Algorithms to prevent attacks in Active Directory environments

Albert Ibars Cubel

Supervisor: Maria José Serna

Department of Computer Science
Universitat Politècnica de Catalunya

This dissertation is submitted for the degree of
Computing

Facultat d'Informàtica de Barcelona October 2022

Acknowledgements

I want to express my deep gratitude to Professor Maria Serna from the Department of Computer Science of UPC for having directed my Bachelor's Thesis. This project would not have been possible without her patient guidance, enthusiastic encouragement, and valuable critiques of this research work.
I would also like to thank my GEP tutor Javier Morales for his advice and assistance in planning the project's beginning.
Finally, I wish to thank my parents and brother for their support and encouragement throughout my study.

Abstract

Microsoft Active Directory is the default management system for Windows domain networks, so convenient that it is used in many organizations worldwide. Because of this popularity, AD has been a focused cyberattack target. An Active Directory environment could be described as an attack graph in which nodes represent users, computers, groups, and services. The problem is formulated with an attacker who has managed to infiltrate the network to a certain number of nodes (entry nodes) and seeks to reach the most privileged node. On the other hand, the defender's objective will be to maximize the attacker's shortest path length.

In this paper, we have conducted a study of the AD-blocking style graph. We have observed that the attack graphs of this Microsoft service have small maximum attack path lengths and are tree-like. The problem is computationally difficult. Six different algorithms have been formulated to solve the problem. Firstly, the problem of blocking only the most vital edge of the attack graph has been investigated, where two algorithms have been formulated and will later become greedy algorithms. An FPT algorithm and three heuristic algorithms have also been developed: a Hill Climbing, a Simulated Annealing, and a genetic algorithm. With these algorithms, we will help IT admins identify high-risk edges in practical Active Directory environments.

Keywords: Active Directory; Graph algorithms; Shortest path; Most vital edges

Table of contents

List of figures **xi**

List of tables **xiii**

1 Context and scope **1**

1.1 Context . . . 1

1.1.1 Introduction . . . 1

1.1.2 Concepts . . . 2

1.1.3 Problem to be resolved . . . 4

1.1.4 Stakeholders . . . 4

1.2 Justification . . . 4

1.2.1 Related work . . . 5

1.3 Scope . . . 6

1.3.1 Objectives . . . 6

1.3.2 Requirements . . . 7

1.3.3 Potential obstacles and risks . . . 7

1.4 Methodology and rigour . . . 8

1.4.1 Methodology . . . 8

1.4.2 Monitoring tools and validation . . . 8

1.5 Work delay . . . 9

2 Project planning **11**

2.1 Task definition . . . 11

2.2 Resources . . . 13

2.2.1 Human resources . . . 13

2.2.2 Material resources . . . 14

2.3 Risk management . . . 15

2.3.1 Deadline of the project 15
2.3.2 Bad decision to describe AD attack graphs parameters 15
2.3.3 Create a program for the experiment 16
2.3.4 Computational power 16
2.4 Gantt chart 17

3 Budget and sustainability **19**
3.1 Budget 19
3.1.1 Identification of costs 19
3.1.2 Cost estimates 20
3.1.3 Management control 23
3.2 Sustainability 24
3.2.1 Self-assessment 24
3.2.2 Economic dimension 24
3.2.3 Environmental dimension 25
3.2.4 Social dimension 26

4 Formal definition and preliminaries **27**
4.1 Preliminaries 27
4.1.1 Basic definitions of graph theory 27
4.1.2 Basic algorithms 30

5 Problem formulation **33**
5.1 Model description 33
5.2 The complexity of the problem 34
5.3 Most vital edge problem 37

6 Algorithms **39**
6.1 Solving the MVE problem 39
6.1.1 Basic sequential algorithm 39
6.1.2 A more efficient solution 40
6.2 A Greedy-based algorithm implementation 42
6.3 An FPT-based algorithm implementation 44
6.3.1 Parameterized complexity 44
6.3.2 Solving the problem with an FPT algorithm 45

7 Heuristics **47**
7.1 Introduction to local search optimization 47
7.1.1 Hill Climbing 47
7.1.2 Simulated Annealing 48
7.2 Representation of the problem as a local search problem 49
7.3 Genetic Algorithm 51
7.4 Representation of the problem as a genetic algorithm 51

8 Experimentation **57**
8.1 Development specifications 57
8.2 Parameter and function selection 57
8.2.1 Justification of Simulated Annealing parameters 58
8.2.2 Justification of genetic algorithm parameters 61
8.3 Experimentation of the proposed algorithms 65
8.3.1 Experimentation with a small AD attack graph 65
8.3.2 Experimentation with a medium AD attack graph 67
8.3.3 Experimentation with a big AD attack graph 69

