Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed
Computing

Region Based File Sharing

Bachelor's Thesis

Adrian Jenny

adjenny@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Robin Fritsch, Béni Egressy
Prof. Dr. Roger Wattenhofer

August 18, 2021

Acknowledgements

I thank my thesis supervisors Béni Egressy and Robin Fritsch. They showed admirable patience and support when I was given the chance to propose my own idea for a thesis and it came to formulating the core question as a concrete topic. I also very much enjoyed the conducive atmosphere in our weekly meetings where we were discussing new ideas and I was given directions as to what to focus on.

Further, I thank Professor Roger Wattenhofer and the whole of the Distributed Computing Group for allowing and encouraging students to come up with their own ideas as thesis topics and supporting them on their way.

Abstract

Large-scale file-sharing infrastructures tend to employ some sort of file replication mechanism between nodes in the network. Choosing the correct network nodes that should hold a copy of a file and thereby serve as a replica site is a challenging task. Many different strategies have been proposed where only the optimization of read accesses was considered. In this thesis, we model the underlying network as a graph and try to choose the best nodes as replica sites such as to both optimize read and write operations. To that end, we propose a method for the spreading of file updates in the network and thereupon investigate different strategies for the selection of the correct nodes. We will see that the algorithms based on these strategies all suffer from not being able to account for all effects that might follow from having made some wrong decision earlier on in their execution.

Contents

Chapter 1

Introduction

We live in a world with an ever-growing need for convenient and fast solutions for data exchange or, in general, new ways for efficient sharing of digital resources. Different scenarios require the consideration of different aspects in the creation of an infrastructure that aims to offer said sharing capabilities. Large scale scientific computing, a company's internal file sharing system, highly-available and low-latency web content or the access to services that run the *Internet of Things* are all examples of scenarios that pose a different set of challenges and requirements to the underlying infrastructure that enables these services.

For large-scale file-sharing services, replication is a commonly used technique to offer availability as well as locality. In this thesis, we strive to approach the problem of optimal replica placement, i.e. choosing the correct nodes in a network to host certain files, from a new angle where the costs incurred by file updates between replica sites are also considered. To that end, we go on in chapter 2 to introduce a model and give some definitions on which the later work will be based. In chapter 3 we will discuss different ways as to how a file update might spread in a network such as to minimize some given notion of costs. The bulk of this work is presented in chapter 4 where we will introduce different algorithms These algorithms aim to approximate an optimal choice of nodes used to serve as replica sites for a specific file. These algorithms all try to choose nodes such that the induced additional costs for the spreading of updates on write operations do not outstrip the benefits gained through offering lower latency access. We will illustrate the challenges these algorithms face through specific examples and show how this might adversely affect their result. Finally, in chapter 5 we will draw our conclusions and point out some limitations to our chosen model. Based on these limitations, we give recommendations as to how these might be addressed through future investigations.

Chapter 2

Background

2.1 General Problem Statement

The fast delivery of digital resources relies on an underlying infrastructure that is both optimized and versatile. Depending on the specific needs and implementation, some of the following considerations are being taken into account when designing such an infrastructure:

- read access latency
- read-only vs. read-write datasets
- usage of storage space
- usage of network bandwidth
- processing overhead
- resiliency to outages
- data consistency
- dynamic vs. static environment
- load balancing
- ...

Combining several desired properties for an optimized infrastructure design presents a significant challenge since the fulfillment of one property may adversely affect others. While there already exist commercial solutions that each optimize for a subset of the given properties (e.g. Content Distribution Networks [1] for low-latency, read-only web content), there are still other areas that did not yet receive enough attention.

2.2 Related Work

There exist many different strategies for sharing mutable resources in reliable networks. The Arvy family of protocols [2] specifically is concerned with regulating access and ownership of a single resource in a network. However, as soon as different types of accesses, e.g. read and write operations, can be discerned, Arvy's way of transferring ownership for every access request may lead to unnecessary traffic when applied to a simple read request from a node that is far away from most other nodes wishing to access that resource. Therefore, most implementations of advanced data sharing or file store infrastructures rely on some sort of caching or data replication strategy. A replica placement algorithm aims to find the best set of nodes in a network to be used as replica sites for a given resource. Replica selection algorithms on the other hand aim to direct an accessing node to at least one of these replica sites. Both of these processes work together to try to optimize for some property (e.g. minimize read access latency given some storage space limits and access patterns). The surveys in [3, 4] give an overview of different approaches to both replica placement and selection while also pointing out their limitations. In [5] we see an approach that tries to reduce access latencies in environments with very limited storage space per node by grouping nodes that are geographically close together into a region, with only a small number of nodes having to serve as a replica site of a file for the whole region. [6] provides functionality for decentralized decision-making for dynamic environments where a global decision-making entity might be a bottleneck.
However, most approaches merely focus on optimizing read accesses, a fact that is also pointed out in [7] and in [8] as *"The optimization problems of write operation in data grid have not been well studied"*. It is further mentioned in [4] that most works of literature only focus on read-only datasets and that the overhead of a consistency model for an updateable dataset can neutralize the benefits that were gained by replication in the first place.
Conversely, we see a discussion of the difference between *"aggressive copy"* vs. *"lazy copy"* in [9]. The aggressive copy mechanism updates all replica sites of a given file with the file's most recent version as soon as a write operation to one of the replica sites occurs, while the lazy copy mechanism only updates other replica sites with the file's most recent version as soon as those other replica sites first need to serve that file. The approach described in [10] uses the underlying replication strategy that was described in [5] and adds a consistency mechanism for write operations resembling a multi-master-slave model that changes its synchronization mode based on access patterns.
Even though [7, 8, 9, 10] all offer some consideration for mechanisms to keep updateable datasets consistent while also trying to reduce the update cost overhead, all of these approaches only focus on adding those mechanisms on top of a given set of replica sites instead of considering the possible costs these write operations might later incur already during replica placement.

2.3 Model

2.3.1 Network

We model the underlying network as a connected Graph: $G = (V, E)$ with V being the set of vertices that represent the nodes in the network and E being the set of strictly-positively weighted, undirected edges representing the physical network connections.
We will need a cost function to help us access the weight associated with an edge and represent it as a cost to our network.

Definition 2.1 (Cost function)**.** We define $c : E \to \mathbb{R}^+$ as the cost function that yields the the weight of an edge. For $\{u, w\} \in E$ we may write $c(u, w)$ instead of $c(\{u, w\})$ for notational convenience. Further, for a set of edges $E' \subseteq E$, $c(E')$ just yields the sum of the contained edges' weights. I.e. $c(E') = \sum_{e \in E'} c(e)$.

It is helpful to have a concept of a connection between nodes in the network. The notion of a general path in G can be thought of as a set of edges $P \subseteq E$ that forms a connection between two endpoint vertices. The cost function from 2.1 can be used to help decide on shortest paths within a given graph. In order to access information about the shortest path, we introduce a path function.

Definition 2.2 (Path function)**.** We may access the set of edges on a shortest path between two nodes with the path function $p : V \times V \to \mathcal{P}(E)$. Assuming $s, t \in V$ and $s \neq t$, then $p(s, t) \subseteq E$ yields the set of edges on the shortest path between s and t meaning that $c(p(s, t))$ yields the lowest possible cost compared to $c(E')$ with $E' \subseteq E$ being any other set of edges that connects s and t. For notational convenience, we may also write $p(s, T)$ with $T \subset V$ to obtain the set of edges on the shortest path from s to the closest node in T. Concretely, $c(p(s, T)) = \min\{c(p(s, t)) | t \in T\}$.

2.3.2 Files and Accesses

Whenever we talk about anything regarding a file or a file access, then we think of a file object as an element in the set of files, e.g. $f \in F$.
Such a file may be stored at none or up to all nodes in the network.

Definition 2.3 (Hosts Set)**.** We use $\mathcal{H} : F \to \mathcal{P}(V)$ to find the set of vertices in our graph that represent the nodes at which a particular file is stored. E.g. $\mathcal{H}(f) = T \subseteq V$ yields the set of vertices T representing the nodes where we can find file f.

In order to access a file f, a node might be interested in the closest replica site for file f. We therefore introduce the concept of a closest host for a particular file:

Definition 2.4 (Closest Host)**.** We define $h : V \times F \to V$ as the function used to find the closest node in the network that stores a file and therefore serves as a host for that file. Assuming $\mathcal{H}(f) \neq \emptyset$ for some file $f \in F$, then $h(s, f) \in \mathcal{H}(f)$ for some node $s \in V$ with $c(p(s, h(s, f))) = \min\{c(p(s, t))|t \in \mathcal{H}(f)\}$ Because all edge weights are strictly positive, we have: $s \in \mathcal{H}(f) \implies h(s, f) = s$.

Reminiscent of a real-world setting, some nodes may only need to work with a particular subset of all the files and also with varying frequencies between different files and types of accesses. We differentiate between a read and a write access and introduce the notion of a corresponding read and write demand for every node-file-pair.

Definition 2.5 (Demands)**.** We model demands for a file as the number of interactions between a given node and a given file. Let $v \in V$ represent a node in our network and let $f \in F$ represent a file in the collection of files. We define:

- Read Demand: $\eta : V \times F \to \mathbb{N}_0$ with e.g. $\eta(v, f)$ as the number of times node v performs a read access on file f.
- Write Demand: $\omega : V \times F \to \mathbb{N}_0$ with e.g. $\omega(v, f)$ as the number of times node v performs a write access on file f.

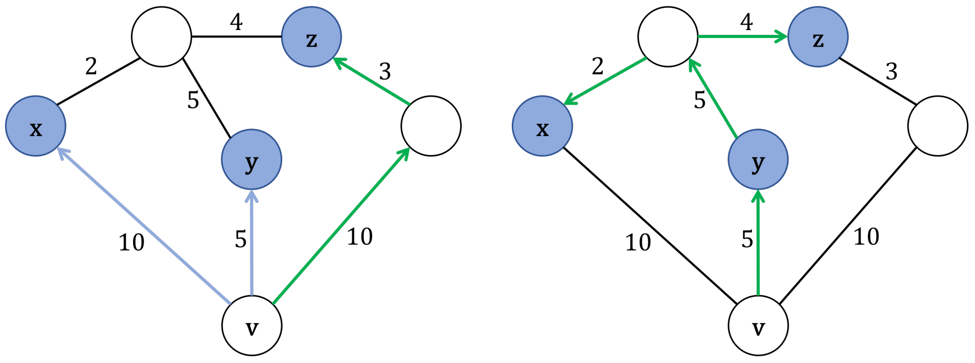
Chapter 3

Replication Network

Assume we have a node in a network that wants to update a file that is potentially stored at multiple replica sites. We need to decide on a way for that update to spread to all replica sites in the network. Based on this mechanism, we can apply our cost function in order to effectively compare different assignments of files to replica sites.
As described in our model, we represent the underlying network as an undirected graph $G = (V, E)$ with positive edge weights and we consider a set F of files. We use an aggressive-copy mechanism as compared to a lazy-copy mechanism in all our considerations in order to keep all replica sites updated with a file's most recent version. The differences between those mechanisms are discussed in detail in [9].

3.1 Time vs. Network Costs

Assume we have the set $\mathcal{H}(f)$ of replica sites for file f and a node $v \in V$ that updates f. For the specific examples presented in figure 3.1 below we have $\mathcal{H}(f) = \{x, y, z\}$ colored in blue.



(a) Time (b) Network

Figure 3.1

We have to decide whether we wish to look at the "time" that was needed for the update to reach the last replica site or rather the overall network cost that is incurred by writing and replicating a file. The first option implicitly assumes that the edge weights represent the bandwidth and physical latency between nodes whereas the second option allows a more general cost function based on other parameters.

- *Time:* A simple mechanism that optimizes the time for the update to spread to the nodes in $\mathcal{H}(f)$ is to send it from v to every node in $\mathcal{H}(f)$ via the shortest path. In this case, the replication network for file f and node v is the shortest-path-tree rooted at v with all leaves in $\mathcal{H}(f)$. The time needed to distribute the update is then equal to the length of the longest of the shortest paths to nodes in $\mathcal{H}(f)$, i.e. $\max\{c(p(v,t)) | t \in \mathcal{H}(f)\}$. In the example depicted in figure 3.1a this is equal to $c(p(v,z)) = 13$. Note how the update gets distributed from v to all nodes in $\mathcal{H}(f)$ along the respective shortest paths with the longest shortest path shown in green.

- *Network costs:* In order to minimize the total network costs incurred by a file update, we want the update of file f to spread along a set $E_f \subseteq E$ of edges that connects $\{v\} \cup \mathcal{H}(f)$ such that the sum of edge weights in S, i.e. $c(E_f)$, is minimal. As shown in figure 3.1b, the update would need to be distributed along the green edges in order to minimize network costs such that $c(E_f) = 16$. Different approaches as to how to get this set of edges are discussed in more detail below.

We see that using a replication network based on optimizing overall network costs allows us to minimize the cost of every write access for a more arbitrarily defined edge cost function. Such a general edge cost function may be defined in such a way as to capture different preferences or to represent real-world costs associated with the usage of specific physical links between nodes. Furthermore, even if one were to try to optimize for time with the approach described above, then it may happen that the corresponding network costs increase significantly. As seen in the example in figure 3.1a, while we could minimize time costs, the network costs of this approach reach $c(p(v,x) \cup p(v,y) \cup p(v,z)) = 28$ compared to 16 as in figure 3.1b. In order to allow for more flexibility, we will therefore focus our considerations on minimizing network costs and go on to discuss different approaches as to how to achieve this.

3.2 Minimizing Network Costs

We consider the choices for a replication network for a single file f. To minimize the total network cost for updates, we need a set of edges $E_f \subseteq E$ in G that connects all nodes in $\mathcal{H}(f)$. We see that the subgraph G_f spanned by E_f would need to be a tree in order to be a replication network of minimal cost. This fact becomes obvious if we assumed G_f to not be a tree, i.e. to contain edges such that loops are formed in G_f. In this case, we could simply remove the edge with the highest cost from each loop in order to lower our costs but let the vertices remain connected.
A tree that provides the required properties is called a Steiner tree:

Definition 3.1 (Steiner tree)**.** For $G = (V, E)$ and $R \subseteq V$, let $ST(G, R) = G' = (V', E')$ be the Steiner tree of G. Then G' is the subgraph of G with $E' \subseteq E$ such that the vertices in $R \subseteq V'$ are connected and the sum of the edge weights in E', i.e. $c(E')$, is minimal.

Let $v \in V$ be a node that updates/writes file f. We recognize that if $v \in V_f$ with V_f being the set of nodes in the Steiner tree $ST(G, \mathcal{H}(f)) = G_f = (V_f, E_f)$, then this Steiner tree spanned across $\mathcal{H}(f)$ trivially provides the best replication network that minimizes overall network load. If $v \notin V_f$ then the best replication network is the Steiner tree that is guaranteed to include v, i.e. $ST(G, \{v\} \cup \mathcal{H}(f))$.

Computing these trees would either have to be done statically or during live operations with f. In both cases, we would reach a limit to the practicability of this approach to a solution as we explore ever-larger networks. This is due to the inherent complexities involved in the computation of Steiner trees. It has been shown that finding solutions to the Steiner tree problem within at most $\frac{96}{95}$ times the cost of an optimal solution is NP-hard [11]. The currently best-known approximation algorithm with polynomial running time performs within an approximation factor of 1.39 of the optimal solution [12].

However, even if the computation were simple, there are potentially up to $|V \setminus \mathcal{H}(f)| + 1$ different replication networks for every file f that could require such a Steiner tree since $ST(G, \{v\} \cup \mathcal{H}(f))$ may be different for every node $v \in V$. One way to avoid this computational overhead is to use a simpler approach in which node v sends its update along the shortest path to the closest node in V_f and then the update gets distributed along the edges of G_f. This way, the tree that connects the designated replica sites of a file remains the same for every write operation on f. Thus, we call this approach the **fixed replication network**. Conversely, we call the approach where we create a new Steiner tree for every write operation a **dynamic replication network**. While both approaches yield trees, the sum of the edge weights in the fixed replication network is never smaller than the sum of the edge weights in the dynamic replication network.

Consider the Steiner tree $G_f = (V_f, E_f) = ST(G, \mathcal{H}(f))$ from before. Conversely, let $G'_f = (V'_f, E'_f) = ST(G, \{v\} \cup \mathcal{H}(f))$ be the Steiner tree that includes v (*dynamic replication network for some* $v \in V \setminus \mathcal{H}(f)$). We wish to find out how much worse it can be to connect v to G_f via the shortest path (*fixed replication network for* v) as opposed to the total cost of an optimal solution provided by G'_f. Concretely, we wish to investigate the factor of α in the following equation:

$$c(p(v, V_f)) + c(E_f) \leq \alpha \cdot c(E'_f) \tag{3.1}$$

An approximate solution for obtaining Steiner trees via shortest paths is presented in [13]. The authors assume S to be the set of nodes that must be included in the Steiner tree. In their approach, the authors start with a single node from S and then continually grow their tree by always adding everything on the shortest path to the closest of the remaining unincluded nodes in S. They show that the cost of their resulting tree has a cost that is never worse than $2 \cdot (1 - \frac{1}{|s|})$ times that of the optimal Steiner tree. We see that this factor approaches 2 for large $|S|$.
However, we cannot use the proof of this bound for our approach because even though we add v to the existing Steiner tree G_f via the shortest path, v might not have been added at that point or in that order. This is because the described approach in [13] always chooses the shortest path to the **closest** of the remaining nodes to be included.

Theorem 3.2. *For any G, and any $f \in F$, the factor of α in equation 3.1 has a constant upper bound of $\alpha < \frac{3}{2}$.*

Proof. We established before that if $v \in \mathcal{H}(f)$, or indeed $v \in V_f$, we then have $G_f = G'_f$. As this situation does not give us any information about how large α could be in a potential worst-case, we will now assume that $v \in V \setminus V'_f$.
Recall that in G'_f we look at the Steiner tree $ST(G, \{v\} \cup \mathcal{H}(f))$ whereas $G_f = ST(G, \mathcal{H}(f))$. Without loss of generality, we can make certain assumptions on the shape of G'_f. Concretely, we can assume G'_f to be of a form in which v connects subgraphs T_i of G_f, for $i \in [n]$ and $n \in \mathbb{N}$, via distinct paths. We therefore have n as the number of distinct subgraphs that are connected through v. Given this, we can distinguish different cases as to how v could be included in G'_f compared to G_f.

Case: $n = 1$

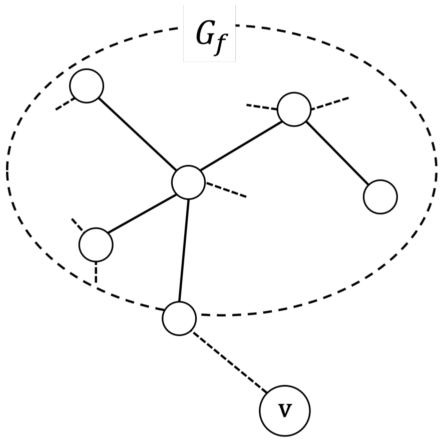


Figure 3.2

If $n = 1$ then v was connected to via the shortest path $p(v, V_f)$ to G_f to form G'_f. In this case we have $c(p(v, V_f)) + c(E_f) = \alpha \cdot c(E'_f)$ with $\alpha = 1$ and our simple approach thus yields the optimal solution.

Case: $n \geq 2$

In this case we have up to n different subgraphs T_i of G_f with $i \in [n]$. These subgraphs' combined vertex sets further contain all the vertices in $\mathcal{H}(f)$. According to our assumptions, every subgraph T_i is connected to v via a distinct path p_i of cost $c(p_i) = c_i$ in order to form G'_f as depicted in figure 3.3. Further, we let V_i and E_i be the vertex and edge sets of every T_i with $i \in [n]$ respectively. Thus, the minimal cost provided by G'_f can be written as:

$$c(E'_f) = \sum_{i \in [n]} (c(E_i) + c_i) \tag{3.2}$$

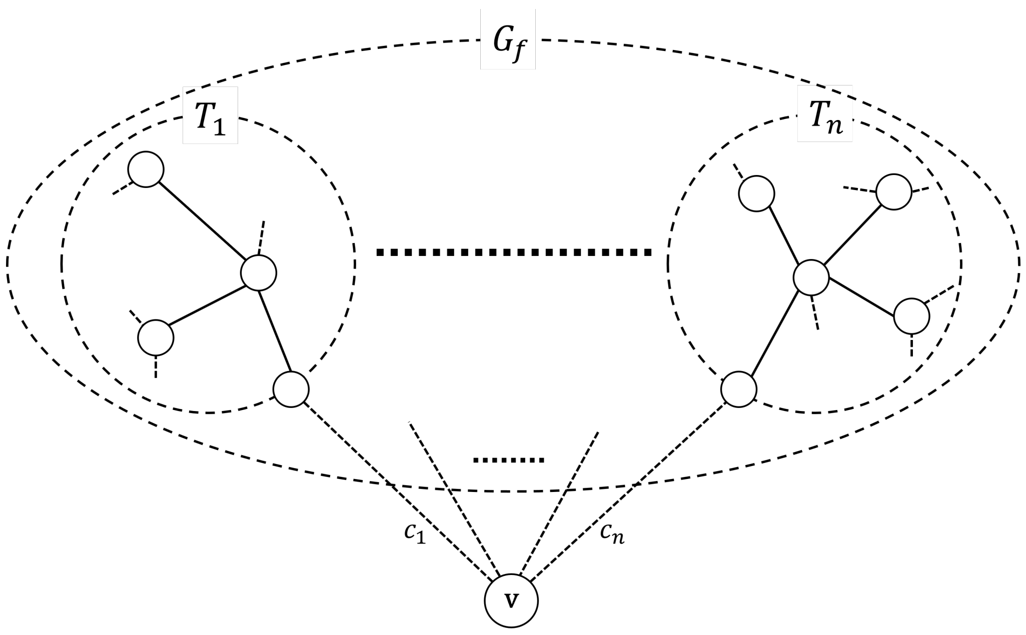


Figure 3.3

We note how $\min\{c_1, \ldots, c_n\} \leq \frac{1}{n} \cdot \sum_{i \in [n]} c_i$ and we use this knowledge to apply the following transformations to the costs resulting from using our simplified approach:

$$\begin{aligned} c(E_f) + c(p(v, V_f)) &\leq c(E_f) + \min\{c_1, \ldots, c_n\} \\ &< \sum_{i \in [n]} (c(E_i) + c_i) + \min\{c_1, \ldots, c_n\} \\ &= \sum_{i \in [n]} c(E_i) + \sum_{i \in [n]} c_i + \min\{c_1, \ldots, c_n\} \\ &\leq \sum_{i \in [n]} c(E_i) + (\frac{n+1}{n}) \cdot \sum_{i \in [n]} c_i \\ &< (\frac{n+1}{n}) \cdot (\sum_{i \in [n]} c(E_i) + \sum_{i \in [n]} c_i) \\ &= (\frac{n+1}{n}) \cdot \sum_{i \in [n]} (c(E_i) + c_i) \\ &= (\frac{n+1}{n}) \cdot c(E'_f) \end{aligned} \tag{3.3}$$

Recognizing that $\max\{\frac{n+1}{n} | n \geq 2\} = \frac{3}{2}$, we could show that α from equation 3.1 always stays below $\frac{3}{2}$ for any $n \in \mathbb{N}$ thereby proving according to theorem 3.2 that $\alpha < \frac{3}{2}$. □

3.3 Fixed vs. Dynamic Replication Network

We have seen that the cost of using a fixed replication network lies within $\frac{3}{2}$ times the cost of a dynamic replication network with many cases being below this worst-case.
While the costs of write operations in a fixed replication network may exceed the same costs of a dynamic replication network, the fixed case has other potential advantages. Firstly, the solution is easier to compute and requires less storage overhead while also getting rid of the need for a lookup overhead during runtime at every node for it to recognize in which situation it is currently in. Secondly and most importantly, every write operation travels along the same set of edges in the fixed Steiner tree that connects the replica nodes. This means that the network costs would not increase if we just designated every node on the Steiner tree as a replica node. Doing this offers us the benefit of decreasing overall read costs because we now offer more and potentially closer replica nodes for read operations while not increasing write costs at all. This also offers the possibility to come up with algorithms that create G_f at the same time as they are exploring the correct nodes to put in $\mathcal{H}(f)$. Even though this thesis does not deal with the effects of mutual exclusion and simultaneous writes, the mentioned second point also provides the opportunity to implement a mechanism to detect simultaneous writes since write updates are bound to meet on the tree spanned by the fixed replication network.

Chapter 4

Algorithms

Having established the fact that we wish file updates to be distributed to all replica sites in an aggressive manner via a replication network, we still need a way to decide on appropriate nodes that should serve as replica sites for a file.
As mentioned in [4, 7, 8] most available research only focuses on minimizing read accesses with respect to limited storage space per node and other considerations. While the approaches described in [7, 8, 9, 10] try to offer mechanisms for efficient data consistency methods in updatable datasets, they still only work on top of given replica sites. In the last chapter we assumed to already be given $\mathcal{H}(f)$ for a file f. However, we still need an efficient way to determine $\mathcal{H}(f)$ for every file. We also know that ideally G_f forms a Steiner tree for all nodes in $\mathcal{H}(f)$. The tree structure of the fixed replication network G_f then allows us to use every node of G_f as a replica site for f such that $\mathcal{H}(f) = V_f$.
We wish to consider the nodes' update behavior already during replica placement in order to choose every file's replica sites such that the resulting replication network serves to minimize total network cost.

The **total network cost** consists of both total read and total write costs in our network.

Definition 4.1 (Total Read Cost)**.** As established before, whenever a node $v \in V$ wants to access a file $f \in F$, it gets that file from its closest replica site for a read cost of $c(p(v, h(v, f)))$ to the network. If the node v serves itself as a replica site for file f, i.e. if $v \in \mathcal{H}(f)$, then the resulting read cost is 0.
This yields our formula for the total read costs in the whole network:

$$\textsc{TotalReadCost} = \sum_{f \in F} \sum_{v \in V} \eta(v, f) \cdot c(p(v, h(v, f)))$$

Assuming $G_f = (V_f, E_f)$ represents the subgraph of G that models our replication network for a file $f \in F$, then we can use G_f to define write costs.

Definition 4.2 (Total Write Cost)**.** As discussed in chapter 3, whenever a node $v \in V$ wants to write or update a file $f \in F$, it sends the updated file to the

closest replica site $h(v, f)$ from where it gets further distributed along all the edges of G_f to all nodes in $\mathcal{H}(f)$.
The formula for the total write costs is therefore:

$$\textsc{TotalWriteCost} = \sum_{f \in F} \sum_{v \in V} \omega(v, f) \cdot (c(p(v, h(v, f))) + c(E_f))$$

We see that the network costs incurred for read operations decrease, the closer a requesting node is to an available replica site. Conversely, while the write cost of a simple write operation may also decrease if a closer replica site is reachable, the availability of that replica site means in turn that it also has to be connected to the other replica sites via some additional edges in E_f, causing write operations of other nodes to increase in cost.
We observe how the total costs are each calculated as the sum of the costs of operations with every file $f \in F$. Since we did not introduce any notion of dependency between the files, we can assume the creations of the corresponding replication networks G_f for $f \in F$ to also be independent of each other. Without loss of generality, we can therefore focus our attention on how an algorithm finds a replication network for a single file. To that end, we introduce $\textsc{Cost}_{G_f}$ as a shorthand for all the read and write costs incurred by operations with a file f given G_f.

In the selection of appropriate nodes for $\mathcal{H}(f)$ and the corresponding creation of G_f, we wish to strike a balance such as to minimize the sum of read and write costs, i.e. $\textsc{Cost}_{G_f}$, incurred by operations with f.

4.1 Special Cases

Only considering a single file $f \in F$, we immediately recognize two special cases:

Case $|\mathcal{H}(f)| = 1$:

There exists only one replica site for file f. In this case, a file update doesn't have to be distributed to other replica sites and therefore we have $E_f = \emptyset$. In this case, a single read access to f incurs the same costs as a single write access to f for all nodes because $c(E_f) = 0$.

Case $\mathcal{H}(f) = V$:

Here the file is stored on all nodes represented by the vertices in G. The read costs for f will be 0 for all nodes in the network because we have $\forall v \in V : h(v, f) = v$. In order for $c(E_f)$ to be minimal, G_f has to be the Minimum-Spanning-Tree (MST) of G. The MST is just the special case Steiner tree $ST(G, V)$ for whose computation several efficient algorithms are available [14, 15].

4.2 Simple GreedyNeighbor

In this first step, we propose a very simple and naive implementation of how we might determine which nodes should belong to $\mathcal{H}(f)$ and how we should create G_f out of G. Our algorithm is based on the observation that, in a fixed replication network G_f, we can use every node in V_f as a replica site. We therefore have $\mathcal{H}(g) = V_f$ and we may use these terms interchangeably.

To be better able to reason about this and further approaches, we introduce the notion of neighborhood in our network:

Definition 4.3 (Neighborhood)**.** Given graph $G = (V, E)$ and $U \subseteq V$, we denote $\mathcal{N}(U) \subseteq V$ as the union of the neighborhoods of all the vertices of U in G. In the case of $|U| = 1$, e.g. $U = \{u\}$, we just write $\mathcal{N}(u)$ for convenience, with $\mathcal{N}(u)$ meaning that $\forall v \in \mathcal{N}(u), \exists e \in E\colon\ e = \{u, v\}$.

Using this, we can now give a description of a first approach to a greedy algorithm.

Algorithm 1: Simple GreedyNeighbor

```
1  $E_f := \emptyset$
2  $V_f := \textsc{ChooseBest}(V)$
3  $G'_f := G_f$
4  while $true$ do
5      $W := \mathcal{N}(V_f) \setminus V_f$
6      foreach $w_i \in W$ do
7          $v := \arg\min_{u \in \mathcal{N}(w_i) \cap V_f} \{c(w_i, u)\}$
8          $T_f := (V_f \cup v, E_f \cup \{w_i, v\})$
9          if $\textsc{Cost}_{T_f} < \textsc{Cost}_{G'_f}$ then
10             $G'_f := T_f$
11         end
12     end
13     if $G_f \neq G'_f$ then
14         $G_f := G'_f$
15     else
16         break
17     end
18 end
```

The algorithm presented above greedily grows G_f given the underlying network represented by G. Initially, G_f does not have any edges. For the first node, we call ChooseBest(V) in line 2 of the algorithm. This initializes V_f to only contain the single node out of V that would have been chosen if we were only allowed to choose one node as a replica site for f. We continue the initialization

by storing a copy of the current G_f in G'_f in line 3.
From there, we iteratively grow G_f until we cannot get any more cost-benefits by adding further neighbors. In each iteration, the neighboring nodes to V_f in G that are not yet part of V_f are considered as candidates to be added to G_f. These candidates are represented by the set W in line 5. For each of these candidates, if one of them has edges to multiple different nodes in V_f, we identify the shortest of these edges in line 7. We then add the candidate and the corresponding edge that brought us the greatest cost reduction to V_f and E_f respectively. After that, we check whether we were able to grow G_f and iterate again until there is no more neighbor to V_f that would yield any benefit if it were added to the replication network G_f. Note how every iteration will reduce the costs and we will therefore never have a situation where our costs are higher than after the algorithm's initialization.

4.2.1 Trees

In a first step, we want to show how the algorithm presented in 1 runs on a restricted set of possible inputs. To that end, we assume for now that G has the form of a tree, i.e. G has no loops. That fact implies that there is only one path between any two nodes. This also trivializes the need to find a suitable replication network G_f between the chosen nodes in $\mathcal{H}(f)$.

In order to better reason about this algorithm, we need a way to describe different partitions of our graph.

Definition 4.4 (Partition)**.** Let $G = (V, E)$ be a graph and let $e = \{u, w\} \in E$ with $u, w \in V$ be an edge in our graph. If e forms a **bridge** in G, then if e were removed, G is no longer connected. Now assuming that e is a bridge, we introduce $part_G : V \times E \to \mathcal{P}(V)$ as the partitioning function for G.
Given the above, $part_G(u, \{u, w\}) \subset V$ represents the set of nodes that remain connected to u in G if the edge $\{u, w\}$ were removed. Note that $part_G(u, \{u, w\})$ will always yield a strict subset of V because we assumed e to be a bridge and we therefore do not have any other paths to connect u and w without using e.

Theorem 4.5. *Algorithm 1 will yield the optimal G_f if G is a tree.*

Proof. Let $s \in V$ be the starting node that was chosen during the algorithm's initialization in line 2, i.e. s is the node that yields the lowest costs if we were only allowed to choose a single node. To prove Theorem 4.5, we need to prove two different parts. We need to show that our stop condition is sound and that there is no node other than s that could yield fewer costs in the resulting G_f if it was used as a starting node instead.

To prove that our stop condition is sound, we need to show that we cannot arrive in a situation where it would make sense to continue growing G_f even if there are no direct neighbors of G_f in G that offer any cost-benefit. Concretely, there cannot be a configuration that includes all the nodes from V_f as replica sites that yields lower costs than just using G_f. We investigate the question of when adding a neighbor offers a cost-benefit. Assume $u \in V_f \land w \in \mathcal{N}(u) \land w \notin V_f$. The node represented by w is therefore a possible candidate to be added. What effect does adding w to G_f have on total costs?

- Effect on $part_G(u, \{u, w\})$: Because w is further away than u, no nodes in $part_G(u, \{u, w\})$ will change their desired replica site if they want to read file f and therefore the read costs will not change for this partition. However, if they want to write file f then their write update has to be distributed to w as well and the cost for every write operation increases by $c(\{u, w\})$.
- Effect on $part_G(w, \{u, w\})$: Because w is closer than u to all nodes in $part_G(w, \{u, w\})$ they will all change their desired replica site for file f to w and therefore decrease costs for every read operation by $c(\{u, w\})$. Costs for write operations will not change because writes needed to be sent along $\{u, w\}$ anyway to reach u even if w were not added.

Note that $part_G(w, \{u, w\}) = V \setminus part_G(u, \{u, w\})$

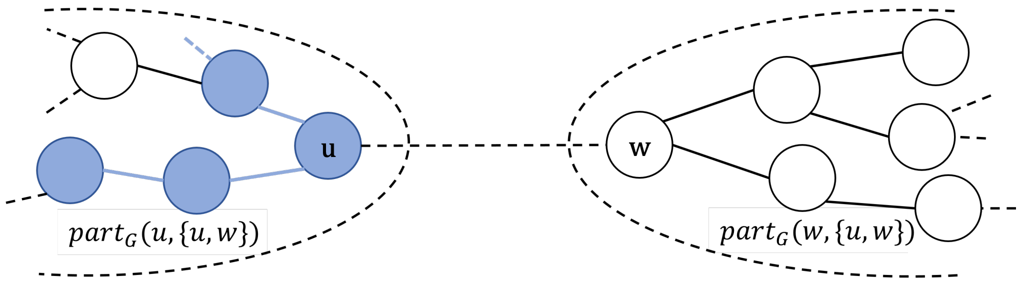


Figure 4.1

It therefore makes sense to add w if the following condition holds:

$$\sum_{v \in part_G(u,\{u,w\})} \omega(v, f) \cdot c(u, w) \quad < \quad \sum_{v \in part_G(w,\{u,w\})} \eta(v, f) \cdot c(u, w) \tag{4.1}$$

As all our edge weights are strictly positive, we see that the multiplication with the edge cost does not change the outcome and the question of adding w is therefore independent of $c(u, w)$ in this scenario. The greedy algorithm terminates as soon as the above condition is not satisfied for any neighbor of V_f in G.

Now assume we added w anyway even though the above condition was not satisfied. Because G is a tree, trying to grow G_f further from w can only decrease the value on the right hand side of the condition shown in 4.1, while it can only increase for the left hand side. We therefore know that if the condition did not hold when we added w, it will also never hold if we try to grow further from w. Thus it made sense to not add w in the first place and stop growing altogether as soon as no neighbor offers any cost-benefit, i.e. no neighbor w to a node $u \in V_f$ satisfies condition 4.1.

Further, we need to show that there exists no starting node $t \neq s$ that gives a better result than starting with s.
Let $G_f = (V_f, E_f)$ be the subgraph that was found when s was chosen as a starting node While $G'_f = (V'_f, E'_f)$ is the subgraph that results from using t as the starting node. We need to differentiate the following cases:

Case $t \in V_f$:

Starting in t, G'_f cannot grow to include nodes from $V \setminus V_f$ since none of these nodes can satisfy the condition in 4.1 because they also did not satisfy it when s was used as starting node. Moreover, we know that G'_f will grow to include s since the algorithm is guaranteed to include every neighbor that offers a cost-benefit and we know that the connection between s and t got included in G_f because it offered lower costs than using s alone which is, in turn, less costly than using t alone. We now know that $V'_f \subseteq V_f$ and $s \in V_f$ as well as $s \in V'_f$. From there, the algorithm will continue to grow G'_f by applying condition 4.1 to every candidate until eventually $G_f = G'_f$.

Case $t \notin V_f$:

In this case, G'_f cannot grow further away from s because, according to the first part of our proof from above, these directions cannot become profitable again since t was already not included in G_f. Therefore, as long as $\mathcal{N}(V'_f) \cap V_f = \emptyset$, G'_f can only grow straight towards s. Analogous to the reasoning from the previous case, upon including the first node from V_f to V'_f it can and will only grow up to the bounds of G_f and include all nodes of G_f. Thus we get a G'_f with $V'_f = V_f \cup nodes(p(t, s))$ with $nodes(p(t, s))$ yielding the set of nodes that appear in the set of edges represented by $p(t, s)$ and therefore $V_f \subset V'_f$. Since condition 4.1 did not hold for the additional nodes in V'_f during the creation of G_f, these nodes were not beneficial and thus using G'_f as the replication network for f yields higher costs than G_f.
We see in both cases that, starting in $t \neq s$, we cannot arrive at a G'_f that offers a lower total cost than G_f. $\square$

4.2.2 General Connected Graph

We have seen that the presented algorithm yields optimal results if we restrict ourselves to only allowing trees as valid inputs for G. However, most modern networks' backbone infrastructures, especially the internet [16], are not built or designed after a tree topology. This has many obvious reasons including redundancy concerns and load balancing. Therefore we now no longer assume the underlying network represented by G to be a tree. We merely require G to be a connected graph. In comparison to trees, G may now have multiple different paths between any two given nodes and we could therefore also have loops in G. This now allows for multiple choices for connections between nodes in V_f. In particular, $part_G$ is not applicable to edges that are part of a loop and the argument about the condition 4.1 never again being true if we continue growing G_f may not be valid in all cases. Furthermore, we recognize that the presented algorithm can only reason about the benefits that the inclusion of neighbors to V_f might incur while it is oblivious to connections between nodes further away from V_f. We illustrate the different challenges arising around G being a general connected graph below. For simplicity, we will assume node $u \in V$ to be an articulation point in G for all of the following examples. That is, if u and all the edges to its neighbors were removed, then G would no longer be connected.

Switching

We wish to describe the phenomenon of "switching" in the context of an iteration of algorithm 1. As established before, whenever a node wishes to perform a read or a write operation with file f, it interacts with the closest replica site. E.g. $h(w, f) = u$ means that node w interacts with file f via node u. As a new replica

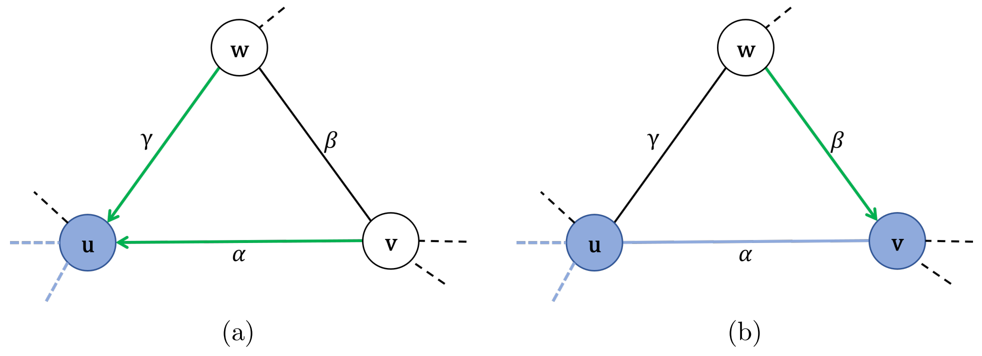


Figure 4.2: Switching Showcase

site gets added to V_f after an iteration of algorithm 1, some nodes will have a new preferred, i.e. closest, replica site. However, we only talk of "switching" when this new replica site was not part of the path to the old replica site. Switching can therefore never occur in trees because there is only one path between any two nodes. However, if we look at the part of a general graph as shown in 4.2a, we see an occurrence of node w switching from node u over to node v to interact with file f as soon as v got added as a replica site as seen in 4.2b. In order for this situation to occur in this fashion, several conditions need to hold about this part of the network. Firstly, the algorithm must have arrived in a situation where node u got added as a replica site in 4.2a and the demands for f of all the participating nodes as well as the edge lengths in G need to be such that node v got included in V_f in a further iteration of the algorithm. The conditions of the edge lengths for the given example are as follows:

(i) $\alpha < \beta + \gamma$

(ii) $\gamma < \alpha + \beta$

(iii) $\beta < \gamma$

Conditions (i) and (ii) explain why nodes v and w chose to contact replica site u directly via their respective edges to u in 4.2a. The condition in (iii) explains why node w chose to switch its path to the replication network in 4.2b after v got included in V_f. In fact, we could even have $\beta < \alpha$, in that case, if node w got included in V_f instead of node v, then node v would be the one to switch its path to a preferred replica site. We extend the example from above with a new

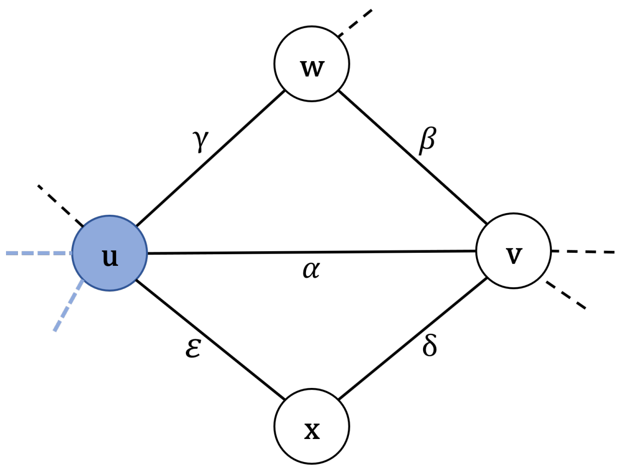


Figure 4.3

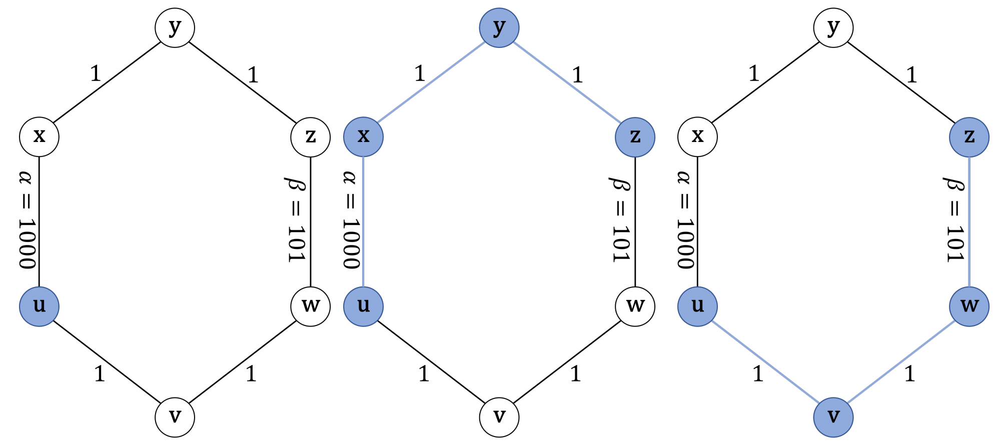
node x and the added condition (iv) $\alpha + \delta < \epsilon$. In this case, if v got included in V_f in the next step, w would be switching as before but x would not. This is because x already reached u over a path that led via v.
We now investigate what the effect on read and write costs would be if v were included in V_f. We see that for all nodes that already reached u via v, i.e. node x because of (iv), the induced read costs decrease by α per read operation while

the induced write costs stay the same. This is the same case as was already shown for trees. Consequently, there could also be further nodes that will remain with node u as preferred replica site irrespective of the inclusion of v to V_f. For those nodes, the read costs would stay the same while the write costs per operation increase by α. However, if we look at the switching node w, we can observe more subtle behavior. The read and write costs induced by w (and every further node only reachable through w) decrease by $\gamma - \beta$ per read or write operation. Yet there is an additional effect on the costs for every write operation. Write updates now have to go along the edge with length α in order to spread to the rest of G_f. Therefore the costs per write increase by α. We remember (ii) $\gamma < \alpha + \beta \iff \gamma - \beta < \alpha$ and see that for write operations, the savings are smaller than the additional write overhead of α. Therefore, switching nodes alone with sufficiently more write demand than read demand could never cause the inclusion of v to V_f according to algorithm 1 because it would only increase the costs at that step.

