texttt{pushable}(u_i, v_i)$

  For each $a \in A_i$ such that $a \sim v_i$ (considered in an arbitrary order)

    Let $\texttt{amnt} \leftarrow \min\{\texttt{rem}, x(a, u_i) r(a)\}$

    Let $\delta \leftarrow \texttt{amnt} / r(a)$

    $x'(a, v_i) \leftarrow x'(a, v_i) + \delta$ and $x'(a, u_i) \leftarrow x'(a, u_i) - \delta$

    $\texttt{rem} \leftarrow \texttt{rem} - \texttt{amnt}$

    If $\texttt{rem} == 0$ exit loop and go to next i.

/* Transfer load from C' to L */

For each node $v \in C'$ and each node $u_i \in L$

  For each client $a \in A_i$ and $a \sim v$

    $x'(a, u_i) \leftarrow x'(a, u_i) + x'(a, v)$ and $x'(a, v) \leftarrow 0$

Figure 3.5: Processing for a Cluster C

Given the above discussion, we iteratively consider each cluster $C_j \in \mathcal{C}$ and perform the above transformation. This results in $(t+1)$ consorts from C_j being fully-opened and all the other nodes in C_j being fully closed. At the end of processing all the clusters, we get a solution in which each node is either fully open or fully closed. For each cluster C_j, we incur an extra cost of at most $(t+1)$ while applying Proposition 2.12 since the number of fully open nodes with which the proposition is invoked is $(t+1)$. Also, an additional cost of $(t+1)$ for opening the consorts. Thus, the cost increases by at most $2(t+1)|\mathcal{C}|$. As argued in Section 2.5, the transformation in Proposition 2.12 takes polynomial time. Identification of consorts for the red nodes associated with a cluster and modifying the assignments takes time polynomial in the input size and t. Further, the number of clusters is polynomial in the number of nodes. Thus, the overall time taken by this procedure is polynomial in the input size and parameter t.

Chapter 4

Replica Placement on Bounded Degree Bounded Tree-width Graphs

4.1 Introduction

In this chapter we study the replica placement problem on BDBT graphs. We first study the problem on the directed version of BDBT graphs defined as follows.

Definition 3. *Directed Bounded Degree Bounded Tree-width graph (directed BDBT graph).* We say that a directed graph is a *directed bounded degree bounded tree-width graph* if the undirected graph obtained by ignoring directions on the edges has bounded degree and bounded tree-width. Moreover, it has a designated node that we call the root of the graph (denoted by $\texttt{Rt}(\texttt{G})$). □

We present a constant factor approximation algorithm for the replica placement problem on directed BDBT graphs. This result appeared in [ACG+14] and is captured by the following theorem. The proof of the theorem is presented in Section 4.2.

Theorem 4.1. *The replica placement problem on directed BDBT graphs admits a polynomial time* $2 \cdot (d + t + 2)$ *factor approximation algorithm, where* d *and* t *are*

the degree and tree-width of the graph respectively.

We then show that a constant factor approximation algorithm can be obtained for the case of (undirected) BDBT graphs by replacing each edge of the graph with a pair of edges, one in each direction and then invoking Theorem 4.1. We loose an additional factor 2 in the approximation ratio. This yields Theorem 1.5 and the proof is presented in Section 4.3.

4.2 Algorithm for directed BDBT graphs

In this section, we shall work with an alternate definition of the problem that is equivalent to the definition of replica placement problem for directed graphs. In this new definition clients are a part of the input graph and therefore participate in all the procedures in the same way as the nodes of the graph do. We call this alternate version *Replica Placement problem with client-nodes*. A formal definition of this alternate version is given below.

Problem Definition: The input consists of a directed graph $G = (V, E)$. Each edge $e \in E$ is associated with a length $w(e)$. Graph G may have nodes with in-degree 0. However, we assume that the out-degree of such nodes is exactly 1. Each leaf node (having no in-edges and one out-edge) represents a client. Let A be the set of all the clients and let $|A| = m$. For each client $a \in A$, the input specifies a request (or demand) $r(a)$ and an integer $d_{\max}(a)$, representing the maximum distance it can travel. The distance constraint and capacity constraint are defined in the same manner as for directed graphs. A feasible solution consists of two parts: (i) a subset of nodes $S \subseteq V$ where servers are opened; (ii) for each client $a \in A$, it assigns the request to a server opened at some node $u \in S$ accessible to a.

The cost of the solution is the number of servers opened, i.e., cardinality of S. The goal is to compute a solution of minimum cost.

□

We next show that the two problem definitions, viz., replica placement for

directed graphs and replica placement with client-nodes, are equivalent. First, consider an instance, I, of the replica placement problem on a directed graph $G = (V, E)$. This can be transformed to an instance I' of the replica placement problem with client-nodes as follows. The underlying graph $G' = (V', E')$ for I' is constructed by modifying the graph G for I. We start with $V' = V$ and $E' = E$. For every client, a in I attached to $\texttt{att}(a) \in V$, introduce a node v_a in V' and a directed edge $(v_a, \texttt{att}(a))$ having 0 length in G' - we call these nodes client-nodes. Note that the directed edges ensure inaccessibility of client-nodes from each other.

A problem solution S' of I' can be converted to a solution S of I as follows. For each node u of G open in S', add u to S as an open replica. Additionally, open dedicated replicas at clients for which the corresponding client-nodes are open in S'. It is easy to see that both these solutions have the same cost.

Next, consider an instance, I', of the replica placement problem with client-nodes on graph $G' = (V', E')$. This can be transformed into an instance I of the replica placement problem; the underlying graph $G = (V, E)$ for I is constructed by modifying the graph G' for I'. We start with $V = V'$ and $E = E'$. For every client-node $a \in V'$ having an edge, say (a, u), to node $u \in V'$, set $V = V' \setminus \{a\}$, add a to the client set A for I and set $\texttt{att}(a) = u$. A problem solution S of I can be converted to a solution S' of I' as follows. For each node u of G' open in S, add u to S' as an open replica. Additionally, open client-nodes corresponding to dedicated replicas open in S. It is easy to see that both these solutions have the same cost.

IP formulation: The following IP formulation is obtained by modifying the IP formulation of the Replica Placement problem given in Section 1.1.2 to capture the Replica Placement problem with client-nodes wherein the clients are an integral part of the graph.

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

$$\sum_{a \in A \,:\, a \sim u} x(a,u) \cdot r(a) \;\leq\; y(u) \cdot W \qquad (\forall u \in V) \tag{4.1}$$

$$x(a,u) \;\leq\; y(u) \qquad (\forall a \in A, u \in V : a \sim u) \tag{4.2}$$

$$\sum_{u \in V \,:\, a \sim u} x(a,u) \;=\; 1 \qquad (\forall a \in A) \tag{4.3}$$

$$y(u) \;\in\; \{0,1\} \qquad (\forall u \in V) \tag{4.4}$$

$$x(a,u) \;\in\; \{0,1\} \tag{4.5}$$

The integrality constraints (4.4) and (4.5) can be relaxed as follows to yield the corresponding linear program:

$$0 \leq y(u) \leq 1$$

$$0 \leq x(a,u) \leq 1$$

4.2.1 Outline of the algorithm

In this section we present an overview of the LP rounding based algorithm for directed BDBT graphs. Consider an optimal solution $\sigma = \langle x_\sigma, y_\sigma \rangle$ to the LP formulation given above. 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 4. *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. The poor nodes, P, are de-capacitated (recall that a node u is said to be de-capacitated if even on transferring to u the assignments of all clients that

can access u, from other partially open nodes, the capacity utilization at u is less than W).

3. Every client is either served 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 served by them, and the second part comprising of the poor nodes and the clients served 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.

The algorithm works in three stages. We start with the LP solution. The first stage transforms the LP solution into a stable solution, the second stage transforms the stable solution into an *integrally-open* solution and finally the third stage transforms the *integrally-open* solution into an *integral* solution. We next provide an outline of these three stages.

Stage 1: Constructing Stable Solutions for Bounded Degree graphs. In the first stage, we transform any feasible solution $\sigma_{in} = \langle x_{\sigma_{in}}, y_{\sigma_{in}} \rangle$ for a bounded degree graph into a stable solution $\sigma_s = \langle x_{\sigma_s}, y_{\sigma_s} \rangle$ as stated in Lemma 4.2 (note that this lemma applies to any bounded degree graph, not just BDBT graphs). We use the notation $\texttt{cost}(.,.)$ to denote the cost of a subset of nodes specified by the second parameter in the solution specified by the first parameter. The proof of the lemma is given in Section 4.2.2.

Lemma 4.2. *Any LP solution σ_{in} for a bounded degree graph can be converted into a stable solution σ_s such that $\texttt{cost}(\sigma_s, R) \leq (d+1) \cdot \text{cost}(\sigma_{in})$ and $\texttt{cost}(\sigma_s, P) \leq \text{cost}(\sigma_{in})$ where R and P are respectively the rich and poor nodes of the stable solution σ_s.*

Stage 2: Converting a Stable Solution to Integrally Open solution for Bounded Tree-width graphs. In the second stage, we show how to transform any stable solution σ_s for a bounded tree-width graph into an integrally open solution σ_{io} as captured in Lemma 4.3 (note that this lemma applies to any bounded tree-width graph, not just BDBT graphs). The proof of the lemma is given in Section 4.2.3.

Lemma 4.3. *Let R and P respectively be the rich and poor nodes of a stable solution σ_s of a graph, 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 nodes of P fully-opened 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.*

Stage 3: Obtaining an Integral Solution for BDBT graphs. Since a BDBT graph has bounded degree, Lemma 4.2 can be used to transform the input solution σ_{in} into a stable solution σ_s, and since a BDBT graph has bounded tree-width, Lemma 4.3 can be used to transform σ_s 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. Our problem definition requires that the request must be wholly assigned to a single node. We address this issue by invoking Proposition 2.12 with F as the set of fully-open nodes in σ_{io} and $\mathcal{A}$ as the set of all the clients (i.e., $\mathcal{A} = A$). In the resultant solution σ_{out} each client is assigned to at most one fully open node and the cost can increase by a factor of at most two. This leads to the following lemma.

Lemma 4.4. *Any integrally open solution σ_{io} can be converted into an integral solution σ_{out} such that* $\text{cost}(\sigma_{out}) \leq 2 \cdot \text{cost}(\sigma_{io})$.

Using the cost analyses stated in Lemmas 4.2, 4.3 and 4.4, we see that

$\text{cost}(\sigma_{out}) \leq 2 \cdot \text{cost}(\sigma_{io}) \leq 2 \cdot (\texttt{cost}(\sigma_s, R) + (t+1) \cdot \texttt{cost}(\sigma_s, P)) \leq 2 \cdot ((d+1) \cdot \text{cost}(\sigma_{in}) + (t+1) \cdot \text{cost}(\sigma_{in})) = 2 \cdot (d+t+2) \cdot \text{cost}(\sigma_{in})$.

This establishes Theorem 4.1.

In Section 4.2.2 we show that the transformation involved in obtaining a stable solution (Lemma 4.2) takes time polynomial in the number of nodes and parameter d. In Section 4.2.3 we show that the transformation involved in obtaining an integrally open solution (Lemma 4.3) takes time polynomial in the input size and parameter t. Also, as argued in Section 2.5, the transformation involved in obtaining an integral solution (Proposition 2.12) runs in polynomial time. Thus, the overall transformation involved in Theorem 4.1 takes time polynomial in the input size, and parameters t and d.

4.2.2 Proof of Lemma 4.2

Let σ_{in} be the input LP Solution. The transformation is performed in two phases called the *de-capacitation* phase and the *stabilization* phase. The former phase is similar to the *de-capacitation* phase described for the case of tree and bounded tree-width graphs (with the exception that clients are also part of the processing in the current scenario).

4.2.2.1 De-capacitation

Consider an LP solution $\sigma_{in} = \langle x_{in}, y_{in} \rangle$ and let u be a partially-open or closed node. The clients that can access u might have been assigned to other partially-open nodes (including client-nodes) in σ_{in}. We call the node u *de-capacitated*, if even when all the above assignments are transferred to u, the capacity utilization at u is less than W; meaning,

$$\sum_{a \sim u} \quad \sum_{v:\ a \sim v\ \wedge\ v \in \mathsf{PO}} x(a,v) \ < \ W,$$

where PO is the set of partially-open nodes under σ_{in} (including u). The solution σ_{in} is said to be *de-capacitated*, if all the partially-open and the closed nodes are

de-capacitated.

This phase transforms the input solution into a de-capacitated solution by performing the pulling procedure (given in Figure 2.3) on the partially-open and closed nodes. Given a partially-open or closed node u, the procedure transfers assignments from other partially-open nodes to u, as long as the capacity at u is not violated. Note that if u is a client-node, then u is the only client that can access the node u. Therefore, since $r(a) \leq W \ \forall a \in A$ (i.e., the total request of any client-node is no more than W), the procedure must be able to transfer to client-node u, the assignments of client u from all other partially open nodes. The following lemma captures the transformation involved in de-capacitation.

Lemma 4.5. *Any LP solution $\sigma_{in} = \langle x_{in}, y_{in} \rangle$ can be transformed into a de-capacitated solution $\sigma' = \langle x', y' \rangle$ such that $\text{cost}(\sigma') \leq 2 \cdot \text{cost}(\sigma_{in})$. Moreover, if FO' and PO' denote the fully and partially open nodes in σ' respectively, then $\texttt{cost}(\sigma', FO') \leq \text{cost}(\sigma_{in})$ and $\texttt{cost}(\sigma', PO') \leq \text{cost}(\sigma_{in})$.*

Proof. We process the partially-open and closed nodes in an arbitrary order, as follows. Let u be a partially-open or closed node. Hypothetically, consider applying the pulling procedure on u. The procedure may terminate in one of two ways: (i) it reaches its capacity limit of W: there may or may not be more assignments that could have been pulled; (ii) all the assignments are pulled and the total capacity utilization at u is still less than W. In the former case, we fully open u and perform the pulling procedure on u. In the latter case, the node u is de-capacitated and so, we leave it as partially-open, without performing the pulling procedure. It is clear that the above method produces a de-capacitated solution σ'. We next analyze the cost of σ'. Let FO and PO denote the fully and partially open nodes in σ_{in} respectively. Let r_{PO} denote the total requests assigned to nodes in PO, i.e., $r_{PO} = \sum_{u \in PO} \sum_{a \in A} r(a) \cdot x(a, u)$. Let s be the number of partially-open or closed nodes converted to be fully-open; note that apart from these conversions, the method does not alter the cost. Note that the request assigned to nodes in s (in σ') were assigned to nodes in PO (in σ_{in}). Therefore,

the extra cost s is at most $\lfloor r_{PO}/W \rfloor$, since any newly opened node is filled to its capacity. Also, due to the capacity constraints, the partially open nodes in input solution σ_{in} must also incur a cost of at least $\lfloor r_{PO}/W \rfloor$. The fully open nodes of σ_{in} remain untouched. It follows that $\texttt{cost}(\sigma', FO') \leq \texttt{cost}(\sigma_{in}, FO) + \text{cost}(s) = \texttt{cost}(\sigma_{in}, FO) + \texttt{cost}(\sigma_{in}, PO) = \text{cost}(\sigma_{in})$. As no new nodes become partially open, it is easy to see that $\texttt{cost}(\sigma', PO') \leq \text{cost}(\sigma_{in})$. □

4.2.2.2 Stabilization

Given Lemma 4.5, assume that we have a de-capacitated solution $\sigma' = \langle x', y' \rangle$. We next discuss how to transform σ' into a stable solution. While performing this transformation, we shall encounter three types of nodes - fully open, partially open and fully closed. The idea is to carefully select a subset of partially open and closed nodes, fully-open them and perform the pulling procedure on these nodes.

Lemma 4.6. *Any de-capacitated solution $\sigma' = \langle x', y' \rangle$ for a BDBT graph can be transformed into a stable solution $\sigma_s = \langle x_s, y_s \rangle$ such that $\texttt{cost}(\sigma_s, R) \leq (d+1) \cdot \texttt{cost}(\sigma', FO')$ and $\texttt{cost}(\sigma_s, P) \leq \texttt{cost}(\sigma', PO')$ where FO' and PO' are respectively the fully-open and partially-open nodes in σ' and R and P are the rich and poor nodes of the stable solution σ_s.*

Proof. To begin with, we color the nodes red, blue and brown as described below. All the fully-open nodes are colored red and the remaining nodes (partially-open nodes and fully-closed nodes) are colored blue. For each red node u, we select all its neighbors that are not already red and declare them to be the *helpers* of u. Multiple red nodes are allowed to share the same helper. Once the helpers have been identified, we color them all brown. We consider these brown nodes in an arbitrary order, open them fully and perform the pulling procedure on them. Thus, while the blue nodes are partially-open, the red and the brown nodes are fully-open, with the brown and the blue nodes being de-capacitated. We take σ_s to be the solution produced by the above process. We take R to be the set of fully-open (red and brown) nodes and P to be the set of partially open (blue)

nodes in the solution σ_s. This completes the description of the procedure. We now show that σ_s is a stable solution with R and P defining the rich and poor nodes respectively.

Claim 4.7. *The solution σ_s is stable.*

Proof. The first two conditions are trivially satisfied as the input solution σ' is de-capacitated. To check the third condition, we prove a stronger claim – we show that any client to which a node in R is accessible, is not assigned to a node in P. Let a be any client to which a node, $u \in R$ is accessible. Observe that either $y_{in}(a) < 1$ or $x_{in}(a, a) = 1$ (i.e., it cannot be that $y_{in}(a) = 1$ and $x_{in}(a, a) < 1$). This is because node a is accessible only to client a, and therefore we can set $y_{in}(a) = x_{in}(a, a)$ (thereby improving the cost of the LP solution in the case when $y_{in}(a) = 1$ and $x_{in}(a, a) < 1$). Thus, if a is itself red, it must be that a is not assigned to any other node as $r(a) \leq W$ - either in the LP solution itself, or the pulling process would have ensured this. If a is not red, then there must be a brown node w on the path from a to u, such that $a \sim w$ since non-red neighbours of red nodes are colored brown. In this case, the pulling process on w would have ensured that a is not assigned to any node in P. This completes the proof of the claim. □

We now analyze the cost. Note that there are at most d neighbours of a red node; thus the total number of brown nodes is at most $d \cdot \texttt{cost}(\sigma', FO')$. This implies that $\texttt{cost}(\sigma_s, R) \leq (d+1) \cdot \texttt{cost}(\sigma', FO')$. As the extent of openness of the remaining nodes is untouched, it follows that $\texttt{cost}(\sigma_s, P) \leq \texttt{cost}(\sigma', PO')$. □

Lemma 4.2 follows by combining Lemmas 4.5 and 4.6. Combining the cost analyses of Lemmas 4.5 and 4.6, together with the fact that the newly opened nodes in the de-capacitation phase become rich, we have:
$\texttt{cost}(\sigma_s, R) \leq (d+1) \cdot \texttt{cost}(\sigma', FO') \leq (d+1) \cdot \text{cost}(\sigma_{in})$ and $\texttt{cost}(\sigma_s, P) \leq \texttt{cost}(\sigma', PO') \leq \text{cost}(\sigma_{in})$.

This completes the proof of Lemma 4.2.

The time taken by the de-capacitation procedure is clearly polynomial in the size of the input. The stabilization step involves identifying the helpers of some of the nodes and performing the pulling procedure on them. Clearly, identification of helpers takes polynomial time as it involves examining the neighbors of some nodes. Also, there are at most d helpers for any node and the pulling procedure runs in polynomial time. Thus, the overall time taken in obtaining a stable solution is polynomial in the number of nodes and parameter d.

4.2.3 Proof of Lemma 4.3

Let σ_s be the input stable solution on a bounded tree-width graph; let R and P be the rich and poor nodes of σ_s respectively. Our procedure to convert σ_s into an integrally open solution σ_{io} 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. Essentially, resolved nodes are those that the algorithm has decided to open or close. 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, $(X = \{X_j : j \in J\}, T = (J, K))$, of the graph G having tree-width t. Fix an arbitrary bag X_r, such that $Rt(G) \in X_r$, to be the root of the tree decomposition. We shall treat the edges of the tree decomposition as directed towards X_r. We say that a client a is *critical* at bag $X_q \in X$ if X_q is the highest bag along the path to X_r such that a can be assigned to some node in X_q. We call a bag *critical* if it is critical for some client.

We will process the bags iteratively in any topological order of T (bottom-

up). Our procedure will maintain the following invariants at the start/end of every iteration:

1. Any client is either assigned to resolved nodes only or unresolved nodes only, not both.

2. On completion of processing of a bag X_q, all clients critical at X_q are resolved.

These invariants are satisfied at the start of the procedure as the initial solution is stable.

Let X_q be the bag being processed in the current iteration. Let π_{old} be the solution at the start of the current iteration; we form a new solution π_{new} in the current iteration. We check if there is any unresolved client, a, that is critical at X_q. If not, we do nothing in this iteration; we simply take $\pi_{new} = \pi_{old}$. Otherwise, consider the partitioning of the set `Unres` based on whether a node appears in X_q or not; let $Y = \texttt{Unres} \cap X_q$ and $\overline{Y} = \texttt{Unres} \setminus X_q$. Further, let Z be the nodes of $\overline{Y}$ that appear in some bag $X_{q'}$ below X_q in the tree decomposition, i.e., $Z = \{u \in \overline{Y} : u \in X_{q'}$ and $\exists$ a path from $X_{q'}$ to X_q in the tree decomposition$\}$. We claim that the extent to which the nodes in Z and Y are open collectively is at least 1.

Claim 4.8. $\sum_{u \in Z \cup Y} y_{\pi_{old}}(u) \geq 1$.

Proof. Recall that the client a critical at X_q is unresolved. Thus a can only be assigned to nodes in $Z \cup Y$ (invariant 1). 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. This implies the claim. □

Given Claim 4.8, we would like to fully open the nodes in Y and close the nodes in Z. This can be done if the clients assigned to nodes in Z can be assigned to the nodes in Y. Towards this, we claim that every client assigned to a node in Z has some node of Y accessible to it.

Claim 4.9. *Let b be any client served by some $u \in Z$. Then some node $v \in Y$ is accessible to b.*

Proof. As b is assigned to an unresolved node, clearly b is also unresolved. But b could not have become critical at any bag below X_q because if it had become critical at some bag, say $X_{q'}$, then after the processing of $X_{q'}$ was completed, it would have been resolved (by invariant 2). Thus b must have accessible to it some node appearing in X_q in the tree decomposition. This implies that some node $v \in X_q$ must be accessible to b (by the separation property of tree decompositions). □

Let $\pi_1 = \pi_{old}$. Consider nodes of Y in an arbitrary order. Let $u \in Y$; we perform the pulling procedure on u from $\overline{Y}$ with π_1 as input; let π_2 be the solution after the pulling procedure. Note that on applying the pulling procedure u cannot exit mid-way because of reaching the capacity limit as $u \in P$, i.e., u is a poor node in the stable solution σ_{in} and is therefore de-capacitated. We take π_2 as π_1 for performing the pulling procedure on the next node of Y. The solution π_2 obtained on performing the pulling procedure on the last node u of Y is taken to be π_{new}. We also open the nodes of Y by setting $y_{\pi_{new}}(u) = 1$ for all $u \in Y$ and color them yellow. All other values of $x_{\pi_{new}}(.,.)$ and $y_{\pi_{new}}(.)$ are retained as in π_{old}. It is easy to see that the solution π_{new} is feasible as the only nodes on which the assignments increase are the nodes of Y and we have fully-opened these nodes. By Claim 4.9, no clients remain assigned to nodes of Z after the pulling. We close all the nodes of Z by setting $y_{\pi_{new}}(v) = 0$ for all $v \in Z$ and color them white. Finally we update the sets `Res` and `Unres` by removing the nodes of Z and Y from `Unres` and adding them to `Res` as they are now resolved. We next claim that the invariants continue to hold after these modifications to the solution.

Claim 4.10. *The stated invariants continue to be maintained in π_{new}.*

Proof. In order to ensure that the first invariant is satisfied, as we have resolved the nodes of Y, we must ensure that all clients assigned to them in π_{old} are resolved as well. The pulling procedure on the nodes of Y ensures that these clients are no longer served by any node in `Unres` in π_{new}.

For invariant 2, consider any client, b, critical at X_q. Similar to the argument for the first invariant, the pulling procedure on the nodes of Y would have ensured

that b is no longer assigned to any node in `Unres` and therefore resolved.

Thus all the invariants continue to be satisfied at the end of the iteration. □

This completes the processing of the current iteration. We take the solution π_{new} as the input π_{old} for the next iteration.

Let σ be the solution π_{new} formed in the last iteration. Note that as every client is critical on some bag and we have processed all the critical bags, all clients are resolved by invariant 2. Thus, by invariant 1, no clients are assigned to any remaining nodes in `Unres`. We close all these nodes by setting $y_\sigma(u) = 0$ for all $u \in$ `Unres` and colour them white. Thus we obtain an integrally open solution. This is taken as the final solution σ_{io} and output by the procedure.