9 Final words **73**
9.1 Conclusion 73
9.2 Future work 74

References **75**

Appendix A Handle with DBCreator data **77**

Appendix B Parameter selection tables **81**

List of figures

1.1 Kanban board using Asana 8

2.1 Gantt chart 17

4.1 Graphical representation of a graph. 28

5.1 Simple graphical representation of the problem. 34
5.2 The reduction 35

6.1 $M_{s_0}(u)$ and $N_{s_0}(u)$ 41
6.2 Example graph, where the MVE is not found among the b most vital edges for $b > 1$. 43
6.3 Example of a graph with substitutable block-worthy edges. 44

7.1 Simulated Annealing vs. Hill Climbing strategy 49
7.2 Crossover example for the N queens problem. 52
7.3 Crossover without overlap 53
7.4 Crossover with overlap 53

8.1 The BloodHound attack graph design 57

A.1 DBCreator console display. 77
A.2 DBCreator graph representation using neo4j. 78

List of tables

2.1 Summary of the tasks to be performed 13

3.1 Budget structure 20
3.2 The total cost of human resources 21
3.3 Amortization costs for the hardware resources 21
3.4 The general cost of the project 22
3.5 Incidental cost of the project 23
3.6 Total project cost 23

8.1 Comparison of two initial generators solution of Hill Climbing 58
8.2 Determination of the number of iterations in Simulated Annealing 59
8.3 Temperature determination in Simulated Annealing 60
8.4 Comparison of crossover operators 61
8.5 Determination of the initial population of the genetic algorithm 62
8.6 Determination of the maximum population of the genetic algorithm 63
8.7 Determination of the number iterations of the genetic algorithm 63
8.8 Determination of mutation rate 64
8.9 Determination of crossover rate 65
8.10 The small AD attack graph results where all edges are blockable and contain unweighted edges. 66
8.11 The small AD attack graph results where not all edges are blockable and contain unweighted edges. 66
8.12 The small attack graph AD results where not all edges are blockable and contain weighted edges $(w(e) \leq 15)$. 67
8.13 The medium AD attack graph results where all edges are blockable and contain unweighted edges. 68

8.14 The medium AD attack graph results where not all edges are blockable and contain unweighted edges. 68
8.15 The medium attack graph AD results where not all edges are blockable and contain weighted edges $(w(e) \leq 15)$. 69
8.16 The big AD attack graph results where all edges are blockable and contain unweighted edges. 70
8.17 The big AD attack graph results where not all edges are blockable and contain unweighted edges. 70
8.18 The big attack graph AD results where not all edges are blockable and contain weighted edges $(w(e) \leq 15)$. 71

Chapter 1

Context and scope

1.1 Context

This work is a computer engineering Degree Final Project (TFG), specialization in Computer Science, done at the Facultat d'Informàtica de Barcelona of the Universitat Politècnica de Catalunya, supervised by Maria Jose Serna Iglesias.

1.1.1 Introduction

Today, almost all information is stored electronically using computers and networks. With the emergence of the Internet, computers were connected through this global network, increasing the efficiency of information exchange and availability. However, the Internet was not primarily built to be secure; it was supposed to communicate between trusted computers and trusted networks. Computers are more vulnerable to information security being attacked and compromised. At the same time, applications and services rely on computers to store their information. For this reason, the importance of information security in computer networks is only growing.

In April 2015, John Lambert described the challenges facing today's defense security teams and introduced a perspective on network attacks [1]. While defenders deal with lists of assets and databases, attackers pay attention to their relationships. Lambert paraphrases this imbalance by stating:

> *"Defenders think in lists. Attackers think in graphs.*
> *As long as this is true, attackers win."*
>
> *—John Lambert, Microsoft Threat Intelligence Center*

Cyber attack graphs model the chain of events to produce a successful attack. After performing reconnaissance (research, search for information, weak points, etc.), an attacker's first step is to breach a network, extend control, and reach other services or resources. Then move through the network undetected to locate the highest-privilege account, called the *Domain Admin* DA.