While we see this difference in behavior between simple trees as compared to general graphs, we also make a further observation. If the algorithm presented in 1 were to be implemented in practice, as long as the underlying network could be represented as a tree, we could simply use the condition mentioned in 4.1 to decide for every neighbor of V_f if it is worth to be included in the replication network. This would allow for a fairly efficient implementation. However, if G is not guaranteed to be a tree and if switching might occur, then if one wouldn't want to have to recalculate all of the costs for every candidate at every iteration, it would become necessary to use a more sophisticated condition that also mentions the different edge lengths instead of only comparing demands.

Wrong Choice

In this section we wish to present a fundamental limitation of algorithm 1. The limitations are inherent to the greedy nature of the algorithm as well as its limited local knowledge about the graph layout around the neighborhood of V_f. The occurring problem is further enabled by the aforementioned switching of paths. Observe the three different instances of a graph shown in figure 4.4. We can think of the underlying network as showing the connections between two regions of nodes that are themselves relatively closely connected. We differentiate between the region consisting of nodes u, v and w and the region consisting of nodes x, y and z. These two regions might represent two geographically distanced office locations or any other situation that could explain why the weight of the edges connecting the corresponding subgraphs is significantly larger than the weight of any of the other edges. Also note how the edge $\{u, x\}$ has a weight that is again significantly larger than the weight of edge $\{w, z\}$.



(a) start (b) wrong (c) right

Figure 4.4

For the purposes of the example, we let none of the nodes have any demand for file f except for the following:

- **Node u:** $\eta(u, f) = 100$, $\omega(u, f) = 1$
- **Node z:** $\eta(z, f) = 10$, $\omega(z, f) = 0$

Given these demands and edge weights, the algorithm will initialize and choose node u as a starting node as seen in figure 4.4a. Because $\omega(u, f) < \eta(z, f)$ and no other demands, we know that it makes sense for the algorithm to grow G_f beyond only u to also encompass node z no matter the length of the path. Also note that right after the initialization, node z has its preferred route to replica site u via the path $p(z, u) = \{\{u, v\}, \{v, w\}, \{w, z\}\}$ with a total cost of $c(p(z, u)) = 103$. However, if we let the algorithm continue, it will add node x to V_f in the first iteration and continue to grow during further iterations until it reaches node z as shown in figure 4.4b whereas the replication network presented in figure 4.4c would have been the optimal choice. We call the replication network found by the algorithm and shown in blue in 4.4b $G_f = (V_f, E_f)$ and we conversely call the replication network depicted in 4.4c $G'_f = (V'_f, E'_f)$. Since all of the nodes that have any demand for file f, i.e. u and z, are themselves replica sites in G_f as well as G'_f, we do not have any induced read costs for either of the two replication networks and we therefore only have to compare the cost of write operations that send their updates along the edges of the corresponding replication network. With $c(E_f) = 1002$ and $c(E'_f) = 103$ we recognize that our algorithm gave a result that is close to 10 times as costly as the optimal result. We will go on to investigate how the algorithm chose such a path and what this means for its approximation ratio compared to an optimal solution.

Comparing figure 4.4a to figure 4.4b we note that the algorithm must have included node x in V_f in the first iteration after the initialization whereas node v would have been on the optimal path. We therefore know, by the design of the algorithm, that it must have been more beneficial to include node x in that iteration and we go on to compare the effects of including either of these two nodes.

- **Inclusion of node v:**
  We see that $h(z, f)$ changes to node v. Therefore we reduce the costs of a single read operation of node z by 1 and we save 10 in read costs compared to the initial state. The inclusion of edge $\{u, v\}$ to E_f causes u's write cost to rise by 1 for every write operation. Since $\omega(u, f) = 1$, we therefore have total cost savings of 9 in the first iteration.

- **Inclusion of node x:**
  The inclusion of edge $\{u, x\}$ to E_f causes a total increase in write costs of 1000. However, since $c(p(x, z)) < c(p(u, z))$ this will cause node z to switch such that in the end $h(z, f) = x$. This yields savings of $10 \cdot (c(p(u, z)) - c(p(x, z))) = 1010$ over all read accesses performed by z. We therefore have total savings of $10 > 9$.

We see how the algorithm decided on this by the greater immediate savings it could gain while only having knowledge about the effects of the possible inclusion of direct neighbors to u.
Admittedly, the respective edge lengths and demands for file f were chosen such as to illustrate a worst-case scenario with that combination. Also the larger the difference in weights of the two edges between the two regions in the graph, the more unlikely it is for such a network to exist in practice.
However, we may still show how this problem causes the algorithm to possibly deliver results whose costs do not lie within a constant factor of the costs of an optimal result.

Theorem 4.6. *The resulting G_f of algorithm 1 yields costs for operations with file f that do not lie within a constant factor of the costs of an optimal solution.*

Proof. We again look at the graph shown in figure 4.4a yet we assume $c(u, x) = \alpha$ and $c(w, z) = \beta$. Since we know that node z will be included anyway because $\eta(z, f) > \omega(u, f)$, we only have to find out how much longer we could make the path $\{\{u, x\}, \{x, y\}, \{z, z\}\}$ compared to the shortest path $p(z, u) = \{\{u, v\}, \{v, w\}, \{w, z\}\}$. Because both paths contain two separate edges each with a weight of 1, we will only focus on the edges $\{u, x\}$ and $\{w, z\}$ respectively. We saw that the switching of z to x caused an increase in write costs that was just set off by the decrease in read costs such that the total savings were just more than if the algorithm would have chosen to include v instead of x. In order for this to be true we need to satisfy

$$\eta(z,f)\cdot 1-\omega(u,f)\cdot 1<\eta(z,f)\cdot(\beta+2-2)-\omega(u,f)\cdot\alpha \tag{4.2}$$

where the left hand side represents the savings incurred by adding v to V_f and the right hand side represents the same for node x.
If we now let n be any arbitrarily large integer, we may choose $\beta = 2$ and $\alpha = 2n$. In order to still satisfy the above condition we can choose $\omega(u,f) = 1$ and $\eta(z,f) = 2n$. Substitution yields

$$2n-1<2n \tag{4.3}$$

We see how the difference in costs between the solutions represented by figure 4.4b and figure 4.4c respectively can be completely described by α and β. With our choice of α and β we get $\frac{2n+2}{2+2}=\frac{n+1}{2}$ as the ratio between these costs. Since we can choose an arbitrarily large n, this ratio can also get arbitrarily large. We therefore know that the algorithm's resulting cost does not lie within a constant factor of the optimal cost as had to be proven. $\square$

As a side note, for the above problem to occur on the given graph, we implicitly assumed $\eta(u,f)$ to be sufficiently large such that u gets chosen as the starting node during the algorithm's initialization.
Even though such a situation becomes ever more unlikely for ever-larger differences in path lengths to be encountered in the real world, we still acknowledge that algorithm 1 therefore cannot be used on general graphs.

4.3 GreedyNeighbor on Shortest Paths

In order to find a way around the problem stated in theorem 4.6 we adjust the algorithm from 1 to only consider neighbors as candidates to be added to G_f if the edge that connects the current node from V_f to that neighbor, lies on the shortest path between that neighbor and V_f.
More concretely, assume $u \in V_f$ and $C'_u = \mathcal{N}(u) \setminus V_f \neq \emptyset$. This means that the set C'_u contains at least one neighbor of u that might be a potential candidate to be added to the replication network. However, we might only consider a subset of C'_u as real candidates, namely only the neighbors of u whose edge to u also lies on the shortest path to u: $C_u = \{w | w \in C'_u \wedge \{u, w\} \in p(u, w)\}$.

This yields the following algorithm:

Algorithm 2: GreedyNeighborSP

```
1  $E_f := \emptyset$
2  $V_f :=$ ChooseBest$(V)$
3  $G'_f := G_f$
4  while $true$ do
5      $W := \mathcal{N}(V_f) \setminus V_f$
6      foreach $w_i \in W$ do
7          $v := \arg\min_{u \in \mathcal{N}(w_i) \cap V_f} \{c(w_i, u)\}$
8          if $\{w_i, v\} \notin p(w_i, v)$ then
9              continue
10         end
11         $T_f := (V_f \cup v, E_f \cup \{w_i, v\})$
12         if $\text{Cost}_{T_f} < \text{Cost}_{G'_f}$ then
13             $G'_f := T_f$
14         end
15     end
16     if $G_f \neq G'_f$ then
17         $G_f := G'_f$
18     else
19         break
20     end
21 end
```

The necessary restrictions outlined above are described in algorithm 2 on the lines 8 to 10. Concretely the algorithm now ensures that if there is a neighbor $v \in \mathcal{N}(V_f) \setminus V_f$ then that neighbor will only be evaluated as a possible candidate if the shortest edge connecting V_f to v (ensured on line 7) also lies on the shortest path from V_f to v. If not, the consideration of v as a possible candidate will be skipped in this iteration (line 9). While it might still be beneficial to include v

in V_f, this will now only happen if v is reached via a shortest path. This change now inhibits the occurrence of problems like those described in figure 4.4.

Furthermore, the introduced check can only restrict our choice if other paths to that neighbor exist. If G were a tree, then there are no nodes between which there are multiple paths. Thus our new algorithm works equivalently to our original algorithm from 1 if G is a tree and therefore our new algorithm is also optimal on trees.

4.3.1 Unnecessary Replica Sites

Having dealt with that problem, we are still left with other adverse effects that are caused by switching on general graphs. We saw that if a node switches to another path to reach its new closest replica site, that the earlier path is now left unused by that and possibly more nodes.

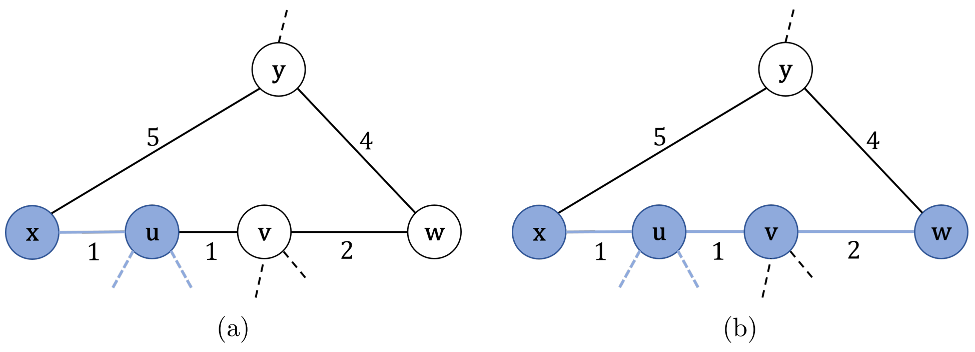


Figure 4.5

To further illustrate this problem, we might imagine a graph as depicted in figure 4.5a with the following demands for the nodes:

Node	η	ω
u	100	50
v	0	0
w	51	0
x	3	0
y	49	0

Algorithm 2 will initially start with node u (step not shown). After that it will consider the nodes $\mathcal{N}(V_f) = \mathcal{N}(u) = \{v, x\}$ as candidates for the first iteration. We know because of $\omega(u, f) < \eta(w, f)$ and because node u is the only node with any write demand, that the algorithm will eventually go on to include node w in V_f. However, while only evaluating v and x, the algorithm decides to include x as shown in figure 4.5a because it offers the greater benefit at that moment. The choice of x satisfies the combined demands of x and y, i.e. $\eta(x, f) + \eta(y, f) = 52$ whereas if node v had been chosen it would only satisfy w's demand $\eta(w, f) = 51 < 52$ for the same cost. Note that the neither demand of x nor y alone would have offset the incurred costs of the inclusion of x. Also note how there did not yet occur any switching up to that point. As stated before the algorithm will go on to include node w and terminate after the situation depicted in figure 4.5b is reached. We recognize how upon the inclusion of w, node y switches to replica site w and therefore redirects the traffic. Now the replica site w serves its own demand of 51 and y's demand of 49 whereas replica site x is left to only serve its own demand of 3. This means that even though the algorithm already terminated and chose earlier on to include node x as a replica site, that it would now yield a cost-benefit if node x could be removed from V_f. Concretely, having $x \in V_f$ like in figure 4.5b saves x's total read costs of $\eta(x, f) \cdot c(u, x) = 3$ and introduces the costs of $\omega(u, f) \cdot c(u, x) = 50$. Therefore if node x were removed from V_f, 47 could be saved from the total cost, yielding $\textsc{Cost}_{G'_f} = 349$ compared to $\textsc{Cost}_{G_f} = 396$ resulting from algorithm 2.

We therefore face the problem that the occurrence of switching during later iterations in algorithm 2 might make entire parts of G_f, that were included earlier on, obsolete, needlessly increasing the total cost $\textsc{Cost}_{G_f}$.

4.3.2 Early Termination

We stated above how algorithm 2 would inevitably go on to include w because $\omega(u, f) < \eta(w, f)$. If we now go on to change $\eta(w, f)$ to 49, the aforementioned condition no longer holds. This will lead us on to discover a much more severe problem involving the algorithm failing to discover all nodes that lower the total cost. Given that assumption, algorithm 2 will terminate after it reaches the state shown in 4.5a. This yields $\textsc{Cost}_{G_f} = 442$ upon termination whereas the optimal cost of 349 remains the same. In this particular case, this happens because the algorithm will never include node v since the demand it carries is not sufficient to offset the cost of including it.
In a more general setting, we may describe the problem as follows. In every iteration, as the algorithm tries to grow V_f, it may only know about the demands that are directly incoming via the respective shortest paths to nodes in V_f as well as about the effects of switching that may occur as it considers each candidate. However, it cannot know about the effects of switching that may occur later on and could therefore miss out on potential future benefits. In the concrete presented case, the algorithm could not know that the later inclusion of w would cause y to switch and yield benefits because it never came to consider w as a candidate since we never had $w \in \mathcal{N}(V_f)$.

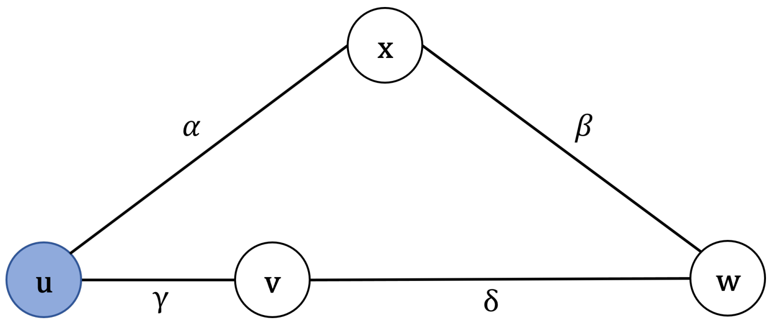


Figure 4.6

We use the specific graph shown in figure 4.6 to learn more about the specific conditions that need to be fulfilled in order for the algorithm to terminate early. We will again assume that after initialization we have $V_f = \{u\}$. In order for that to be the case, we can merely think of $\eta(u)$ being large enough. We further assume that the algorithm will not go on to include any of the neighbors $\{v, x\}$ and terminate early. However, the choice of $V'_f = \{u, v, w\}$ with $E'_f = \{\{u, v\}, \{v, w\}\}$ would have been optimal.
Since the algorithm terminates with only $V_f = \{u\}$ we know that neither of the neighbors of u, if they were included, could have served enough read demand such as to offset $\omega(u)$. We therefore know that either using $\{u, v\}$ or $\{u, x\}$ as replica sites would have increased the costs for operations with f. V'_f being the optimal case shows that after the inclusion of v, node w was viewed as a viable candidate that, in addition to serving its own demand, could also serve as replica site for node x when x switches over to w. That being the case, we see that w chose to reach replica site $h(w, f) = u$ through v right after the initialization phase because otherwise enough demand would have been routed through x such as to justify the inclusion of x. We therefore know the following:

(i) $\gamma + \delta < \alpha + \beta$
because $p(u, w) = \{\{u, v\}, \{v, w\}\}$ right after initialization

(ii) $\alpha < \beta + \gamma + \delta$
because otherwise v would have been included

We may further assume that no switching occurs during the consideration of either v or x since they would otherwise have been included.

(iii) $\gamma + \delta < \beta$
because v and w do not switch if x were included (stronger (i))

(iv) $\alpha < \beta + \delta$
because x does not switch if v were included (stronger (ii))

(v) $\eta(x,f) < \omega(u,f) + \omega(v,f) + \omega(w,f)$
because of (iii) and because x was not included

(vi) $\eta(v,f) + \eta(w,f) < \omega(u,f) + \omega(x,f)$
because of (iv) and because v was not included

(vii) $\beta < \alpha$
because x switches to w as soon as w gets included

Having the above as the necessary conditions for the problem of early termination to occur in the depicted situation, we see that the optimal solution extracts its cost-benefit from x's switch to a closer replica site and from the fact that the read demands of v and w are being served directly.
We can use this knowledge to create a situation with yet another obstacle.

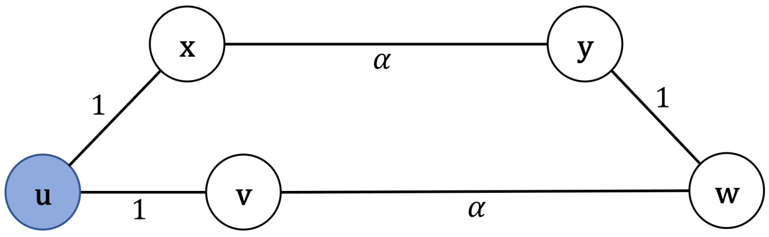


Figure 4.7

We again assume node u's read demand to be large enough such that it is the starting node. Further we have $\eta(y,f) = \eta(w,f) = \omega(u,f) - 1$ and α arbitrarily large. We again see that algorithm 2 will just terminate early and will neither include node v or x. This is because the demand served on the respective path is too low to offset the additional write costs and because none of the other nodes would switch, and thereby reinforce, to another path upon inclusion of either v or x. Thus the cost resulting from algorithm 2 are

$$\textsc{Cost}_{G_f} = (\eta(w,f) + \eta(y,f)) \cdot (\alpha + 1) \tag{4.4}$$

However the optimal solution would be to either also include $\{x,y\}$ or $\{v,w\}$. Given the latter as an example for an optimal solution G'_f, the total costs would be

$$\textsc{Cost}_{G'_f} = \omega(u,f) \cdot (\alpha + 1) + \eta(y,f) \tag{4.5}$$

Given these costs and necessarily $\eta(y, f) < \omega(u, f)$, we see that, in the graph shown in figure 4.7, the costs $\textsc{Cost}_{G_f}$ resulting from the algorithm will stay below a factor of 2 times the costs $\textsc{Cost}_{G'_f}$ of an optimal solution.

$$\begin{aligned}
\frac{\textsc{Cost}_{G_f}}{\textsc{Cost}_{G'_f}} &= \frac{(\eta(w, f) + \eta(y, f)) \cdot (\alpha + 1)}{\omega(u, f) \cdot (\alpha + 1) + \eta(y, f)} \\
&= \frac{2 \cdot \eta(y, f) \cdot (\alpha + 1)}{\omega(u, f) \cdot (\alpha + 1) + \eta(y, f)} \\
&< \frac{2 \cdot \eta(y, f)}{\omega(u, f)} \\
&< \frac{2 \cdot \eta(y, f)}{\eta(y, f) + 1} \\
&< 2
\end{aligned} \tag{4.6}$$

Thus far, the situation depicted in 4.7 and the occurring problem are relatively similar to the situation shown before. However, the additional node on the path between u and w through x now allows us to make the following claim.

Theorem 4.7. *For certain types of underlying graphs G algorithm 2 yields a replication network G_f whose induced cost $\textsc{Cost}_{G_f}$ can be arbitrarily larger than $\textsc{Cost}_{G'_f}$ provided by an optimal replication network G'_f.*

Proof. To prove theorem 4.7 we extend the graph from figure 4.7.

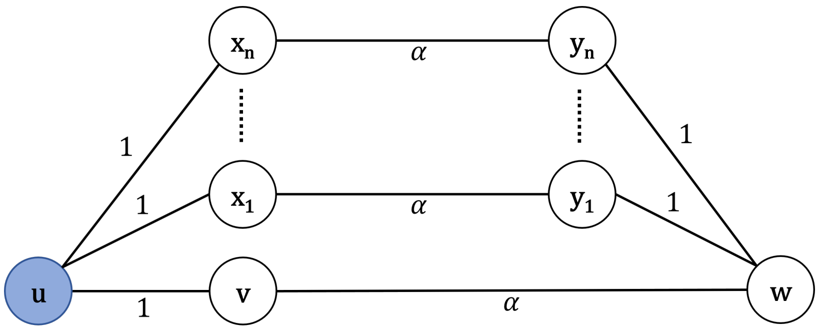


Figure 4.8

We now have up to n paths between u and w through x_i and y_i for $i \in [n]$. We can follow a similar argument as before in first assuming α to be arbitrarily large and $\forall i \in [n] : \eta(y_i, f) = \omega(u, f) - 1$ and also $\eta(w, f) = \omega(u, f) - 1$. Additionally, we again have $\eta(u, f)$ large enough such that u is chosen as the starting node. If we didn't have both x_i and y_i on their respective path between u and w but instead a direct edge between u and some y_j (similar as before in 4.6), then at some large enough n the algorithm would have included y_j and the other y_i with

