Bachelor Thesis

Protection of networks through failure injection

Belmin Oruc

November 3, 2024

Reviewer:
Prof. Dr. Klaus-Tycho Förster
Erik van den Akker

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl V - Programmiersysteme
Fachgruppe Softwaretechnik sicherer Systeme
https://sse.cs.tu-dortmund.de

Abstract

As Software as a Service (SaaS) becomes more prevalent globally, networks have emerged as a critical infrastructure in our society. As with any infrastructure, networks are susceptible to failure. It is therefore of the utmost importance to ensure the integrity and reliability of networks in order to provide an uninterrupted experience for both users and service providers. Consequently, the largest software-as-a-service providers have initiated comprehensive network testing procedures. In 2012, Netflix released a program called Chaos Monkey. The application is responsible for randomly terminating instances in production, thereby ensuring that engineers implement their services in a resilient manner with regard to instance failures. Other software-as-a-service (SaaS) providers, including Google and Amazon, have developed their own versions of Chaos Monkey. This illustrates the significance and necessity of such testing applications. In reference to the work of Shelly (2015) [31], the author presents an algorithm, designated as "Greedy Killer," that systematically eliminates all edges of a network graph when feasible. Although the Greedy Killer algorithm represents an improvement over previous implementations, numerous network invariants remain unchecked, thereby providing opportunities for further algorithmic enhancement. This thesis will investigate these potential enhancements by implementing, testing, and verifying their efficacy and runtime. Furthermore, it will provide an overview of the concept of chaos engineering from a variety of perspectives. The implementation will focus on a variety of algorithmic approaches to gain insight into the optimal method for ensuring the continuous functionality of one of our most critical infrastructures.

Mit der zunehmenden weltweiten Verbreitung von Software as a Service (SaaS) haben sich Netzwerke zu einer wichtigen Infrastruktur in unserer Gesellschaft entwickelt. Wie jede Infrastruktur sind auch Netze anfällig für Ausfälle. Es ist daher von größter Bedeutung, die Integrität und Zuverlässigkeit der Netze zu gewährleisten, um sowohl den Nutzern als auch den Dienstanbietern einen ununterbrochenen Betrieb zu ermöglichen. Folglich haben die größten Software-as-a-Service-Anbieter umfassende Netztestverfahren eingeführt. Im Jahr 2012 veröffentlichte Netflix ein Programm namens Chaos Monkey. Die Anwendung ist für die zufällige Beendigung von Instanzen in der Produktion verantwortlich und stellt so sicher, dass die Ingenieure ihre Dienste in Bezug auf Instanzausfälle belastbar implementieren. Andere Software-as-a-Service (SaaS)-Anbieter, darunter Google und Amazon, haben ihre eigenen Versionen von Chaos Monkey entwickelt. Dies verdeutlicht die Bedeutung und Notwendigkeit solcher Testanwendungen. Unter Bezugnahme auf die Arbeit von Shelly (2015) [31] stellt der Autor einen als „Greedy Killer“ bezeichneten Algorithmus vor, der systematisch alle Kanten eines Netzwerkgraphen eliminiert, wenn dies möglich ist. Obwohl der Greedy-Killer-Algorithmus eine Verbesserung gegenüber früheren Implementierungen darstellt, bleiben zahlreiche Netzwerkinvarianten unberücksichtigt, wodurch sich Möglichkeiten zur weiteren Verbesserung des Algorithmus ergeben. In dieser Arbeit werden diese potenziellen Verbesserungen untersucht, indem ihre Wirksamkeit und Laufzeit implementiert, getestet und verifiziert werden. Darüber hinaus wird ein Überblick über das Konzept des Chaos-Engineering aus verschiedenen Perspektiven gegeben. Die Implementierung wird sich auf verschiedene algorithmische Ansätze konzentrieren, um einen Einblick in die optimale Methode zur Gewährleistung der kontinuierlichen Funktionalität einer unserer kritischsten Infrastrukturen zu erhalten.

Contents

1 Introduction

The first chapter serves to introduce the issues that will be examined in this thesis, to provide a rationale for our work, to outline the structure of the thesis, and to present the research questions that will be addressed.

1.1 Motivation

In recent years, communication networks have become a fundamental component of global infrastructure, supporting essential services and facilitating communication across diverse sectors. For this reason, the reliability of these networks represents a primary concern for many software as a service (SaaS) providers. The intrinsic nature of communication networks renders the prevention of all forms of failure an impractical undertaking. This underscores the necessity of guaranteeing the dependability of route switching and fail-safe mechanisms to forestall the disruption of services in the event of such failures. As discussed in Nunes [25], the advent of Software-Defined Networking (SDN) has the potential to streamline network management, thereby facilitating innovation and evolution. SDN decouples control decisions from the networking hardware, thereby enabling software developers to rely on network resources in a manner analogous to their reliance on storage and computing resources. The importance of networks in modern infrastructure, coupled with developments in network management, has created a fertile ground for innovation, which may be a contributing factor to the emergence of chaos engineering (CE) in relatively short order. In Basiri [15], the Chaos and Traffic Team at Netflix defines chaos engineering as ”the discipline of experimenting on a distributed system in order to build confidence in its capability to withstand turbulent conditions in production.” This is applicable to a multitude of failures, including hardware failures and client surges. By identifying four principles for chaos engineering, Netflix aims to make the use of chaos engineering widely accessible. These four principles are as follows: 1. Construct a hypothesis based on steady-state behavior 2. Vary real-world events 3. Conduct experiments in production 4. Automate experiments to run continuously, with further explanation provided in chapter 3.

By adhering to the aforementioned principles, Netflix employs the use of unavoidable failures to their advantage. This is achieved by introducing failures into their operational networks with the objective of identifying unforeseen behaviors and undetected system defects. To elaborate further, Chaos Monkey randomly selects virtual machine instances that host their production services and terminates them.

This approach has been demonstrated to be effective, allowing Netflix's engineering team to develop their services in a manner that is capable of handling instance failures. The success of Chaos Monkey has encouraged other software-as-a-service (SaaS) providers, including Amazon [6], Facebook [8], Microsoft [14], and Google [10], to adopt a similar approach. These examples demonstrate the efficacy of the process of ”breaking things during production.” The extensive use of Chaos Engineering in recent years has encouraged other fields to adopt the strategy as well. Fields that have been influenced by CE include:

Security: As described in Torkura in [34] and [35], the application of security chaos engineering (SCE) has enhanced the security of infrastructure as a service (IaaS) through the deliberate introduction of vulnerabilities. This approach, which involves proactive experimentation to build confidence in the system's resilience against malicious attacks, has been shown to be an effective method for ensuring the security of IaaS systems.

Automated Failure Recovery: In reference to the work of Ikeuchi et al. [20] , the authors utilize CE in conjunction with deep reinforcement learning (DRL) to terminate various resources through CE, subsequently employing DRL to examine, observe, and attempt diverse episodes of recovery actions within a chaotic environment.

Performance Debugging: The utilization of CE in performance debugging is delineated in reference [22]. The resulting framework, PerCe, employs chaos engineering as a means of stressing various system events, thereby generating sufficient and authentic abnormal data.

The aforementioned fields, along with the interest expressed by several prominent SaaS providers, demonstrate not only a keen interest in CE but also its efficacy. Consequently, it is highly probable that Chaos Engineering will persist as a pivotal subject, with potential applications in hitherto unexplored domains. The integration of AI, in particular, could be a promising avenue for expanding the scope of CE, as previously outlined in reference to the framework proposed by [20].

1.1 illustrates the implementation of CE in various contexts, showcasing the diversity of target stacks, resilience attributes, and application layers. This evidence substantiates the assertion that CE is a versatile concept, applicable to a multitude of applications, and that its efficacy is enhanced by the contributions of numerous entities.

Framwork	**Traget Stack**	**Resilience Attributes**	**Application Layer**
Chaos Kong [16]	AWS Region	availability (non-security)	cloud network
Chaos Gorilla [36]	AWS Availability Zones	availability (non-security)	cloud network
Chaos Monkey	AWS Availability Zones	availability (non-security)	cloud network (VMs))
Chaos Monkey for Spring Boot	Spring Boot Applications	availability (non-security e.g. latency, terminations)	Java Applications (e.g. REST, inter-service calls)
Royal Chaos [?]	Java Applications	availability (non-security)	JVM
Chaos Toolkit [24]	AWS, Azure Google & Kubernetes	availability (non-security)	cloud & kubernetes networks
PowerfulSeal [7]	Kubernetes	availability (non-security) availability (non-security)	kobernetes network (e.g. pods, microservices)
ChaosMesh [7]	Kubernetes	availability (non-security)	kubernetes network
ChaoSlingr [9]	Kubernetes	security (confidentiality, integrity & avaiability)	cloud services (e.g. S3, IAM)
CloudStrike [33]	AWS & GCP	security (confidentiality, integrity & avaiability)	cloud servicesr (e.g. S3, IAM)

Table 1.1: Chaos Engineering frameworks [34]

In recent years, both providers and users of network services have demonstrated a growing expectation for enhanced capabilities and services. The study, [28]indicates that users are up to 123% more likely to abandon a website if the loading time exceeds 10 seconds. Similarly, software as a service and network providers seek to reduce costs wherever possible. The objective of this paper is to take chaos engineering to its logical extreme. To what extent can a network be compromised without compromising usability? Furthermore, how might the cost be reduced as much as possible without compromising the usability of the network?

To this end, we will examine the work of Nick Shelly and others, as referenced in [31]. This paper introduces Armageddon, an application that eliminates edges in accordance with specific invariants. Specifically, the objective is to achieve reachability while simultaneously optimizing coverage and speed. The objective of this Bachelor's thesis is to: The objective of this study is to evaluate the impact of cost, flow, and biconnectivity on the chaos engineering process. To this end, additional invariances will be incorporated into the evaluation of edge removal. Furthermore, the efficacy of different algorithmic approaches will be assessed, both with and without the added invariances. The ultimate goal is to identify a sustainable method for network destruction, thereby enhancing the security, reliability, and availability of networks in diverse contexts.

1.2 Structure

The first chapter is designed to familiarize the reader with the subject matter of Chaos Engineering and this document. Subsequently, we will examine pertinent literature to gain a more profound comprehension of the applications and operational principles of chaos engineering. The third chapter will address the terminology, fundamental concepts, and algorithms utilized in the development of the proposed approach. The fourth chapter will provide an explanation of the methods and frameworks utilized in the implementation. The following two chapters examine the experimental design and the implementation of the algorithms. The evaluation chapter assesses the effectiveness of the implementation through the examination of test scenarios and the stress testing of the algorithm with different network graph libraries. The conclusion reflects on the test results, discusses potential issues, evaluates the usability of the results, and considers improvements to the algorithms. It also provides insights into future directions.

2 Related Work

As previously indicated in the introductory section, a considerable amount of research in this field is conducted by private enterprises, such as Netflix. The inaugural instance of chaos engineering was employed by the aforementioned entity in 2011. Consequently, the methodology is still relatively novel. This study will examine the methods employed by various Software as a Service (SaaS) providers in the implementation of their Chaos Engineering (CE) solutions. In the academic realm, a plethora of algorithms have been proposed, each with the objective of solving the intelligent disruption of networks. The approaches have been examined from a variety of perspectives, including the examination of different invariants, the analysis of varying run times, and the investigation of disparate network configurations. Consequently, the results of the research vary considerably.

2.1 Chaos Engineering in SaaS

This study will initially examine the various methodologies employed by SaaS providers in the implementation of their CE services.

2.1.1 Netflix

In their seminal work, Netflix [15] identified four principles that they believe encapsulate the Chaos Engineering approach to experiment design. These principles were first introduced by Netflix, which is the entity that first introduced the concept of Chaos Engineering.

1. Build a hypothesis around steady state behavior

The term "works properly" is insufficiently precise to serve as the basis for designing experiments for Netflix. The rationale behind this approach is that since their services are so diverse, if one service encounters an issue, it does not significantly impact the overall system availability. The ultimate objective is to ensure that users can easily access content and consume it. To this end, they utilize a metric called (stream) start per second (SPS). SPS, among other metrics, is used to characterize the steady-state behavior. Based on this observation, they propose that the hypothesis should be built around the functionality of this steady-state behavior.

2. Vary real world events

In the real world, errors can occur in a multitude of ways. To ensure that its services align as closely as possible with real-world conditions, Netflix deliberately introduces a variety of failure scenarios. These experiments involve a range of inputs, including:

- terminate virtual machine instances
- inject latency into requests between services
- fail requests between services
- fail an internal services
- make an entire Amazon region unavailable

These inputs occur at the hardware and software levels. To ensure uninterrupted operation, these experiments are automated.

3. Run experiments in production

One of the most crucial aspects of CE is the implementation of experiments within the actual production environment. This approach is essential to guarantee comprehensive integration testing. It is, in practice, exceedingly challenging to accurately recreate the full spectrum of system characteristics during the testing phase.

4. Automate experiments to run continuously

Given the significant alterations occurring within the Netflix system, it is imperative that the tests be conducted on an ongoing basis to guarantee that the present state of the system is adequately evaluated.

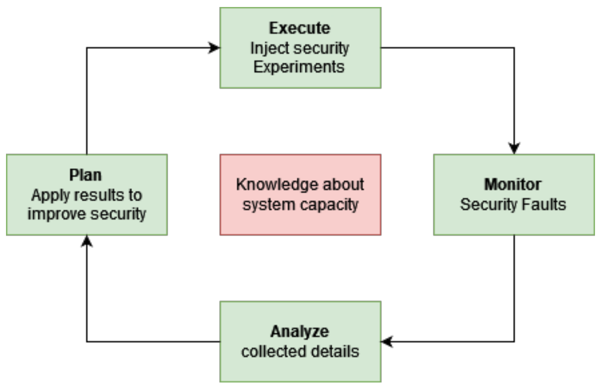

Execute
Inject security Experiments
Plan
Apply results to improve security
Knowledge about system capacity
Monitor
Security Faults
Analyze
collected details

Figure 2.1: Experiment Design

As outlined in [13], the implementation of lineage-driven fault injection (LDFI) has enabled Netflix to automate the process of failure testing.

2.1.2 Google

In [10] Google presents the introduction of their disaster recovery testing (DiRT) program. They have a generally similar approach to Netflix. The approach taken is, for the most part, similar to that of Netflix. A dedicated disaster test team is responsible for conducting tests during a period when system issues can be addressed and resolved. In [11] Google discusses the practice of whitelisting specific services that have been identified as prone to failure in the event of an outage. They assert that "in essence, they have already failed the test and there is no point in causing an outage for them when the failing condition is already well understood." In contrast, every service that has not been whitelisted must demonstrate resilience in the face of these tests, as there is no mechanism for bypassing DiRT. In conclusion, Google exemplifies two pivotal aspects of chaos engineering:

It is futile to deliberately compromise a system that has identified failure points, and no system is invulnerable.

2.1.3 Amazon

In 2021 Amazon introduces AWS Fault INjection Simulator (FIS), a managed service that performs fault injection experiments in AWS workload. These can target some of the biggest services AWS provides. Amazon Elastic Compute Cloud (EC2), Amazon Elastic Container Service (ECS), Amazon Elastic Kubernetes Service (EKS), Amazon Relational Database Service (RDS) and Amazon IAM Role [26]. Amazon describes FIS [3] as a fully managed service for running fault injection experiments to improve an application's performance, observability and resilience. That way they make it simple to run fault injection for their users and thus making it available for over one million AWS users.

2.1.4 Microsoft

In [14], Microsoft acknowledges the influence of Netflix's Simian Army and their own replication of a comparable system for their Search Services. It is specified that the potential operations available to the Search Chaos Monkey (SCM) are contingent upon the level of chaos selected for the test environment.

In the case of low chaos the system is capable of recovering from significant issues with minimal or no disruption to the availability of its services. Accordingly, notifications generated by a low chaos monkey are regarded as software defects.

Medium chaos entails a moderate level of disruption, allowing the system to recover gracefully with minimal or no interruption to service availability. Failures may recover in a graceful manner, but may result in a degradation of service performance or availability. In such instances, low-priority alerts are generated and conveyed to engineers on call.

In the event of a catastrophic failure (High Chaos) that results in the interruption of service availability, a high-priority alert will be transmitted to the engineers on call, necessitating manual intervention for repair.

The final category is that of extreme chaos. Failures that result in an ungraceful degradation of the service, leading to data loss or failure without the generation of an alert. Extreme Chaos is an unpredictable phenomenon that is unlikely to be utilized by Microsoft.

Amazon's AWS, Microsoft's Azure, and Google Cloud have a combined cloud infrastructure market share of approximately 67%. Consequently, the increasing reliance on Chaos Engineering by these entities not only signifies the significance of further development in this domain but also provides compelling evidence that it has already become an indispensable component of our network infrastructure.

2.2 Academic Chaos Engineering

A suitable introduction to the subject of chaos engineering is provided by [24]. This work introduces chaos engineering with the objective of understanding its historical development and the underlying principles that govern it. Additionally, the book provides instructions for implementing automated chaos engineering using Python 3 and Chaos Toolkit CLI, thus serving as a comprehensive guide for learning CE. Similarly, [29] offers a detailed examination of chaos engineering, encompassing its theoretical foundations, practical applications across diverse industries, the human and business factors influencing its adoption, and the evolution of CE. Consequently, the book presents a comprehensive account of the impact of CE on software engineering.

(Security) chaos experimentation employs the scientific method. In chaos engineering, we pose questions with the objective of proposing a hypothesis, which is then subjected to experimentation to either confirm or refute the hypothesis previously proposed. Subsequently, we initiate a new line of questioning, leading to the formulation of new hypotheses and the undertaking of additional experiments. This process is repeated until a conclusion is reached that is supported by a substantial body of evidence.

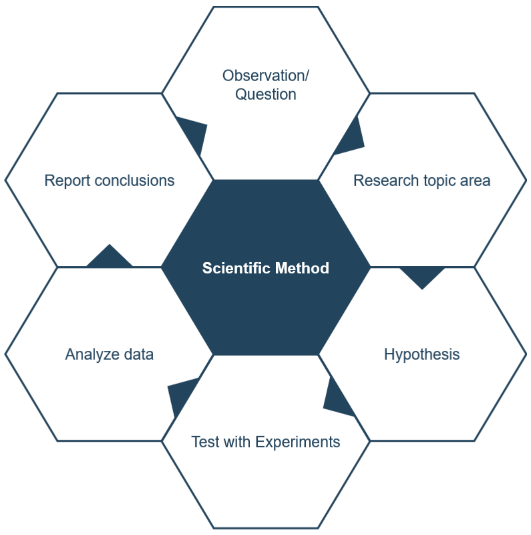


Figure 2.2: Scientific Method

As previously stated in the introduction, academic literature has begun to explore the application of chaos engineering in a number of different fields. For further details, please refer to the following papers: [34], [35], [20], [21], [22].

As illustrated in 1.1, various implementations of chaos engineering have been proposed for diverse applications.

Past studies have employed a range of approaches to chaos engineering research. For instance, [23] utilized risk analysis techniques to enhance understanding of CE principles, while [37] employed the network simulator ns-3 to elucidate the underlying principles of chaos engineering.

3 Background

We will now examine some foundational concepts and definitions that will be referenced throughout this work. Additionally, we will delve deeper into the fundamental principles and algorithms utilized in my implementation.

3.1 Network Graphs

The use of network graphs facilitates experimentation and enables a concentration on the algorithmic process itself. As stated in [18], a graph is defined as:
A **graph** G is a pair $G = (V, E)$ consisting of a finite set $V \neq \emptyset$ called vertices or nodes. The elements of E are called *edges*. Each edge has a set of one or two vertices associated to it, which are called its endpoints. Graphs are often represented by pictures in a plane. The vertices of a graph $G = (V, E)$ are represented by (bold type) points and the edges by lines (preferably straight lines) connecting the end points.

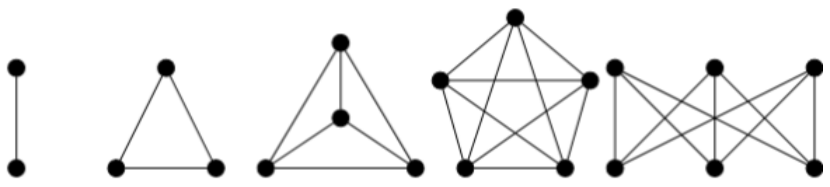


Figure 3.1: Examples of Graphs

A **directed edge** is an edge, one of whose endpoints is designated as the tail, and whose other endpoint is designated as the head.
A **directed graph** (digraph) is a graph each of whose edges is directed.
A **weighted graph** is a graph in which each edge is assigned a number, called its edge-weight.

3.2 Minimum Spanning Tree

Connected Graph

A Graph G is connected if and only if there exists an edge $e = vw$ with $v \in V_1$ and $w \in V_2$ whenever $V = V_1 \cup V_2$ (that is $V_1 \cap V_2 = \emptyset$) is a decomposition of the vertex set of G.
We also say a graph is biconnected if, and only if, it cannot be disconnected by removing only one node.

Subgraph

A **subgraph** or **tree** of a graph G is a graph H whose vertices and edges are all in G. If H is a subgraph of G, we may also say that G is a supergraph of H.

Spanning Tree

A Spanning Tree $G_S = (V_S, E_S)$ is a connected subgraph of a Graph G where $V_S = V$ and $E_S \in E$.

Minimum Spanning Tree

A minimum spanning tree (MST) of a weighted connected graph is a spanning subgraph of the original graph that is both connected and has the smallest possible total edge weight. Figure 4.1 illustrates this concept with an example of a connected graph (top left), a subgraph (top right), a spanning tree (bottom left), and a minimum spanning tree (bottom right).

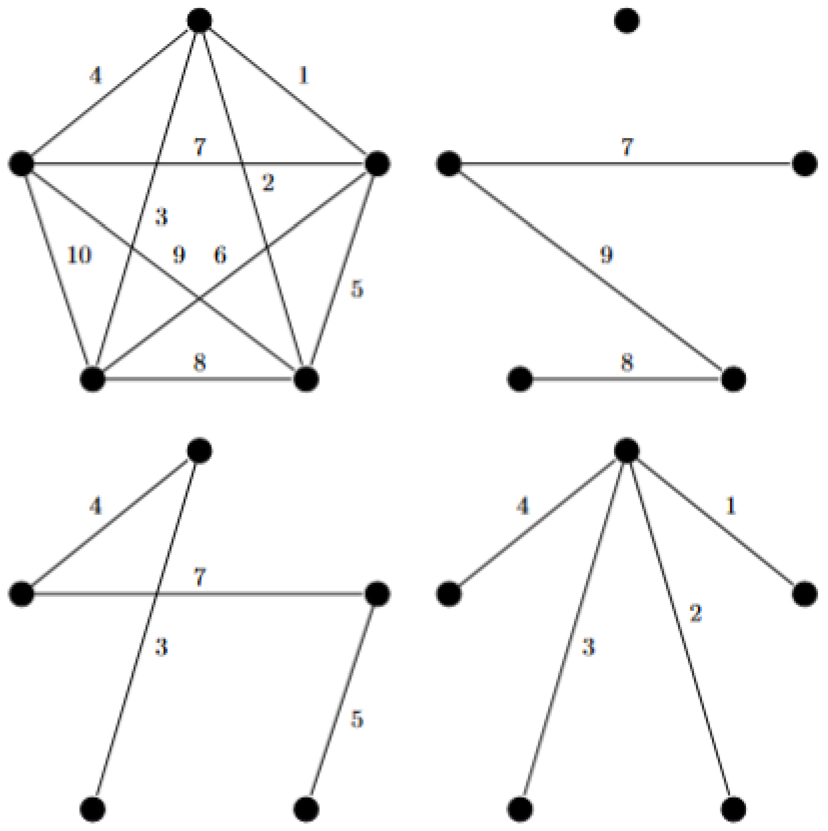


Figure 3.2: Minimum Spanning Tree Example

In [5] an algorithm to find the minimum spanning tree of a weighted graph is shown.
Let $G = (V, E)$ be a connected graph with $V = \{1, ..., n\}$, and let $w : E \longrightarrow \mathbb{R}$ be a weight function for which two distinct edges always have distinct weights.

Algorithm 1 Boruvka $(G = (V, E), w; T)$

```
1: for $i = 1$ to $n$ do $V_i \leftarrow \{i\}$
2: $T \leftarrow \emptyset; M \leftarrow \{V_1, ..., V_n\}$
3: while $|T| < n - 1$ do
4:     for $U \in M$ do
5:         find an edge $e = uv$ with $u \in U, v \notin U$ and $w(e) < w(e')$
6:         for all edges $e' = u'v'$ with $u' \in U, v' \notin U$
7:         find the component $U'$ containing $v$
8:     for $U \in M$ do $MERGE(U, U')$
```

With a runtime of $O(|E|log|V|)$ the algorithm will always find a minimum spanning tree if possible.

3.3 Disjoint Graphs

A mutually disjoint set is defined as follows [1]: A set, A, and a second set, B, are disjoint if the conjunction of the two sets, $A \wedge B$, is the empty set. In a more general sense, S is defined as a system of mutually disjoint sets if $A \wedge B = \emptyset$ for all $A, B \in S$ such that $A \neq B$.

This definition can be applied to graphs, where a disjoint (or mutually exclusive) graph is defined as follows: Two graphs, $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, are considered disjoint when $E_1 \wedge E_2 = \emptyset$ for all $E_1 \in G_1$ and $E_2 \in G_2$. This signifies that all edges of the two graphs are distinct from one another. While the same could be defined for the nodes, it is not a necessary component of this work.