This thesis focuses specifically on protecting against attackers in an Active Directory environment. The problem is formulated as a Stackelberg game between a defender and an attacker. The defender has a budget to block sure edges of the attacker's graph movement. In contrast, the attacker knows which edges have been secured[1] and intends to find the shortest path to the *Domain Admin*. For a coordinated attack, the Bloodhound tool allows you to enumerate, collect information and show the domain structure from graphs. One of the main functionalities of BloodHound is to automatically generate the shortest attack path from the attacker's entry node to DA, where the distance is defined as the number of relations. Before the invention of BloodHound, attackers explored network environments blindly, hoping to discover a privilege escalation pathway. The defender seeks to maximize the attacker's expected shortest path length.

1.1.2 Concepts

This part introduces some Active Directory-specific notions to understand the project better and a software tool for analyzing and visualizing AD environments.

Active Directory

Active Directory (AD) is a technology belonging to Microsoft company that provides a service for managing Windows domain networks [2]. Small, mid-sized businesses and large enterprises commonly use it. Active Directory centralizes the management of users and resources, providing an integrated environment that allows organizations to maintain a virtual representation of themselves.

Administrators can manage domains, users, groups, computers, services, and applications. For example, an administrator account can create a group of users authorized to specific server resources. The administrator can also take control of local resources at any time simply by changing the user's rights and permissions. All this Active Directory data is stored as objects on the network. AD uses a structured data store to form a logical, hierarchical organization

[1] In practice, Active Director attackers can use a tool called SharpHound to scan the entire environment and obtain information from all edges.

of directory information. However, it can be difficult for administrators to monitor users' access data and permissions as a network grows. To solve this, Active Directory offers a way to organize users into logical groups and subgroups while providing access control at each level [3].

The Active Directory structure comprises three main components: domains, trees, and forests. Specifically, a domain tree is a collection of domains, and a forest is a collection of trees. The central component of an Active Directory network is the Active Directory Domain Controller (DC). The DC is a computer running a Windows Server operating system. It is the most privileged computer, as it authenticates and manages communication between users and computers in the network and enforces access policies.

BloodHound

BloodHound is a graphical viewer that uses graph theory to reveal the hidden and often unintended relationships within an Active Directory environment. It is used by attackers (red teams) and defenders (blue teams) [4]. Attackers can use BloodHound to identify complex attack paths that would be impossible to locate quickly. Defenders can use BloodHound to identify, eliminate, or block those same attack paths. BloodHound is a tool that can automatically enumerate a domain, collect information, and provide a deeper understanding of privileged relationships in an Active Directory environment. Bloodhound is composed of 2 main parts:

- Ingestors are the applications responsible for enumerating the domain and extracting all the information. The best-known example is SharpHound [5].
- The visualization application shows all the previously collected information and offers different ways to escalate privileges in the domain. This application is built with *Javascript* with a *Neo4j* database.

DBCreator

The BloodHound team provided a randomly synthetic database generator called DBCreator [6] to test use cases of a real-world Active Directory environment. It consists of a python script that generates a randomized data set for testing BloodHound features and analysis. It allows you to modify the characteristics and size of the database as a parameter. The resulting graph will simulate an accurate AD environment. With this tool, we created all the attack graphs for the investigation.

1.1.3 Problem to be resolved

Microsoft Active Directory is the default management system for Windows domain networks, which is so convenient that it is used in many organizations worldwide. Because of this popularity, AD has been a focused cyberattack target. An Active Directory environment could be described as an attack graph in which nodes represent users, computers, groups, and services. If an attacker manages to gain access to the system, it will usually be by obtaining a user account, either by social engineering, phishing, or any other technique. The attacker then starts with low privileges and will try to increase those privileges by moving to a highly privileged node. This type of attack is called *identity snowball attacks* [7], so we will try to prevent this attack and formulate it as a Stackelberg game between an attacker and a defender. The problem consists of an attack graph containing a target node and multiple entry nodes. The defender chooses which edges of the graph to block, limited by his budget. The attacker aims to reach the highest-privileged node, the Domain Controller (DC), from the shortest unblocked attack path. The defender's goal is to maximize the expected shortest path length for the attacker. So, in this thesis, we will take the *blue team* role and study the problem of the most vital edges to help IT admins identify high-risk edges in practical Active Directory environments.

1.1.4 Stakeholders

The main stakeholders that will benefit from the project are users that use Active Directory environments. These organizations will strengthen network security to protect their data from adversaries and security threats from this project. Also, a powerful guild that will benefit is IT administrators tasked with improving safety to those most vital edges of the attacker's shortest paths. Computing the most vital edges will enable network designers to construct networks with better robustness.

On the other hand, indirectly involved stakeholders would be the scientific community, which gains access to the study and can use the information and conclusions for further studies in blocking edges in attack graphs.