$i \neq j$ would switch over to y_j thus putting an upper bound on the algorithm's worst-case result.
However, in the situation as seen in figure 4.8, we can follow the exact same reasoning as before for an arbitrarily large n. The algorithm again produced $G_f = (\{u\}, \emptyset)$ whereas an optimal solution could have been provided by $G'_f = (\{u, v, w\}, \{\{u, v\}, \{v, w\}\})$. We see how we could cause up to n nodes to switch if only we grew G_f up to either w or some y_i with $i \in [n]$. More plainly, considering the relationship between the given demands, we can make the following transformations:

$$\begin{aligned}\frac{\textsc{Cost}_{G_f}}{\textsc{Cost}_{G'_f}} &= \frac{(\eta(w,f) + \sum_{i\in[n]} \eta(y_i, f)) \cdot (\alpha+1)}{\omega(u,f)\cdot(\alpha+1) + \sum_{i\in[n]} \eta(y_i,f)} \\ &= \frac{(\eta(w,f) + n \cdot \eta(y_1,f)) \cdot (\alpha+1)}{\omega(u,f)\cdot(\alpha+1) + n\cdot\eta(y_1,f)} \\ &= \frac{(n+1)\cdot\eta(w,f)\cdot(\alpha+1)}{(\eta(w,f)-1)\cdot(\alpha+1) + n\cdot\eta(w,f)}\end{aligned} \tag{4.7}$$

Consequently, for large α and large $\eta(w,f)$ we have

$$\lim_{(\alpha,\eta(w,f))\to(\infty,\infty)} \frac{(n+1)\cdot\eta(w,f)\cdot(\alpha+1)}{(\eta(w,f)-1)\cdot(\alpha+1) + n\cdot\eta(w,f)} = n+1 \tag{4.8}$$

To allow for large $\eta(w,f)$ and large α without causing the algorithm to choose another node than u as a start node, we just assume $\eta(u,f)$ to be even larger in turn. Nonetheless, we see how, in such a situation, the resulting $\textsc{Cost}_{G_f}$ approaches $n+1$ times the optimal costs $\textsc{Cost}_{G'_f}$ with n being the number of additional paths between nodes u and w. Therefore we do not have a fixed approximation ratio for the costs caused by the resulting G_f from algorithm 2. $\square$

Since a situation similar to the one depicted in figure 4.8 might indeed occur in the real world where lots of redundant connections are very widespread, we cannot use algorithm 2 on general graphs.

4.4 GreedyGlobal

The algorithms presented up to now were a very natural way of probing different techniques to grow G_f on top of a given graph G. However, we clearly identified limitations to the proposed strategies on general graphs. We noticed that only being able to reason about the effects of including nodes in $\mathcal{N}(V_f)$ in each iteration makes it impossible to give reasonable estimations about the consequences to $\textsc{Cost}_{G_f}$ that might or might not have occurred otherwise. Therefore we need a way to look further than just within $\mathcal{N}(V_f)$. After what we learned in the subchapter where we described how algorithm 1 might make the wrong choice we know that we should clearly follow growth directions that form a shortest path. This consideration leads us to the following algorithm.

Algorithm 3: GreedyGlobal

```
 1 $E_f := \emptyset$
 2 $V_f := \textsc{ChooseBest}(V)$
 3 $G'_f := G_f$
 4 while $true$ do
 5     $W := V \setminus V_f$
 6     $B := \mathcal{N}(W) \cap V_f$
 7     foreach $b_i \in B$ do
 8         foreach $w_j \in W$ do
 9             $T_f := (V_f \cup nodes(p(b_i, w_j)), E_f \cup p(b_i, w_j))$
10             if $\textsc{Cost}_{T_f} < \textsc{Cost}_{G'_f}$ then
11                 $G'_f := T_f$
12             end
13         end
14     end
15     if $G_f \neq G'_f$ then
16         $G_f := G'_f$
17     else
18         break
19     end
20 end
```

The initialization phase is the same as in algorithm 1 and 2. We again try to grow the graph in a greedy manner. At every iteration, the algorithm evaluates all possible shortest paths from nodes in V_f to nodes in the rest of the graph and includes the nodes and edges on the path that yields the greatest cost-benefit. In line 5 we store all nodes outside of V_f in W. Consequently in line 6 we store all nodes from V_f that have neighbors outside of V_f in B, i.e. B contains the nodes on the "border" of V_f. In lines 7 to 14 we evaluate every possible shortest path and include the best path found in that iteration. The above specification merely serves as a description as to how the algorithm works while a real-world implementation could take advantage of several possible optimizations.

4.4.1 Trees

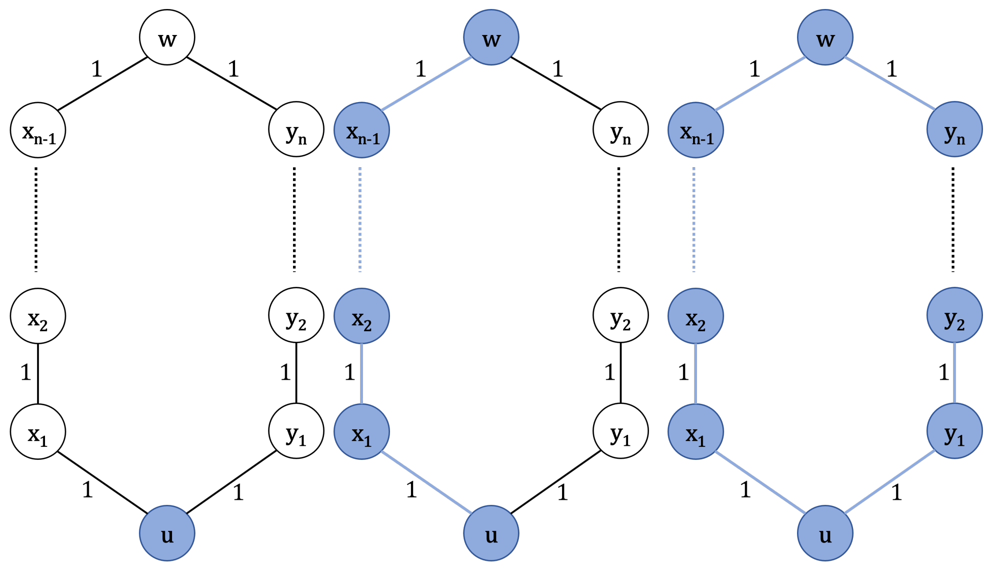
If we think about how this new algorithm works on trees in every iteration we see how, whenever a path and all its nodes get included, the resulting G_f from that iteration is similar to the G_f's produced by several iterations of algorithm 1 or 2. However, the other algorithms might also have had some iterations where they grew G_f along different branches therefore performing the steps in a different order. Yet including all viable nodes in any order will always produce the same result on a tree since there are no loops and nodes cannot suddenly switch to a new replica site on a different branch. Because algorithm 3 simply includes nodes in a different order but uses the same mechanism to determine the viability of a node, it therefore produces the same result as algorithms 1 and 2 on trees and is consequently also optimal thereon.

4.4.2 General Connected Graph

Recalling both the examples from figure 4.5 and 4.8 we see that at least in these specific cases algorithm 3 would have produced the optimal result albeit with more computational effort put into a single iteration. This is because at every iteration algorithm 3 has knowledge of every shortest-path-tree rooted at any of the nodes in V_f and can reason about including any of those trees' branches or parts thereof. It would therefore know that e.g. in the example from 4.5 it could include nodes v and w since $p(u,w) = \{\{u,v\},\{v,w\}\}$ is part of the edge set of the shortest-path-tree rooted at u and it would favor this choice instead of the inclusion of the unnecessary node x. However, we must keep in mind that knowledge of this subset of all shortest-path-trees in G is still not sufficient to reason about all possible effects.

Limitation

If we look at the situation depicted in figure 4.9a where we start with $V_f = \{u\}$ right after the initialization of the algorithm, we see that there exist two different paths to reach node w from node u. Further we have $\forall e \in E : c(e) = 1$. Note how we only have $n-1$ nodes in between u and w on the path on the left hand side while we have n nodes in between on the right hand side. Since the shortest-path-tree rooted at u has no knowledge of the edge $\{w, y_n\}$ we can only include up to n nodes in the first iteration of algorithm 3.



(a) start (b) wrong (c) end

Figure 4.9

These n nodes could either consist of the nodes on the path on the left hand side from u up to and including node w or they could consist of the nodes on the path on the right hand side from u up to and including node y_n.
Suppose node u is the only node with any write demand for f and further $\omega(u, f) < \eta(w, f)$. We also let all the nodes on the right hand side path have the same read demand, $\forall i, j \in [n] : \eta(y_i) > 0 \wedge \eta(y_i, f) < \omega(u, f) \wedge \eta(y_i, f) = \eta(y_j, f)$. We go on to calculate the costs savings potential if the algorithm decided to include either of these maximum length paths. We note that since node u is the only node with write demand and because the two paths are both of cost $c(p(u, w)) = c(p(u, y_n)) = n$ that the incurred write costs would be the same for both choices. Therefore we only have to compare the possible savings on read costs. As seen in equation 4.9, the savings upon including the nodes on the right hand path consist of savings incurred by the switching of w to y_n and the fact that all the nodes y_i, for $i \in [n]$, can now serve themselves and no longer have to contact node u.

$$\begin{aligned}\textsc{Savings}_{RHS} &= (n-1) \cdot \eta(w, f) + \sum_{i \in [n]} (i \cdot \eta(y_i, f)) \\ &= (n-1) \cdot \eta(w, f) + \eta(y_1, f) \cdot \sum_{i \in [n]} i \\ &= (n-1) \cdot \eta(w, f) + \frac{n \cdot (n+1)}{2} \cdot \eta(y_1, f)\end{aligned} \tag{4.9}$$

Conversely, equation 4.10 shows the savings incurred upon including the nodes on the path on the left hand side. For ease of calculation we may assume, without loss of generality, that n is even. These savings consist of node w now being able to serve its own demand and additionally, half of the nodes on the right hand path will switch to w and thus save $\frac{n}{2}$ on average per read access.

$$\begin{aligned}\textsc{Savings}_{LHS} &= n \cdot \eta(w,f) + \sum_{i \in [\frac{n}{2}]} (\frac{n}{2} \cdot \eta(y_i, f)) \\ &= n \cdot \eta(w,f) + (\frac{n}{2})^2 \cdot \eta(y_1, f)\end{aligned} \tag{4.10}$$

Now assume we have $\textsc{Savings}_{RHS} < \textsc{Savings}_{LHS}$ and the algorithm therefore decides to include the left hand path as shown in figure 4.9b.

$$\begin{aligned} & \textsc{Savings}_{RHS} < \textsc{Savings}_{LHS} \\ \iff \quad & (n-1) \cdot \eta(w,f) + \frac{n \cdot (n+1)}{2} \cdot \eta(y_1, f)) < n \cdot \eta(w,f) + (\frac{n}{2})^2 \cdot \eta(y_1, f) \\ \iff \quad & \frac{n \cdot (n+1)}{2} \cdot \eta(y_1, f)) < \eta(w,f) + (\frac{n}{2})^2 \cdot \eta(y_1, f) \\ \iff \quad & \frac{2n \cdot (n+1) - n^2}{4} \cdot \eta(y_1, f)) < \eta(w,f) \\ \iff \quad & \frac{n}{2} \cdot (1 + \frac{n}{2}) \cdot \eta(y_1, f)) < \eta(w,f)\end{aligned}$$

The algorithm will make this choice as long as $\eta(w,f)$ is more than $\frac{n}{2} \cdot (1 + \frac{n}{2})$ times the value of any of the pairwise equal y_i's read demand
Note how under the above condition there is nothing to prevent us from assuming that $\forall i \in [n] : \omega(u,f) < \eta(y_i, f)$. This would guarantee that after the inclusion of w via the left hand path algorithm 3 would continue to include all of the nodes y_i for $i \in [n]$ in the following iterations and reach the state depicted in 4.9c. Obviously, since all of the y_i were included anyway, the optimal choice would have been to choose the right hand path in the first iteration and then go on to include w in the second iteration right before termination. We call this optimal result G'_f with $V'_f = \{u, w, y_1, \ldots, y_n\}$ and $c(E'_f) = n+1$. The algorithm however provides a solution G_f with $V_f = V$ and $c(E_f) = 2n$.

$$\lim_{n \to \infty} \frac{c(E_f)}{c(E'_f)} = \lim_{n \to \infty} \frac{2n}{n+1} = 2 \tag{4.11}$$

This therefore serves as an example that there exist situations for which algorithm 3 may produce results that are close to twice as costly compared to an optimal solution.

Note how the algorithm produced $V_f = V$ which is a special case discussed in subchapter 4.1. In our example, the algorithm did not include the last edge to prevent a cycle from forming such that G_f became a Minimum-Spanning-Tree of G. However if one of the other edges $e \in E$ would have been a little bit costlier, e.g. $c(e) = 1 + \epsilon$, then the resulting G_f would not have been a Minimum-Spanning-Tree. This could be easily remedied via introducing a check at the end whether indeed $V_f = V$ and just outputting a Minimum-Spanning-Tree of G as G_f.

As a side note, for large enough n, the algorithm would still continue to include many of the y_i even if we had $\forall i \in [n] : \eta(y_i, f) < \omega(u, f)$ since the demand of multiple nodes y_i incoming through a single path could still offset the costs caused by $\omega(u, f)$.

One may also think about several optimizations or additional checks for algorithm 3 to be able to satisfyingly recognize and prevent the displayed behavior. However, those checks would need to become increasingly more complex if one could not just simply discard some nodes after an iteration (they may become useful for later connections or switches) or if one thinks about how loops within other loops of a graph might behave.
The same counts for fixes to algorithm 2 for the problem presented in subchapter 4.3.1. Such fixes would quickly need to explore all possible paths between nodes while avoiding loops leading to the effort put into those fixes quickly becoming exponential in nature and thereby defeating the purpose of finding a reasonable approximation algorithm.

Upper Bound

We continue to state G_f as the result of algorithm 3 and G'_f as an optimal solution. With the example just presented above we know that the upper bound for $\frac{\text{Cost}_{G_f}}{\text{Cost}_{G'_f}}$ is surely not lower than 2.

Recalling that an optimal G_f should form a Steiner tree for $\mathcal{H}(f) \subseteq V$ on G, we look back on our mention of [13] in subchapter 3.2. The authors create an approximated Steiner tree on G for $S \subseteq V$ by always connecting the next closest vertex remaining in S to the already included vertices via the shortest path. The resulting tree approximates the sum of the costs of the included edges by a factor of at most 2 of an optimal Steiner tree. This approach is reminiscent of algorithm 3 in its style and greedy behavior. However, the important difference is that we do not know $\mathcal{H}(f)$ (or S in the authors' case) beforehand but we try to identify $\mathcal{H}(f)$ and the way its nodes should be connected within G as we go along in the algorithm. Attempts to adapt the algorithm and proof from [13] for our purposes sadly weren't successful.

We deem that in order for algorithm 3 to give a result with $\textsc{Cost}_{G_f}$ higher than $2 \cdot \textsc{Cost}_{G'_f}$, it would need to include a path $P \subset E$, that makes one or multiple other paths already included in G_f superfluous. Those other paths would further need to have a totaling cost larger than $c(P)$ because we would otherwise stay within a 2-approximation. We were not able to find such an example, however we were sadly also not able to come up with a definitive proof that guarantees that always $\frac{\textsc{Cost}_{G_f}}{\textsc{Cost}_{G'_f}} < 2$.

Having to consider all possible interactions and how multiple nodes might interfere with their write demands and how switches might occur at any possible future point in time presents a considerable challenge in coming up with a proof in a natural way. However, to conclude we state our intuition as to why we think the upper bound to our resulting costs might indeed be 2. To this end, we continue our reasoning from above concerning the mentioned path P. If we can indeed assume (as is not proven) that the algorithm can only perform worse if P is included at a point in time where it makes enough other nodes and edges redundant, then it had to be the case that the possibility of including P only just appeared after the last iteration. Otherwise, P would already have been included earlier on as an extension of another path, thus not being able to make enough other nodes superfluous because they were not yet included. If P only just appeared as a possibility to be included, then this means that P was not part of any of the edge sets of any of the shortest-path-trees rooted at any node in V_f. This would mean that at least one edge $e \in P$ would need to have been in a loop of G but undetected by all shortest-path-trees (like edge $\{w, y_n\}$ in figure 4.9a). As seen above, it might happen that all nodes within such a loop get included in V_f but this worst-case is still bounded by an approximation factor of 2. We therefore believe that indeed $\frac{\textsc{Cost}_{G_f}}{\textsc{Cost}_{G'_f}} < 2$ yet without having a proof to definitely show it.

Chapter 5

Conclusions and Future Work

In this thesis we tried to come up with a replica placement algorithm that also considered the spreading of write updates as a criterion. To that end, we first explored different ways to efficiently distribute updated files in chapter 3. We then went on and investigated different possible algorithms for replica placement in chapter 4 that were all focused on greedily growing an appropriate replication network.

- **Simple GreedyNeighbor**
  This first algorithm follows the natural idea to always include the next node that yields the greatest cost-benefit. However, the problem discussed in subchapter 4.2.2 concerning the algorithm making the wrong choice in some instances left us to abandon this mechanism.

- **GreedyNeighbor on Shortest Paths**
  While this algorithm is an extended version of the one before that fixes the mentioned problem, it still suffered from not being able to reason about the switching behavior of nodes not neighboring the already explored area.

- **GreedyGlobal**
  In the end, we presented yet another algorithm that is able to evaluate all shortest paths extending from the already explored area to the rest of the graph. This lets us reason about the possible inclusion of all nodes in the graph. However, the algorithm lacks knowledge of edges that do not lie on shortest paths. This led to the discovery of a case where the algorithm gives a result with costs close to 2 times the optimal costs.

While the first two algorithms were failed because they could not efficiently deal with effects that might or might not occur during the further exploration of yet undiscovered nodes, the third algorithm did not show any behavior that might indicate it producing a result with costs larger than 2 times the optimal costs. In the end, we gave some intuition as to why we think the last algorithm might indeed never perform worse than in the presented example.

With the ever-growing needs and demands for interconnectivity as well as with increasing file sizes and new technologies it remains to be seen what possible mechanisms might be looked into to enable large-scale file sharing. It is certain that with the popular move to the cloud, even outside the realm of scientific research and read-only datasets, there will need to be new ways to account for the costs of update propagation

While we can clearly state that the continued search for a proof for the approximation ratio of algorithm 3 will give a sensible impression how well we might perform within the confines of the described model, we should also talk about the limitations of said model. Overall lots of the simplifying assumptions are not well grounded in reality and it remains to be seen whether investigations into lifting some of these assumptions get along well with the rest of the presented theories. The points to be considered include:

- The presented model has no concept of simultaneous accesses or read after write dependencies. It was shortly mentioned in subchapter 3.3 how a fixed replication network might be of use to detect simultaneous writes. A mechanism is needed for this detection as well as a strategy for dealing with this event.
- In a real-world setting, we would need to account for a much more dynamic scenario where files are created and deleted and where demands and general access patterns change over time. This could be dealt with via the introduction of some heuristics that periodically resample access patterns and translate them to demands for specific files.
- The introduction of time allows differentiation in access times and a more specific notion of bandwidth as edge weights. For example, if nodes are in geographically different regions, it may be possible to steer away from an aggressive-copy mechanism as means for update distribution between those regions and thus save on write costs.
- The fixed replication network G_f uses every node within it as a replica site for f. This was only possible because we assumed no additional overhead and merely focused on the induced network costs. Furthermore, not all nodes might represent systems with enough storage capacity for us to just assume that we never run out of space. Were this the case, then replica placement for large read-only datasets were as simple as just storing everything everywhere. This assumption was introduced to allow us to focus on finding a balance between read and write costs. However, this assumption cannot hold for every dataset on every set of nodes.

Bibliography

[1] G. Peng, "Cdn: Content distribution network," in *ArXiv, Computer Science*, 2004.

[2] P. Khanchandani and R. Wattenhofer, "The arvy distributed directory protocol," in *SPAA '19: The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, 2019.

[3] R.KingsyGrace and R.Manimegalai, "Dynamic replica placement and selection strategies in data grids— a comprehensive survey," in *Journal of Parallel and Distributed Computing, Volume 74, Issue 2*, 2014.

[4] T. Hamrouni, S. Slimani, and F. B. Charrada, "A survey of dynamic replication and replica selection strategies based on data mining techniques in data grids," in *Engineering Applications of Artificial Intelligence, Volume 48*, 2015.

[5] S.-M. Park, J.-H. Kim, Y.-B. Ko, and W.-S. Yoon, "Dynamic data grid replication strategy based on internet hierarchy," in *Lecture Notes in Computer Science, volume 3033*, 2004.

[6] Y. Yuan, Y. Wu, G. Yang, and F. Yu, "Dynamic data replication based on local optimization principle in data grid," in *Sixth International Conference on Grid and Cooperative Computing*, 2007.

[7] A. Guroob and M. D. H, "Efficient replica consistency model (ercm) for update propagation in data grid environment," in *International Conference on Information Communication and Embedded Systems (ICICES)*, 2016.

[8] X. PengZhi, W. YongWei, H. XiaoMeng, Y. GuangWen, and Z. WeiMin, "Optimizing write operation on replica in data grid," in *Science China Information Sciences volume 54*, 2011.

[9] Y. Sun and Z. Xu, "Grid replication coherence protocol," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings*, 2004.

[10] R.-S. Chang and J.-S. Chang, "Adaptable replica consistency service for data grids," in *Third International Conference on Information Technology: New Generations (ITNG'06)*, 2006.

[11] M. Chlebík and J. Chlebíková, “The steiner tree problem on graphs: inapproximability results,” in *Theoretical Computer Science 406, 3, 207–214*, 2008.

[12] J. L. Byrka, F. Grandoni, T. Rothvoß, and L. Sanità, “Steiner tree approximation via iterative randomized rounding,” in *Journal of the ACM, Volume 60*, 2016.

[13] H. Takahashi and A. Matsuyama, “An approximate solution for the steiner tree problem in graphs,” in *Math. Japonica 24(1980)573-577*, 1980.

[14] J. B. Kruskal, “On the shortest spanning subtree of a graph and the traveling salesman problem,” in *Proc. Amer. Math. Soc. 7 (1956), 48-50*, 1956.

[15] R. C. Prim, “Shortest connection networks and some generalizations,” in *The Bell System Technical Journal (Volume: 36, Issue: 6, Nov. 1957)*, 1957.

[16] B. Donnet and T. Friedman, “Internet topology discovery: a survey,” in *IEEE Communications Surveys and Tutorials, Volume: 9*, 2007.