3.4 Greedy Killer

In [32] Greedy Killer gets introduced as a algorithm that fails all edges in 2 iterations if N contains at least 2 disjoint spanning trees. Else at most |V| iterations are needed.
The Greedykiller functions by first looking if the Graph G has two spanning trees T_1 and T_2 that have no shared edges. If this is the case It will fail the edges E_1 first and afterwards E_2. If there are no disjoint spanning trees we set all Edge weights to 1. Afterwards we compute minimum weight spanning trees (MST) and fail all links that are not inside that MST. We also set all the failed edge weights to 0. This ensures that we will compute a different minimum weight spanning tree in the next iteration. If the sum of the edges that should be failed is already zero or all the edges in the graph are zero. The loop ends.

Algorithm 2 Greedy Killer ($G = (V, E)$)

```
1:  if G has spanning trees $T_1 = (V, E_1), T_2 = (V, E_2), E_1 \wedge E_2 = \emptyset$ then
2:      fail $E \backslash E_1$ and fail $E \backslash E_2$
3:  else
4:      $\forall e \in E$ set link weights of $c(e) = 1$
5:      repeat
6:          compute minimum weight spanning tree (MST) $T = (V, E')$
7:          fail all links $E'' = E \backslash E'$ not in the MST T
8:          set sum of new edge failures $\lambda = \sum_{\forall e \in E''} c(e)$
9:          $\forall e \in E''$ set $c(e)$
10:     until $\lambda = 0$ or $\forall e \in E : c(e) = 0$
```

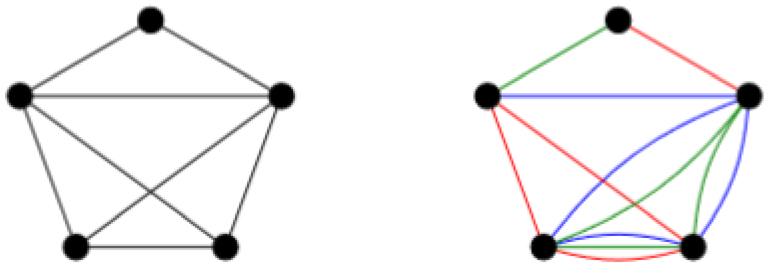


Figure 3.3: Example of killing with Greedykiller. One color simbolizes all edges that are killed together

3.5 Network Flow

[19]
A flow network is modeled as a directed graph, denoted by $G = (V, E)$. The set of vertices, V, represents the nodes through which the network is constructed, while the set of edges, E, comprises the connections between these nodes. The graph is assumed to be connected, although not every edge is necessarily present. A special source vertex, designated by the variable s, produces units of a commodity that flow through the edges of the graph to be consumed by a sink vertex, represented by the variable t, which is also known as the target or terminus. In a flow network, the supply of units produced is assumed to be infinite, and the sink vertex is assumed to be able to consume all units it receives.

For each edge (u, v), a flow f(u, v) is defined, representing the number of units of the commodity flowing from u to v. Additionally, an edge has a fixed capacity c(u, v), limiting the maximum number of units that can traverse that edge. For a flow f through the network to be feasible, the following criteria must be satisfied:

Capacity Constraint The flow $f(u, v)$ through an edge is constrained to be non-negative and cannot exceed the edge capacity $c(u, v)$. This implies that $f(u, v)$ is bounded by 0 and $c(u, v)$. In the absence of an edge from u to v, we set $c(u, v)$ to 0.

Flow Conservation Except for the source vertex s and sink vertex t each vertex $(v, u) \in E$ (the flow into u) must equal the sum of $f(u, w)$ for all edges $(u, w) \in E$ (the flow out of u). THis property ensures that flow is neither produced nor consumed in the network, except at s and t.

3.6 Minimum Cost Flow

In Definition 11.1.1 of [5] the Minimum Cost Flow Problem is defined as following: Let $G = (V, E)$ be a connected digraph, and let the following data be given:

- upper and lower capacity functions $b : E \longrightarrow \mathbb{R}$ and $c : E \longrightarrow \mathbb{R}$, respectively;
- a cost function $\gamma : E \longrightarrow \mathbb{R}$;
- a demand function $d : V \longrightarrow R$ with $\sum_{v \in V} d(v) = 0$

The minimum cost flow problem (MCFP) is concerned with the determination of a mapping $f : E \longrightarrow \mathbb{R}$ with minimal cost, $\gamma(f) = \sum_{e \in E} \gamma(e) f(e)$, subject to the following two conditions:
(F1) $b(e) \leq f(e) \leq c(e) \forall e \in E$ (capacity restrictions);
(F2) $\sum_{e^+ = v} f(e) - \sum_{e^- = v} f(e) = d(v) \forall v \in V$ (demand restrictions).
Vertices with a negative demand are referred to as sources, while vertices with a positive demand are designated as sinks. All other vertices may be regarded as transshipment nodes. A flow is a map $f : E \longrightarrow \mathbb{R}$ that satisfies the demand restrictions for all $v \in V$. If, in addition, the capacity restrictions hold for all $e \in E$,then f is said to be an admissible flow. Therefore, the MCFP seeks an admissible flow of minimum cost.

This definition facilitates the advancement of the concept of ”killing edges.” By employing the network simplex or alternative minimum cost flow algorithms, it is possible to ascertain edges that can be eliminated without affecting the efficiency of the network while ensuring that the network retains sufficient capacity to operate normally. In my implementation, I intend to utilize the NetworkX method min_cost_flow.

3.7 Breadth first search (BFS)

One of the most fundamental methods in algorithmic graph theory is the breadth-first search (BFS), which we will use to identify the minimum flow of the graph. Initially proposed by Moore in 1959, the modified version employed in this thesis is outlined as follows:

Algorithm 3 bfs_min_flow ($G = (V, E), source$)

```
1: $visited \leftarrow \{all\ nodes\ with\ value\ false\}$
2: $queue \leftarrow (source, inf)$
3: $min\_flow \leftarrow inf$
4: while Queue not empty do
5:     $current_node, flow = queue.pop()$
6:     $visited[current\_node] = True$
7:     for $neighboringraph[current\_node]$ do
8:         $capacity \leftarrow capacityofedgebetweencurren\_nodeandneighbor$
9:         if $notvisited[neighbor]$ then
10:            $new\_flow \leftarrow min(flow, capacity)$
11:            $min\_flow \leftarrow min(min\_flow, new\_flow)$
12:            $queue.append((neighbor, new\_flow))$
```

A breadth-first search would be conducted in accordance with the following graph, which would examine the nodes in alphabetical order, commencing with the source node, s.

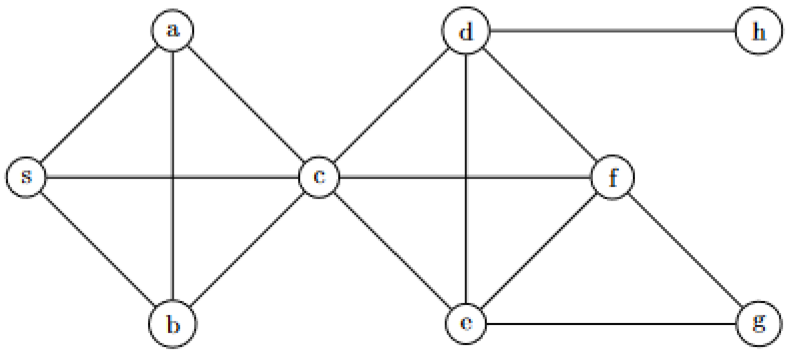


Figure 3.4: Minimum Spanning Tree Example

4 Methods

The following section will present the methods employed in the implementation of the application and the design of the experiment.

4.1 Python

The decision to utilize Python was a relatively straightforward one. Its extensive library of packages, concise syntax, and support for functions all contributed to an efficient and effective development process. The ability to rapidly prototype initial implementations of algorithms, the ease of code reuse through the use of functions, and the flexibility in defining invariants for a killable set of edges were particularly advantageous. Despite my limited prior experience with Python, these factors collectively influenced my decision to employ it in this project.

4.2 NetworkX

For the majority of the implementation process, I utilized the NetworkX Python package. NetworkX characterizes itself as a software package designed for the creation, manipulation, and analysis of complex networks, encompassing their structural, dynamic, and functional attributes. [2]. It provides a plethora of useful functions and algorithms, facilitating the evaluation of given graphs. It is open source and can load, store, and analyze networks, generate new networks, build network models, and draw networks. While it is not considered the state of the art for large-scale problems with fast processing requirements, it is straightforward to use and an excellent option for real-world network analysis. Some of the benefits of NetworkX include [12]:

- Most of the core algorithms rely on extremely fast legacy code
- It uses standard graph algorithms (Dijkstra, Simplex etc.)
- It has an extensive set of native readable and writable formats
- It is easy to install and use on the main platforms with a strong online documentation
- It is ideal for representing networks of different types
- Takes adcantage of Pythons ability to import data from otuside
- NetworkX includes many graph generator functions and facilities to read and write graphs in many formats such as .edgelist, .adjlist, .gml, .graphml, .pajek, etc.

4.2.1 Installation

Assuming pip is installed on your system you can install the networkX library using it and the following command.

```
$ pip install --upgrade networkx[default]
```

4.3 SNDLib

In the context of algorithmic problems, there are a number of standardized benchmark libraries that have been developed for specific problems. One example is TSPlib [4], which is used for the traveling salesman problem. In contrast to these libraries, CE does not have a similar resource, which means that it is necessary to create test scenarios using different network libraries. The Survivable Network Design Library (SNDlib), which was first introduced in 2005, represents a standardized test instance for the design of survivable fixed telecommunication networks [27]. The following code illustrates a condensed example of the Abilene.xml file from the SNDLib. This file identifies the nodes with an ID and their respective coordinates. Subsequently, the links are defined, including the source and target, as well as the capacity and cost. Finally, the SNDLib specifies the demands between sources and edges.

```
<networkStructure>
<nodes coordinatesType="geographical">
<node id="ATLAM5">
<coordinates>
<x>-84.3833</x>
<y>33.75</y>
</coordinates>
</node>
<node id="ATLAng">
.
.
.
</nodes>
<links>
<link id="ATLAM5_ATLAng">
<source>ATLAng</source>
<target>ATLAM5</target>
<preInstalledModule>
<capacity>9920.0</capacity>
```

Listing 4.1: Abilene Example part 1

```
<cost>0.0</cost>
</preInstalledModule>
</link>
<link id="ATLAng_HSTNng">
.
.
.
</links>
</networkStructure>
<demands>
<demand id="IPLSng_STTLng">
<source>IPLSng</source>
<target>STTLng</target>
<demandValue>3580.0</demandValue>
</demand>
<demand id="CHINng_ATLAM5">
.
.
.
</network>
```

Listing 4.2: Abilene Example part 2

4.3.1 Data Concepts

A SNDLib network planning problem consists of two parts:

- A network, describing nodes, links, demands, capacities, cost and some other planning data
- A network planning model, which specifies the planning parameter, e.g., whether links are directed or undirected, whether the routing a demand may be split on several paths or not, or which capacity model or survivablitity concept should be used

SNDLib Network

An SNDlib Network is defined as a representation of the network structure, the traffic to be routed, and a set of admissible routing paths. The set of links defines the potential connections between the nodes that may be used to carry traffic; parallel links are permitted. For each link, the capacity and cost information is specified in terms of the pre-installed capacity and its associated cost, as well as a list of potential capacity modules with their respective costs. Furthermore, a fixed-charge cost for utilizing the link is defined, as well as the flow cost incurred by routing traffic through that link. The set of communication demands delineates the traffic to be routed. Each demand is defined by its end-nodes and the volume of traffic that must be routed through the network, expressed in multiples of a base routing unit. The routing unit of a demand defines the amount of link capacity consumed by one unit of the demand's traffic. This may be expressed in terms of a specific data rate, such as 2 Mbit/s, 155 Mbit/s, or 2.5 Gbit/s. Ultimately, a list of permissible routing paths must be defined. This list may be empty, indicating that all potential paths are acceptable, or it may contain a non-empty set of paths for each demand. In the latter case, a hop limit may be imposed for each demand, representing the maximum number of links permitted for each admissible routing path. If the SNDlib model specifies the use of this hop limit, it further restricts the set of permissible paths.

Network	Nodes	Edges	Demands
Atlanta	15	22	210
Cost266	37	57	1332
DFN-Bwin	10	45	90
DFN-Gwin	11	47	110
Di-Yuan	11	42	22
France	25	45	300
Germany50	50	88	662
Guil39	39	172	1471
Janos-US	26	84	650
Janos-US-CA	39	122	1482
Newyork	16	49	240
Nobel-EU	28	41	378
Nobel-Germany	17	26	121
Nobel-US	14	21	91
Norway	27	51	702
PDH	11	34	24
Pioro40	40	89	780
Polska	12	18	66
Sun	27	102	67
TA1	24	55	396
TA2	65	108	1869
Zib54	54	81	1501

Table 4.1: SNDLib networks

SNDLib Model

An SNDlib model is a specification of a set of selected design parameters. These are defined by the set of attributes and their permissible values given in 4.1, with each attribute requiring a single value. The sole design objective currently under consideration is to minimize total link cost, which includes fixed charge cost, cost of pre-installed capacity, cost of selected capacity modules, and routing cost. This paper will primarily focus on SNDLib Networks.

4.3.2 Parser

It was of great importance to ensure that the edges in the SNFLib Topology had the appropriate capacity, cost, and demand settings when parsing them into NetworkX Graphs. As a preliminary step, I examined the parser located at GitHub [9] and adapted it to suit my requirements. I extracted the method *read_sndlib_topology* from the graph.py file and incorporated the extraction of cost and capacity for each edge.

The utilization of this methodology facilitated the generation of a NetworkX graph and a list of demands, thereby enabling the execution of experiments with the algorithms that will be introduced in chapter 6.

4.4 Topology Zoo

The topology zoo, analogous to SNDLib, is a compendium of over two hundred network topologies from assorted network providers. The collection was established for academic purposes, educational pursuits, and the advancement of knowledge, and is accessible in both GML and GraphML. This enables its direct interpretation by NetworkX, facilitating seamless integration. The sole challenge encountered in parsing the Topologyzoo graphs was the inability of NetworkX to process graphs comprising identical edges on multiple occasions.

The accuracy of the networks is guaranteed by the fact that they are provided by the network providers themselves, thereby ensuring a basis in ground truth. However, as the paper acknowledges, some of the providers produce the maps manually, which may potentially lead to issues. Additionally, some providers may simplify their networks, which could further enhance the reliability of the data.

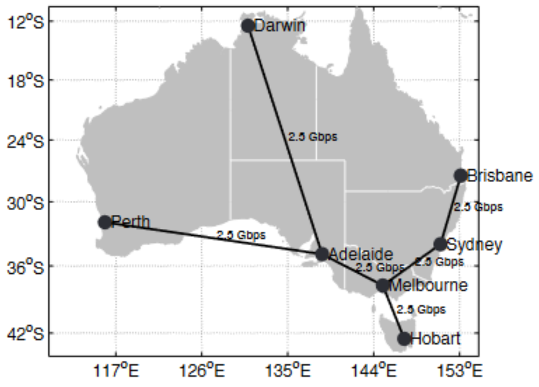


Figure 4.1: Network aus_simple from Topology Zoo

```
graph [
    directed 1
    GeoLocation ""Australia
    GeoExtent ""Country
    Network "aus "simple
    Classification "Backbone, "Transit
    Creator "Topology Zoo "Toolset
    LastAccess ""03/12/10
    Layer ""IP label "aus "simple
    Source ""Example Version ""1.0
    Developed ""Developed
    NetworkDate ""
    hierarchic 1
    Type "COM "
    DateObtained ""
    node [
        id 0
        label ""Perth
        label ""Perth
        Internal 1
    ]
```

Listing 4.3: Abilene Example part 1

```
    node [
        id 1
        label ""Adelaide
        Internal 1
    ]
    .
    .
    edge [
        source 0
        target 1
        Speed ""2.5
        Label "2.5 Gbps, "Ethernet
        Units ""G
        Type ""Ethernet
    ]
    .
    .
]
```

Listing 4.4: Abilene Example part 2

In order to parse the topology zoo, it is possible to simply use the NetworkX function readgml. The decision was taken to use gml files in preference to GraphML because the former are slightly smaller and thus require less storage space. Once the gml files have been parsed, a network graph can be produced which can then be analysed by the algorithms.

5 Experimentdesign

The following section will examine the methodology employed in the experiments. It will discuss the specific parameters utilized and the manner in which the implementation was analyzed. Additionally, it will provide a brief overview of the apparatus utilized to conduct the experiments, with the objective of contextualizing the CPU times that will be subsequently evaluated. When designing the experiment the most important things to consider are the following things:

What can the algorithms do and where might restrictions arrise?

What datasets will be used to test?

What can we test with the datasets given?

Which format do the test results have?

As with Greedy-Killer, the objective of all graphs is to eliminate as many edges as possible within a single iteration. Consequently, the objective is to complete as few iterations as possible. In the event that an edge cannot be eliminated, it will be appended to a list of "survivors," which will be included in the final test results. In contrast to the Greedy-Killer algorithm, this one not only assesses the connectivity of the graphs but also permits the examination of biconnectivity, minimum flow, and maximum cost. For a comprehensive delineation of the constraints imposed on the graphs under examination, please refer to chapter 6.1, subchapter "check_requirements."

In order to test as many different graphs as possible, two different datasets have been selected, both of which exhibit distinct characteristics.
Topologie Zoo: The topology zoo will be utilized due to its extensive data set. The dataset comprises over 522 graphs, exhibiting diverse sizes and topologies.
SNDLib: in contrast, is considerably smaller than the topology zoo. However, it offers the valuable insight of edge-capacity and cost data. This limitation, however, stems from the fact that the tests are designed for algorithms have less data to work with. Nevertheless, the results will undoubtedly provide compelling evidence for the benefits of examining the flow and cost of a graph when removing edges.

In order to ascertain which of the algorithms performs best under different circumstances, three unique runs will be conducted using the given datasets. **First run:** The initial run will be conducted with the following parameters: The initial trial will be conducted with a bot topology and the SNDLib library. This

implies that no flow or cost restrictions will be imposed on the algorithms. This should provide a suitable baseline for the evaluation of the algorithms' performance in a controlled environment.
The results of the SNDLib run will be of particular significance in this instance, as the run is more comparable to the subsequent runs. The initial run is the sole instance in which the topology zoo will be employed.
Second Run: The second run can be divided into three sub-runs. The aforementioned runs differ in their restrictions regarding the maximum cost. The initial subrun will establish the maximum cost such that between 1 and 5 percent of all edges must be eliminated for the algorithm to accept the graph as valid. The second subrun will perform the same function, but with a cost threshold of 10-20%. The final run will ensure that 50% of the edges have a cost that exceeds the threshold for validity. The significant margins for the "must kill edges" are a consequence of the considerable variation in the number of edges across the graphs. If the maximum cost is set in a way that one edge has a cost that is too high (i.e., 5% of a graph with 20 edges, but 1% for a graph with 100).
Note: It should be noted that given the nature of the data in SNDLib, it is not particularly meaningful to impose constraints on the minimum flow. This is due to the fact that the capacity of all edges is identical. Consequently, if the capacity is less than that of a single edge, it is necessarily less than the capacity of all edges. This precludes the possibility of eliminating any edges, which would render the test results uninterpretable. The option of not eliminating edges due to flow restrictions remains available, and it may be worth considering this in future tests. **Third Run:** The third run will be analogous to the second run, with the exception of the method of checking for single connectivity, which will be replaced by a check for biconnectivity. Additionally, the run will proceed at a slower pace, with the objective of limiting the maximum cost.

The tests will be conducted in the following manner:
Each algorithm will be tested in the following order: WorkingKiller, BacktrackKiller, DynamicKiller, GreedyKiller/FlowKiller, Random, BruteKiller. Each algorithm will be executed on each graph, and the results will be recorded in a log file. This file will contain the number of iterations required, the edges that have not been killed, and the number of edges that have not been killed. Additionally, the results will be automatically plotted on a graph, with a blue line representing the iterations and a red line representing the surviving edges.

6 Implementation

We will now examine the implementation of the code itself. Initially, we will analyze the helper functions utilized. Subsequently, we will examine each algorithm, delineating its fundamental concept, implementation, and an evaluation of its advantages and limitations in comparison to the overarching approach. For each algorithm, the pseudocode will be provided as a visual aid to facilitate comprehension of the implementation.

6.1 Helper Functions

As implied by their designation, helper functions facilitate the evaluation of requisite elements for a given graph, or alternatively, the assessment of data yielded by algorithms. They enhance readability and obviate the need for repetitive code blocks.

check_requirements

This function constitutes the primary component of the requirement verification process, which is designed to ascertain the following: The following items are to be considered in the context of the overall requirement check:

- Is the graph connected?
- Checks if the edges of the given graph have both a capacity and a cost.

  In the event that this is not the case, we can return a true value, as in this instance, connectivity is the sole requisite check.
- Checks if the requirement for the minimum flow is given.

  To do this, we use the breadth first search function bfs_min_flow to find the minimum flow. We iterate through all the nodes as sources for this.
- Looks for the lowest cost using find_lowest_weight.

If any of the requirements are not met the function returns false. If all are met it returns true.

find_lowest_weight

Finds the lowest weight (and thus the lowest cost) path of a given graph. For that it checks every source node and ever target node in a graph.

build_residual_network

Returns a residual network of a given graph.

bfs_min_flow

Returns the minimum value of a given residual network using breadth first search.

get_ remaining_edges

Returns the edges that have not been killed by the algorithm and thus determining the survivers.

All the other helper functions should be self explanatory either by name or by code.

6.2 Flowkiller

The Flowkiller is an optimization of the Greedy-Killer, as presented in [32]. In essence, the algorithmic approach remains analogous. The algorithm continues with the next iteration if there are two mutually exclusive minimum spanning trees (MSTs). In the event that no minimum spanning trees are identified, the algorithm proceeds to identify the edges that fail to belong to any of the discovered spanning trees. This process continues until the value of lambda reaches zero. The optimization occurs in two stages. First, the algorithm does not merely ascertain whether the graph remains connected after edge removal; it also verifies that the flow and cost requirements are satisfied and that the graph's specified demands are met. Another optimization was implemented at a relatively late stage of the project, primarily due to the manner in which the NetworkX Minimum Spanning Tree (MST) iterator sorts the MSTs. NetworkX employs a sorting mechanism that places lambda in a relatively early state of zero. This is due to the manner in which the spanning trees are ordered, with edges occurring in the trees at an early stage. This is why I included an additional statement following the initial instance of lambda reaching zero. We verify that all edges have been considered at least once by creating a copy of the original graph's edges. This significantly increases the algorithm's runtime, particularly in cases where edges cannot be eliminated. However, it is a crucial modification to guarantee the algorithm's functionality. As we will discuss in the subsequent chapter, the flowkiller encountered some issues that regrettably resulted in its inoperability.

Pros

- Efficient
- Prioritizes edges with the least impact

Cons

- Evaluating the impact of each edge can be computationally intensive

Algorithm 4 Greedy Flow Killer

```
function GREEDY_FLOW_KILLER($G, demands, max\_cost, min\_cap$)
    if N has spanning trees $T_1 = (V, E_1), T_2 = (V, E_2), E_1 \wedge E_2 = \emptyset$ then
        if $T_1$ and $T_2$ meets requirements then
            fail $E \setminus E_1$ and fail $E \setminus E_2$
    else
        $\forall e \in E$ set link weights of $c(e) = 1$
        repeat
            Iterate through spanning trees $T = (V, E')$
            if $E \setminus E'$ meets requirements then
                fail all links $E'' = E \setminus E'$ not in the MST T
                set sum of new edge failures $\lambda = \sum_{\forall e \in E''} c(e)$
                $\forall e \in E''$ set $c(e)$
            if $\lambda = 0$ then
                if MST with not yet considered edges exists then
                    Return to Line 16 with new MST
                else
                    Break
        until Break
```

6.3 DynamicKiller

Dynamic programming is a recursive method for solving sequential decision problems (SDP). It can be employed to identify optimal strategies and equilivria for a diverse array of sequential decision problems (SDPs) and multiplayer games. It is a practical method for finding solutions to extremely complicated problems [30]. This makes dynamic programming a suitable approach for attempting to resolve the issue raised in this paper.

The DynamicKiller algorithm employs a dynamic programming approach. The general concept is to perform an iterative evaluation of each edge in order to ascertain whether it can be removed without contravening the constraints of the graph. The algorithm employs a dynamic programming table to maintain a record of the maximum number of edges that can be removed while preserving the requisite properties of the graph. The process is continued until all potential removable edges have been identified.

The algorithm initiates by creating a list of all edges in the graph and also generates a dynamic programming table and a list of removable edges. Subsequently, the algorithm iterates through all edges, creating a temporary copy of the graph to remove the edge in question. It then assesses whether the graph still fulfills the specified requirements after the edge removal. The DP table is populated with edges that do not breach the requisite conditions. This process continues until all potential edges have been removed.

Pros

- Structured Approach: Provides a clear and structured method for edge removal.

Cons

- Can be memory-intensive due to the need to store intermediate results in the DP table
- Relatively Complex

Algorithm 5 Dynamic Killer

```
1: function DYNAMIC_KILLER($G$, $demands$, $max\_cost$, $min\_cap$)
2:     $original\_edges \leftarrow$ list of edges in $G$
3:     $dp \leftarrow$ 2D array of size $(n+1) \times (n+1)$ initialized to $0$
4:     $removable\_edges \leftarrow []$
5:     for i in original edges do
6:         $edge\_to\_remove \leftarrow original\_edges\ temp = G$
7:         if $temp$ has edge $edge\_to\_remove$ then
8:             remove $edge$ from $temp$
9:         if check_requirements then
10:            $dp[i][1] \leftarrow 1$
11:            append $[edge\_to\_remove]$ to $removable\_edges$
12:        for $j$ from $2$ to $n$ do
13:            $dp[i][j] \leftarrow dp[i-1][j]$
14:            if $dp[i-1][j-1] > 0$ then
15:                if $temp\_graph$ has $edge\_to\_remove$ then
16:                    remove $edge$
17:                if check_requirements then
18:                    $dp[i][j] \leftarrow \max(dp[i][j], dp[i-1][j-1]+1)$
19:                    if length of $removable\_edges < j$ then
20:                        append $[]$ to $removable\_edges$
21:                    if $edge\_to\_remove$ not in $removable\_edges[j-1]$ then
22:                        append $edge\_to\_remove$ to $removable\_edges[j-1]$
23:                add $edge\_to\_remove$ back to $temp\_graph$
24:    return $removable\_edges$
```

6.4 Iterative Killer

The iterative killer algorithm is arguably the simplest presented in this paper. It iterates through all edges of a graph and eliminates them in a sequential manner. It eliminates all edges that can be eliminated in the same number of iterations as there are edges that can be eliminated. If it were not for a significant drawback discussed in chapter 7, it could be considered a baseline for the other algorithms. Due to its straightforward structure, it is computationally simple, requiring only one loop and no recursion.

Algorithm 6 IterativeKiller

```
1: function ITERATIVE_KILLER($G, max\_cost, min\_cap$)
2:     $original\_edges \leftarrow$ edges in G
3:     $link\_failures \leftarrow []$
4:     for edge in original_edges do
5:         $temp\_graph \leftarrow$ G.copy
6:         remove $edge$ from $temp$
7:         if check_requirements then
8:             append $[edge]$ to $link\_failures$
9:         return $removable\_edges$
```

6.5 WorkingKiller

Although the strategy of eliminating edges in a greedy manner may appear to be a somewhat myopic approach, it has demonstrated a notable degree of efficacy in addressing the issue of edge removal. As demonstrated previously with the Greedy Killer, the object that is best in the moment is selected.
This indicates that the WorkingKiller is analogous to the FlowKiller, which is a greedy algorithm designed to remove edges from a graph in an iterative manner, ensuring that the graph remains connected and meets specific flow and cost requirements. However, there are notable differences in their approach to selecting and removing edges. In contrast to the FlowKiller, which focuses on identifying minimum spanning trees and removing edges that do not belong to them, the Workingkiller employs a distinct methodology.
To identify edges that can be removed, the Working Killer employs a function, find_killable_edges, which initially sorts the remaining edges in order of cost. This guarantees that all edges that are cost-inefficient are eliminated. Subsequently, the edge is eliminated, and the graph's connectivity is assessed. If the graph remains connected, the edges are appended to a list of edges that may be removed to optimize the graph's cost. In the event that the edge has not been previously removed, it is simply added back to the graph. Upon completion of its traversal of the edges, the Working Killer returns the list of killable edges. To avoid modifying the original graph directly, the Working Killer creates a copy of the graph and then identifies the edges that can be removed without violating the requirements. These edges are then removed from the copy graph. The Working Killer then repeats this process until all edges that can be removed without violating the requirements have been removed.

Pros

- Batch Processing: Removes multiple edges in batches, which can be efficient in certain scenarios.
- Simplicity: Easier to implement and understand compared to more complex algorithms.

Cons

- Less Granular: Batch removal may not always lead to the most optimal solution.
- Potential Overkill: Removing multiple edges at once can sometimes lead to unnecessary removals.

Algorithm 7 Working Killer

```
function WORKING_KILLER($G$, $demands$, $max\_cost$, $min\_cap$)
    $killed\_links \leftarrow []$
    $remaining\_edges \leftarrow$ list of edges in $G$
    while $remaining\_edges$ do
        $iteration\_links \leftarrow []$
        $graph\_copy \leftarrow$ copy of $G$
        $killable\_edges \leftarrow$ find_killable_edges
        if $killable\_edges$ is empty then
            break
        for $killable\_edges$ do
            append $edge$ to $iteration\_links$
            remove $edge$ from $remaining\_edges$
        append $iteration\_links$ to $killed\_links$
    return $killed\_links$
```

6.6 BruteKiller

Brute force algorithms are capable of solving a vast array of problems and are relatively straightforward to comprehend. They are employed primarily in the domains of searching, sorting, string matching, and matrix multiplication. Some of these problems exhibit analogous characteristics to those observed in the edge-killing problem, as exemplified by the Brute Force Algorithm Implementation of Dictionary Search. This is why the brute force approach is a viable contender for solving the proposed problem.
The BruteKiller algorithm is an example of a brute-force approach. The fundamental concept is to assess all potential combinations of edges to ascertain which sets of edges can be removed without contravening the graph's constraints. This exhaustive approach guarantees that all potential removable edges are taken into account, thereby providing a comprehensive solution. However, this is achieved at the cost of higher computational complexity.
The algorithm commences with the creation of a list comprising all edges of the graph. Additionally, an empty list is generated for the purpose of monitoring the edges that have been eliminated. Subsequently, all possible subsets of the edges are considered. For each subset of edges, the algorithm creates a temporary copy of the graph and removes the edges in the subset from this copy. If the temporary graph still meets the requirements, the edges are added to the list of killed links.
Another method for implementing a brute force approach is to attempt to kill all edges individually. If the list of remaining edges is empty at any point, the algorithm stops. This results in the Brutekiller requiring |E| iterations to kill all edges, rendering it a less compelling approach.

Pros

- Comprehensive: Evaluates all possible combinations of edge removals, ensuring no potential solution is missed.
- Simplicity: Conceptually simple and easy to understand

Cons

- Computationally Expensive: Extremely high computational cost due to evaluating all combinations.
- Scalability: Not suitable for large graphs due to exponential growth in combinations.

Algorithm 8 Brute Killer

```
function CLEVER_BRUTE_KILLER($G$, $demands$, $max\_cost$, $min\_cap$)
    $edges \leftarrow$ list of edges in $G$
    $remaining\_links \leftarrow edges.copy$
    $link\_failures \leftarrow []$
    for $r$ from 1 to length of $edges$ do
        for each subset of $edges$ of size $r$ do
            $temp\_graph \leftarrow$ copy of $G$
            remove edges in $subset$ from $temp\_graph$
            if check_requirements($temp\_graph$, $demands$, $max\_cost$, $min\_cap$) then
                link_failures.append(subset)
                for edge in subset do
                    if edge in remaining_edges then remaining_edges.remove(edge)
    return $killed\_links$
```

6.7 Backtrackkiller

The BacktrackKiller algorithm employs a backtracking approach to identify and remove edges. The general concept is to systematically examine all potential combinations of edge removals, employing a backtracking approach whenever a removal results in a violation of the constraints. This approach enables the algorithm to identify an optimal or near-optimal set of removable edges by exploring different paths and undoing changes when necessary.
After initializing in a manner similar to that used by other algorithms, it proceeds through the original edges in a recursive function. Subsequently, the edges are temporarily removed and the requirements are evaluated. Thereafter, the recursive function attempts to remove additional edges. In the event that a removed edge violates the requirements, it is restored. The recursion terminates when the remaining_edges is empty.

- Systematic Exploration: Uses backtracking to explore different combinations, potentially finding optimal solutions.
- Flexibility: Can be adapted to various constraints and requirements.

Cons

- Complexity: Can be complex to implement and understand.
- Performance: May still be computationally expensive, especially for large graphs.

Algorithm 9 Backtrack Killer

```
1: function BACKTRACK_KILLER($G$, $demands$, $max\_cost$, $min\_cap$)
2:     $original\_edges$ ← list of edges in $G$ with data
3:     $killed\_edge\_sets$ ← []
4:     $all\_edges\_killed$ ← empty set
5:     function RECURSIVE_REMOVE($temp\_graph$, $current\_set$)
6:         for $original\_edges$ do
7:             if not in $current\_set$ then
8:                 remove $edge$ from $temp\_graph$
9:                 add $edge$ to $current\_set$
10:                if check_requirements then
11:                    append $current\_set$ to $killed\_edge\_sets$
12:                    update $all\_edges\_killed$ with $current\_set$
13:                    if $all\_edges\_killed$ equals set of all edges in $original\_edges$ then
14:                        Try removing edge
15:                        Except return
16:                    RECURSIVE_REMOVE($temp\_graph$, $current\_set$)
17:    return $killed\_edge\_sets$
```

6.8 Randomkiller

The general concept is to incorporate randomness into the edge removal process, which may occasionally result in the identification of removable edges that other, more systematic algorithms may overlook. Currently, the random killer is largely analogous to the brute killer. The primary distinction between this approach and the brute killer is that it first consolidates the list of original edges before attempting to remove them. The outcomes are highly variable, and a detailed examination of this phenomenon will be presented at a later stage. At this juncture, it is crucial to emphasise that this method does not introduce a novel approach to edge removal, as it employs a similar strategy to the brute killer.

7 Evaluation

This chapter will present the findings of the conducted tests. The aforementioned tests have been previously discussed in the chapter dedicated to the design of the algorithm. As previously stated, the focus of this discussion will be on the analysis of the algorithms from a connectivity standpoint, in alignment with the definition presented [32].
Subsequently, the outcomes of the tests conducted with heightened requirements will be examined. Following this, the discussion will shift to the analysis of runtime, CPU time, and the identification of any potential issues encountered during the implementation and the generation of results by the algorithm.

7.1 Flowkiller

The implementation of the flowkiller encountered some difficulties, which occasionally made it challenging to work with. The Greedykiller is primarily concerned with minimum spanning trees, with the exception of instances where two distinct spanning trees are present. This resulted in the Greedykiller being highly inflexible and difficult to optimize. A variety of approaches have been employed in an effort to develop a functional Flowkiller. In particular, the evaluation phase revealed the source of the issues.
To address these challenges, it is necessary to revisit the Greedykiller. As previously indicated, in the absence of two distinct spanning trees, the algorithm will examine the minimum spanning tree of the graph and eliminate all edges not present in it. Subsequently, the weight of all edges that have been eliminated is set to zero. This renders it highly probable that the subsequent minimum spanning tree will encompass the previously eliminated edges, thereby resulting in the removal of edges that have not yet been affected. The issue arises when edges that are not in the minimum spanning tree cannot be eliminated due to, for instance, cost constraints. This results in the edge weight of these edges not being set to zero, preventing the minimum spanning tree from changing. Consequently, a loop is formed that will persist indefinitely, as the spanning trees remain constant and lambda is unable to reach zero.
Over time, I have made multiple attempts to modify the greedy killer to a version I have named "Flowkiller." In each iteration, I have retained the original identity while modifying the functionality to align with the requirements of my thesis. The most promising approach was to utilize a spanning tree iterator. Iterating through all the spanning trees would be an extremely inefficient use of computational resources, resulting in a significant increase in runtime. Consequently, I attempted a comparable methodology to that employed by the original greedy killer. The use of lambda as a means of terminating the loop at the earliest opportunity, once a set of killable edges that have already been killed has been identified, represents a potential solution to this problem. The issue with this approach was related to the manner in which the spanning tree iterator sorted the spanning trees. The iterator had the spanning trees arranged in a manner that resulted in the same edges being examined repeatedly before reaching those situated later in the graph. This rendered the aforementioned approach impractical. To address this issue, a provisional solution was implemented whereby, upon reaching zero, the lambda variable was set to iterate through

all edges not yet set to zero, with the objective of identifying a spanning tree comprising these edges. This approach proved effective for smaller graphs but was infeasible for larger ones, as it required as many iterations as there were edges, rendering it impractical.
For the time being, we will limit our testing to the original Greedy Killer, which does not allow additional constraints.

7.1.1 Without Requirements

The test results of the Greedy Killer without additional requirements are satisfactory. The runtime was consistent with the aforementioned description by [32].

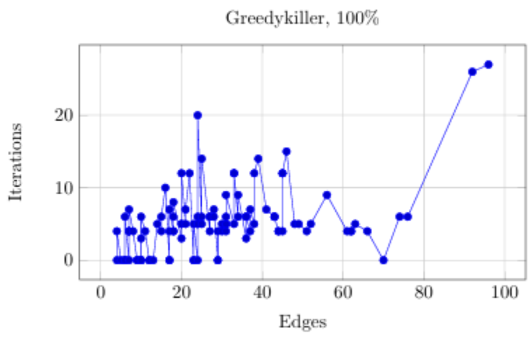


Figure 7.1: Topology Zoo run

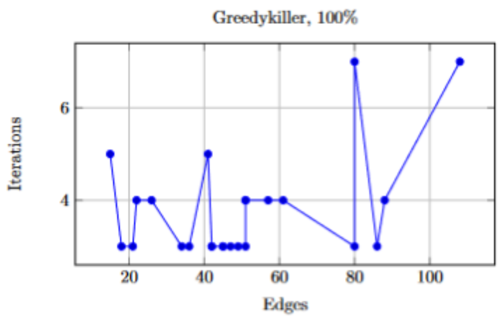


Figure 7.2: SNDLib Run

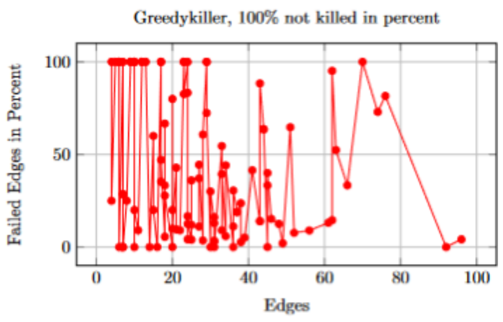


Figure 7.3: Topology Zoo run

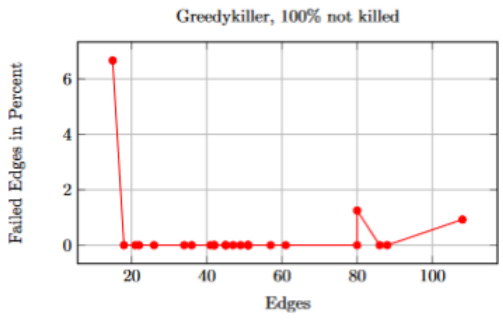


Figure 7.4: SNDLib Run

7.2 Dynamickiller/ Iterative Killer

Dynamic Killer is somewhat anomalous. It exhibits a tendency to throw out one edge at a time, with a low frequency of instances where two edges are thrown out simultaneously. This results in a relatively slow pace. It is conceivable that a method could be devised to facilitate the simultaneous deployment of multiple edges by the DynamicKiller; however, no viable approach could be identified. At this time, the runtime of the DynamicKiller is reasonable in comparison to that of Brutekiller. The algorithm exhibits some degree of intelligence but is not particularly effective. This results in a runtime of approximately

$|E|$, necessitating an iteration for each edge that can be eliminated. This indicates that a more computationally efficient approach would be to iterate through all edges and eliminate them individually, provided that the edge elimination does not contravene the graph's underlying constraints. This indicates that the IterativeKiller illustrated in chapter 6 yields identical outcomes to the DynamicKiller, yet with a considerably superior runtime. Henceforth, the results will be presented by the IterativeKiller, as previously stated, given that its introduction renders the DynamicKiller redundant.

7.2.1 Without Requirements

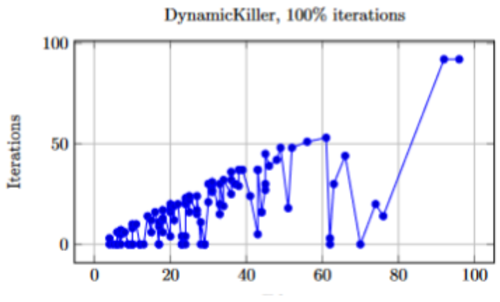


Figure 7.5: Topology Zoo run

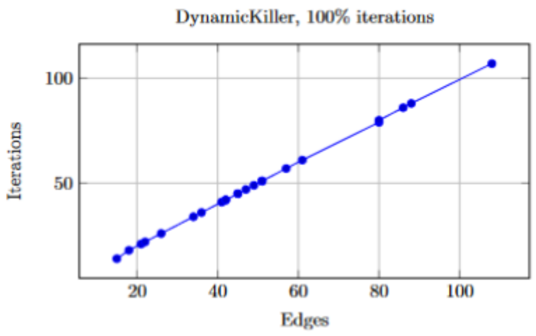


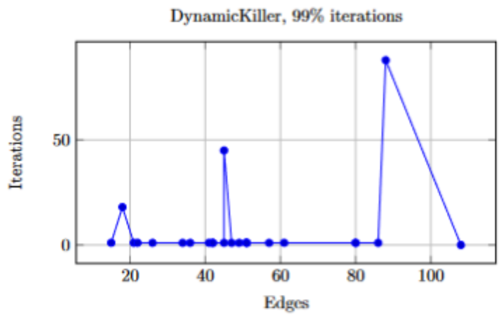
Figure 7.6: SNDLib Run

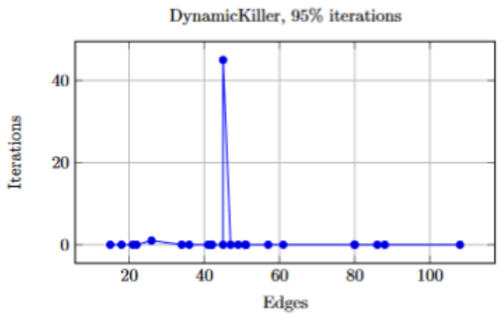
Topologyzoo: An examination of 7.5 reveals that the interative killer requires a greater number of iterations for a greater number of edges. This is in contrast to the other algorithms, which demonstrate a more consistent pattern. The other algorithms are more dependent on the specific layout of the graph than on the number of edges.

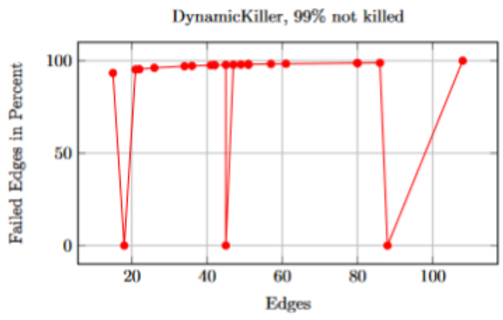
SNDLib: Upon examination of 7.6, it becomes evident that the image is even more discernible. It is observed that the graph displays a nearly linear trajectory, with only slight deviations due to the presence of edges that cannot be eliminated due to connectivity constraints. The data suggests that dynamic programming may not be an optimal approach for chaos engineering.

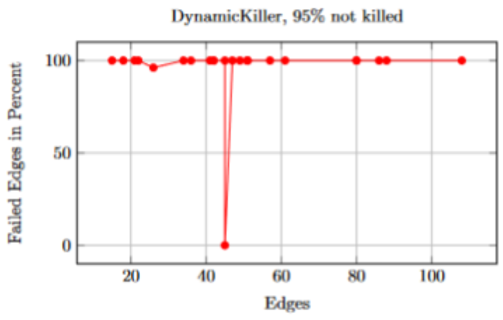
7.2.2 With Requirements

In examining the case of runs with requirements, it becomes evident that IterativeKiller encounters significant challenges in eliminating edges as soon as the requirements are tightened. This results in the survival of the entire graph in instances where a few edges cannot be removed. This is due to the fact that it has difficulties in identifying a valid graph, as all the graphs that it examines have those edges with excessive costs. The only instances in which it is able to eliminate the majority of edges are when there is only one edge that needs to be removed for the rest of the graph to be valid. Similarly to the other algorithms, once only 95% of all edges are killable, it simply allows the majority of graphs to pass through.


DynamicKiller, 99% iterations
Iterations
Edges
0
50
20
40
60
80
100


DynamicKiller, 95% iterations
Iterations
Edges
0
20
40
20
40
60
80
100


DynamicKiller, 99% not killed
Failed Edges in Percent
Edges
0
50
100
20
40
60
80
100


DynamicKiller, 95% not killed
Failed Edges in Percent
Edges
0
50
100
20
40
60
80
100

7.3 Workingkiller

It can be reasonably argued that the working killer approach represents a superior methodology for the implementation of a greedy algorithm. This is due to the fact that the method in question does not employ minimum spanning trees. The efficacy of the greedy approach can be discerned by examining the diagrams. The Working Killer algorithm requires the fewest iterations to eliminate all edges that can be killed. It is not only the most efficient algorithm in terms of the number of iterations but also one of the fastest and most consistent in terms of real-world runtime. In many cases, it terminates each edge of the graph in two iterations, similar to the greedy killer algorithm. The performance of the Working Killer algorithm is generally comparable to that of the greedy killer algorithm, which is to be expected given the similarities in their implementations.

7.3.1 Without Requirements

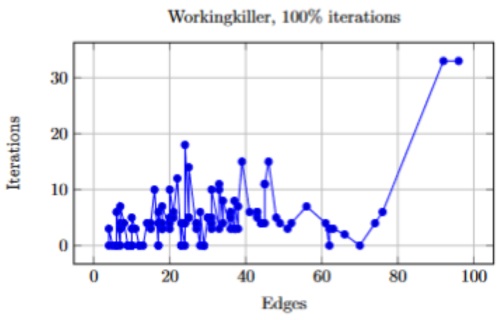


Figure 7.7: Topology Zoo run

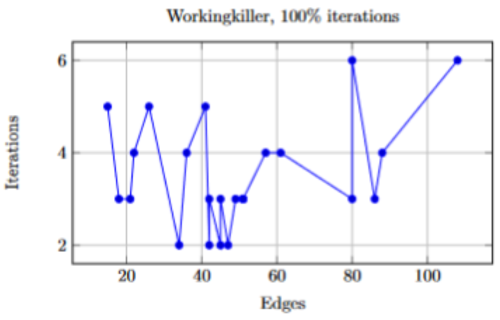
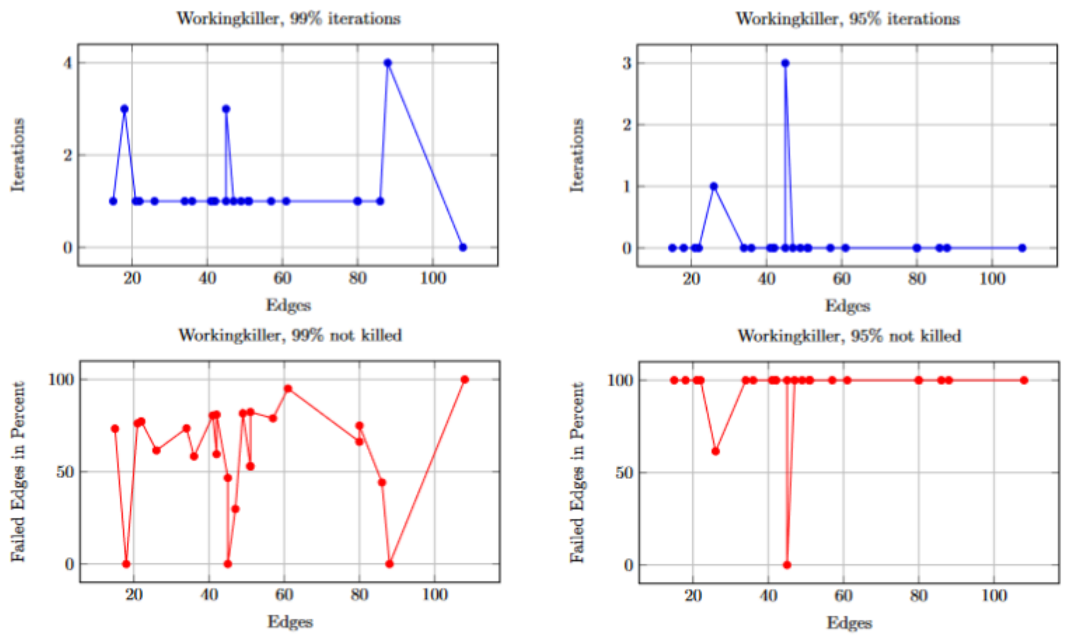


Figure 7.8: SNDLib Run

The Surviving edges look the same as with the two algorithms before. Thus there is no need to look at them again.

7.3.2 With Requirements

The WorkingKiller program is able to meet the more challenging specifications with reasonable effectiveness. In particular, in comparison to GreedyKiller and DynamicKiller. The termination of the algorithm is not impeded by the more stringent requirements and is also less sensitive to a few ”must-kill” edges. However, it should be noted that its performance in terms of requirement testing is not as robust as that of some other algorithms. In this regard, the most favorable aspect of Working Killer is that it does not unduly affect the algorithm’s overall performance when it is unable to eliminate all edges.



7.4 Backtrackkiller

7.4.1 Without Requirements

Even without requirements, the Backtrack Killer demonstrates suboptimal performance. It frequently necessitates a greater number of iterations than the number of edges, and even in the infrequent instances where it requires a lesser number of iterations than edges, the number of iterations it requires remains relatively high in comparison to the other algorithms. This renders it largely inoperable in these scenarios.

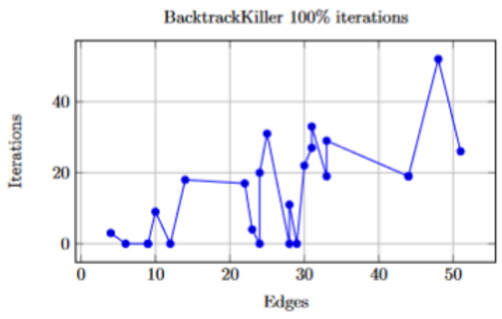


Figure 7.9: Topology Zoo run

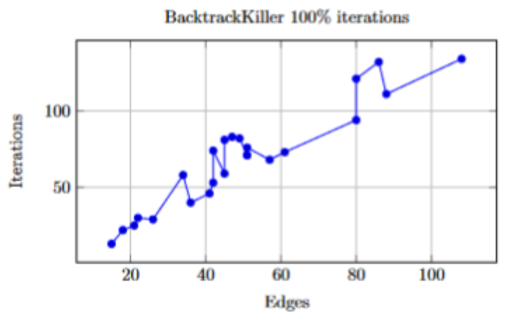


Figure 7.10: SNDLib Run

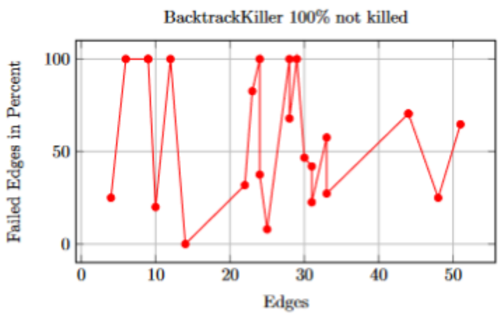


Figure 7.11: Topology Zoo run

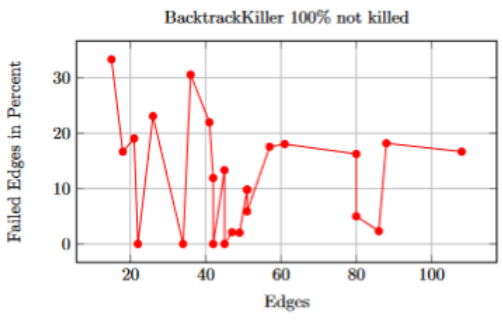
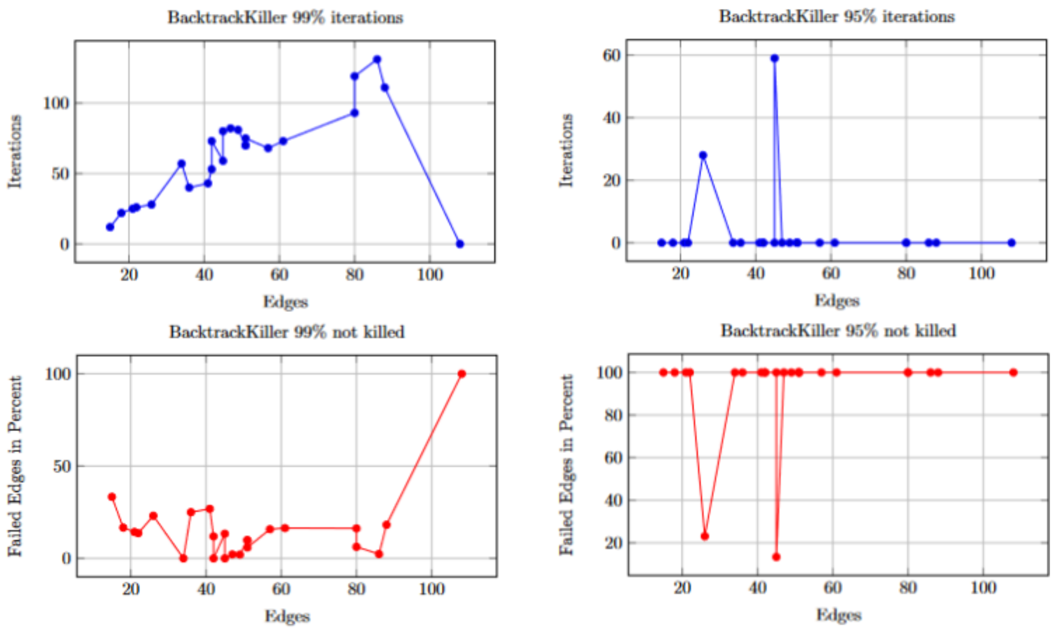


Figure 7.12: SNDLib Run

7.4.2 With Requirements

This represents a significant advantage of the BacktrackKiller. The algorithm is highly effective at circumventing "must-kill" edges while still eliminating a significant portion of the graph, despite the inherent uncertainty in this process. However, its performance remains relatively slow. As evidenced by the fact that it encounters the same limitations as other algorithms when only 95% of the graph remains viable, it is evident that the graph's functionality is contingent upon the presence of the top 5% of the most costly edges.



7.5 Brutekiller/Randomkiller

In the following section, the Brutekiller and Randomkiller will be discussed in conjunction with one another, as they exhibit a high degree of functional similarity. Neither of the two algorithms was terminated by either the dataset nor the requirement. Therefore, there is no data to be discussed. As evidenced in the log file, the real-world runtime is exceptionally lengthy, necessitating a minimum of as many iterations as there are edges in order to eliminate all the edges. The inability to terminate is attributed to the inefficiency of the algorithm, which is incapable of processing large-scale graphs. Even if it were to function, as previously discussed, the algorithm necessitates a considerable number of iterations to eliminate all edges. Consequently, it was determined that an investigation into the brute force and, by extension, random algorithms was not a fruitful endeavor.

7.6 Overall

In summary, it can be stated that two algorithms have been identified which exhibit clear advantages over the remainder. Firstly, the Working Killer is the fastest algorithm in terms of both speed and iterations. Furthermore, it is capable of eliminating the majority of edges, even when the requirements are more stringent. The Backtrack Killer has the advantage of being able to eliminate a greater number of edges when the requirements are more stringent. This results in a situation in which the optimal algorithm for chaos engineering with specific requirements is dependent on the particular use case and the relative importance of efficiency versus thoroughness. Additionally, five algorithms are evidentally unsuitable for this particular use case. The GreedyKiller is unsuitable for use in this context, as it is unable to handle requirements that are not connectivity. Similarly, the Dynamic/Iterative Killer has numerous issues when edges must be killed for a graph to function. Finally, the Brute/Random Killer is so inefficient that it is unable to function at all.
7.1 provides a comparison of the results of our algorithms, which supports this conclusion. It is likely that some of the algorithms could be optimized to enhance their effectiveness and improve their performance. The evidence suggests that Greedy Algorithms are the optimal choice for this use case. In the conclusion, we will discuss alternative approaches, but for now, we can state with certainty that Chaos Engineering algorithms based on Greedy Algorithms are effective, fast, and elegant.
7.2 illustrates the test runs that demonstrate biconnectivity. As only the WorkingKiller, IterativeKiller, and BacktrackKiller have yielded usable results with reduced edge counts, the table will focus exclusively on these algorithms. The tests with biconnectivity demonstrate two key findings: firstly, that biconnectivity is a necessary condition for the algorithms to perform optimally; and secondly, that there are instances where no edges can be eliminated in any given graph. It can also be stated that, in general, the majority of graphs in the TopologyZoo are not biconnected. This is a cause for concern, given that the TopologyZoo contains real provider networks.

Mean number of edges in SNDLib: 49.83
Mean number of edges in Topology Zoo: 29.2

Algorithm	Dataset	Mean Iterations	Mean Survivers in percent	Realworld Runtime in seconds
Greedy Killer	Topology Zoo	5.42	41.58	0.5
	SNDLib 100%	3.8	0.37	0.1
IterativeKiller	Topology Zoo	17.67	43.54	1
	SNDLib 100%	49.71	0.37	0.3
	SNDLib 99%	7.125	85.33	0.2
	SNDLib 95%	1.91	95.67	0.1
	SNDLib 50%	1.88	95.83	0.1
Working Killer	Topology Zoo	4.8	43.54	0.3
	SNDLib 100%	3.54	0.37	0.5
	SNDLib 99%	1.25	60.28	0.5
	SNDLib 95%	0.167	94.23	0.5
	SNDLib 50%	0.125	95.83	0.5
Backtrackkiller	Topology Zoo	14.95	58.30	0.1
	SNDLib 100%	68.58	12.23	1
	SNDLib 99%	62.13	15.95	1
	SNDLib 95%	3.63	93.18	1
	SNDLib 50%	2.46	96.39	1
Brute- & Randomkiller	Topology Zoo	NaN	NaN	NaN
	SNDLib 100%	NaN	NaN	NaN
	SNDLib 99%	NaN	NaN	NaN
	SNDLib 95%	NaN	NaN	NaN
	SNDLib 50%	NaN	NaN	NaN

Table 7.1: Testresults

Algorithm	**Dataset**	**Mean Iterations**	**Mean Survivers in percent**	**Realworld Runtime in seconds**
Iterative Killer	Topology Zoo	0.76	97.06	1
	SNDLib 100%	32.21	32.21	6
	SNDLib 99%	3.96	92.26	5
	SNDLib 95%	3.46	93.67	5
	SNDLib 50%	0	100	5
Working Killer	Topology Zoo	0.13	97.06	1
	SNDLib 100%	2.71	34.25	1
	SNDLib 99%	0.875	65.91	5
	SNDLib 95%	0.375	92.1	5
	SNDLib 50%	0	100	5
Backtrackkiller	Topology Zoo	0.81	99.10	1 21087
	SNDLib 100%	41.42	51.39	5
	SNDLib 99%	33.25	62.01	5
	SNDLib 95%	4.375	94.6	5
	SNDLib 50%	0	100	5

Table 7.2: Biconnected Testresults

8 Conclusion

As the evaluation has demonstrated, the elimination of edges with supplementary invariants is feasible; however, it markedly affects the performance of the utilized algorithm and the number of edges that can be removed from a graph. These outcomes are anticipated and the conclusion should not be to disregard those edge cases. In my estimation, the most pertinent insights to be derived from the tests are as follows:

- It is imperative that network graphs be designed to withstand a multitude of potential failures. Furthermore, it is essential to recognize that costs and flows are variables that may fluctuate in the event of any type of disaster.
- It is imperative that network invariants be realistic. Furthermore, the expected cost should be zero, and the flow should exceed the capacity of the edges in question. The application of chaos engineering will inevitably yield unfeasible results.

It is my contention that Netflix's approach demonstrates both of these points in an exemplary manner. The company ensures network resilience through the implementation of a comprehensive testing strategy, which is designed to identify and address potential weaknesses through a process known as chaos engineering. Additionally, they have established attainable objectives, namely, to enable users to efficiently locate and view content. This is the extent of the functionality that the majority of users require. As previously stated in the introduction, users may be inclined to abandon a service if they perceive it to be unresponsive, leading to potential subscription cancellations. Therefore, it is evident that Netflix should integrate network flow into their experimentation, if this has not already been done. Additionally, it is likely that Netflix is interested in reducing networking costs for their services. Consequently, the two main invariants considered in this thesis are beneficial when further experimenting with Chaos Engineering.

Another significant insight to be derived from the evaluation is the distinction between various algorithmic approaches, as previously discussed in Chapter 7. Greedy algorithms demonstrated superior performance in most scenarios, except when the objective was to eliminate as many edges as possible in challenging situations. In such cases, a backtracking algorithm proved more effective. It is recommended that multiple Chaos Engineering algorithms be implemented to ensure optimal results.

In conclusion, it can be stated that network security should be considered from multiple perspectives, and that Chaos Engineering will become a standard practice across industries and fields due to its distinctive approach and efficacy.

8.1 Problems and Challenges

8.1.1 Finding an efficient maximum cost algorithm

A variety of approaches were attempted to identify an effective method for integrating cost and flow management. One might initially assume that the max_flow_min_cost function is the optimal approach for addressing this metric. However, this may not be the case when working with large graphs. The issue with this approach is that the NetworkX library employs the simplex algorithm to obtain the necessary data. However, the simplex algorithm has been shown to be highly inefficient, leading to a significant slowdown in the algorithms. In some cases, the runtime exceeded 30 minutes per run, with failures occurring for reasons that could not be identified. This made it impractical to run the tests overnight, as was originally intended.

8.2 Outlook

There are numerous avenues for further enhancement of the methodology. One potential avenue for improvement would be to enhance the datasets, either by utilizing larger ones or by identifying and examining previously unconsidered edge cases. This encompasses, but is not limited to, the following:

The creation or identification of datasets that facilitate the evaluation of the flow requirements of the graphs. As has been previously indicated, neither SNDLib nor the Topology Zoo can be considered as a factor in the capacity of the datasets, as this is not a parameter that can be evaluated by the algorithm. The issue with SNDLib is that all of the capacities are identical, while in the case of Topology Zoo, the edges lack any associated values.

Furthermore, the inclusion of additional factors, such as demands, in addition to cost and flow requirements, is essential. In particular, it is essential to examine the demands as defined by SNDLib. In such a scenario, there is a compelling rationale for employing an algorithm akin to the simplex algorithm, which can simultaneously consider cost, flow, and demands.

The identification of larger datasets that challenge the limits of the algorithms represents a promising avenue for further investigation, with the WorkingKiller algorithm offering particular potential for experimentation.

The combination of diverse invariants could also be a fruitful area of exploration, contingent on the definition of a valid user experience. There may be additional invariants that have not been considered in this context.

A second potential avenue for improvement is the modification of the algorithms themselves. As one of the primary objectives of this thesis was to evaluate and enhance the efficacy of diverse methodologies, I did not dedicate sufficient attention to fully optimizing each approach. Naturally, efforts were made to ensure comparability; however, as evidenced in Chapter 7, the results remained unsatisfactory. There are multiple avenues for enhancing the algorithms. One approach would be to rewrite the algorithm to yield superior outcomes or a more comprehensive result set. Another potential avenue would be to modify the requirements verification process to better align with the specific approach. Currently, all algorithms employ the same verification function, and depending on the approach, there might be a more optimal approach for that particular use case.

It may also be of interest to consider alternative approaches. As previously noted, given the objectives of this thesis, I did not focus on entirely new ideas with regard to the development of algorithms. Instead, I conducted a comparative analysis, which I believe represents a valuable baseline for evaluating any future approach that may emerge.

One potential avenue for exploration is the application of machine learning. The application of machine learning has had a profound impact on numerous fields, particularly in the domain of computer science.

Given the success of this approach in other areas, it seems reasonable to conclude that it could also be a valuable tool in the field of chaos engineering. The evidence presented in this study [17] indicates a growing interest in the integration of artificial intelligence (AI) and machine learning in the field of chaos engineering. An algorithm that is capable of discerning the crucial invariance of a network and adjusting its parameters and methodologies in accordance with this understanding could prove to be a highly effective tool, not only in terms of overall performance but also in the context of personalization. Service Providers A and B may have disparate priorities for their networks. Some may prioritize maximizing network flow or minimizing cost, while others may prioritize maintaining connectivity at multiple levels. A machine learning algorithm could potentially address these diverse objectives through individualized solutions.

As previously indicated in the reference [32] , an additional measure that could be employed is the utilisation of more robust coverage properties, which would serve to mitigate the likelihood of failure scenarios.

In conclusion, this thesis examines the effect of cost and flow requirements on different edge-killing algorithms. It is clear that greedy algorithms perform best for this approach. The field of chaos engineering is a promising avenue for future network security and reliability.

9 Use of AI

I utilized the DeepL Write software to rectify grammatical and spelling errors, as well as to identify optimal wording. I am solely responsible for all content included in this thesis; the AI tool merely served to facilitate the editing process.

Example Query:

The algorithms that are commented out are those that have difficulties when processing certain data sets and therefore cannot be expected to function smoothly.
⇒ The commented-out algorithms are those that that encounter difficulties when processing specific data sets and therefore cannot be expected to function optimally.

Bibliography

[1] [n. d.]. Introduction to Set Theory, Revised and Expanded | Karel Hrbacek, Thom. https://www.taylorfrancis.com/books/mono/10.1201/9781315274096/introduction-set-theory-revised-expanded-karel-hrbacek-earl-taft-thomas-jech-zuhair-nashed

[2] [n. d.]. NetworkX — NetworkX documentation. https://networkx.org/

[3] [n. d.]. Resilience Testing Tools - AWS Fault Injection Service - AWS. https://aws.amazon.com/fis/

[4] [n. d.]. TSPLIB—A Traveling Salesman Problem Library | ORSA Journal on Computing. https://pubsonline.informs.org/doi/abs/10.1287/ijoc.3.4.376

[5] 2005. The Network Simplex Algorithm. In *Graphs, Networks and Algorithms*, Dieter Jungnickel (Ed.). Springer, Berlin, Heidelberg, 321–339. https://doi.org/10.1007/3-540-26908-8_11

[6] 2014. EC2 Maintenance Update II | AWS News Blog. https://aws.amazon.com/blogs/aws/ec2-maintenance-update-2/ Section: Amazon EC2.

[7] 2017. PowerfulSeal: A testing tool for Kubernetes clusters | Bloomberg LP. *Bloomberg L.P.* (Dec. 2017). https://www.bloomberg.com/company/stories/powerfulseal-testing-tool-kubernetes-clusters/

[8] 2024. Facebook Turned Off Entire Data Center to Test Resiliency. https://www.datacenterknowledge.com/hyperscalers/facebook-turned-off-entire-data-center-to-test-resiliency

[9] 2024. Optum/ChaoSlingr. https://github.com/Optum/ChaoSlingr original-date: 2019-03-27T13:52:02Z.

[10] 2024. Using SRE and disaster recovery testing principles in production. https://cloud.google.com/blog/products/management-tools/shrinking-the-time-to-mitigate-production-incidents

[11] 2024. Weathering the Unexpected - ACM Queue. https://queue.acm.org/detail.cfm?id=2371516

[12] Mohammed Zuhair Al-Taie and Seifedine Kadry. 2017. *Python for Graph and Network Analysis.* Springer International Publishing, Cham. https://doi.org/10.1007/978-3-319-53004-8

[13] Peter Alvaro, Kolton Andrus, Chris Sanden, Casey Rosenthal, Ali Basiri, and Lorin Hochstein. 2016. Automating Failure Testing Research at Internet Scale. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC '16)*. Association for Computing Machinery, New York, NY, USA, 17–28. https://doi.org/10.1145/2987550.2987555

[14] Microsoft Azure. 2015. Inside Azure Search: Chaos Engineering. https://azure.microsoft.com/en-us/blog/inside-azure-search-chaos-engineering/

[15] Ali Basiri, Niosha Behnam, Ruud de Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal. 2016. Chaos Engineering. *IEEE Software* 33, 3 (May 2016), 35–41. https://doi.org/10.1109/MS.2016.60 Conference Name: IEEE Software.

[16] Netflix Technology Blog. 2017. Chaos Engineering Upgraded. https://netflixtechblog.com/chaos-engineering-upgraded-878d341f15fa

[17] Lorenzo Barberis Canonico, Vimal Vakeel, James Dominic, Paige Rodeghero, and Nathan McNeese. 2020. Human-AI Partnerships for Chaos Engineering. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW'20)*. Association for Computing Machinery, New York, NY, USA, 499–503. https://doi.org/10.1145/3387940.3391493

[18] Noah Chang and Jenny Nichols. [n. d.]. Introduction to Graph Theory. ([n. d.]).

[19] George T. Heineman, Gary Pollice, and Stanley Selkow. 2009. *Algorithms in a Nutshell*. "O'Reilly Media, Inc.". Google-Books-ID: nrrNCwAAQBAJ.

[20] Hiroki Ikeuchi, Jiawen Ge, Yoichi Matsuo, and Keishiro Watanabe. 2020. A Framework for Automatic Failure Recovery in ICT Systems by Deep Reinforcement Learning. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. 1310–1315. https://doi.org/10.1109/ICDCS47774.2020.00170 ISSN: 2575-8411.

[21] Hiroki Ikeuchi, Yosuke Takahashi, Kotaro Matsuda, and Tsuyoshi Toyono. 2021. Recovery Process Visualization based on Automaton Construction. In *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. 10–18. https://ieeexplore.ieee.org/abstract/document/9464009 ISSN: 1573-0077.

[22] Zhenlan Ji, Pingchuan Ma, and Shuai Wang. 2023. Perfce: Performance Debugging on Databases with Chaos Engineering-Enhanced Causality Analysis. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1454–1466. https://doi.org/10.1109/ASE56229.2023.00106 ISSN: 2643-1572.

[23] Dominik Kesim, André van Hoorn, Sebastian Frank, and Matthias Häussler. 2020. Identifying and Prioritizing Chaos Experiments by Using Established Risk Analysis Techniques. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. 229–240. https://doi.org/10.1109/ISSRE5003.2020.00030 ISSN: 2332-6549.

[24] Russ Miles. 2019. *Learning Chaos Engineering: Discovering and Overcoming System Weaknesses Through Experimentation*. "O'Reilly Media, Inc.". Google-Books-ID: DwqiDwAAQBAJ.

[25] Bruno Astuto A. Nunes, Marc Mendonca, Xuan-Nam Nguyen, Katia Obraczka, and Thierry Turletti. 2014. A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks. *IEEE Communications Surveys & Tutorials* 16, 3 (2014), 1617–1634. https://doi.org/10.1109/SURV.2014.012214.00180 Conference Name: IEEE Communications Surveys & Tutorials.

[26] Nyker Matthew C. King. 2022. Chaos Engineering Synthesis. https://doi.org/10.13140/RG.2.2.17008.74243

[27] S. Orlowski, R. Wessäly, M. Pióro, and A. Tomaszewski. 2010. SNDlib 1.0—Survivable Network Design Library. *Networks* 55, 3 (May 2010), 276–286. https://doi.org/10.1002/net.20371

[28] Samantha Persson. 2019. *Improving perceived performance of loading screens through animation.* https://urn.kb.se/resolve?urn=urn:nbn:se:lnu:diva-86080

[29] Casey Rosenthal and Nora Jones. 2020. *Chaos Engineering: System Resiliency in Practice.* "O'Reilly Media, Inc.". Google-Books-ID: iVjbDwAAQBAJ.

[30] John Rust. [n. d.]. New Palgrave Dictionary of Economics. ([n. d.]).

[31] Nick Shelly, Brendan Tschaen, Klaus-Tycho Förster, Michael Chang, Theophilus Benson, and Laurent Vanbever. 2015. Destroying networks for fun (and profit). In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks.* ACM, Philadelphia PA USA, 1–7. https://doi.org/10.1145/2834050.2834099

[32] Nick Shelly, Brendan Tschaen, Klaus-Tycho Förster, Michael Chang, Theophilus Benson, and Laurent Vanbever. 2015. Destroying networks for fun (and profit). In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks.* ACM, Philadelphia PA USA, 1–7. https://doi.org/10.1145/2834050.2834099

[33] Kennedy A. Torkura, Muhammad I.H. Sukmana, Feng Cheng, and Christoph Meinel. 2019. Security Chaos Engineering for Cloud Services: Work In Progress. In *2019 IEEE 18th International Symposium on Network Computing and Applications (NCA).* 1–3. https://doi.org/10.1109/NCA.2019.8935046 ISSN: 2643-7929.

[34] Kennedy A. Torkura, Muhammad I. H. Sukmana, Feng Cheng, and Christoph Meinel. 2020. CloudStrike: Chaos Engineering for Security and Resiliency in Cloud Infrastructure. *IEEE Access* 8 (2020), 123044–123060. https://doi.org/10.1109/ACCESS.2020.3007338 Conference Name: IEEE Access.

[35] K. A. Torkura, Muhammad I. H. Sukmana, Feng Cheng, and Christoph Meinel. 2021. Continuous auditing and threat detection in multi-cloud infrastructure. *Computers & Security* 102 (March 2021), 102124. https://doi.org/10.1016/j.cose.2020.102124

[36] Ariel Tseitlin. 2013. The antifragile organization. *Commun. ACM* 56, 8 (Aug. 2013), 40–44. https://doi.org/10.1145/2492007.2492022

[37] Anton Zubayer and Tai Luong. 2018. *Simulation of chaos engineering for Internet-scale software with ns-3.* https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-230214

Eidesstattliche Versicherung

(Affidavit)

Belmin Oruc	221231
Name, Vorname (surname, first name)	Matrikelnummer (student ID number)

[✓ Bachelorarbeit] (Bachelor's thesis) ☐ Masterarbeit (Master's thesis)

Titel
(Title)

Schutz von Netzwerken durch Fehlereinspeisung /
Protection of networks through failure injection

Ich versichere hiermit an Eides statt, dass ich die vorliegende Abschlussarbeit mit dem oben genannten Titel selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

I declare in lieu of oath that I have completed the present thesis with the above-mentioned title independently and without any unauthorized assistance. I have not used any other sources or aids than the ones listed and have documented quotations and paraphrases as such. The thesis in its current or similar version has not been submitted to an auditing institution before.

02.11.2024
Ort, Datum
(place, date)

Unterschrift
(signature) [illegible]

Belehrung:
Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -).

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird ggf. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin") zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Official notification:
Any person who intentionally breaches any regulation of university examination regulations relating to deception in examination performance is acting improperly. This offense can be punished with a fine of up to EUR 50,000.00. The competent administrative authority for the pursuit and prosecution of offenses of this type is the Chancellor of TU Dortmund University. In the case of multiple or other serious attempts at deception, the examinee can also be unenrolled, Section 63 (5) North Rhine-Westphalia Higher Education Act (*Hochschulgesetz, HG*).

The submission of a false affidavit will be punished with a prison sentence of up to three years or a fine.

As may be necessary, TU Dortmund University will make use of electronic plagiarism-prevention tools (e.g. the "turnitin" service) in order to monitor violations during the examination procedures.

I have taken note of the above official notification:*

02.11.2024
Ort, Datum
(place, date)

Unterschrift
(signature) [illegible]

***Please be aware that solely the German version of the affidavit ("Eidesstattliche Versicherung") for the Bachelor's/ Master's thesis is the official and legally binding version.**