1.2 Justification

As discussed, Active Directory is the most widely used technology for managing identities in modern enterprises. The Active Directory service controls entire domains, making it an ideal target for attackers. Currently, many studies focus on blocking or removing a certain

number of edges to maximize the attacker's shortest path. However, apart from studies, we have not seen tools on the market that provide similar capabilities to the ones studied. We believe it would be good to implement this search for the most vital edges as an add-on to BloodHound, thus strengthening Active Directory environments.

In this project, a new open-source tool called BlueHound [8] has emerged. It helps blue teams identify security issues (vulnerabilities) and uses attack graphs like BloodHound. So the addition of finding the most vital edges would be a perfect fit for this new tool as it is more of a defensive tool than BloodHound, which is more of an attacker's utensil.

1.2.1 Related work

The problem of blocking or eliminating a certain number of edges to maximize the attacker's shortest path appears in the literature under many names. Most authors hold it as the *most vital edges*, *arcs*, or *links* problem [9–11]; other contributors call it *bounded cut* [12–14] or *edge interdiction* problems [15]. There is also a variant in which only one link can be removed, sometimes named with the prefix *single*.

Ball et al. [16] is the first to define and motivate the most vital edges in a network in the context of directed graphs with non-negative cost $c(e)$ of removing. They show this problem to be NP-Hard. They show that a closely related issue, whose solution provides a lower bound on the value of the optimal solution of the most vital arc problem, is polynomially solvable.

Golovach and Thilikos [14] focus on blocking b edges to ensure that the shortest path is more significant than a parameter l. They proposed a parameterized algorithm for the bounded length cut problem family and demonstrated that the problem with unit-length edges is W[1]-hard respect to k. The authors also showed that the problem is fixed-parameter tractable if the maximum path length and the budget are small.

Bazgan et al. [17] also studied the perspective of parameterized complexity. He studied the single-source single-destination shortest path edge interdiction. The author proposed a kernelization algorithm including feedback edges as one of his parameters.

Dvořák and Knop [13] also show that the studied bounded length cut is W[1]-hard with a parameterized algorithm. They proposed an FPT algorithm concerning l and the graph's treewidth w with complexity $O(l'^{12w^2}n)$.

Finally, the article inspired us to do this project and create a version of it. Mingyu Go et al. [18] proposed three fixed-parameter algorithms and a GCN-based approach for

defending Active Directory-style attack graphs. The observation is that attack graphs have small maximum attack path lengths and are similar to trees.

1.3 Scope

In this section, we will define the main objectives of the theoretical and practical parts, the requirements, and possible obstacles and risks that may arise in the realization of the thesis.

1.3.1 Objectives

The project aims to understand how attackers attack an Active Directory environment. We provide edge-blocking algorithms to prevent these attacks and establish the complexity of the associated computational problem. We adopt parameterized complexity analysis, heuristics techniques, greedy design, and fixed-parameter tractable algorithms. Finally, we compare the results on different attack graphs using our program built from the topology created by DBCreator.

Theoretical Part

- Study Active Directory environments, BloodHound, and DBCreator.
- Present the model description of the attack graph.
- Propose and study the computational complexity of the algorithms for three different versions of the model and the different algorithms used.
  - All edges are blockable with unit-length edges.
  - Not all edges are blockable with unit-length edges.
  - Not all edges are blockable and have a different weight.

Practical Part

- Program the algorithms studied.
- Modify and extract the DBCreator topology for our experiment.
- Compare the results obtained for the different algorithms studied for the three versions.
- Conclude the results.

1.3.2 Requirements

Some requirements are needed to ensure the quality of the project.

- Create Active Directory attack graphs with small maximum attack path lengths and structurally close to trees.
- Define the comparison algorithms that are going to be used.
- Use good programming practices with a readable style of the algorithms studied.
- Possess sufficient computational power to handle algorithms on graphs with many nodes.

1.3.3 Potential obstacles and risks

Throughout the project's development, there may be possible obstacles and risks that could delay the realization of the project, and if this is the case, we will have to deal with them. Here we put some examples:

- **Deadline of the project.** We must achieve the project deadline. Therefore, it will be necessary to sufficiently organize and plan the project's development to finish it on time.
- **Bad decision to describe Active Directory attack graphs parameters.** We need to adopt a list of appropriate parameters that facilitate and describe a natural Active Directory environment to develop the algorithms and their respective experiments.
- **Create a program for the experiment.** The experiments in this project will be carried out with synthetic graphs generated using DBCreator, from which we will extract the topology to carry out the study. