MITIGATING THE IMPACT OF CORRELATED HARDWARE FAILURE ON DATA AVAILABILITY THROUGH SURVIVABLE REPLICA PLACEMENT

by

K. Alex Mills

APPROVED BY SUPERVISORY COMMITTEE:

Neeraj Mittal, Chair

Balaji Raghavachari

S. Venkatesan

I-Ling Yen
Dedicated to my mother,

who probably won’t understand

a word of this...
MITIGATING THE IMPACT OF CORRELATED HARDWARE FAILURE ON DATA AVAILABILITY THROUGH SURVIVABLE REPLICA PLACEMENT

by

K. ALEX MILLS, BSSE

THESIS
Presented to the Faculty of
The University of Texas at Dallas
in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE IN
COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS
May 2013
ACKNOWLEDGMENTS

First of all, I would like to thank God for His continuous grace and mercy over my life. Next, I would like to thank Dr. Mittal for his support as my advisor. He has given me the opportunity to perform this research, and has served as a helpful guide concerning viable research directions. I must also acknowledge Dr. Raghavachari’s insight concerning the method for visiting subproblems, and for many long, fruitful discussions. I would also like to thank Dr. Chandra and Dr. Venky for their helpful comments and especially for their patience concerning frequent visits outside of their regular office hours. We would also like to thank Dr. Shashidhar Gandham, a Ph.D graduate from UT Dallas currently working at Google, for his helpful suggestions concerning applications of our failure model to network failures. I cannot close without mentioning the support I have received from the Erik Jonsson School of Engineering and Computer Science by way of a fellowship which has given me the opportunity to perform this research. Finally, I wish to thank my family. Without their support, I would not have been able to come this far. Thank you all.

April 2012
PREFACE

This thesis was produced in accordance with guidelines which permit the inclusion as part of the thesis the text of an original paper or papers submitted for publication. The thesis must still conform to all other requirements explained in the “Guide for the Preparation of Master’s Theses and Doctoral Dissertations at The University of Texas at Dallas.” It must include a comprehensive abstract, a full introduction and literature review, and a final overall conclusion. Additional material (procedural and design data as well as descriptions of equipment) must be provided in sufficient detail to allow a clear and precise judgment to be made of the importance and originality of the research reported.

It is acceptable for this thesis to include as chapters authentic copies of papers already published, provided these meet type size, margin, and legibility requirements. In such cases, connecting texts which provide logical bridges between different manuscripts are mandatory. Where the student is not the sole author of a manuscript, the student is required to make an explicit statement in the introductory material to that manuscript describing the student’s contribution to the work and acknowledging the contribution of the other author(s). The signatures of the Supervising Committee which precede all other material in the thesis attest to the accuracy of this statement.
In distributed storage systems, data availability is a chief concern. Many time-sensitive applications require access to data within certain time bounds. While many solutions have been proposed which address the failure of an individual hard drive or the corruption of a file, only a few solutions have considered the impact of hardware failure on data availability elsewhere in the data center. In this thesis, we present a model for correlated failure based upon a hierarchical relationship between multiple points of failure. Our model is generic in the sense that it makes no assumptions concerning the hardware or system domain — it can be applied to hierarchical failure in multiple contexts. We provide a dynamic programming algorithm which computes an optimal placement of a single data replica with respect to our hierarchical failure model in polynomial time. Finally, the limitations of our failure model are discussed, along with some improvements we plan in our future work.
# TABLE OF CONTENTS

**ACKNOWLEDGMENTS** ....................................................... v

**PREFACE** .................................................................. vi

**ABSTRACT** ................................................................. vii

**LIST OF FIGURES** ........................................................... ix

**LIST OF NOTATIONS** ........................................................ x

**CHAPTER 1 INTRODUCTION** ............................................... 1

**CHAPTER 2 FAILURE POINT MODELS** .............................. 7

  2.1 Failure and Dependency ............................................... 7

  2.2 Survivable Replica Placement ....................................... 10

  2.3 Failure Models Described by Trees ............................... 12

**CHAPTER 3 A DYNAMIC PROGRAMMING SOLUTION** .......... 16

  3.1 Cut-Placement Correspondence ...................................... 16

  3.2 Optimal Substructure .................................................. 21

  3.3 A Dynamic Programming Algorithm .............................. 25

  3.4 Retrieving the Valid Cut of Minimum Size ...................... 29

**CHAPTER 4 CONCLUSIONS** ............................................... 33

  4.1 Solving $k$-SRP ......................................................... 34

  4.2 Integer Capacity Nodes ............................................... 36

**APPENDIX** .................................................................. 40

  A.1 Subroutine Definitions ............................................... 40

  A.2 Additional Observations ............................................ 40

**REFERENCES** ............................................................... 42

**VITA**
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>An illustration of an undesirable replica placement.</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>An illustration of a desirable replica placement.</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>An example of a failure point model as described by a directed acyclic graph.</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>An illustration of a formal failure point model.</td>
<td>11</td>
</tr>
<tr>
<td>3.1</td>
<td>An example of a tree with a transverse cut.</td>
<td>17</td>
</tr>
<tr>
<td>3.2</td>
<td>An illustration of tree decomposition.</td>
<td>22</td>
</tr>
<tr>
<td>3.3</td>
<td>Pseudo-code for a dynamic program which computes the minimum size of any valid transverse cut.</td>
<td>27</td>
</tr>
<tr>
<td>3.4</td>
<td>Pseudo-code for subroutine MAX-A</td>
<td>28</td>
</tr>
<tr>
<td>3.5</td>
<td>Pseudo-code for a dynamic program which computes the optimal size transverse cut.</td>
<td>31</td>
</tr>
<tr>
<td>3.6</td>
<td>Pseudo-code for subroutine RETRIEVE-CUT</td>
<td>32</td>
</tr>
<tr>
<td>4.1</td>
<td>A counter-example to the optimization of $\rho$-SRP using a naïve approach.</td>
<td>35</td>
</tr>
<tr>
<td>4.2</td>
<td>An example of the reduction described by Theorem 4.2.1</td>
<td>37</td>
</tr>
</tbody>
</table>
LIST OF NOTATIONS

$B_I(U)$ Set of inner border nodes of transverse cut $U$.

$B_O(U)$ Set of outer border nodes of transverse cut $U$.

$c(P,k)$ Number of $k$-safe nodes in a graph given placement $P$.

$ch(n)$ Set of children of node $n$, a member of some tree.

$C(T)$ Set of transverse cuts of tree $T$.

$D(u)$ Dependence set of vertex $u$.

$e(U)$ Number of edges which cross transverse cut $U$.

$\sigma(u,P)$ Survival number of node $u$ given placement $P \subseteq S$.

$L$ Set of leaf nodes in a given tree.

$\rho$ Replication factor for a given model.

$S$ Set of storage nodes for given failure model, $G = (V, E)$. $S \subseteq V$.

$U$ A transverse cut of some tree.

$V_k$ Set of $k$-safe nodes for a tree with vertex set $V$.

$G = (V, E)$ Graph $G$ with vertex set $V$ and edge set $E$.

$T = (V, E)$ A tree, $T$ with vertex set $V$ and edge set $E$.

$\mathbb{Z}^+$ Set of positive integers.
CHAPTER 1
INTRODUCTION

Consider a distributed data center, where many blocks of data are stored on thousands of servers. The owners of this data center are being paid both to store data for their customers and to ensure that the data is available whenever they may need to access it. In today’s competitive market, many customers have strict requirements concerning the quality of service which the data center must provide. These requirements are formalized in a contract, agreed to by both parties, which states the minimum level of service which must be met by the data center. In industry, this contract is known as a Service Level Agreement (SLA). SLAs typically include agreed upon requirements concerning various performance indicators including system availability, downtime, unplanned outages, and mean time to failure (MTTF), to name only a few (Wustenhoff, 2002). For time sensitive applications, these performance indicators may also include the amount of time taken to service a request.

For many customer applications, data availability is a key requirement. In their book on high availability systems, Piedad and Hawkings state that availability encompasses four key elements, which we include here verbatim (Piedad and Hawkins, 2008):

1. Reliability – the ability to perform under stated conditions for a stated period of time
2. Recoverability – the ability to easily bypass and recover from component failure
3. Serviceability – the ability to perform effective problem determination, diagnosis, and repair
4. Manageability – the ability to create and maintain an environment that limits the negative impact people may have on the system.
Each of these four elements impacts data center management. As an example, consider reliability. In 2010, Google published results of an extensive one year study on data availability in their cloud storage infrastructure. In this study, they found that “a large fraction of failures happen in bursts”, indicating that, rather than failing one at a time, servers tend to fail in groups. Furthermore, the study also found that a large majority of these failures are “associated with rack- or multi-rack level events”, indicating that the cause of failure is not due to the failure of a single device on the server, but due to some other failure event which has implications for multiple servers throughout the data center. Google concluded that “correlation among node failures dwarfs all other contributions to unavailability in our production environment” (Ford et al. 2010).

How can we mitigate the impact of these correlated failures on data availability? In order to answer this, we must first identify the impact of correlated failures. We will accomplish this by illustrating a few undesirable scenarios. Consider two scenarios where three identical replicas of the same block of data are distributed on nodes throughout the data center. Each node receives power from a surge protector which is located on each server rack. In scenario A, each replica is located on nodes which share the same rack. In scenario B, each replica is located on separate racks. As can be seen from the diagram of Scenario A (Figure 1.1), a failure in the power supply unit on a single rack could result in a situation where an data block is completely unavailable, whereas in scenario B, (Figure 1.2) three power supply units would need to fail in order to achieve the same result.

Network failures may also have an impact on the availability of data. Consider a time-critical application where data must be made available within a few seconds. Consider also that a large volume of requests are made on a consistent basis. The network fabric must be able to accommodate these requests. In a typical data center, each rack of servers is equipped with a top of rack (ToR) switch, which links to the aggregation layer. The failure of these ToR switches involves a decrease in the bandwidth available to the servers on the
Figure 1.1. Scenario A — Identical replicas of a single data-block, labeled A, B, and C, are placed on the servers indicated. As mentioned in the text, the failure of a single rack-level, or switch-level component could result in the potential loss of availability of this block of data.

Figure 1.2. Scenario B — Identical to Scenario A, except that the replicas have been placed differently. Notice that multiple failure events are required to make the block of data corresponding to the replicas unavailable.
rack. Consider the same scenarios as before. It should be clear that in scenario A, a single rack-level failure event would decrease the amount of bandwidth available to transfer the data block, and in scenario B, multiple rack-level events are required in order to achieve the same result. Therefore, it’s possible that failures in the network switching fabric can adversely affect the recoverability of data in time-sensitive scenarios.

Both of these scenarios can be avoided by ensuring that each replicas is placed on nodes which lie on separate racks, however, what may not be immediately clear is that this simple heuristic can be suboptimal under certain conditions. Extending our first illustration, consider a failure in the power supply unit which services multiple racks. Such a failure could potentially impact the availability of every data replica on the rack. Furthermore, these power supply units could depend on the proper performance of one or more power transformers in the data center, which may themselves depend upon other power supply components. A failure in any one of these components could impact the availability of one or more data blocks.

While designers of data placement schemes are aware of the problem of correlated failure, very few data placement schemes surveyed have considered the problem of finding an optimal data placement strategy. CRUSH, a distributed object-based storage system, allows the user to configure a set of data placement rules (Weil et al., 2006). These rules serve as a configurable strategy which allows the user to indicate which row, rack, and server a set of replicas should be stored on. We believe that this places the burden of finding an optimal solution on the user, which could lead to user-error, or the use of a sub-optimal solution in order to save development time. RADPA attempts to make an optimal assignment of data replicas to storage nodes, solving an integer linear program to do so (Chen et al., 2009). However, in order to use RADPA one must measure the reliability of each device and determine requirements for the minimum reliability of each data block. In practice, these measures may not be available, or may be estimated, which impacts the reliability of the
solution. While RADPA focuses on placing replicas to meet availability constraints, it is not intended to model correlated failure. We present these examples as evidence that the problem of making effective data placements has been previously studied. However, as part of our literature survey we were unable to find any work in which optimal replica placement was used to mitigate the effects of correlated failure.

A reasonable alternative to using data placement strategies is to increase the reliability of the hardware itself, typically by adding redundancy. Unfortunately, this approach can only improve the aggregate reliability so far. Even in a scenario where each individual piece of hardware in the data-center lasts on average one-million hours before failing, when many hardware components of this quality are incorporated into a large-scale system the odds of a single device failing dramatically increases. Specifically, when 64,000 or more of these devices are running simultaneously, data center managers can expect to see, a single failure per day on average (Reed et al., 2006). We emphasize that this phenomenon is not merely the product of theory, it has also been observed in practice. In 2009, Google reported daily failures of their hardware devices (Dean, 2009). In 2011, Microsoft published a study concerning the impact of redundancy in network hardware. Their study concluded that “network redundancy is only 40% effective in reducing the median impact of failure” (Gill et al., 2011). Furthermore, they noticed that ToR switches are more susceptible to correlated failure than other network components. We propose that in scenarios like these, where correlated failure is ubiquitous, any strategy for data availability should include the intelligent placement of data replicas.

In this thesis, we investigate the problem of making an intelligent placement of data replicas to improve data availability. In particular, we focus on techniques for ensuring high availability under a model which takes correlated hardware failure into account. The question we seek to answer is as follows: How can we mitigate the effects of correlated hardware failure on data availability by making an intelligent placement of data replicas?
We propose a hierarchical model for correlated failure, and show that, for our model, we can find an optimal placement of $\rho$ replicas of a single data-block.

The remainder of the thesis is organized as follows. In Chapter 2 we will describe a novel technique for modeling hierarchical failure and introduce the problem of Survivable Replica Placement (SRP). In Chapter 2 we will narrow our focus and consider the problem of solving a special case of SRP when the model given is a tree. Chapter 3 concerns the algorithm itself, which is an application of dynamic programming and a combinatoric analysis of the problem. Finally, we conclude in Chapter 4 with a brief discussion of extensions to our model, and some open problems.
CHAPTER 2
FAILURE POINT MODELS

In Chapter 1, we illustrated two scenarios where data replicas were placed on servers in a data center. We gave an informal argument as to which scenario was preferable, based upon the potential for failures effecting the availability of the data-block. In this chapter, we will formalize these notions, and describe what we term as failure-point models. Our description will focus on the assumptions inherent in our model, in order to ensure that we have clearly communicated exactly where it can be applied. Once we have described the model, we will give a precise definition of the intuitive notion of “preferable placement of data replicas” from the previous chapter.

This chapter is organized into three sections. Section 2.1 describes the types of failure we expect to encounter, and describes our notion of dependency which lays the foundation for the problem which we wish to solve. In Section 2.2 we describe the problem of placing replicas on storage nodes in a data center. Sections 2.1 and 2.2 concern themselves with a general model of dependency which can be represented by a directed acyclic graph, however our algorithm is only concerned with the case where the dependency model is a tree. In this case, we are immediately able to derive some useful properties, the discussion of which comprises the content of Section 2.3.

2.1 Failure and Dependency

A distributed system is made up of multiple hardware entities acting in concert to achieve a goal. Each of these entities has a set of responsibilities which it must fulfill in order for the system as a whole to function properly. As designers of distributed systems, we would like to
ensure that when a single hardware entity fails to perform one or more of its responsibilities that the system’s goal can still be achieved. In general, when a hardware entity fails to perform as expected we say that it has failed. As this definition is exceedingly broad, we further classify failure into several types. Within distributed systems there are at least three models of failure which are widely accepted (Garg, 2002). Below, we paraphrase these classes in terms of hardware entities as we have described them above.

- **Crash Failure** – occurs when a hardware entity ceases to perform all of its responsibilities.
- **Omission Failure** – occurs when a hardware entity only performs a subset of the responsibilities expected of it.
- **Byzantine Failure** – occurs when a hardware entity exhibits behavior which appears arbitrary.

For the purposes of the discussion which follows, we make the assumption that all failures are crash failures. However, we note that any of these classes of failures may adversely impact data availability. For instance, in time-sensitive applications the omission of a single packet of data could cause a transaction to miss its deadline, invoking an SLA violation for which the data center could be required to reimburse their customer.

In order to account for the impact which failure of a single entity may have on other entities in the system we will define what we call a failure point model. A failure point model consists of a set of hardware entities represented as failure points. A failure point is simply an entity which is capable of failure. Additionally, in order to ensure its own proper operation each entity requires that other entities on which it depends continue to function.

---

1Garg also mentions Crash + Link Failure, however we omit this case because for the purposes of our model we consider links to be first-class hardware entities. In our model, we can express the failure of a link as the failure of a node.
properly. A specific example of this from data centers is the relationship between a server and its top of rack (ToR) switch. The server is responsible for serving network requests. To do so, it routes its packets through the ToR switch. In such a scenario, the server depends upon the proper functioning of the ToR switch in order to service its request. If the ToR switch fails, there is no longer any guarantee that the server can fulfill its own responsibility.

When two entities participate in such a dependency relationship, we say that entity A depends on entity B. If a dependency relationship between A and B exists and entity B fails, we say that A is at risk of failure. Clearly, given that B has failed, we cannot guarantee the successful operation of A. At the very least, its success is in jeopardy because there is no longer any guarantee that B will perform all of its operations as expected. For the remainder of this thesis we will take the conservative approach and consider the worst-case failure scenario. That is, we assume that if entity B fails then entity A fails as well.

This set of hardware entities and dependence relationships can be represented as a directed graph, $G = (V, E)$, where $V$ is the set of failure points, and $(u, v) \in E \iff v$ depends on $u$. We will discuss several assumptions concerning this graph. First of all, we make the assumption that $G$ is acyclic. While there may be situations in practice which introduce cyclic dependencies among failure points, such situations introduce theoretical issues which are beyond the scope of this thesis. We also assume that there is some node $r \in V$ such that there is a path from $r$ to every other node $u \in V$. The failure of $r$, can be interpreted as representing a catastrophic failure which effects every other node in the system. In terms of our application to data centers, $r$ could represent a disaster scenario such as an earthquake or flood, during which the availability of all data in the data center is at risk. See Figure 2.2 for an illustration of this model.

Formally, we say that if a node $u$ fails, then every node $v$ such that there exists a path from $u$ to $v$ also fails.\footnote{Notice that this implies that the dependency relationship is transitive.} Said another way, every node which is reachable from $u$ fails.
Figure 2.1. An example of a failure point model described by a directed acyclic graph. An edge between nodes $u$ and $v$ indicates that $v$ depends on $u$ for its successful operation. $D(a)$ is the set of nodes which will fail if $a$ fails, i.e. the dependence set of $a$.

definition of failure induces what we call the dependence set of node $v$. A node, $v$, is in the dependence set of node $u$ if and only if the failure of $u$ implies the failure of $v$. That is to say, the dependence set of $u$ is exactly the set of nodes in $V$ which are reachable from $u$. We denote the dependence set of $u$ by $D(u)$.

We conclude this section with some summarizing definitions.

**Definition 2.1.1.** A failure point model is defined as a directed, acyclic graph $G = (V,E)$, where $V$ is a set of failure points, and $(u,v) \in E \iff v$ depends on $u$.

**Definition 2.1.2.** Given a failure point model $G$, the set of nodes which are reachable from $u$ is termed the dependence set of $u$, denoted $D(u)$. The failure of $u$ implies the (possible) failure of every $v \in D(u)$.

### 2.2 Survivable Replica Placement

In order to discuss replica placement we must first introduce our data replication model. For convenience, we consider that the data center is required to store $B$ unique blocks of data.
It is common practice in data centers to store several replicas of each block of data, both to ensure high availability and to ensure that data is retained in case of hardware failure. We presume that the data center stores $\rho$ replicas of each data block, and that the data center has sufficient capacity to store at least $\rho \cdot B$ blocks of data.

Given a particular failure point model represented by graph $G$, and a subset of nodes, $S \subseteq V$, which are designated as storage nodes, we would like to make a placement of replicas on the nodes of $G$ such that the failure of any single node impacts the availability of each data block as little as possible. More precisely, for each data block, we are required to select $\rho$ nodes from $S$ on which these data blocks will be stored. We seek a subset $P \subseteq S$ with $|P| = \rho$, termed a *placement*, which satisfies certain properties to be explained shortly.

At this point, for the purpose of clarity, we will illustrate a failure point model derived from the data center used in the examples illustrated in the previous chapter. In Figure 2.2, one can see the failure point model constructed from the scenarios depicted on page 3.

To quantify the differences between Scenarios A and B from the previous chapter, we need to derive some function, $c$, which maps every subset of $S$ to a real number. This is the goal of the discussion in the remainder of this section.
Given a subset of $S$ and a node $u \in V$, define the survival number of $u$ as the number of replicas which are guaranteed to survive if node $u$ fails. Formally, if $P \subseteq S$ is the set of storage nodes and $u \in V$, then,

$$\sigma(u, P) = |P - (P \cap D(u))|.$$ 

In terms of our previous discussion, $\sigma(u, P)$ is the number of nodes in $P$ which are not also in the dependence set of $u$.

In the context of a placement $P$, we say that a node $u \in V$ is $k$-safe if it has $\sigma(u, P) \geq k$, and $k$-unsafe if it has $\sigma(u, P) < k$. Equivalently, node $u$ is is $k$-safe if at least $k$ replicas would survive if $u$ were to fail. Given this definition of $k$-safety, we can describe the $k$-safety of an entire placement, $P$, by counting the number of nodes which are $k$-safe. Formally, given placement $P$, and an integer $k \in \mathbb{Z}^+$ let $c(P, k)$ be the number of $k$-safe nodes under placement $P$. Note that the concrete definitions of functions $c$ and $\sigma$ implicitly depend upon the definition of graph $G$.

**Definition 2.2.1.** Given a failure model $G = (V, E)$ with storage nodes $S \subseteq V$, and placement $P \subseteq S$, define the survival number of $u \in V$, as $\sigma(u, P) = |P - (P \cap D(u))|$. 

**Definition 2.2.2.** Given a failure model $G = (V, E)$, and placement $P$, any node $u \in V$ with $\sigma(u, P) \geq k$ is termed $k$-safe. Any node which is not $k$-safe, is termed $k$-unsafe.

**Definition 2.2.3.** Given a failure model $G = (V, E)$, and placement $P$, $c(P, k)$ is the number of $k$-safe nodes in $G$ under placement $P$. 

$$c(P, k) = |\{ u \in V \mid \sigma(u, P) \geq k \}|$$

2.3 Failure Models Described by Trees

We now turn our attention to the special case where $G$ is a tree. To do this, we will require the following properties of graph $G$. 

**Property 2.3.1.** For each \( s_i \in S \), there is no node \( v \) such that \((s_i, v) \in E\).

**Property 2.3.2.** For every leaf node, \( \ell \) there exists a unique path from the root to \( \ell \).

The first property ensures that every storage node is a leaf node, the second property is the standard tree property for any graph. In this section, we will present some elementary results concerning the properties of \( \sigma \) and \( c \) for the special case in which our failure point graph satisfies the properties above. To emphasize the difference, we will denote trees using variable \( T \) (rather than \( G \)) for the remainder of this thesis.

Our first set of properties are intuitive. If \( T \) is a tree, then for any \( \rho > 0 \), no placement \( P \), with \(|P| = \rho\) may ensure that all nodes in \( T \) are \( \rho \)-safe. A single node could always be made to fail by an adversary to ensure that some replica fails. In fact, if the adversary were particularly lazy, they would simply choose the root of the tree, which always has survival number 0 for any non-empty placement. Therefore, if even one replica is placed on the tree, some node must be \( \rho \)-unsafe.

The following two lemmas should also be immediately apparent.

**Lemma 2.3.1.** For any tree \( T \) and placement \( P \), if node \( u \) has successor \( s \) then \( \sigma(s, P) \geq \sigma(u, P) \).

**Lemma 2.3.2.** For any tree \( T \) and placement \( P \), if node \( u \) has ancestor \( a \), then \( \sigma(a, P) \leq \sigma(u, P) \).

We can convince ourselves of the validity of the above lemmas by considering the child of some node \( u \in V \), denoted \( c \). Consider Lemma 2.3.1 and suppose that when \( c \) fails, three replicas survive. Notice that when \( u \) fails, \( c \) must also fail, therefore the survival number of \( u \) can be no less than that of \( c \). A similar argument can also be made for Lemma 2.3.2.

These lemmas suggest that the survival number of a node is hereditary in some sense. This immediately implies the following corollary to Lemmas 2.3.1 and 2.3.2 concerning \( k \)-safety (and \( k \)-unsafety).
Corollary 2.3.1. In a tree, if any node is $k$-safe then all of its descendants are also $k$-safe, and if any node is $k$-unsafe then all of its ancestors must also be $k$-unsafe.

These lemmas also indicate something more interesting, at least in a theoretical sense. Namely, they indicate that any selection of placement $P \subseteq S$ induces a partial order on the set of failure points in some sense. This partial order shall prove indirectly useful as we consider techniques for finding an optimal placement of $\rho$ replicas.

We have now come to the point in our discussion where we can completely define the problem which we would like to be able to solve. The problem is as follows: given a failure-point model represented by a tree, $T$, and a set of storage nodes, $S$, which are the leaves of $T$, we wish to place $\rho$ replicas on $\rho$ nodes of $S$ such that the number of $(\rho - 1)$-safe nodes in $T$ (i.e., $c(P, \rho - 1)$) is maximized. We focus on $(\rho - 1)$-safety rather than $\rho$-safety for reasons which are explained in Chapter 4.

Problem 2.1. Survivable Replica Placement (SRP) — Given tree $T$ with leaf nodes $L$, find $P^* \subseteq L$, with $|P^*| = \rho$ such that,

$$P^* = \arg \max_{P \subseteq L} c(P, \rho - 1).$$

Rather than immediately focus our attention on this problem, we shall perform several simplifications and then combine them to achieve our goal of solving Problem 2.1.

For the sake of completeness, consider a variant of Problem 2.1 where we are asked to find $P^*$ which satisfies:

$$P^* = \arg \max_{P \subseteq L} c(P, k).$$

Suppose $k = 1$. For any tree $T$ with root $r$, if $r$ has at least 2 children, two of which are denoted by $a$ and $b$, then any placement which includes at least one replica assigned to some leaf node in the subtree rooted at $a$, and another replica to some leaf node in the subtree rooted at $b$ will be an optimal solution to our problem. To see this, notice that if node $a$ were
to fail, then at least one replica would survive — namely the replica in the subtree rooted at $b$. A similar situation occurs if node $b$ were to fail. Likewise, if any other child of $r$ were to fail, then at least two replicas would survive (one each in the subtrees rooted at $a$ and $b$ respectively). Therefore, for this placement, $P$, by Lemma 2.3.1, every node $n \in V - \{r\}$, has $\sigma(n, P) \geq 1$, which implies that any such $P$ is optimal for the case when $k = 1$. If $r$ instead had only one child, we would recursively consider its child to be the root until we find a node which has at least two children.

In the preceding discussion we were able to find $P^*$ when $k = 1$ by considering a single case, (when $r$ has two children). A similar brute-force approach can be taken to construct optimal solutions for $k = 2, 3, ..., \rho$, however, as $k$ increases, the number of special cases which must be considered increases exponentially when using such an algorithm. As it turns out, we can solve the case where $k = \rho - 1$ (i.e. Problem 2.1) in polynomial time, after proving some additional properties concerning the structure of our problem. However, before we do so, we will need to perform some further analysis concerning survival numbers, which is one of the main concerns of the next chapter.
CHAPTER 3
A DYNAMIC PROGRAMMING SOLUTION

In this chapter, we develop an algorithm for solving the $k$-SRP problem for the case where $k = \rho - 1$. The majority of our analytical effort will be focused on proving a correspondence between desirable placements and minimum cardinality transverse cuts, which are defined in Section 3.1. Once this theoretical analysis has been performed, we will arrive at a polynomial time solution via an application to dynamic programming. Dynamic programming is an algorithmic technique attributed to Richard Bellman for solving problems which exhibit optimal substructure with overlapping subproblems (Cormen et al., 2001b). In Section 3.2, we will apply this technique by constructing a recurrence which we can use to solve $(\rho - 1)$-SRP. We will prove this recurrence correct, and show how to compute values for the recurrence in polynomial time in Sections 3.3 and 3.4.

3.1 Cut-Placement Correspondence

As the title of this section suggests, herein we outline a correspondence between transverse cuts and the number of $k$-safe nodes in a tree. We show that the set of $k$-safe nodes must be exactly the set of nodes in $V$ which are not in a transverse cut with certain properties. This section culminates in a formal proof of the correspondence between transverse cuts having a certain property and the number of $k$-safe nodes in a tree for a certain class of placements.

In a tree $T = (V, E)$ with root $r$, a cut $U$ is a subset of nodes in $V$. Let $V_k$ be the set of $k$-safe nodes in $V$. We say that a cut $U$ is a transverse cut if it is closed under the ancestor relationship. That is to say, for any $u \in U$, any ancestor of $u$ must also be present in any transverse cut. Because of this property, any non-empty transverse cut must contain $r$. For
Given a transverse cut, $U$, we define a **standard placement**, $P$, as any placement of replicas such that, for each $v \in B_O(U)$, at most one replica from $P$ is present in the subtree rooted at $v$. It is easy to see that if such a standard placement $P$ exists then each node in $V - U$ must be $(|P| - 1)$-safe. This notion is formalized in the following Theorem.
Theorem 3.1.1. Given tree $T$ and a transverse cut $U$, if $e(U) \geq \rho$, then a standard placement $P$ such that $|P| = \rho$ exists. Further, $P$ ensures that each node in $V - U$ is $(\rho - 1)$-safe. More precisely, if $V_{\rho-1}$ is the set of $(\rho - 1)$-safe nodes, then $V - U \subseteq V_{\rho-1}$.

Proof. Consider the forest of at least $\rho$ trees which is formed by $V - U$. Select $\rho$ arbitrary trees, and place one replica on the leaves of these trees. This forms a standard placement, $P$, by definition.

Suppose $T_i$ is a tree in $V - U$. Each $T_i$ either has a replica placed on it, or it doesn’t. If $T_i$ doesn’t have any replicas placed on it, then all nodes in $T_i$ are $\rho$-safe, and therefore $(\rho - 1)$-safe as well. If $T_i$ has a replica placed on it, and the root of $T_i$ fails, then $\rho - 1$ replicas must survive. By Corollary 2.3.1, this implies that all nodes in $T_i$ are $(\rho - 1)$-safe. Since the union of all $T_i$ is exactly $V - U$, all nodes in $V - U$ are $(\rho - 1)$-safe, therefore $V - U \subseteq V_{\rho-1}$.

According to Theorem 3.1.1, given any transverse cut $U$ with $e(U) \geq \rho$, we can form a placement $P$ such that all nodes in $V - U$ are $(\rho - 1)$-safe. We observe that there are many such placements which could be made. Viewed in terms of our objective function, this theorem provides us with a lower bound on the possible value of $c(P, \rho - 1)$, given any standard placement $P$ for cut $U$.

A natural next question is to ask whether or not $c(P, \rho - 1) = |V - U|$. If this were the case, then we could maximize $c(P, \rho - 1)$ by minimizing the size of $U$, while ensuring that $e(U) \geq \rho$. We can derive such a result in the usual way by proving that $c(P, \rho - 1) \leq |V - U|$. In order to do so, we will need to impose one additional property on cut $U$, namely that of minimality. For clarity, by a minimal cut $U$, with $e(U) \geq \rho$, we mean a cut $U$ such that no node $u \in U$ can be removed to form $U' = U - \{u\}$, such that $e(U') \geq \rho$ still holds. Note that there may be multiple such minimal cuts in a tree. Given this additional property, we can prove the following theorem.
Theorem 3.1.2. Given tree $T$, for any minimal transverse cut $U$ such that $e(U) \geq \rho$, let $P$ be a standard placement of $\rho$ replicas with respect to $U$, then, the set of $(\rho - 1)$-safe nodes, denoted $V_{\rho-1}$, is precisely $V - U$, that is: $V_{\rho-1} = V - U$

Proof. Let $P$ be any arbitrary, standard placement with respect to $U$. By Theorem 3.1.1 all nodes in $V - U$ are $(\rho - 1)$-safe. That is, $V - U \subseteq V_{\rho-1}$, we will prove that $V - U = V_{\rho-1}$ using a proof by contradiction.

Suppose $V - U \subset V_{\rho-1}$. This implies that there must be some $u \in U$ which is $(\rho - 1)$-safe, yet is not in $V - U$. Without loss of generality$^1$ let $u \in B_I(U)$. Construct $U' = U - \{u\}$, and notice that $u \in B_O(U')$. We claim that $e(U') \geq \rho$, we will prove this fact using contradiction.

Suppose $e(U') < \rho$, then by the pigeonhole principle, some $(\rho - 1)$-unsafe node must exist in $B_O(U')$. To see this, note that $e(U') < \rho$ implies that there are at most $\rho - 1$ nodes in $B_O(U')$. Therefore, since $|P| = \rho$, at least one of the nodes in $B_O(U')$ must have at least two replicas placed on it, and therefore at least one node in $B_O(U')$ must be $(\rho - 1)$-unsafe, i.e.

$$\exists v \in B_O(U') \ni v \notin V_{\rho-1}.$$  \hspace{1cm} (3.1)

However, before $U'$ was formed, we determined that $V - U \subset V_{\rho-1}$. Therefore, every outer border node of $U'$ which is also in $V - U$ must be $(\rho - 1)$-safe, i.e.,

$$(V - U) \cap B_O(U') \subset V_{\rho-1}.$$  \hspace{1cm} (3.2)

Notice that $u$ is the only node currently in $B_O(U')$ which is not also in $(V - U)^2$ and $u$ was also previously shown to be $(\rho - 1)$-safe. Therefore, every node in $B_O(U')$ is $(\rho - 1)$-safe. That is,

$$\forall v \in B_O(U') \ni v \in V_{\rho-1}$$  \hspace{1cm} (3.2)

$^1$If $u \notin B_I(U)$, then we can simply select some child of $u$, $u'$ such that $u' \in B_I(U)$. By Corollary 2.3.1 we are guaranteed that this child is also $(\rho - 1)$-safe

$^2$Since $u \in U$, it cannot possibly be in $V - U$. 

Equations 3.1 and 3.2 form a contradiction, therefore \( e(U') \geq \rho \).

But if \( e(U') \geq \rho \) and \(|U'| < |U|\) then \( U \) must not have been a minimal cut such that \( e(U') \geq \rho \). And so, we have found a contradiction for our original assumption, therefore,

\[
(V - U \not\subset V_{\rho-1}) \land (V - U \subseteq V_{\rho-1}) \implies V - U = V_{\rho-1},
\]

which is our desired result. \(\square\)

Theorem 3.1.2 immediately yields the following corollary.

**Corollary 3.1.1.** For every standard placement \( P \) with respect to a minimal transverse cut \( U \), where \( e(U) \geq \rho \),

\[
c(P, \rho - 1) = |V - U|.
\]

Notice that there are two conditions which are required in order to apply Corollary 3.1.1. \( U \) must have \( e(U) \geq \rho \), and \( U \) must be a minimal. That is, no \( u \in U \) can be removed to form \( U \) while ensuring that \( e(U') \geq \rho \). We say that a *valid* transverse cut \( U \) satisfies the first of these conditions, (i.e. \( e(U) \geq \rho \)).

Given this Corollary 3.1.1 we can reduce Problem 2.1 to the following problem.

**Problem 3.1.** Given a tree \( T \), find a transverse cut, \( U^* \) such that \( e(U) \geq \rho \), and \( U \) has minimum cardinality\(^3\) among all such transverse cuts, i.e.,

\[
U^* = \arg \min_{U : e(U) \geq \rho} |U|.
\]

By Theorem 3.1.2 any standard placement of \( \rho \) replicas on a cut found as solution to Problem 3.1 would yield an optimal solution to Problem 2.1. In the next section, we will show how we can solve Problem 3.1 in polynomial time using dynamic programming, and use this solution to solve SRP.

\(^3\)Recall that any set satisfying some property and having minimum cardinality is also minimal.
3.2 Optimal Substructure

To ensure ease of reading, we choose to focus on a graphical description of transverse cuts first, before providing the recurrence used to compute valid transverse cuts. The recurrence which we will eventually derive follows from a few elementary observations concerning the way in which transverse cuts can be decomposed as the union of several other transverse cuts. There are many ways to decompose a tree, but the method presented here is designed to lead us to the recurrence which we desire.

Consider a tree $T$, in which the root has $n$ children. We aim to mathematically express all possible transverse cuts of $T$ in terms of the transverse cuts of its children. We call the process of arriving at such an expression decomposition. In order to construct a method for decomposing any given tree, we will have to consider three cases. In the first two cases we decompose trees where the root has between zero and one children. In the third case, we consider how to iteratively decompose a tree with an arbitrary number of children. Since the ideas presented in this discussion are intuitive, we will present several observations as lemmas without proof.

Consider the case where $T$ has only one child. Name the subtree rooted at this child $T_1$, and let the set of transverse cuts of a tree $T$ be denoted by $C(T)$.

Lemma 3.2.1. “Up” Case — Consider a tree $T = (V,E)$ formed starting with root $r$ and attaching tree $T_1$, with root $r_1$, by adding $(r,r_1)$ to $E$. All transverse cuts of $T$ can be described as the union of some $U_1 \in T_1$, and $\{r\}$. More precisely, for any $U \in C(T)$, where $U \neq \{r\}$, there exists a transverse cut $U_1 \in C(T_1)$ such that $U = U_1 \cup \{r\}$.

Likewise, we can make a similar observation concerning any tree $T$ with root $r$, to which we graft in a new tree, $T_1$, by attaching the root of $T_1$ as a child of $r$. This observation allows us to inductively “peel off” children until we reach the case where the tree only has one child, at which point we can apply Lemma 3.2.1.
Figure 3.2. An illustration of tree decomposition, as explained in Lemmas 3.2.1 and 3.2.2.
(a) When we consider a tree with a single child, we observe that each transverse cut in tree $T$ can be described as the union of some transverse cut in $T_1$ and the set $\{r\}$. (b) Likewise, when we consider a tree, $T_1$ which has had an additional child “grafted on”, we note that each transverse cut of the new tree, $T$, can be described as the union of some transverse cut in $T_1$ and a transverse cut in $T_2$.

Lemma 3.2.2. “Out” Case — Consider the tree $T = (V,E)$ formed starting with tree $T_1$ having root $r_1$, and attaching tree $T_2$ having root $r_2$ by adding edge $(r_1,r_2)$ to $E$. All transverse cuts of $T$ can be described as the union of some $U_1 \in T_1$, and some $U_2 \in T_2$. More precisely, for any $U \in C(T)$, there exists a transverse cut $U_1 \in C(T_1)$, and a (possibly empty) transverse cut $U_2 \in C(T_2) \cup \{\emptyset\}$, such that $U = U_1 \cup U_2$.

Lemmas 3.2.1 and 3.2.2 are illustrated in Figure 3.2 parts (a) and (b) respectively.

Using Lemmas 3.2.1 and 3.2.2 we can decompose any tree, $T$, starting from the top and “peeling off” nodes one at a time. What may not be immediately apparent is that this decomposition can also be performed in reverse, starting from the leaves of any tree $T$, to “grow the tree” all the way up to the root. Specifically, we can start at any arbitrary leaf node of $T$, i.e. $\ell$ and alternately apply the previous lemmas until every node has been included in the tree. Any such process would start with Lemma 3.2.1 to include the parent of leaf $\ell$, then would repeatedly apply Lemma 3.2.2 until all siblings of $\ell$ had been included. This process then continues by including the grandparent of $\ell$, and so on until the entire tree has been visited. Intuitively, Lemma 3.2.1 allows us to move “up” the tree by including parents,
and Lemma 3.2.2 allows us to move “out” by including siblings. This simple observation will enable us to derive a recurrence which we can use to solve \((\rho - 1)\)-SRP.

Given a tree, \(T = (V, E)\), let function \(A : V \times \mathbb{Z}^+ \to \mathbb{Z}^+\), take as input \(n \in V\), and \(k \in \mathbb{Z}^+\). \(A(n, k)\) is defined as the maximum number of edges which can be obtained by any transverse cut of cardinality \(k\) in the subtree rooted at node \(n\). Suppose that we had calculated \(A(r, k)\), where \(r\) is the root of \(T\), for all \(0 \leq k \leq |V|\). By means of a simple linear scan, we could then find the minimum value of \(k\) for which \(A(n, k) \geq \rho\). Finding values for \(A(r, k)\) only gives us the cardinality of the minimum valid set. However, if during the calculation of \(A(r, k)\) we could store the cut \(U\) which induces the value of \(A(r, k)\) being calculated, we could simply retrieve and return \(U\) once our calculation is finished. Since leaf nodes have no children, and cannot themselves be part of a transverse cut, we will use the convention that if \(\ell\) is a leaf node \(A(\ell, k)\) is undefined for any values of \(k\).

Based upon Lemmas 3.2.1 and 3.2.2 we know that we only need to consider two additional cases, “up” and “out” to construct a recurrence for \(A\) which will cover all possibilities for internal nodes. Therefore, we only need to concern ourselves with two possibilities, each of which takes at most two trees into account.

The “up” case is fairly simple, so we present it first. Refer to Figure 3.2 and recall that \(r_1\) is the root of tree \(T_1\), and that \(T\) is formed by appending \(T_1\) to root \(r\). We assume that \(A(r_1, k)\) has been previously computed for tree \(T_1\), for all \(1 \leq k \leq |V|\). Clearly, for the “up” case, \(A(r, 1) = 1\), since the only possible singleton cut is precisely \(U = \{r\}\), for which only one edge will cross the cut. For every \(k > 1\), \(A(r, k)\) will be the maximum number of edges in \(T_1\) for any cut of cardinality \(k - 1\). Therefore, \(A(r, k)\) is defined as follows:

\[
A(r, k) = \begin{cases} 
1 & \text{if } k = 1, \\
A(r_1, k - 1) & \text{if } k > 1. 
\end{cases}
\]  (3.3)
The “out” case takes a bit more thought. In this case, there are three trees to consider. Tree $T$ is formed by combining trees $T_1$ and $T_2$ (having roots $r_1$ and $r_2$ respectively) by adding an edge from $r_1$ to $r_2$. We define $A(r_1, k)$ as follows,

$$A(r_1, k) = \begin{cases} A(r_1, 1) + 1 & \text{if } k = 1, \\ \max \left\{ \max_{i+j=k, \ i\neq 0, j\neq 0} [A(r_1, i) + A(r_2, j)], A(r_1, k) + 1 \right\} & \text{if } k > 1. \end{cases} \quad (3.4)$$

We encourage the reader to make constant reference to Figure 3.2 in the discussion that follows. The only possible value for the singleton cut, $\{r_1\}$ is precisely the same as the singleton cut previously computed in $T_1$ (i.e. $A(r_1, 1)$), except that the additional edge from $r_1$ to $r_2$ must also be added. In order to compute $A(n, k)$, we must check every possible combination of cuts in $T_1$ and $T_2$ and determine the maximum. To accomplish this, we use $A(r_1, i) + A(r_2, j)$, for all non-zero values of $i$ and $j$ which sum to $k$. Were we to use this value on its own, we would leave out an important case, the case where only a single cut from $T_1$ is used. To account for this case, we take the maximum of $A(r_1, k) + 1$ and the previous formula.

We have just defined $A$ recursively. So far, for ease of reading, we have considered that $1 \leq k \leq |V|$, however, for certain values of $k$, it could be the case that $A(n, k)$ is undefined. Recall that $A(n, k)$ is undefined for leaf nodes. The recurrence above implies that the value of $A(n, k)$ for an interior node $n$ may eventually depend upon the value of some leaf nodes. We resolve this by ensuring that for any $n \in V$, if $A(n, k)$ requires a previous value of $A$ which is undefined, then $A(n, k)$ itself is undefined. Furthermore, all arithmetic operations on undefined values are left undefined, and when taking the maximum operation we consider undefined values to be less than any defined value. We give an informal argument that $A(r, k)$ will be defined for any possible set of values we could reasonably desire. More specifically,

**Proposition 3.2.1.** For any tree $T = (V,E)$, with root $r$, $A(r, k)$ is defined for all $1 \leq k \leq |V| - |L|$, where $L$ is the set of leaf nodes.
The above proposition is understood to be true, since the maximum size of any transverse cut is \(|V| - |L|\). This easily follows from the fact that no transverse cut can include leaf nodes. In addition it can be demonstrated (by “unwinding” the recursion if necessary) that any definition of \(A(r, k)\) where \(k > |V| - |L|\) must depend on some undefined leaf case. This hypothetical dependence of \(A(r, k)\) on \(A(\ell, m)\) would imply that the transverse cut with cardinality \(k\) which is constructed includes node \(\ell\), but \(\ell\) is a leaf node, and therefore cannot be included as a member in any transverse cut. Therefore, the only values of \(k\) for which \(A(r, k)\) is undefined are precisely those values of \(k\) for which any \(U \subseteq V\) with \(|U| = k\) cannot possibly be a transverse cut. This informal argument should suffice to show that our definition of recurrence \(A\) is consistent with the definition of transverse cuts given in Section 3.1.

3.3 A Dynamic Programming Algorithm

We will now show how to use dynamic programming to compute \(A(r, k)\) for all possible values of \(k\). During the process of computing \(A\), we will additionally store data which will be used to reconstruct \(U^*\). To accomplish this, we will associate with each node, \(n\), two vectors, \(A_n\) and \(S_n\), where \(A_n[k]\) stores \(A(n, k)\), and \(S_n[0], ..., S_n[k]\) is used to describe the set of nodes which comprise the transverse cut with \(A_n[k]\) edges, where \(1 \leq k \leq |V| - |S|\).

To ease our exposition, we have split the presentation of the algorithm into two parts. In this section, we describe \textsc{min-trans-cut-card} a procedure which computes the cardinality of \(U^*\), the valid transverse cut of minimum size. In Section 3.4, we make modifications to \textsc{min-trans-cut-card} to describe \textsc{min-trans-cut}, an algorithm which will compute \(U^*\) itself. We will defer a complete description of \(S_n\) until then.

Before describing the algorithm, we must briefly discuss some definitions which are used. It will be convenient to refer to the children of a node. Let \(ch(n)\) be the set of children of
node $n \in V$. We also require the use of MAX-A, a subroutine defined in Figure 3.4. MAX-A is used to compute

$$\max_{i+j=k, \ i\neq0, j\neq0} [A(r_1, i) + A(r_2, j)].$$

Note that MAX-A returns multiple values, strictly for later convenience. Specifically, MAX-A returns $x$, the maximum number of edges used by the union of any pair of cuts, and $(i', j')$, the size of the two sets whose union was taken to form the new cut. $x$ is used directly in the computation of $A_n$, and $(i', j')$ will be used in Section 3.4 to compute $S_n$.

We will now describe the reasoning behind the first version of the algorithm, which uses $A_n$ to compute $|U^*|$. The dynamic program, described in Figure 3.3, starts from nodes at the bottom of the tree, and “grows” the tree upwards and out while computing $A$. To ensure that $A(n, k)$ is computed for all children of a node before each node is visited, we visit each node in order of a post-order traversal. The set of nodes can be sorted in this order simply by running a recursive post-order tree-traversal algorithm of the kind which are presented in many data structures courses\textsuperscript{4}. Any such traversal algorithm takes time $O(|V|)$. Such an algorithm is implicitly invoked at line 2 to sort the set of nodes. Each node is visited in this order during the for loop which comprises lines 4-24. We also need some way to keep track of which nodes have already been visited, and access the value for $A$ which was computed.\textsc{min-trans-cut-card} uses a stack to store the indices of the nodes which have already been visited. Using post-order traversal guarantees that, at any point in time, the set of nodes at the top of the stack will be precisely the set of children of the current node being visited, (assuming the node has any children).

Each node $n \in V$ falls into one of two cases, either $n$ is a leaf node and is handled in lines 5-6, or $n$ is an internal node and is handled in lines 7-23. When $n$ is a leaf node, we fall into the trivial case where where $A(n, k)$ is undefined for all values of $k$ (i.e. $A(n, k) = \bot$); this

\textsuperscript{4}More information on tree traversal can be found in (Cormen et al., 2001a).
Algorithm: MIN-TRANS-CUT-CARD

Input: Tree $T = (V, E)$ with $V = \{1, \ldots, m\}$ and root node $v_r$, $r \in \mathbb{Z}^+$.  
Output: $|U^*|$, where $U^* = \arg \min_{U : e(U) \geq \rho} |U|$.  

1: $A_i \leftarrow \text{EMPTY-ARRAY}(m)$, $\forall i \leftarrow 1 \ldots m$  
2: $N[] \leftarrow V$, sorted by post-order traversal.  
3: $L \leftarrow \text{EMPTY-STACK}()$.  // Stores visited nodes.  
4: for $i \leftarrow 1 \ldots m$ do  
5: \hspace{1em} if $\text{ch}(N[i]) = \emptyset$ then  
6: \hspace{2em} $A_i \leftarrow \lfloor \bot, \bot, \bot, \ldots \rfloor$  // Leaf node  
7: \hspace{1em} else  
8: \hspace{2em} $j \leftarrow \text{POP}(L)$  // “Up” case  
9: \hspace{2em} $A_i[1] = 1$;  
10: for $k \leftarrow 2 \ldots m$ do  
11: \hspace{2em} $A_i[k] = A_j[k - 1]$;  
12: end for  
13: while $\text{PARENT-OF}(T, i, \text{peek}(L))$ do  
14: \hspace{2em} $A_i' \leftarrow A_i$  // “Out” case  
15: \hspace{2em} $j \leftarrow \text{POP}(L)$;  
16: \hspace{2em} $A_i[1] \leftarrow A_i'[1] + 1$  
17: for $k \leftarrow 2 \ldots m$ do  
18: \hspace{2em} $\langle B, (k_1, k_2) \rangle \leftarrow \text{MAX-A}(A_i', A_j, k)$;  
19: \hspace{2em} $A_i[k] \leftarrow \max\{B, A_i'[k] + 1\}$;  
20: end for  
21: end while  
22: end if  
23: $\text{PUSH}(L, i)$.  
24: end for  
25: for $k \leftarrow 1 \ldots m$ do  
26: \hspace{1em} if $A_r[k] \geq \rho$ then  
27: \hspace{2em} return $k$  
28: end if  
29: end for  
30: return $0$  

Figure 3.3. Pseudo-code for a $O(|V|^3)$ dynamic program which, given a tree $T$, computes $|U^*|$, the minimum size of any valid transverse cut. See Section 3.3 for a discussion of the algorithm, and Figure 3.4 for details concerning subroutine MAX-A.
Algorithm: max-a — calculates \[
\max_{i,j: i+j=k} [A(r_1, i) + A(r_2, j)]
\]

Input: Arrays \(A'_i\) and \(A_i\), \(k \in \mathbb{Z}^+\).

1: \(\text{max} \leftarrow -\infty\)
2: \((i', j') \leftarrow (0, 0)\)
3: for \(i \leftarrow 1 \ldots m\) do
4: \(j \leftarrow k - i\)
5: if \(\text{max} < A'_i[i] + A_i[j]\) then
6: \(\text{max} \leftarrow A'_i[i] + A_i[j]; (i', j') \leftarrow (i, j)\);
7: end if
8: end for
9: return \((\text{max}, (i', j'))\)

Figure 3.4. Pseudo-code for a sub-routine used in the implementation of MIN-TRANS-CUT, shown in Figure 3.3.

is described on line 6. When \(n\) has children, we must attempt to merge \(n\) with its children, one after the other. Since all children of \(n\) are at the top of stack \(L\), we can simply pop nodes from \(L\) until we find some child which is not a child of \(n\). Lines 8-12 comprise the “up” case as described in Equation 3.3. Notice that, at this point, \(n\) is guaranteed to have at least one child \(j\), and this child must be the node at the top of the stack. We pop node \(j\), and update the value of \(A_n\), given the definition of \(A_j\).

Lines 13-22 comprise the “out” case, which is the main portion of the algorithm. To ensure that we have yet not popped all of the children of \(n\), we use a peek operation to ensure that node \(i\) is a child of the node at the top of stack \(L\) at line 13. If additional children remain, they must fall into the “out” case. This case recursively depends on whatever value of \(A_n\) we computed in the “up” case. In order to preserve this information, we copy \(A_n\) into \(A'_n\). \text{MAX-A} is used to find the pair of cuts with size \(k\) which has the maximum number of edges, and \(A_n\) is updated according to the definition of Equation 3.4. Each iteration of the loop at line 13 “grafts in” an additional child to the tree rooted at node \(n\). As this loop progresses, \(A_n\) is continuously updated until all children are added.

Finally, the minimum value of \(k\) for which \(A(r, k) \geq \rho\) is determined via a linear scan on lines 25-30. If no valid cut exists, then MIN-TRANS-CUT-CARD outputs 0.
One advantage of dynamic programming is that the running time is typically simple to analyze. Observe first that each node only participates in the while loop at lines 13-21 at most once. This fact can be seen by observing that each node is only pushed onto stack $L$ at most once, and only after it has been visited. Therefore, the while loop at line 13 is only called $O(|V|)$ times throughout the entire execution of the algorithm. Each execution of this loop makes $O(|V|)$ calls to MAX-A, which itself runs in $O(|V|)$ time, for a total running time of $O(|V|^3)$ for algorithm MIN-TRANS-CUT-CARD.

3.4 Retrieving the Valid Cut of Minimum Size

The algorithm developed in the previous section computes the cardinality of a minimum-size cut, $U^*$. In order to retrieve $U^*$ itself, we will need to store some additional information in a separate table, $S_n$. This table has an entry for each node, $n$, and each entry is only defined when $A_n$ itself is defined. Recall that each cut considered during the execution of MIN-TRANS-CUT-CARD is the union of two cuts from previously computed cuts, $U_1$, and $U_2$, rooted at $n_1$ and $n_2$, respectively. In each entry in $S_n$ we will store four values: lookup indices for $n_1$, and $n_2$, and the cardinalities of the cuts which were used, $|U_1|$ and $|U_2|$. Each entry is denoted by a 4-tuple, $(n_1, n_2, |U_1|, |U_2|)$. We will refer to $n_1$ and $n_2$ collectively as indices, and $|U_1|$ and $|U_2|$ collectively as cardinalities. This information can be used to recursively construct $U^*$, given a complete $S_n$ table for any tree. In the previous section, we designed MAX-A to accommodate our need for the values $|U_1|$ and $|U_2|$. Indeed the pair $(i', j')$ which is returned by MAX-A comprises the values of $|U_1|$ and $|U_2|$ which we require (i.e. $|U_1| = i'$, and $|U_2| = j'$).

We present a modified version of MIN-TRANS-CUT in Figure 3.5. It is identical to MIN-TRANS-CUT-CARD, except where $S_n$ is computed and defined at lines 2, 7, 10, 12, 17, and 20-23. We will give an overview of these changes and describe a convention for denoting empty cuts which we will later find convenient. At lines 10, 12, 17, and 23, the value of $A_n$
which is computed only depends upon a single cut. To ensure that $S_n$ is consistent with $A_n$, we signify that an empty cut was present by storing a 0 as both the index and the cardinality. Notice that at line 7, when $S_\ell$ of a leaf node, $\ell$ is computed, all entries are set to zero, indicating that no cut is present at $\ell$. One can observe that in the majority of the cases considered by the code, only one cut is recursively considered. If one were to compare algorithms MIN-TRANS-CUT-CARD and MIN-TRANS-CUT, one would notice that the max operation at line 19 of the former algorithm is replaced by an if-then construct in the latter. Despite this difference, it is trivial to see that the computation of $A_n$ is not significantly changed\textsuperscript{5}.

We will now describe RETRIEVE-CUT, the recursive procedure which is used to retrieve cut $U^*$ given $S_n$. Pseudo-code for this procedure can be found in Figure 3.6 on page 32. RETRIEVE-CUT takes as input a node $n$, and a cardinality $k$ for which both $A_n$ and $S_n$ are defined. Given this input, it recursively constructs $U$ by using $S_n$ to visit and add additional nodes. When RETRIEVE-CUT visits a node, it first ensures the node it is visiting exists ($n \neq 0$), and that it’s adding a non-empty set to cut $U$ ($k \neq 0$). If either of these do not hold, the subroutine simply returns the partial cut $U$ which was given to it, making no additional changes. When both conditions hold $n$ is added to the partially constructed cut. Additionally, if $k > 1$, then additional nodes must have been added from the children of $n$ in order to form $U$. In this case, RETRIEVE-CUT uses $S_n$ to recursively examine the cuts which were paired to form the cut rooted at $n$ having cardinality $k$. Notice that the recursion halts when either a leaf node is reached, or when a node which only contributes itself to $U$ is reached (i.e. when $k = 1$, in which case the if statement at line 3 is skipped).

This recursive procedure suffices to retrieve set $U^*$ which is computed using MIN-TRANS-CUT. At worst, this subroutine will only visit each node once, performing a constant amount

\textsuperscript{5}The only difference is as follows: when considering two cuts, $U_1$ and $U_2$, with $e(U_1) = e(U_2)$, MIN-TRANS-CUT will prefer to select the cut which consists solely of the cut in the previous tree (since the inequality at line 19 is strict) whereas MIN-TRANS-CUT-CARD has no such preference. While this decision was arbitrary, it does not effect the correctness of the algorithm.
Algorithm: **MIN-TRANS-CUT**

**Input:** Tree $T = (V, E)$ with $V = \{1, \ldots, m\}$ and root node $r$, $\rho \in \mathbb{Z}^+$.  

**Output:** $|U^*|$, where $U^* = \arg \min_{U : e(U) \geq \rho} |U|$.  

1: $A_i \leftarrow \text{EMPTY-ARRAY}(m), \forall i \leftarrow 1 \ldots m$  
2: $S_i \leftarrow \text{EMPTY-ARRAY}(m), \forall i \leftarrow 1 \ldots m$  
3: $N[\cdot] \leftarrow V$, sorted by post-order traversal.  
4: $L \leftarrow \text{EMPTY-STACK}().$ // Stores visited nodes.  
5: for $i \leftarrow 1 \ldots m$ do  
6: if $\text{ch}(N[i]) = \emptyset$ then  
7: $A_i \leftarrow [\bot, \bot, \bot, \ldots]; S_i \leftarrow [(0, 0, 0, 0), \bot, \bot, \ldots]$; // Leaf node  
8: else  
9: $j \leftarrow \text{POP}(L)$ // “Up” case  
10: $A_i[1] = 1; S_i[1] = (i, 0, 1, 0);$  
11: for $k \leftarrow 2 \ldots m$ do  
12: $A_i[k] = A_j[k - 1]; S_i[k] = (j, 0, k, 0);$  
13: end for  
14: while $\text{PARENT-OF}(T, i, \text{PEEK}(L))$ do  
15: $A'_i \leftarrow A_i$ // “Out” case  
16: $j \leftarrow \text{POP}(L);$  
17: $A_i[1] \leftarrow A'_i[1] + 1; S_i[1] = (i, 0, 1, 0);$  
18: for $k \leftarrow 2 \ldots m$ do  
19: $\langle B, (k_1, k_2) \rangle \leftarrow \text{MAX-A}(A'_i, A_i, k);$  
20: if $B > A'_i[k] + 1$ then  
21: $A_i[k] \leftarrow B; S_i[k] = (i, j, k_1, k_2);$  
22: else  
23: $A_i[k] \leftarrow A'_i[k] + 1; S_i[k] = (i, 0, A'_i[k] + 1, 0);$  
24: end if  
25: end for  
26: end while  
27: end if  
28: $\text{PUSH}(L, i).$  
29: end for  
30: for $k \leftarrow 1 \ldots m$ do  
31: if $A_r[k] \geq \rho$ then  
32: return $\text{RETRIEVE-CUT}(S_n, r, k, \emptyset);$  
33: end if  
34: end for  
35: return 0

Figure 3.5. Pseudo-code for a $O(|V|^3)$ dynamic program which computes $U^*$. See Section 3.4 for details concerning the algorithm, Figure 3.4 for a description of MAX-A, and Figure 3.6 for a description of RETRIEVE-CUT.
Algorithm: \textsc{retrieve-cut} — retrieves a cut $U$ from $S_n$.

\begin{itemize}
  \item[Input:] Arrays $S$, node index $i \in \{1...m\}$, $k \in \mathbb{Z}^+$, and partial cut $U \subseteq V$.
  \item[Output:] Cut $U \subseteq V$.
\end{itemize}

\begin{itemize}
  \item[1:] if $i \neq 0 \land k \neq 0$ then
  \item[2:] $U \leftarrow U \cup \{i\}$
  \item[3:] if $k > 1$ then
  \item[4:] $(j, k, |U_1|, |U_2|) \leftarrow S_n[k]$;
  \item[5:] $U \leftarrow \textsc{retrieve-cut}(S, j, |U_1|, U)$;
  \item[6:] $U \leftarrow \textsc{retrieve-cut}(S, k, |U_2|, U)$;
  \item[7:] end if
  \item[8:] end if
  \item[9:] return $U$
\end{itemize}

Figure 3.6. Pseudo-code for a sub-routine used to retrieve a cut $U$, given $S_n$. See page 29 for the definition of $S_n$.

of computation at each node, so its running time is bounded by $O(|V|)$. This implies that this recursive procedure will not dominate the $O(|V|^3)$ term which we derived during the analysis of \textsc{min-trans-cut-card}. Further note that each of the additions we made to \textsc{min-trans-cut-card} in this section can be computed in a constant amount of time. Therefore, the $O(|V|^3)$ running time analysis of \textsc{min-trans-cut-card} holds also for \textsc{min-trans-cut}.

Recall that in Section 3.1 we showed that \textit{any} standard placement of $r$ replicas on a transverse cut $U$ is sufficient to solve $(\rho - 1)$-SRP. Therefore, any standard placement of $r$ replicas on the subtrees of the outer border nodes of $U^*$ will ensure that the minimum number of nodes in $T$ are $(\rho - 1)$-unsafe, thereby maximizing $c(P, \rho - 1)$. The process of constructing a standard placement given any transverse cut $U$ is simple. All one needs to do is place a single replica on the subtrees rooted at each of the outer border nodes of $U^*$. Notice that for most cuts there will be multiple standard placements from which to select. Theorem 3.1.2 guarantees that, regardless of which set of leaf nodes are selected as part of the standard placement, the result will maximize $c(U, \rho - 1)$. However, this gives us a large amount of freedom in selecting specific leaf nodes for forming our standard placement. This point is addressed further in Chapter 4.
CHAPTER 4
CONCLUSIONS

In Chapter 2 we described a mathematical model for describing hardware dependency using graphs and developed a framework for placing replicas of data in this model to ensure that failures impact data availability as little as possible. In Chapter 3 we described a polynomial-time solution for the problem of Survivable Replica Placement in the case that our graph can be described by a tree. In this chapter we highlight several extensions of our work and describe what remains to be done in this area.

Currently, the most pertinent factor which limits the applicability of our model is the fact that the solution we describe in Chapter 3 is only applicable for the case when the graph is a tree. In domains where dependency between failure points is better described by a directed acyclic graph, or even a general graph, our solution will be not be able to be directly used. We emphasize that while it is our eventual goal to extend this work to a more general class of graphs, we seek after the much loftier goal of a full description of a theory of failure point models. One of the first steps in the description of such a theory is to determine a relevant application, and demonstrate the usefulness of the model. In this thesis we have demonstrated that there exist polynomial-time solvable problems in this area. We hope to later extend our model to encompass a broader class of graphs and to consider a broader class of problems which may naturally arise from failure point models.

Further extensions include handling multiple failures. Notice from our definition of $\sigma$ that we are restricted to considering scenarios where only a single node may fail at any given time. One possible extension of our work would be to consider variations of $\sigma$ which could take into account multiple failures. Additionally, we have made the assumption that data
is only placed on the leaf nodes of a tree. While the problem of placing data on arbitrary nodes in is interesting in its own right, we could not justify making this assumption due to our desired application to data center management.

In conclusion, we would like to mention two variants of SRP we have identified which remain unsolved at this time. In doing so, we can also describe a simple optimization to the algorithm presented in the previous chapter.

4.1 Solving $k$-SRP

The previous sections have focused on finding a valid transverse cut (i.e. a transverse cut $U$ with $e(U) \geq r$) of minimum cardinality. At the end of Section 3.1, we showed how such a cut could be used to solve ($\rho - 1$)-SRP. In this section, we will describe how to use MIN-CUT to solve ($\rho - 1$)-SRP, and then perform some additional computation to optimize the number of $\rho$-unsafe nodes in the optimal solution to ($\rho - 1$)-SRP.

Recall that in Section 3.1 we observed that there may be many valid standard placements for any transverse cut. After having found $U^*$, we may be able to select a placement which satisfies secondary optimality criteria. In fact, a simple addition suffices to ensure that the placement $P$ we find minimizes the number of $\rho$-unsafe nodes in the tree in addition to minimizing the number of ($\rho - 1$)-unsafe nodes. First, notice that every ($\rho - 1$)-unsafe node is also $\rho$-unsafe by definition. This implies that the set of nodes in $U^*$ are hopeless — we can do nothing to make them $\rho$-safe. Therefore, whatever optimization that remains to be performed needs only to be performed in $V - U$. Recall that $V - U$ forms a forest of $n$ subtrees, denoted $T_i$, for $1 \leq i \leq n$. Suppose that the root of tree $T_i$ is $r_i$, and that for each tree, leaf $\ell_i \in P$, where $P$ is a placement. Consider the number of $\rho$-unsafe nodes in each subtree $T_i$. Each of these trees contains a chain of $\rho$-unsafe nodes which is precisely the path from $r_i$ to $\ell_i$. Therefore it would seem that placing the replica on the storage node which
\( \rho \)-SRP optimal solution \hspace{1cm} (\rho - 1)\text{-SRP optimal solution}

Figure 4.1. A counter-example to the optimization of \( \rho \)-SRP using the naïve approach described. Tree \( T \) is presented in each of the two cases, the solution which optimizes \((\rho - 1)\text{-SRP} \) and the solution which optimizes \( \rho \)-SRP. Light grey nodes have replicas placed upon them, and dark grey nodes are \( \rho \)-unsafe.

minimizes the length of this chain is clearly preferable. This corresponds to selecting the storage node with minimum depth from each \( T_i \).

However, we must be careful in what precisely we claim. All we have done in the previous paragraph is minimize the number of \( \rho \)-unsafe nodes in an optimal solution to \((\rho - 1)\text{-SRP} \). The above technique is not sufficient to solve \( \rho \)-RRP; in fact, we present a counter-example to such a claim in the following paragraph.

Consider a tree, \( T \) with root node \( r \), onto which we wish to place \( \rho = 3 \) replicas. Let \( r \) have three children, and let the subtrees rooted at these children be chains of length \( k \). Arbitrarily select one of these three chains, say on child \( c' \), and let it have three leaf nodes. To maximize \( c(P, \rho - 1) \), we should place one replica on each of the three chains. However, let us examine what occurs when we desire to maximize \( c(P, \rho) \). If we use the same solution as in the \( \rho - 1 \) case, each internal node in each of the chains will be \( \rho \)-unsafe, since if any
one of them fails, the replicas placed at the end of the chain will also fail. This yields a total of $3k - 2$ $\rho$-unsafe nodes. However, if we were to place three replicas at the three leaf nodes on child $c'$, we would have a total of $k - 1$ $\rho$-unsafe nodes, which beats any of the $\rho - 1$ solutions. This scenario is illustrated in Figure 4.1. This example calls into question the applicability of optimizing for $\rho$-safety, rather than $(\rho - 1)$-safety, since the optimal solution to the $\rho$-safe example is extremely similar to replica placement under the current status quo, which motivated this research in the first place.

Based upon this example, we can determine that it is not necessarily the case that the set of optimal solutions to $(\rho - 1)$-SRP contains optimal solutions to $\rho$-SRP. In the previous example, only three optimal solutions to $(\rho - 1)$-SRP existed, and the optimal solution for $\rho$-SRP is not among them.

### 4.2 Integer Capacity Nodes

We would also like to consider another variant of a possible replica placement problem. This variant considers the case where each storage node $s_i \in S$ has integer capacity, $c_i \in \mathbb{Z}^+$, of data-blocks which it can store. We show that without taking additional assumptions concerning data placement into account, the integer data-block case cannot necessarily be solved in polynomial time using our algorithm. However, the assumption which is required is extremely reasonable, and is typically already made in practice.

**Problem 4.1. Integer Capacity** — Given tree $T$ and $k \in \mathbb{Z}^+$, with storage nodes $S = \{s_1, ..., s_n\}$, where each $s_i$ has capacity $c_i \in \mathbb{Z}^+$ for $i = 1, ... n$, find $P^* \subseteq S$, which maximizes $c(P,k)$.

**Problem 4.2. Single-block Capacity** — Given tree $T$ with storage nodes $S = \{s_1, ..., s_n\}$, where each $s_i$ has capacity $c_i = 1$, for $i = 1, ... n$, find $P^* \subseteq S$, which maximizes $c(P,k)$ for some $k$.
Figure 4.2. An example of the reduction of Theorem 4.2.1. An instance of the integer-capacity problem can be seen on the right. In this instance, storage nodes $a, b, c, d,$ and $e$, have capacities $c_a = 2, c_b = 1, c_c = 3, c_d = 2, c_e = 2$ respectively. The reduction, seen on the left, adds $c_i$ leaves to each of these nodes.

We can show that any algorithm $A$ which solves Problem 4.2 can also be used to solve Problem 4.1. See Figure 4.2 for an illustration of the reduction of the below Theorem.

**Theorem 4.2.1.** Any algorithm $A$ which solves the Single-block Capacity case, can be used to solve the Integer Capacity case.

**Proof.** Given an instance of the Integer Capacity case, $(G, S)$ with graph $G = (V, E)$, and storage node set $S \subseteq V$. Construct $(G', S')$, an instance of the Single-block Capacity case as follows: Let $G' = (V', E')$. $V' = V$, with the following additions: for every $s_i \in S$, with capacity $c_i$, add $c_i$ nodes to $V$, labeled $v'_{ij}$, where $j = 1, \ldots, c_i$. Additionally, for every $v'_{ij}$, add $(s_i, v'_{ij})$ to $E$, to form $E'$. More precisely,

$$V' = V \cup \{v'_{ij} \mid s_i \in S, j = 1, \ldots, c_i\}$$

$$E' = E \cup \{(s_i, v'_{ij}) \mid s_i \in S, j = 1, \ldots, c_i\}$$

Clearly, algorithm $A$ can also be used to solve $(V', E')$. This algorithm would place each replica on a single node. $A$ guarantees that an optimal solution is found, and it’s easy to see that the optimality conditions will not be violated by the addition of children for each
leaf. Furthermore, the reduction guarantees that no storage node will have its capacity exceeded.

Notice that, while algorithm $A$ can solve the integer case, it may not necessarily do so in polynomial time. Because the reduction of Theorem [4.2.1] converts the capacity of each storage node into multiple nodes, if the running time of $A$ is given in terms of $|V|$, then algorithm $A$, when run on the converted instance, will depend upon the sum of the vertices and the capacities present in the original instance. To see why this matters, construct an integer-capacity instance $I$ where each storage node has capacity $2^{|V|}$. After performing the reduction of Theorem [4.2.1], the single-block instance computed, $I'$, consists of graph $G' = (V', E')$, where $|V'| = |V| + |S| \cdot 2^{|V|}$. Therefore, a polynomial running time for the single-block capacity case does not imply a polynomial running time for the integer capacity case. This is of special interest because the algorithm which we present in Chapter 3 solves the single-block case in polynomial time, but is not guaranteed to solve the integer capacity case in polynomial time precisely because of this issue.

However, in practice we typically desire that each replica be placed on distinct storage blocks (since we gain nothing in terms of availability by storing multiple copies of identical data on the same storage media). If we take this desiderata as an additional constraint a trivial reduction can be used to convert an instance of the integer capacity problem to an instance of the single-block capacity problem. To form such a reduction it suffices to set the capacities of the storage nodes to 1 for all nodes $s_i \in S$, where $c_i > 0$. This reduction, unlike the previous one, preserves the polynomial running time of any algorithm used to solve the single-block capacity case. Because this is a reasonable assumption to make in practice, we claim that our algorithm is applicable to the integer capacity case any practical scenario where this assumption can be safely made.

In this thesis, we have presented a novel model for correlated hardware failure, and shown how this model can be used to make intelligent decisions concerning data placement in a
replicated data center. We formulated a hierarchical failure point model and illustrated how such a model can be used to optimize replica placement for certain safety criteria. We also showed that this optimization problem can be solved in polynomial time using dynamic programming. Besides being a novel problem formulation and algorithm, we have additionally demonstrated that there are optimization problems for failure point models which are computationally tractable. However, as we have also indicated, there still remains much work to be done before a general solution is found which may be of use to data center managers.
APPENDIX

A.1 Subroutine Definitions

Readers who are interested in the pseudo-code included as Figures 3.3 and 3.5 may wish to reference explicit definitions for the subroutines used therein. Subroutines which are defined in the main text (i.e. MAX-A and RETRIEVE-CUT) are not included.

\texttt{PARENT-OF}(T, i, j) — Given tree \(T\), and nodes \(i\) and \(j\), returns \texttt{true} if and only if node \(i\) is the parent of node \(j\).

\texttt{EMPTY-ARRAY}(n) — Constructs an returns an empty array of size \(n\).

\texttt{EMPTY-STACK()} — Constructs and returns an empty stack.

\texttt{PUSH}(L, i) — Pushes object \(i\) onto stack \(L\).

\texttt{POP}(L) — Removes item from the top of stack \(L\) and returns it.

\texttt{PEEK}(L) — Returns the item at the top of stack \(L\) without removing it.

A.2 Additional Observations

We include here a list of observations concerning failure point models which were not used in developing the main result of this thesis, but which may be of future use nonetheless. Also included are several lemmas which were alluded to in the main body of the text, but which are trivial enough that their inclusion would have disrupted the flow of reading. Every graph \(G\) and tree \(T\) mentioned in the lemmas that follow refer to failure point models.
Proposition A.2.1. For any tree $T$, and any $\rho > 0$, no placement $P$, with $|P| = \rho$ may ensure that all nodes in $T$ are $\rho$-safe.

Proposition A.2.2. For any tree $T$, with root $r$, for any placement $P$, $\sigma(r, P) = 0$.

Lemma A.2.1. For any tree $T$ with less than than $\rho$ interior nodes, where $\rho \geq 2$, there does not exist any placement $P$ of size $\rho$ for which $\sigma(n, P) \geq \rho - 1$ for all interior nodes (excepting the root).

Proof. By the pigeonhole principle, some interior node must have at least two replicas as successors. Therefore, if this node fails, at least two replicas will fail as well. \qed
REFERENCES


VITA

K. Alex Mills was born in Kansas City, Missouri to parents Gary and Michelle Mills, both Texans by birth. Despite the unfortunate circumstances of his birth, the author considers himself a native Texan since his parents, themselves Texans by birth, had the good sense to leave the Midwest and return to the Lone Star State before he was even four years old. He is eternally grateful to them in this regard.

In his early years, Alex was home-schooled by his mother along with his two siblings, Kellen and Maggie. He regards the excellent academic foundation laid by his mother at an early age as having been fundamental to his later success. He attended community college at Brookhaven and Richland Colleges before transferring to The University of Texas at Dallas. Despite being a transfer student, he was successful in the undergraduate program, passing two Ph.D Qualifying Exams before graduating summa cum laude with a Bachelor’s in Software Engineering in the Spring of 2011. After graduation, he was accepted into the Ph.D Computer Science program at UTD, where he is currently performing research with his advisor, Dr. Neeraj Mittal. This Master’s Thesis represents original research which he hopes to extend as part of his dissertation. His areas of academic interest change weekly, but typically include algorithms, combinatorial optimization, linear and discrete optimization, and the math and theory underpinning the techniques in each of these fields.