We now analyse the cost. We need to bound the cost of the yellow nodes. Consider an iteration where a bag X_q is critical and we open the nodes of Y. Note that there are at most $t+1$ nodes in Y. The nodes of $Z \cup Y$ are unresolved at the beginning of the iteration. Also, by Claim 4.8, the $\sum_{u \in Z \cup Y} y_{\pi_{old}}(u) \geq 1$. Thus we charge the opened yellow nodes (of X_q) to the set $Z \cup Y$. Also note that no node is charged multiple times as the nodes of $Z \cup Y$ are resolved at the end of the iteration. Thus $\texttt{cost}(\sigma_{io}, P) \leq (t+1) \cdot \texttt{cost}(\sigma_s, P)$. The nodes of R are resolved to begin with and are left untouched. Thus we have that $\text{cost}(\sigma_{io}) \leq \texttt{cost}(\sigma_s, R) + (t+1) \cdot \texttt{cost}(\sigma_s, P)$.

This completes the proof of Lemma 4.3

The above procedure involves examining the nodes in each bag and performing the pulling procedure on the nodes in some bags. Examining the $(t+1)$ nodes in a bag requires time polynomial in the input size and t. Also, the pulling procedure runs in polynomial time. Thus, since the number of bags is bounded by the number of nodes in the graph, the overall time taken in obtaining an integrally open solution is polynomial in the input size and parameter t.

4.3 Algorithm for (undirected) BDBT graphs

In this section, we present an approximation algorithm for the replica placement problem on (undirected) BDBT graphs that uses the algorithm for directed BDBT graphs (recall that in both these variants, clients are not part of the network nodes). Let I be an instance of the replica placement problem on an (undirected) BDBT graph $G = (V, E)$. We create an instance I' of the replica placement problem on a directed BDBT graph as follows. The graph $G' = (V, E')$ is obtained from G by adding two directed edges $e_1 = (u, v)$ and $e_2 = (v, u)$ to E' for every edge $e = (u, v) \in E$ and taking $w(e_1)$=$w(e_2)$=$w(e)$. The capacity W for the nodes of the graph, $r(.)$ and $d_{\max}(.)$ for every client remain the same in the new instance. Moreover, for any client a, $\texttt{att}(a)$ remains the same. Note that the instances I and I' are equivalent in the sense that every solution of one is also a solution of the other.

Let d' and t' denote the degree and tree-width of G' respectively. Observe that $d' = 2d$ and tree-width $t' = t$. We use Theorem 4.1 to find a $2(d' + t' + 2)$ approximate solution for I'. As I and I' are equivalent, this is also a $2(d' + t' + 2)$-approximate solution to I. Since $d' = 2d$ and $t' = t$, this yields a $2(2d + t + 2)$-approximate solution to I. This establishes Theorem 1.5. Since the transformation involved in Theorem 4.1 takes time polynomial in the input size and parameters d and t, the time taken by the transformation in Theorem 1.5 is also polynomial in the size of the input, and parameters t and d.

Chapter 5

Replica Placement on Trees of Bounded Degree Bounded Tree-width Graphs

5.1 Introduction

In this chapter we study the replica placement problem on TBDBT graphs. We first study the problem on the directed version of TBDBT graphs defined as follows.

Definition 5. *Directed Tree of BDBT graphs (directed TBDBT graph). A directed TBDBT graph G is a pair $\langle \{G_j | j \in J\}, \mathcal{T} \rangle$ where $G_1, G_2, \ldots, G_h$ are directed BDBT graphs, $J = [1, h]$ and $\mathcal{T}$ is a (skeletal) tree with the elements of J as nodes and labeled edges. We consider an arbitrary element of J to be the root of $\mathcal{T}$. This imposes ancestor-descendant relationships between the elements of J. A directed TBDBT graph satisfies the following properties:*

- *The vertices of the directed BDBT graphs are disjoint, i.e., $V(G_i) \cap V(G_j) = \phi \ \forall \ i, j \in J, \ i \neq j$.*

- *The vertex set of G is the union of the vertices of directed BDBT graphs, i.e., $V(G) = \cup_{j \in J} V(G_j)$.*

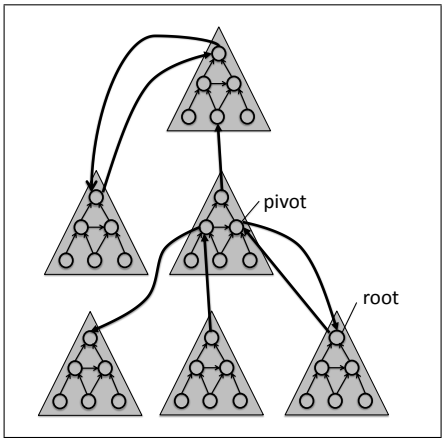


Figure 5.1: Example of directed TBDBT graph. The figure also shows the root of one of the component directed BDBT graphs and the pivot that it connects to in another directed BDBT graph.

- *The edges of all the directed BDBT graphs are contained in G, i.e., $E(G_j) \subseteq E(G)$ for all $j \in J$.*

- *For every edge $e = (i, j)$ in $\mathcal{T}$ where j is ancestor of i, the label $\ell(e)$ on e represents a vertex in $V(G_j)$ and is said to be a pivot node.*

- *For every edge $e = (i, j)$ in $\mathcal{T}$ where j is ancestor of i, there are one or two directed edges between $\texttt{Rt}(\texttt{G}_\texttt{i})$ and $\ell(e))$ in G. The edge(s) may be directed from $\texttt{Rt}(\texttt{G}_\texttt{i})$ to $\ell(e)$ or from $\ell(e)$ to $\texttt{Rt}(\texttt{G}_\texttt{i})$ or both. We refer to $\ell(e)$ as the pivot of $\texttt{Rt}(\texttt{G}_\texttt{i})$ and denote it as pivot$(\texttt{Rt}(\texttt{G}_\texttt{i}))$.*

Let $\texttt{Roots}(\texttt{G})$ *denote the roots of all directed BDBT graphs of G, i.e.,* $\texttt{Roots}(\texttt{G}) = \{\texttt{Rt}(\texttt{G}_\texttt{j}) : \texttt{j} \in \texttt{J}\}$. □

The class of directed TBDBT graphs clearly generalizes directed trees (wherein each component directed BDBT graph consists of a single vertex) (see Figure 5.1). The overall graph has bounded tree-width, but may not have bounded degree. We make the following observation regarding the tree-width of TBDBT graphs.

Claim 5.1. *If the maximum tree-width of any component BDBT graph is t, then the TBDBT graph has tree-width* $\max\{t, 1\}$.

Proof. This follows from the fact that if $e = (f, g)$ is an edge of $\mathcal{T}$ between two BDBT graphs with label $\ell(e) \in V(G_g)$, then we can combine their tree decompositions $(X_1 = \{X_j : j \in J_f\}, T_f = (J_f, K_f))$ and $(X_2 = \{X_j : j \in J_g\}, T_g = (J_g, K_g))$ by introducing a new bag, X_z, between the root bag of T_f (say X_q), and any bag of T_g containing $\ell(e)$ (say $X_{q'}$). X_z contains the nodes $\texttt{Rt}(\texttt{G}_\texttt{f})$ and $\ell(e)$. Add the edges (X_q, X_z) and $(X_z, X_{q'})$. Repeat this for every edge $e \in \mathcal{T}$. Since every newly added bag has 2 nodes, the tree-width of the TBDBT graph is no more than $\max\{t, 1\}$; max is attained at 1 when the BDBT graphs have tree-width 0 (isolated vertices). □

We present a constant factor approximation algorithm for the problem on directed TBDBT graphs. This result appeared in [ACG+14] building upon the work in [ACG+13]. The result is captured by the following theorem and its proof is presented in Section 5.2.

Theorem 5.2. *The replica placement problem on directed TBDBT graphs admits a polynomial time* $136 \cdot (d + \max\{t, 1\} + 4)$ *factor approximation algorithm, where* d *and* t *are respectively the maximum degree bound and the maximum tree-width bound of any component directed BDBT graph of the directed TBDBT graph.*

We then show that a constant factor approximation algorithm can be obtained for the case of (undirected) TBDBT graphs by replacing each edge of the graph with a pair of edges, one in each direction and then invoking Theorem 5.2. We lose an additional factor 2 in the approximation ratio. This yields Theorem 1.6 and the proof is given in Section 5.3.

5.2 Algorithm for directed TBDBT graphs.

In this section we present an approximation algorithm for the problem on directed TBDBT graphs. This algorithm handles clients in the same way as nodes (similar to the algorithm for directed BDBT graphs). Hence we shall work with the definition for the problem with client-nodes, given in Section 4.2.

5.2.1 Outline of the algorithm

In this section we present an overview of the algorithm. The algorithm builds on the procedure for handling directed BDBT graphs presented in Section 4.2. Recall that the procedure for directed BDBT graphs works in three 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, in the third stage, the solution σ_{io} is converted into an integral solution using a cycle cancellation procedure. In the current context of directed TBDBT graphs, it is difficult to obtain a stable solution, because these graphs do not have bounded degree (the pivots may have arbitrary degree). Instead, our algorithm goes via a similar, but a weaker notion of what we call as 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 such solutions can be transformed into integral solutions. Before defining pseudo-stable and hierarchical solutions and, outlining our two transformations, we recall the following terms- (i) recall that a node u is said to be de-capacitated if even on transferring to u the assignments of all clients that can access u, from other partially open nodes, the capacity utilization at u is less than W (ii) recall that a client a is said to be attachable to a node u if it can access u.

Definition 6. *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. *The poor nodes, P, are de-capacitated.*
3. *Every client is either serviced by*
   (a) *only nodes in R.*
   (b) *only nodes in P.*

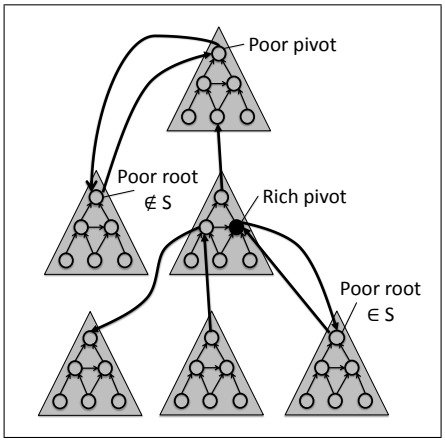


Figure 5.2: The figure shows a root in S and a root in P but not in S.

(c) *nodes in both R and P. Any client a in this category should be attachable to exactly one node in S, where $S \subseteq \texttt{Roots(G)}$ is the set of all poor roots having a rich pivot, i.e., $S = \{u \in \texttt{Roots(G)} \cap \texttt{P} : \texttt{pivot(u)} \in \texttt{R}\}$. 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 graphs. The next (and more sophisticated) stage of the algorithm converts a pseudo-stable solution into a hierarchical solution, defined below.

Definition 7. *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. 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 4.3, since TBDBT graphs have bounded tree-width (recall that Lemma 4.3 only

requires the input graph to have bounded tree-width and not bounded degree).

- 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 graph consists of a skeletal tree $\mathcal{T}$, where each node is in turn a BDBT graph. The skeletal tree structure allows us to obtain the hierarchical solution σ_2'.

We finally merge σ_1' and σ_2' into a single hierarchical solution σ_h serving all the clients.

We next present the two procedures used for transforming an arbitrary LP solution into a pseudo-stable solution and transforming a pseudo-stable solution into a hierarchical solution.

5.2.1.1 Stage 1: Converting an LP solution to a Pseudo-stable Solution

In the first stage, we transform any feasible solution $\sigma_{in} = \langle x_{in}, y_{in} \rangle$ for a TBDBT graph into a pseudo-stable solution $\sigma_{ps} = \langle x_{ps}, y_{ps} \rangle$ as stated in the following lemma.

Lemma 5.3. *Any LP solution σ_{in} can be converted into a pseudo-stable solution σ_{ps} such that $\texttt{cost}(\sigma_{ps}, R) \leq (d+3) \cdot \text{cost}(\sigma_{in})$ and $\texttt{cost}(\sigma_{ps}, P) \leq \text{cost}(\sigma_{in})$ where R and P are respectively the rich and poor nodes of the pseudo-stable solution σ_{ps}.*

Proof Sketch. This transformation is a minor modification of that used in obtaining stable solutions for BDBT graphs. We give a detailed sketch of the proof deferring the formal details to Section 5.2.2. The transformation is divided into the *de-capacitation phase* and *pseudo-stabilization phase*.

The de-capacitation phase is same as in case of BDBT graphs. We process the partially-open and closed nodes iteratively in an arbitrary order. Given a partially-open or closed node u, the procedure transfers assignments from other partially-open nodes to u, as long as the capacity at u is not violated. The transformation is captured in Lemma 4.5.

Pseudo-Stabilization The pseudo-stabilization phase is a variation of the stabilization phase performed in case of BDBT graphs. Since the degree of a node in a TBDBT graph is not bounded, we cannot color all the neighbors of a red node brown, as we did for BDBT graphs. However, we note that it is sufficient to color brown all the in-neighbors and a selected out-neighbor of a red node. Consider any red node u. Let G_j be the BDBT graph to which it belongs. We color brown all of its in-neighbors that are in G_j and not already red. In addition, we also color its nearest non-red out-neighbor brown (even if it is not in G_j). In this process we color brown all the in-neighbors of all the non-pivot nodes. However, some in-neighbors of the pivot-nodes may not get colored brown (note that these are the roots of some BDBT graphs). Next, if u is the root of G_j, we also color its pivot brown (if it is not already red or brown). This process is repeated for all the red nodes. The following proposition follows from the above procedure:

Proposition 5.4. *If an in-neighbor, v, of a red node is neither red nor brown, then $v \in \texttt{Roots}(\texttt{G})$.*

Note that every node has a bounded number of neighbors (and hence in-neighbors) in the BDBT to which it belongs and therefore the number of nodes colored brown is bounded (recall that in general a node may have an arbitrary number of neighbors in the TBDBT).

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. The first two properties are trivially satisfied. For the third property, consider any unsettled 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 in such a case the brown node must have pulled the assignments of a from nodes in P and therefore a would not be assigned to v_2. This also implies that v_1 is red (not brown).

We now show that a is attachable to at least one node in S. Consider the directed path from a to v_1. Let v_3 be the non-red node closest to v_1 on this path and let v_4

be its red out-neighbor in this path. Note that by proposition 5.4, v_3 must either be brown or in $\texttt{Roots}(\texttt{G}) \cap \texttt{P}$. But since a is not attachable to any brown node, v_3 cannot be brown, implying that $v_3 \in \texttt{Roots}(\texttt{G}) \cap \texttt{P}$. Further, observe that v_4 cannot be in the same BDBT as v_3 - otherwise because v_4 is red, v_3 is its in-neighbor and hence we would have colored v_3 brown in the browning process. This implies that v_4 is the pivot of v_3. Moreover, it is rich. Therefore $v_3 \in S$.

Next, we will prove that a cannot be attachable to more than one nodes in S. For this, we observe the following.

Claim 5.5. *Let a be a client attachable to a node $u \in P$. Then, the (shortest) directed path from a to u cannot include a rich pivot.*

The intuition for this claim is as follows (formal proof can be found in Section 5.2.2). If the shortest directed path from a to u includes a brown pivot, then a is attachable to a brown node. On the other hand, in case a directed path from a to u passes through a red pivot, a would be attachable to the nearest non-red out-neighbor of the red node farthest from a on this path. As the nearest non-red out-neighbor of a red node is colored brown, in this case too a would be attachable to a brown node. Both the cases contradict our inference that an unsettled client cannot be attachable to a brown node. Hence the above claim holds.

Finally, we show that a cannot be attachable to two nodes in S. Suppose a is attachable to two nodes, $u_1, u_2 \in S$. Since $S \subseteq P$, $u_1, u_2 \in P$. Also, $pivot(u_1), pivot(u_2) \in R$. By Claim 5.5, the (shortest) directed paths from a to u_1 and u_2 do not include any rich pivot. Hence the path from a to u_1 does not include either $pivot(u_1)$ or $pivot(u_2)$. Similarly, the path from a to u_2 does not include either $pivot(u_1)$ or $pivot(u_2)$. Using the above observation, it can be shown that a cannot be attachable to both u_1 and u_2. A detailed explanation is given in the formal proof. Hence a must be attachable to exactly one node in S implying that the solution is pseudo-stable.

The cost analysis is similar to that in Lemma 4.2. The change in factor arises because earlier, for every red node, we colored at most d neighbors brown, but now

we may color two additional neighbors brown. Firstly, in a TBDBT the nearest non-red out-neighbor of a red node may be outside the BDBT. Secondly, for a red node that is a root, we may color its pivot (not in the BDBT) brown. This establishes Lemma 5.3.

5.2.1.2 Stage 2: Converting a pseudo-stable solution to a hierarchical solution

In the second stage, we transform the pseudo-stable solution σ_{ps} into a hierarchical solution σ_h. Let R and P respectively be the rich and poor nodes of σ_{ps}. Let t be the maximum tree-width of any of the component BDBT graphs of the input TBDBT graph G. Further, let A_1 and A_2 be the sets 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 graph, 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 for every node $u \in V$, we set $x_{\sigma_1}(a, u) = 0$ for all unsettled clients a, and $x_{\sigma_2}(b, u) = 0$ for every settled client b.

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 4.3 (since a TBDBT graph has tree-width $\max\{t, 1\}$). 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 5.6. *Let R and P respectively be the rich and poor nodes of a pseudo-stable solution σ_2 of a TBDBT graph 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. We give a detailed sketch of the proof deferring formal details to Section 5.2.3. Let $\hat{A}$ be the set of unsettled clients. We process the nodes that are special roots for some (unsettled) client iteratively in topological order (bottom-up) of $\mathcal{T}$. Let u be the node currently being processed. Let B_u be the set of clients for which u is a special root and let Z_u be the set of nodes of P to which the clients in B_u are assigned (this may include u itself). We shall argue that all the clients assigned to nodes in Z_u have the same special root, i.e., u. We shall perform the pulling procedure on u from $Z_u \setminus \{u\}$. Note that all the clients in B_u will be reassigned to u and no clients will remain assigned to nodes in $Z_u \setminus \{u\}$. We shall close down all the nodes of Z_u and open u to the extent $\min\{\texttt{cost}(\sigma_2, Z_u), 1\}$; thus the solution remains feasible. We shall account for the cost of opening u by charging the extent to which the nodes of Z_u are open in the solution σ_2. Since the nodes in Z_u have been closed, any node is charged at most once. This completes the processing of u.

We now outline why all the clients assigned to nodes in Z_u have u as their special root. Consider any client b assigned to a node v in Z_u. By definition of Z_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 in P will have the same special root (this follows from Claim 5.5 and the fact that the roots of the BDBT graphs are arranged in a tree-like manner in a TBDBT graph 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 nodes in Z_u have u as the special root. A detailed explanation is provided in the formal proof.

Note that the clients in B_u are not assigned to any poor node passing via $pivot(u)$. Therefore u and B_u will not participate in any further processing (of other nodes that are special roots). Thus, any client in B_u is assigned to at most one special root, viz. u, and hence at most one partially open node (u). Therefore σ_2' is a hierarchical solution. 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 graph as follows. Let Z_1 be the set of nodes of P that are fully open in σ'_1 and Z_2 be the set of 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 all nodes u; (iv) set $x_{\sigma_h}(a,u) = x_{\sigma'_2}(a,u)$ for all unsettled clients a and all nodes u. Note that no node in $Z_1 \cup Z_2$ can exceed its capacity limit, 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 analyses as stated in Lemmas 4.3 and 5.6, we see that

$$\begin{aligned} & \texttt{cost}(\sigma_h, R) &=& \texttt{cost}(\sigma_{ps}, R) \\ \text{and} \quad & \texttt{cost}(\sigma_h, P) &\leq& \texttt{cost}(\sigma'_1, Z_1) + \texttt{cost}(\sigma'_2, Z_2) \\ & &\leq& (\max\{t,1\}+1) \cdot \texttt{cost}(\sigma_1, P) + \texttt{cost}(\sigma_2, P) \\ & &\leq& (\max\{t,1\}+2) \cdot \texttt{cost}(\sigma_{ps}, P). \end{aligned}$$

This establishes the following lemma.

Lemma 5.7. *Let R and P respectively be the rich and poor nodes of a pseudo-stable solution σ_{ps} of a TBDBT graph, G. Let t be the maximum tree-width of any of its component BDBT graphs. 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, $\texttt{cost}(\sigma_h, P) \leq (\max\{t,1\}+2) \cdot \texttt{cost}(\sigma_{ps}, P)$.*

In Section 5.2.3 we show that the transformation involved in Lemma 5.6 takes

time polynomial in the input size and parameter t. Also, as argued in Section 4.2.3, the time taken by the transformation in Lemma 4.3 is polynomial in the input size and parameter t. Thus, the overall time taken by the transformation involved in Lemma 5.7 takes time polynomial in the input size and parameter t.

5.2.1.3 Stage 3: Constant Factor Approximation Algorithm for TB-DBT graphs

We now put together the different transformations and establish a constant factor approximation algorithm for TBDBT graphs. Combining Lemmas 5.3 and 5.7, 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 given by the following lemma.

Lemma 5.8. *Any hierarchical solution σ_h can be converted into an integral solution σ_{out} such that* $\text{cost}(\sigma_{out}) \leq 136 \cdot \text{cost}(\sigma_h)$.

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[SK12] present a 34-approximation. The reduction and proof of the lemma are given in Section 5.2.4.

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

We have thus established Theorem 5.2.

In Section 5.2.2 we show that the transformation involved in obtaining a pseudo-stable solution (Lemma 5.3) takes time polynomial in the number of nodes and parameter d. As argued earlier, the transformation involved in obtaining a hierarchical solution (Lemma 5.7) takes time polynomial in the input size and parameter t. Also, in Section 5.2.4 we show that the transformation in Lemma 5.8 takes time polynomial in the input size. Thus, the overall transformation involved

in Theorem 5.2 takes time polynomial in the input size, and parameters t and d.

5.2.2 Proof of Lemma 5.3

Let σ_{in} be the input LP Solution. The transformation is performed in two phases called the *de-capacitation* phase and the *pseudo-stabilization* phase. Recall that a solution σ is said to be *de-capacitated*, if all the partially-open and the closed nodes in σ are de-capacitated. The de-capacitation phase transforms the input LP solution into a de-capacitated solution. This phase is the same as described for BDBT graphs, captured by Lemma 4.5.

Pseudo-stabilization Phase. We next discuss how to transform a given de-capacitated solution σ' into a pseudo-stable solution σ_{ps}. While performing this transformation, we shall encounter three types of nodes - fully open, partially open and fully closed. Let FO' and PO' be respectively the sets of fully-open and partially open nodes in σ'. The idea is to carefully select a subset of partially open and closed nodes, fully-open them and perform the pulling procedure on these nodes. We maintain sets R and P of *rich* and *poor* nodes. Initialize R to be the set of fully-open nodes in σ' and P to be the set of partially open or closed nodes in σ'. The transformation is captured by the following lemma.

Lemma 5.9. *Any de-capacitated solution $\sigma' = \langle x', y' \rangle$ for a TBDBT graph can be transformed into a pseudo-stable solution $\sigma_{ps} = \langle x_{ps}, y_{ps} \rangle$ such that $\texttt{cost}(\sigma_{ps}, R) \leq (d+3) \cdot \texttt{cost}(\sigma', FO')$ and $\texttt{cost}(\sigma_{ps}, P) \leq \texttt{cost}(\sigma', PO')$ where FO' and PO' are the fully-open and partially-open nodes in σ' respectively and R and P are the rich and poor nodes of the stable solution σ_s.*

Proof. To begin with, we color the nodes red, blue and brown as described below. All the fully-open nodes are colored red and the remaining nodes (partially-open nodes and fully-closed nodes) are colored blue, i.e., the nodes in R are colored red and the nodes in P are colored blue. We next identify the set of nodes, Br, to be colored brown. For this, we first identify for every red node, v, the set of nodes, $Br(v)$, to be colored brown. We then take Br to be the union of the sets $Br(v)$

corresponding to the red nodes. We now describe how to determine $Br(v)$ for a red node v. Let G_j be the BDBT graph to which v belongs. Define $N^-(v)$ to be the set of all non-red in-neighbors of v in G_j, i.e.,

$$N^-(v) = \{u \in P \cap V(G_j) : u \text{ is an in-neighbor of } v\}.$$

Define $N^+(v)$ to be the set of all non-red out-neighbors of v, i.e.,

$$N^+(v) = \{u \in P : u \text{ is an out-neighbor of } v\}.$$

Further, define $NN^+(v)$ to be the nearest non-red out-neighbor of v (ties broken arbitrarily), i.e.,

$$NN^+(v) = argmin_u\{dist(v,u)|\ u \in P \text{ is an out-neighbor of } v\}.$$

Now, if $v \in Roots(G)$ and its pivot is not red, we take $Br(v) = N^-(v) \cup \{NN^+(v)\} \cup \{pivot(v)\}$ else we simply take $Br(v) = N^-(v) \cup \{NN^+(v)\}$. We now take $Br = \cup\{Br(v) : v \text{ is red}\}$. We process the nodes of Br iteratively in an arbitrary order. The processing for a node $u \in Br$ is as follows. Let π_{old} be the solution at the start of the current iteration; we form a new solution π_{new} in the current iteration. We perform the pulling procedure from P onto u with π_{old} as input; let π_{new} be the solution after the pulling procedure. We also open u and set $y_{\pi_{new}}(u) = 1$. All other values of $x_{\pi_{new}}(.,.)$ and $y_{\pi_{new}}(.)$ are retained as in π_{old}. It is easy to see that the solution π_{new} is feasible as the assignments only increase on u but we have fully-opened node u. Finally, we update sets R and P by moving u to R, i.e., $R = R \cup \{u\}$ and $P = P \setminus \{u\}$. We color the node u brown. This completes the processing of the current iteration (for u). We take the solution π_{new} as the input π_{old} for the next iteration. The solution π_{new} at the end of the last iteration is taken as the solution σ_{ps} output by this procedure. The sets R and P obtained at the end are taken as the rich and poor nodes. The set of all nodes in $\texttt{Roots(G)} \cap \texttt{P}$ having a red pivot are taken as the set S output by the procedure. This completes the description of the procedure. We now show that σ_{ps} is a pseudo-stable solution.

Claim 5.10. *The solution σ_{ps} is pseudo-stable.*

Proof. The first condition is trivially satisfied since R consists of red and brown nodes, both of which are fully open. The second condition is ensured by the decapacitation process. To check the third condition, consider an unsettled client a assigned to $v_1 \in R$ and $v_2 \in P$. We first observe that a cannot be attachable to any brown node since in that case, the brown node must have pulled the assignments of a from nodes in P and hence a would not be assigned to v_2. We now need to show that any client assigned to a node in R as well as to a node in P must be attachable to exactly one node in S. This is established by Claims 5.11 and 5.12 below. Hence σ_{ps} is pseudo-stable solution. □

We recall and prove Claim 5.5 required in proving the subsequent claims.

Claim 5.5. Let a be a client attachable to a node $u \in P$. Then, the (shortest) directed path from a to u cannot include a rich pivot.

Proof. Consider a client a attachable to $u \in P$. Further, suppose that the shortest directed path from a to u includes a rich pivot v. Clearly, v is either red or brown. If v is brown, then a is attachable to a brown node. This contradicts the fact that an unsettled client cannot be attachable to a brown node. Now suppose v is red. Consider the red node, say v^* farthest from a on this directed path from a to u. Then, a is attachable to the non-red out-neighbor of v^* lying on this path (which may be u itself). Thus, a must be attachable to the nearest non-red out-neighbor of v^*. But since the nearest non-red out-neighbor of a red node is colored brown, this would imply that a is attachable to a brown node. Therefore in this case also we arrive at a contradiction since an unsettled client cannot be attachable to a brown node. Hence the directed path from a to u does not include any rich pivot. □

Claim 5.11. *Let a be a client assigned to $v_1 \in R$ and $v_2 \in P$; then a is attachable to at least one node in S.*

Proof. First observe that v_1 must be red since a is not attachable to any brown node. Now, consider the directed path from a to v_1. Let v_3 be the node closest

to v_1 on this path such that $color(v_3) \neq red$. Further, let v_4 be its red out-neighbor on this path. Clearly, a is attachable to v_3. We next show that v_3 is in S, i.e., $v_3 \in \texttt{Roots(G)} \cap \texttt{P}$ and it has a rich pivot. Firstly, observe that by Proposition 5.4, either $color(v_3) = brown$ or $v_3 \in Roots(G) \cap P$. But since a is not attachable to any brown node, v_3 cannot be brown, implying that $v_3 \in \texttt{Roots(G)} \cap \texttt{P}$. Also, observe that v_3 and v_4 cannot be in the same BDBT since in such a case, $v_3 \in N^-(v_4) \subseteq Br(v_4)$ and would have therefore been colored brown during the browning procedure. This implies that v_4 is the pivot of v_3. Moreover, since $v_4 \in R$, so $v_3 \in S$. □

Claim 5.12. *Let a be a client assigned to $v_1 \in R$ and $v_2 \in P$; then a is attachable to at most one node in S.*

Proof. We prove the claim by contradiction. Suppose a is attachable to two nodes u_1 and u_2 in S. Since $S \subseteq P$, we have that $u_1 \in P$ and $u_2 \in P$. Also, by the definition of S, $pivot(u_1) \in R$ and $pivot(u_2) \in R$. Now, by Claim 5.5, the directed path from a to u_1 does not include $pivot(u_1)$. Similarly, the directed path from a to u_2 does not include $pivot(u_2)$. Then, since a TBDBT consists of BDBT graphs arranged in a tree-like manner through a skeletal tree, it must be that the directed path from a to one of u_1 and u_2 must include the pivot of the other. Again, by Claim 5.5, the directed path from a to u_1 does not include $pivot(u_2)$. Similarly, the directed path from a to u_2 does not include $pivot(u_1)$. Hence, our assumption is wrong and therefore a must be attachable to at most one node in S. □

We now analyse the cost. There are at most d in-neighbors of a red node within the BDBT graph that are colored brown ; one out-neighbor which is colored brown; and if the red node is a root then there is an additional pivot that is colored brown. Thus the total number of brown nodes is at most $(d+2) \cdot \texttt{cost}(\sigma', FO')$ (where FO' is the set of fully-open nodes in σ'). This implies that $\texttt{cost}(\sigma_{ps}, R) \leq (d+3) \cdot \texttt{cost}(\sigma', FO')$. As the extent of openness of the remaining nodes is untouched, it follows that $\texttt{cost}(\sigma_{ps}, P) \leq \texttt{cost}(\sigma', PO')$ (where PO' is the set of partially-open nodes in σ'). □

Lemma 5.3 follows by combining Lemmas 4.5 and 5.9. Combining the cost analyses of Lemmas 4.5 and 5.9, together with the fact that the newly opened nodes in the de-capacitation phase become rich, we have:
$\texttt{cost}(\sigma_{ps}, R) \leq (d+3) \cdot \texttt{cost}(\sigma', FO') \leq (d+3) \cdot \text{cost}(\sigma_{in})$ and $\texttt{cost}(\sigma_{ps}, P) \leq \texttt{cost}(\sigma', PO') \leq \text{cost}(\sigma_{in})$.

This completes the proof of Lemma 5.3.

The time taken by the de-capacitation procedure is clearly polynomial in the size of the input. The pseudo-stabilization step involves identification of the following for some nodes - in-neighbors within their own BDBT graphs, closest out-neighbor and (possibly) the corresponding pivot. Once identified, the pulling procedure is performed on all these nodes. Any node has at most d in-neighbors within its own BDBT graph and the pulling procedure runs in polynomial time. Thus, the overall time taken in obtaining a pseudo-stable solution is polynomial in the number of nodes and parameter d.

5.2.3 Proof of Lemma 5.6

Let σ_2 be the input pseudo-stable solution. As σ_2 is a *pseudo-stable* solution, each client is attachable to exactly one node in S. We process the nodes of S iteratively in topological order (bottom-up) of $\mathcal{T}$, i.e., we process a node $x \in S$ only after we have processed all nodes $y \in S$ such that if x is the root of BDBT G_i and y is the root of BDBT G_j then i is an ancestor of j in $\mathcal{T}$. Let $u \in S$ be the root being processed in the current iteration and π_{old} be the solution at the beginning of the iteration. Let $\hat{A}$ be the set of unsettled clients. If no unsettled client is attachable to u, we proceed to the next root. Define $B_u = \{a \in \hat{A} : u \text{ is the special root of } a\}$ and $Z_u = \{v \in P : x(a,v) > 0 \text{ for some } a \in B_u\}$. Note that $u \in Z_u$. We perform the *pulling* procedure from $Z_u \setminus \{u\}$ onto u to form a new solution π_{new}. In order to maintain feasibility, we increase the capacity of u as follows: set $y_{\pi_{new}}(u) = min\{1, \texttt{cost}(\sigma_2, Z_u)\}$. We use a special color, say green, to color u. We next show that the nodes in $Z_u \setminus \{u\}$ have no more clients assigned to them, i.e., all the clients assigned to Z_u have the same special root, i.e., u. Consider any

client b assigned to a node v in Z_u. By definition of Z_u, there must be a client, c, assigned to v and having special root as u. The following claim shows that clients attachable to the same node will have the same special root.

Claim 5.13. *Unsettled clients attachable to the same node in P have the same special root.*

Proof. Let a and a' be two unsettled clients attachable to a node $v \in P$. Further, let s and s' be the special roots of a and a' respectively and let G_i and $G_{i'}$ be the BDBT graphs for which s and s' are respectively the roots. Suppose that $s \neq s'$. Since $S \subseteq P$, we have that $s, s' \in P$. Also, by the definition of S, $pivot(s) \in R$ and $pivot(s') \in R$. Now, by Claim 5.5, the paths from a to v and a to s cannot include $pivot(s)$. This implies that both v and a belong to BDBTs that are descendants of i (including possibly i itself) in $\mathcal{T}$. By the same argument, v and a' belong to BDBTs that are descendants of i'. This implies that v belongs to a BDBT that is a descendant of both i and i'. This means that either i is an ancestor of i' or i' is an ancestor of i in $\mathcal{T}$. Without loss of generality, assume that i' is an ancestor of i. Now, since the path from a' to v cannot pass through a rich pivot, it must be that a' also belongs to a BDBT that is a descendant of i. But then the directed path from a' to s' passes through $pivot(s)$ contradicting the fact that this path cannot pass through a rich pivot. Hence it must be that $s = s'$. □

Since c and b are attachable to the same node, v, they must have the same special root; therefore the special root of b must also be u. This implies that all the clients assigned to Z_u have u as the special root. Thus, after the *pulling* procedure, no more clients are assigned to nodes in $Z_u \setminus \{u\}$. Hence we close these nodes by setting $y_{\pi_{new}}(v) = 0 \ \forall v \in Z_u \setminus \{u\}$. Clearly, the LP remains feasible. Let σ_2' be the solution at the end of these iterations, i.e., σ_2' is taken to be the solution π_{new} formed in the last iteration.

We next analyse the cost. Since the nodes in R remain untouched, we only bother about the partially-open nodes. Let $Z_2 \subseteq P$ be the nodes that were colored green (these are the only open nodes other than nodes of R). We charge the extent

to which any green node u of Z_2 is opened to the corresponding set Z_u obtained in the corresponding iteration when u was colored green; note that the sets Z_u are disjoint for different nodes $u \in S$. We have $\texttt{cost}(\sigma_2', Z_2) = \sum_{u \in Z_2} y_{\sigma_2'}(u) \leq \sum_{u \in Z_2} \texttt{cost}(\sigma_2, Z_u) \leq \texttt{cost}(\sigma_2, P)$. The above transformation involves performing the pulling procedure on the roots of some of the component BDBT graphs during a bottom up traversal of the TBDBT graph. Since the pulling procedure runs in polynomial time, the above transformation takes time polynomial in the input size.

5.2.4 Proof of Lemma 5.8

The transformation from the input hierarchical solution σ_h to an integral solution σ_{out} works in three stages. We first convert the solution σ_h into a dual assigned solution σ_{da}, wherein a client is assigned to at most one partially open and one fully open node. Next σ_{da} is converted into an integrally open solution σ_{io} and finally σ_{io} is transformed into the final integral solution σ_{out}.

The hierarchical solution σ_h can be transformed into a dual assigned solution σ_{da} at a factor 2 loss in approximation as follows. Any client $a \in A$ can be serviced by at most one partially-open node and possibly many fully-open nodes in σ_h. We invoke Proposition 2.12 with F as the set of fully open nodes in σ_h and $\mathcal{A}$ as the set of all clients A, to obtain a new solution σ_{da} in which every client is serviced by at most one fully-open node. Clearly, $\text{cost}(\sigma_{da}) \leq 2 \cdot \text{cost}(\sigma_h)$. This establishes Lemma 5.14. The result is formally captured in the following lemma.

Lemma 5.14. *Any hierarchical solution σ_h can be converted into a dual assigned solution σ_{da} such that* $\text{cost}(\sigma_{da}) \leq 2 \cdot \text{cost}(\sigma_h)$.

The dual assigned solution σ_{da} can be transformed into an integrally open solution σ_{io} at a factor 34 loss in approximation by reducing the solution σ_{da} to an LP solution for the capacitated vertex cover problem. Saha and Khuller[SK12] present a 34-approximation LP rounding procedure for the capacitated vertex cover problem. Using their procedure we can get an integrally open solution. This result is

formally captured in the following lemma. The proof is provided in Section 5.2.4.1.

Lemma 5.15. *Any dual assigned solution σ_{da} can be converted into an integrally open solution σ_{io} such that* $\text{cost}(\sigma_{io}) \leq 34 \cdot \text{cost}(\sigma_{da})$.

Finally, we transform the integrally open solution σ_{io} into an integral solution σ_{out} by invoking Lemma 4.4 (as done for BDBT graphs).

This completes the procedure. Using the cost analyses as stated in Lemmas 5.14, 5.15 and 4.4, we see that $\text{cost}(\sigma_{out}) \leq 2 \cdot \text{cost}(\sigma_{io}) \leq 68 \cdot \text{cost}(\sigma_{da}) \leq 136 \cdot \text{cost}(\sigma_h)$. This completes the proof of Lemma 5.8. Clearly, since Proposition 2.12 runs in polynomial time (as argued in Section 2.5), the transformation in Lemma 5.14 takes polynomial time. We show in Section 5.2.4.1, that the transformation in Lemma 5.15 takes time polynomial in the input size. Also, as argued in Section 4.2.1, the transformation in Lemma 4.4 takes polynomial time. Thus, the overall transformation in Lemma 5.8 takes time polynomial in the input size.

5.2.4.1 Proof of Lemma 5.15

The proof goes via the capacitated vertex cover problem. In the capacitated vertex cover problem, we are given a multi-graph $\widehat{G} = (\widehat{V}, \widehat{E})$ and a capacity $w(u)$ for each vertex $u \in V$. A feasible solution is to choose a subset of vertices $U \subseteq V$ and assign each edge $e = (u, v)$ to one of its endpoints such that the following constraints are satisfied: (i) for any edge $e = (u, v)$, the endpoint to which e is assigned must belong to U; (ii) for any node $u \in U$, the number of edges assigned to u is at most $w(u)$. The goal is to minimize the cardinality of the set U.

Consider the natural LP formulation for the problem (similar to that of our LP for the replica placement problem). For each vertex u, we introduce a variable $y(u)$ and for each edge $e = (u, v)$, we introduce two variables $x(e, u)$ and $x(e, v)$.

We write $e \sim u$ to mean that e is incident on u.

$$\begin{aligned} & \textit{Minimize} && \sum_{u \in \widehat{V}} y(u) && \\ \text{s.t.} \quad & \sum_{e \sim u} x(e,u) && \leq \quad y(u) \cdot w(u) && \forall \quad u \in V \\ & x(e,u) + x(e,v) && \geq \quad 1 && \forall \quad e = (u,v) \\ & x(e,u) && \leq \quad y(u) && \forall \quad e \sim u \end{aligned}$$

Saha and Khuller [SK12] provide a procedure for rounding fractional solutions of the above LP.

Lemma 5.16 ([SK12])**.** *There exists a polynomial time procedure that takes as input a multi-graph $\widehat{G}$ and an LP solution σ and outputs an integral solution U along with edge assignments such that $|U| \leq 34 \cdot \text{cost}(\sigma)$.*

Consider the input dual assigned solution σ_{da}. For each node $u \in V$, add a vertex u in $\widehat{V}$ with capacity $w(u) = W$. Consider any client $a \in A$; since σ_{da} is a dual-assigned solution, a is serviced by at most two nodes; let u_a and v_a be the two nodes (if a is assigned to only one node, then choose any arbitrary node to which a is attachable as the second node). [1] Add $r(a)$ parallel edges between the vertices u_a and v_a (called edge copies of a). The above construction yields an instance of the capacitated vertex cover problem. We can construct an LP solution $\widehat{\sigma}$ for the above instance from the solution σ_{da}, in a straightforward manner. For each node $u \in V$, set $y_{\widehat{\sigma}}(u) = y_{\sigma_{da}}(u)$; for each client a and each edge copy e of a, set $x_{\widehat{\sigma}}(e, u_a) = x_{\sigma_{da}}(e, u_a)$ and $x_{\widehat{\sigma}}(e, v_a) = x_{\sigma_{da}}(e, v_a)$. The cost of $\widehat{\sigma}$ is the same as that of σ_{da}. Using Lemma 5.16, we can convert $\widehat{\sigma}$ into an integral solution U, along with an assignment $\widehat{f}$ that assigns each edge to one of its end points. The solution U and $\widehat{f}$ can be readily converted back into an integrally open solution σ_{io} for our problem. Notice that σ_{io} is not an integral solution since the request of a client

[1] In case some client is not attachable to any other node, it has to be fully open and fully serving itself. Also, recall that no other client can be assigned to it. Such a client can therefore be removed from consideration. Thus, we assume that the input does not include any such client.

has been mapped to $r(.)$ parallel edges which might be assigned to different nodes in $\widehat{\sigma}$; thus, a client may be assigned to more than one nodes in σ_{io}. The above reduction clearly takes polynomial time. Also, Lemma 5.16 involves a polynomial time procedure. Therefore, the overall time taken by the procedure in Lemma 5.15 is polynomial in the input size.

5.3 Algorithm for (undirected) TBDBT graphs

In this section we present an approximation algorithm for the replica placement problem on (undirected) TBDBT graphs that uses the algorithm for directed TBDBT graphs (recall that in both these variants, clients are not part of the network nodes).

Let I be an instance of the replica placement problem on an (undirected) TBDBT graph $G = (V, E)$. We create an instance I' of the replica placement problem on a directed BDBT graph as follows. The graph $G' = (V, E')$ is obtained from G by adding two directed edges $e_1 = (u, v)$ and $e_2 = (v, u)$ to E' for every edge $e = (u, v) \in E$ and taking $w(e_1)$=$w(e_2)$=$w(e)$. The capacity W for the nodes of the graph, $r(.)$ and $d_{\max}(.)$ for every client remain the same in the new instance. Moreover, for any client a, $\texttt{att}(a)$ remains the same. Note that the instances I and I' are equivalent in the sense that every solution of one is also a solution of the other. Let d' denote the degree of G', and t' denote the tree-width of any component directed BDBT graph of G'. Observe that $d' = 2d$ and tree-width $t' = t$. We use Theorem 5.2 to find a $136 \cdot (d' + \max\{t', 1\} + 4) \cdot \text{cost}(\sigma_{in})$ approximate solution for I'. As I and I' are equivalent, this is also a solution of cost $136 \cdot (d' + \max\{t', 1\} + 4) \cdot \text{cost}(\sigma_{in})$ to I. Since $d' = 2d$ and $t' = t$, this yields a solution of cost $136 \cdot (2d + \max\{t, 1\} + 4) \cdot \text{cost}(\sigma_{in})$ to I. This establishes Theorem 1.6. Since the transformation involved in Theorem 5.2 takes time polynomial in the input size and parameters d and t, the time taken by the transformation in Theorem 1.6 is also polynomial in the size of the input, and parameters t and d.

Chapter 6

Typed Data Placement

6.1 Introduction

In this chapter we give a 4-approximation algorithm for the typed data placement problem to prove Theorem 1.8. This result appeared in [AGK+14].

We begin with an overview of the algorithm. Recall that the typed data placement problem has a bound on the number of services that can be offered by a server. Also, the cost model does not involve facility opening costs; only the distance travelled by the clients contributes to the cost. The main challenge in this problem lies in dealing with clients having both types of demands with the co-located facility having storage capacity 1. We refer to such clients as dual-clients. The key idea is to resolve one of the demands of each dual-client by carefully assigning it to some facility (this may involve opening the facility). For this, we determine the average cost per unit demand paid by each client, j, for each object-type, o, in the relaxed LP solution; let this be denoted by C_j^o. Thereafter we discard the LP solution, i.e., we do not use the assignments made in the LP solution and build a new solution. The algorithm proceeds iteratively, opening some facilities and assigning some clients in each iteration. The average cost paid by the group of clients assigned in an iteration is no more than 4 times the cost paid by them in the LP solution. This is shown by amortized analysis. An iteration begins by constructing a ball around each client for each object-type with radius proportional

to the corresponding average cost. Observe that a dual-client will have two balls around it. We define the *outer-ball* of a dual-client j (denoted by OB(j)) as the larger of its balls. We then process the dual-clients iteratively in increasing order of the size of their outer-balls. Let j be the client with the smallest outer ball. In the current iteration, we group demands to yield a simpler structure – no other outer balls overlap with the outer ball of j (this is similar to the consolidation due to Shmoys et al.[STA97] and Baev et al.[BRS08] in the sense that the structure is simplified by removal of overlapping balls). We then assign the demand associated with the outer-ball of j and the demands grouped with it to appropriate facilities. Note that our algorithm performs grouping in iterations.

6.2 Algorithm

The algorithm is formally described in Figure 6.2 and the various procedures used are described in Figure 6.3. The input is an instance, I, specifying the clients, D, along with their demands and the facilities, F, along with their storage capacities, such that $D \subseteq F$. We also refer to a client as a 'facility' depending on the context.

The algorithm starts by solving the relaxed LP on the instance I; let $< x, y >$ denote the optimal LP solution. For every client and object-type pair, we define $C_j^o = \sum_{i \in F} c_{ij} \cdot x_{ij}^o$; this is the average cost per unit demand paid by the client j for the object type o in the LP solution. We construct a ball around each client j, for each object-type, o, with radius $\frac{4}{3}C_j^o$. Observe that at least one-fourth of the demand must be satisfied from within this ball.

We define some terms that will be useful in describing the algorithm further. We call a facility to be *open* if it is open for o_1 or o_2. We call a facility to be *free* if it is neither open nor has any demand associated with it. A facility is said to be a *dual-client* if it has demand for both types of objects and a *single-client* if it has demand for only one type of object. A dual-client is categorized as o_1o_2 *dual-client* if $C_j^{o_1} < C_j^{o_2}$, and as o_2o_1 *dual-client* otherwise. A client is said to be *independent* if it does not lie in the outer ball of any dual-client and has demand

for only one object-type, and *dependent* if it lies within the outer ball of a dual-client and has demand for only one object-type. A single-client is referred to as an o_1 client if it has an o_1 demand and an o_2 client otherwise. We treat co-located facilities as follows. If there is a single demand at that location, we categorize one of the facilities as a single-client with the associated demand, and the other as a free facility. If the location has both types of demands, then we treat each of the facilities as a single-client with one of the two demands associated to it.

We next construct an integral solution $< \hat{x}, \hat{y} >$. We first open all the independent-single clients for their own type, i.e., the type for which they have a positive demand; they are then assigned to themselves. These demands can now be removed from consideration as they have been assigned. Therefore, we update the set D to include only the unassigned demands. The algorithm then iteratively processes the dual-clients.

The input to an iteration is (i) a set of unassigned clients, D (along with their demands), (ii) a set of facilities F (where $D \subseteq F$), (iii) two disjoint subsets F_1, $F_2 \subseteq F$ that are open for o_1 and o_2 respectively, (iv) distances c_{ij} $\forall$ $i \in F$, $j \in D$ (satisfying the metric property), and (v) for every client j having demand for object type o, C_j^o, the average cost paid by this demand in the LP solution. Moreover, the input satisfies the additional property that for any dual-client, j, there is no open facility within the outer ball of j. Since a subset of the unassigned demands gets assigned in every iteration, we encounter a reduced problem at the end of every iteration - this forms the input to the next iteration. Clearly, there is no open facility within the outer-ball of any dual-client at the start of the first iteration. We maintain the following invariant to ensure this condition holds at the start of every iteration:

(Inv) For any dual-client, j, there is no open facility within the outer ball of j.

An iteration starts by partitioning the facilities into the following sets: (i) open facilities (F_{open}), (ii) free facilities (F_{free}), (iii) dual-clients (D_{dual}), (iv) independent-single-clients (D_{ind}); and (v) dependent-single-clients (D_{dep}). These sets dynamically change as the algorithm proceeds. We then determine the dual-

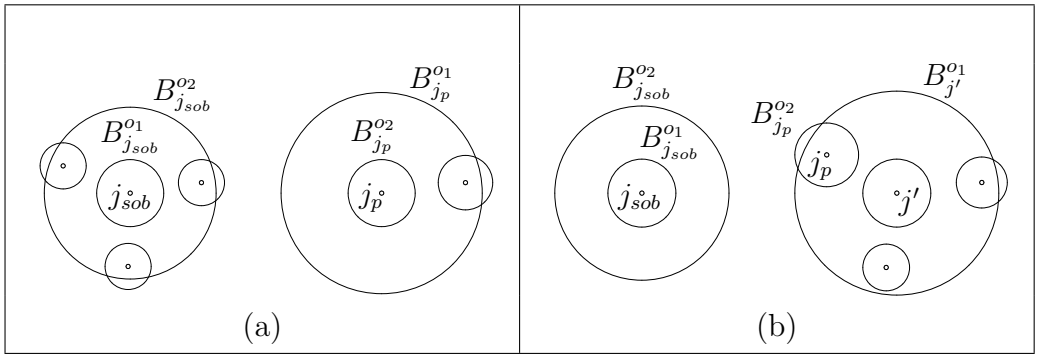


Figure 6.1: Pair-dissolve Scenarios. Here B^o_j denotes j's ball for type o. (a) j_p is o_2o_1 dual-client (b) j_p is dependent-single.

client having the smallest outer ball; we denote this client by j_{sob} and its outer ball by SOB. Let R be the radius of the SOB. Without loss of generality, assume that the SOB is of type o_2 (i.e. j_{sob} is of o_1o_2 category). We say that a ball is *dissolved* if the demand corresponding to it is assigned to some facility. We now try to *dissolve* the SOB by applying one of the following procedures: *Simple-Dissolve*, *Pair-Dissolve* or *Group-Dissolve* in that order.

We first examine if *Simple-Dissolve* is applicable. If there is a 'close-by' facility j_s that is open for o_2, we assign the o_2 demand of j_{sob} to j_s and thus dissolve the SOB. Here, 'close-by' means at a distance of $\leq 2R$ from j_{sob}. Observe that in this case invariant (Inv) is trivially satisfied since no new facility is opened.

If *Simple Dissolve* is not possible, we check for *Pair-Dissolve*. If there exists a client (say j_p), that is either a 'close-by' dependent-single-client of type o_2 or a 'close-by' o_2o_1 dual-client, we open j_{sob} for o_1 and j_p for o_2 (this can be done as (Inv) implies that j_{sob} and j_p are not already open). The two cases are shown in Figure 6.1. We then assign j_p to itself for its o_2 demand and j_{sob} to itself for its o_1 demand. Also, we assign j_{sob} to j_p for its o_2 demand. Now, we assign the outer balls of all the dual-clients containing j_{sob} or j_p to either j_p or j_{sob} appropriately. Invariant (Inv) is satisfied at the end of this case as a facility inside the outer-ball of a dual-client may be opened in (S3) or (S4) but this outer-ball gets dissolved in (S5).

If *Simple* and *Pair Dissolve* are not possible, we perform *Group-Dissolve*. We

The 4-approximation algorithm

Solve relaxed LP; let solution be $< x, y >$

Maintain assignments and open facilities of our algorithm in $< \hat{x}, \hat{y} >$
Open indep-clients for own type and assign to self ...($S_{1.1}$)
While there is any dual-client remaining
  Let j_{sob} be the dual-client having the smallest outer ball
  Assume that j_{sob} is of o_1o_2 category (o_2o_1 case is similar)
  Let $R = \frac{4}{3}C^{o_2}_{j_{sob}}$

  If $\exists$ a facility j_s open for o_2 within a distance of $2R$ from j_{sob}
    Invoke *Simple-Dissolve*
  else if $\exists$ a client j_p within a distance of $2R$ from j_{sob} such that
      j_p is either an o_2 dependent-single or an o_2o_1 dual-client
    Invoke *Pair-Dissolve*
  else **Invoke** *Group-Dissolve*
  Open new indep-clients for own type and assign to self ...($S_{1.2}$)
End-While
Return $< \hat{x}, \hat{y} >$ – new solution: assignments & facilities open

Figure 6.2: The 4-approximation algorithm

begin by '*grouping*' onto the SOB, the outer balls of all dual-clients that *overlap* with it (i.e., $\exists$ a common facility in both). *Grouping* means the following: the demand corresponding to the outer-ball of the client is temporarily removed. We ensure that at the end of the iteration, these demands are introduced back and assigned to some facility not very far from j_{sob}. As these facilities are yet to be identified, we keep track of the *grouped* demands in the set $Commit$. Now, no dual-clients overlap with the SOB. We next *dissolve* the SOB.

We first consider the simple case wherein there is a facility $j' \neq j_{sob}$ within the SOB that has no demand for o_1 (i.e., a dependent-single of type o_2 or a free

Simple-Dissolve, Pair-Dissolve and Group-Dissolve procedures

Procedure *Simple-Dissolve*

Assign j_{sob} to j_s for its o_2 demand ...(S_2)

Procedure *Pair-Dissolve*

Open j_{sob} for o_1 and assign it to itself for o_1 ...(S_3)

Open j_p for o_2 and assign to it both j_p and j_{sob} for o_2 ...(S_4)

For each dual-client, j_d, whose outer ball contains j_p or j_{sob}

If it is of o_1o_2 category, set $\hat{x}^{o_2}_{j_d j_p} = 1$, else set $\hat{x}^{o_1}_{j_d j_{sob}} = 1$ ...(S_5)

Procedure *Group-Dissolve*

Let *Commit* = {dual-clients j_c: OB(j_c) and OB(j_{sob}) overlap }

'*Group*' the outer-ball of each client in *Commit* with the SOB

If OB(j_{sob}) contains a free facility or an o_2 client (say j')

Open j' for o_2 and assign j_{sob} to it for o_2 ...(S_6)

If j' is an o_2 client, then assign it to itself for o_2 ...(S_7)

Open j_{sob} for o_1 and assign it to itself for o_1 ...(S_8)

For each client, j_c in *Commit*

If it is of o_1o_2 category, set $\hat{x}^{o_2}_{j_c j'} = 1$, else set $\hat{x}^{o_1}_{j_c j_{sob}} = 1$ ...(S_9)

else

Let $j^* \in$ SOB having minimum demand-weighted service cost

to its nearest-neighbour, $nn(j^*)$

Open j^* for o_2 and assign j_{sob} to it for o_2 ...(S_{10})

Open $nn(j^*)$ for o_1 (if it is not already open)

Assign j^* and $nn(j^*)$ to $nn(j^*)$ for o_1...(S_{11})

For each dual-client, j_c s.t. $j_c \in Commit$ or $nn(j^*) \in$ OB(j_c)

If it is of o_2o_1 category, set $\hat{x}^{o_1}_{j_c nn(j^*)} = 1$, else set $\hat{x}^{o_2}_{j_c j^*} = 1$...($S_{12}$)

Figure 6.3: Simple-dissolve, Pair-dissolve and Group-dissolve procedures

facility). We open j' for o_2 and assign j_{sob} to it. Also, j' is assigned to itself (if it has an o_2 demand). We open j_{sob} for o_1 and assign it to itself. This can be done since both j_{sob} and j' are not already open (by (Inv)). We then assign all the demands in *Commit* to j_{sob} or j' appropriately. This takes care of assigning the 'grouped' demands. This completes the simple case. Note that the invariant (Inv) is satisfied at the end of this case as a facility inside the outer-ball of a dual-client may be opened in (S6) or (S8) but this outer-ball gets dissolved in (S9).

We now consider the more involved cases wherein either all dependent-single-clients in SOB are of type o_1 (Figure 6.4a) or there is no dependent-single-client in SOB (Figure 6.4b). Let j^* be the o_1 client in the SOB whose demand-weighted service cost, when served by its nearest neighbor (denoted by $nn(j^*)$), is the minimum (j^* could be a dependent-single in SOB or j_{sob} itself). We open j^* for o_2 and assign the o_2 demand of j_{sob} to it. This can be done because j^* is not already open (by (Inv)). We open $nn(j^*)$ for o_1 if it is not already open (we shall shortly argue that $nn(j^*)$ cannot be an o_2 client and moreover, if it is already open, it must be for o_1). We then assign j^* and $nn(j^*)$ to $nn(j^*)$ for their o_1 demands. Note that all the other o_1 clients in the SOB become independent; these will be opened for o_1 at the end of the current iteration. We will later bound the cost paid by j^* for its o_1 demand by amortizing it over the cost paid by the other o_1 clients in the SOB. Since j^* and $nn(j^*)$ have been opened for different types, we dissolve the outer balls that contain j^* or $nn(j^*)$ to one of them appropriately. Finally, we assign all the demands in *Commit* to j^* or $nn(j^*)$ appropriately. This takes care of assigning the 'grouped' demands. Note that invariant (Inv) continues to hold as a facility inside the outer-ball of a dual-client may be opened in (S10) or (S11) but this outer-ball gets dissolved in (S12).

We still need to prove the following for $nn(j^*)$ above.

Claim 6.1. *At S_{10}, $nn(j^*)$ is not already open for o_2.*

Proof. It can be argued that at step S_{10}, $c_{j_{sob}nn(j^*)} \leq 2R$ (cf. Proposition 6.3). Thus, if $nn(j^*)$ was already open for o_2 before the current iteration, the SOB

would have dissolved in step S_2 (*simple-dissolve*) of this iteration. On the other hand, if $nn(j^*)$ was opened for o_2 in this iteration, it must have been done so in step S_3 (*pair-dissolve*); but in this case the SOB itself would have dissolved and therefore we would not have reached step S_{10}. □

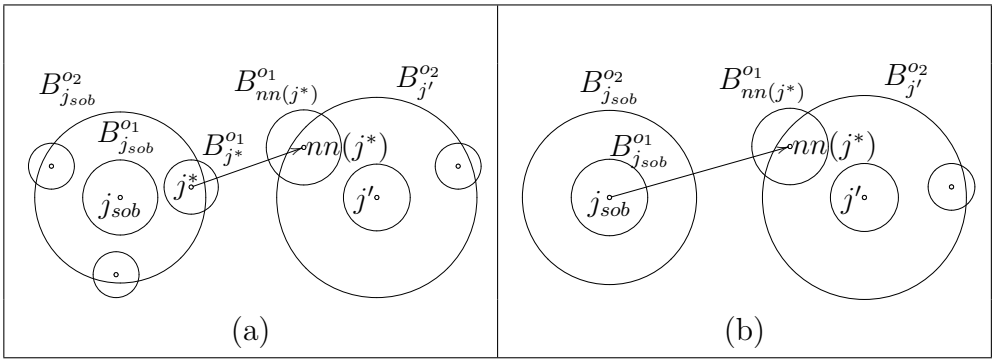


Figure 6.4: Some involved Group-Dissolve Scenarios. Here B^o_j denotes j's ball for type o. (a) All dependent-single-clients in SOB are of type o_1 and $nn(j^*) \notin$ SOB (b) There is no dependent-single-client in SOB.

At the end of each iteration, we resolve all the newly formed independent-single-clients (dual-clients whose outer-balls have been dissolved) by opening them for their own types and assigning their demands to themselves. Clearly, (Inv) continues to hold after $S_{1.2}$. We can now remove the demands that are already assigned – this yields the input for the next iteration.

Finally, the algorithm returns the newly formed solution. This completes the description of the algorithm. Clearly the solution is integral. It is easy to see that the solution is feasible as independent clients are all assigned and outer-balls of dual-clients are eliminated through *simple-dissolve*, *pair-dissolve* or *group-dissolve* thereby making the dependent-singles independent, which are subsequently assigned. Hence, at the end of the algorithm, all demands are assigned. This yields the following result.

Lemma 6.2. $< \hat{x}, \hat{y} >$ *is an integral feasible solution.*

6.3 Analysis

We now show that the algorithm presented in Figure 6.2 is a 4-factor approximation algorithm. The following results will be useful in our analysis.

Proposition 6.3. *At step S_{10}, if there exists some client j other than j_{sob} in the SOB, then $c_{j^*nn(j^*)} \leq R$ else $c_{j^*nn(j^*)} \leq 2R$. In either case, $c_{j_{sob}nn(j^*)} \leq 2R$.*

Proof. The claim is trivial when there exists some client j ($\neq j_{sob}$) in the SOB. Else, the claim follows from the fact that more than 1/2 of both o_1 and o_2 type of demands of j_{sob} must lie within a distance of $2C^{o_2}_{j_{sob}} \leq 2R$. □

Lemma 6.4. *Consider a particular iteration of the algorithm. Let $\mathcal{A}sg$ be the set of demands ($< client, object >$ pairs) that are assigned to facilities in this iteration. Then, the cost paid by the demands in $\mathcal{A}sg$ is no more than 4 times the cost paid by these demands in the LP solution, i.e., $\sum_{<j,o>\in\mathcal{A}sg} d_j \cdot c_{\sigma(j,o)j} \leq 4 \cdot \sum_{<j,o>\in\mathcal{A}sg} d_j \cdot C^o_j$ where $\sigma(j,o)$ denotes the facility in $\mathcal{A}sg$ to which j's demand for o is assigned.*

Proof. We prove the claim by considering the cases under which the demands are added to the set $\mathcal{A}sg$.

The claim is trivial when a client is assigned to itself in $S_{1.1}$. For S_2 in *Simple-Dissolve*, the claim follows as $c_{j_{sob}j_s} \leq 2R$ and $R = (4/3) \cdot C^o_{j_{sob}}$.

We next focus on the assignments in *Pair-Dissolve*. The claim is trivial for S_3. For S_4, it follows as $c_{j_{sob}j_p} \leq 2R$ and $R = (4/3) \cdot C^o_{j_{sob}}$. In S_5, since j_p or j_{sob} (as the case may be) belongs to the outer ball of j_d, the assignment costs $\leq 2R + R_d \leq 3R_d$, where R_d is the radius of the outer ball of j_d and $R \leq R_d$ as R corresponds to the SOB.

We now focus on the assignments in *Group-Dissolve*. We first consider the simple case wherein there is a free facility or an o_2 client in the SOB. For S_6, the claim follows as j' lies in the SOB. S_7 and S_8 trivially satisfy the claim. For S_9, $c_{j'j_c} \leq c_{j_{sob}j_c} + R \leq R_c + R + R \leq 3R_c$, where R_c is the radius of the outer ball of j_c.

Lastly, we focus on the case wherein dependent-single-clients in the SOB (if any) are only of o_1 type. Let X be the set of all the o_1 demands in the SOB. For S_{10}, the claim follows as j^* lies in the SOB.

Now consider S_{11}, where j^* is assigned to $nn(j^*)$ for o_1. Since the SOB is resolved in the current iteration and all the clients inside it become independent and are subsequently resolved in step $S_{1.2}$ of this iteration, it follows that $X \subseteq \mathcal{A}sg$. The following claim shows that X satisfies the inequality of this lemma.

Claim 6.5. *The set X of all the o_1 demands in the SOB satisfies:* $\sum_{<j,o>\in X} d_j \cdot c_{\sigma(j,o)j} \leq 4 \cdot \sum_{<j,o>\in X} d_j \cdot C_j^o$

Proof. Since $\sum_{<j,o_1>\in X} y_j^{o_2} \geq 1/4$, at least $1/4$ fraction of the o_1 demands in X are not being served locally (i.e. from their co-located facility). Note that these demands will have to travel at least to their closest facilities (other than themselves) to get served. Since, j^* pays the minimum cost in getting services from its closest facility, this implies that the cost paid by the LP solution for the demands in X must be at least $(1/4) \cdot (d_{j^*} \cdot c_{j^* nn(j^*)})$. In our integral solution, $\sigma(j^*, o_1) = nn(j^*)$. Moreover, all the o_1 demands in X other than j^* are assigned to themselves in step $S_{1.2}$; therefore $\sigma(j, o_1) = j$ for all $j \in X$, $j \neq j^*$ and hence they pay a cost of 0. Thus, the set X satisfies the claim. □

Next, consider the assignments in S_{12}. If j_{sob} lies in the intersection of the outer balls of j_{sob} and j_c, then since $nn(j^*)$ and j^* lie within a distance of $2R$ from j_{sob} by Proposition 6.3, the cost of assignment is $\leq R_c + 2R \leq 3R_c$, where R_c is the radius of the outer ball of j_c and $R \leq R_c$ as R corresponds to dual-client with the smallest outer ball. If j_{sob} does not lie in the intersection of the outer balls of j_{sob} and j_c then there exists a client j' in the SOB having demand for o_1. Thus by Proposition 6.3, $c_{j^*,nn(j^*)} \leq R$ and hence the cost of assignment is again $\leq R_c + R + R \leq 3R_c$. For all the remaining clients in $\mathcal{A}sg$ ($\notin X$) assigned to themselves in $S_{1.2}$, the claim follows trivially. This completes the proof of the lemma. □

Theorem 1.8 is implied by the following: (i) observation that for all the demands allocated in step $S_{1.1}$, $c_{\sigma(j,o)j} = 0$; (ii) applying Lemma 6.4 over all the iterations and summing up the inequalities; and (iii) correctness follows from Lemma 6.2. It is easy to see from Figures 6.2 and 6.3 that the algorithm runs in polynomial time.

Chapter 7

Conclusion

In this work we studied two types of data placement problems - Replica Placement and *typed* Data Placement.

We began by studying the replica placement problem on tree graphs with unit edge lengths and presented a constant factor approximation algorithm. We next considered generalizations of this basic variant along two dimensions - considering more general graphs and allowing for arbitrary lengths on the edges. We first studied a variant allowing for more general graphs and showed that the replica placement problem with hop counts on bounded tree-width graphs admits an $O(t)$-approximation algorithm where t is the tree-width of the input graph. We next considered variants of the replica placement problem allowing for arbitrary edge lengths. In this direction we first studied graphs having bounded degree and bounded tree-width (BDBT graphs); we gave an $O(d + t)$-approximation algorithm for these graphs where d and t are respectively the degree and tree-width of the graph. Thereafter, in order to generalize the above result to include trees, we studied a more general class of graphs called Tree of BDBT graphs (TBDBT graphs) and presented $O(d+t)$-approximation algorithm for these graphs where d and t are respectively the degree and tree-width of any component BDBT graph. The following problems/questions remain open in the context of replica placement problem.

- replica placement on bounded tree-width graphs having arbitrary edge-lengths
- determining the lower bound on approximation ratio for replica placement on graphs with bounded tree-width having unit edge-lengths

Regarding the *typed* data placement problem, we studied the variant with two object types and no facility opening costs; we presented an 4-approximation algorithm for this problem. It remains open to find an algorithm with approximation factor smaller than 10 for the variant with facility opening costs.

Bibliography

[AAG+10] D. Applegate, A. Archer, V. Gopalakrishnan, S. Lee, and K. Ramakrishnan. Optimal content placement for a large-scale vod system. In *The Conference on emerging Networking EXperiments and Technologies*, page 4, 2010.

[ACG+13] S. Arora, V. T. Chakaravarthy, N. Gupta, K. Mukherjee, S. Roy, and Y. Sabharwal. Replica placement via capacitated vertex cover. In *Foundations of Software Technology and Theoretical Computer Science*, pages 263–274, 2013.

[ACG+14] S. Arora, V. T. Chakaravarthy, K. Gupta, N. Gupta, and Y. Sabharwal. Replica placement on directed acyclic graphs. In *Foundations of Software Technology and Theoretical Computer Science*, pages 213–225, 2014.

[AGK+14] S. Arora, N. Gupta, S. Khuller, Y. Sabharwal, and S. Singhal. Facility location with red-blue demands. *Operations Research Letters*, 42(6-7):462–465, 2014.

[BA10] J. Byrka and K. Aardal. An optimal bifactor approximation algorithm for metric uncapacitated facility location problem. *Society for Industrial and Applied Mathematics Journal of Computing*, 39:2212–2231, 2010.

[BLRG12] A. Benoit, H. Larchevêque, and P. Renaud-Goud. Optimal algorithms and approximation algorithms for replica placement with distance con-

straints in tree networks. In *IEEE International Parallel and Distributed Processing Symposium*, pages 1022–1033, 2012.

[BR01] I. D. Baev and R. Rajaraman. Approximation algorithms for data placement in arbitrary networks. In *Symposium On Discrete Algorithms*, pages 661–670, 2001.

[BRS08] I. D. Baev, R. Rajaraman, and C. Swamy. Approximation algorithms for data placement problems. *Society for Industrial and Applied Mathematics Journal of Computing*, 38(4):1411–1429, 2008.

[BRSR08] A. Benoit, V. Rehn-Sonigo, and Y. Robert. Replica placement and access policies in tree networks. *IEEE Transactions on Parallel and Distributed Systems*, 19:1614–1627, 2008.

[CGTS99] M. Charikar, S. Guha, E. Tardos, and D. B. Shmoys. A constant-factor approximation algorithm for the k-median problem. In *Symposium on Theory Of Computing*, pages 1–10, 1999.

[CGW14] W. Cheung, M. Goemans, and S. Wong. Improved algorithms for vertex cover with hard capacities on multigraphs and hypergraphs. In *Symposium On Discrete Algorithms*, pages 1714–1726, 2014.

[CKS02] I. Cidon, S. Kutten, and R. Soffer. Optimal allocation of electronic content. *Computer Networks*, 40:205–218, 2002.

[CN06] J. Chuzhoy and J. Naor. Covering problems with hard capacities. *Society for Industrial and Applied Mathematics Journal on Computing*, 36(2):498–515, 2006.

[DJGT99] R. Diestel, T. R. Jensen, K. Y. Gorbunov, and C. Thomassen. Highly connected sets and the excluded grid theorem. *Journal of Combinatorial Theory, Series B*, 75(1):61–73, 1999.

[GHK+06] R. Gandhi, E. Halperin, S. Khuller, G. Kortsarz, and A. Srinivasan. An improved approximation algorithm for vertex cover with hard capacities. *Journal of Computer and System Sciences*, 72(1):16–33, 2006.

[GHKO03] S. Guha, R. Hassin, S. Khuller, and E. Or. Capacitated vertex covering. *Journal of Algorithms*, 48(1):257–270, 2003.

[GK99] S. Guha and S. Khuller. Greedy strikes back: Improved facility location algorithms. *Journal of Algorithms*, 31(1):228–248, 1999.

[HKK10] M. Hajiaghayi, R. Khandekar, and G. Kortsarz. Budgeted red-blue median and its generalizations. In *European Symposium on Algorithms*, pages 314–325, 2010.

[JMM+03] K. Jain, M. Mahdian, E. Markakis, A. Saberi, and V. V. Vazirani. Greedy facility location algorithms analyzed using dual fitting with factor-revealing lp. *Journal of the ACM*, 50(6):795–824, 2003.

[JMS02] K. Jain, M. Mahdian, and A. Saberi. A new greedy approach for facility location problems. In *Symposium on Theory Of Computing*, pages 731–740, 2002.

[KDW01] K. Kalpakis, K. Dasgupta, and O. Wolfson. Optimal placement of replicas in trees with read, write, and storage costs. *IEEE Transactions on Parallel and Distributed Systems*, 12:628–637, 2001.

[KL09] M.J. Kao and C.S. Liao. Capacitated domination problem. *Algorithmica*, 4835:1–27, 2009.

[Li11] S. Li. A 1.488 approximation algorithm for the uncapacitated facility location problem. In *International Colloquium on Automata, Languages and Programming*, pages 77–88, 2011.

[LLW06] Y. Lin, P. Liu, and J.J. Wu. Optimal placement of replicas in data grid environments with locality assurance. In *International Conference on Parallel and Distributed Systems*, pages 465–474, 2006.

[LSS12] R. Levi, D. Shmoys, and C. Swamy. LP-based approximation algorithms for capacitated facility location. *Mathematical Programming*, 131(1-2):365–379, 2012.

[MYZ06] M. Mahdian, Y. Ye, and J. Zhang. Approximation algorithms for metric facility location problems. *Society for Industrial and Applied Mathematics Journal of Computing*, 36(2):411–432, 2006.

[RKN+11] R.Krishnaswamy, A. Kumar, V. Nagarajan, Y. Sabharwal, and B. Saha. The matroid median problem. In *Symposium On Discrete Algorithms*, pages 1117–1130, 2011.

[SK12] B. Saha and S. Khuller. Set cover revisited: Hypergraph cover with hard capacities. In *International Colloquium on Automata, Languages and Programming*, pages 762–773, 2012.

[STA97] D. B. Shmoys, É. Tardos, and K. Aardal. Approximation algorithms for facility location problems. In *Symposium on Theory Of Computing*, pages 265–274, 1997.

[Svi02] M. Sviridenko. An improved approximation algorithm for the metric uncapacitated facility location problem. In *Integer Programming and Combinatorial Optimization*, pages 240–257, 2002.

[TX05] X. Tang and J. Xu. QoS-aware replica placement for content distribution. *IEEE Transactions on Parallel and Distributed Systems*, 16:921–932, 2005.

[WLL08] J. Wu, Y. Lin, and P. Liu. Optimal replica placement in hierarchical data grids with locality assurance. *Journal of Parallel and Distributed Computing*, 68:1517–1538, 2008.

[Wol82] L. Wolsey. An analysis of the greedy algorithm for the submodular set covering problem. *Combinatorica*, 2(4):385–393, 1982.

List of Publications

- Sonika Arora, Venkatesan T. Chakaravarthy, Neelima Gupta, Koyel Mukherjee, Yogish Sabharwal, *Replica Placement via capacitated vertex cover*, in 33rd International Conference on Foundations of Software Technology and Theoretical Computer Science 2013, December 12-14, 2013, Guwahati, India, 2013, pp. 263-274.

- Sonika Arora, Venkatesan T. Chakaravarthy, Kanika Gupta, Neelima Gupta, Yogish Sabharwal, *Replica Placement on directed acyclic graphs*, in 34th International Conference on Foundations of Software Technology and Theoretical Computer Science 2014, December 15-17, 2014, New Delhi, India, 2014, pp. 213-225.

- Sonika Arora, Neelima Gupta, Samir Khuller, Yogish Sabharwal, Swati Singhal, *Facility Location with red-blue demands*, Elsevier Journal of Operations Research Letters, vol. 42, no. 6-7, pp. 462-465, 2014.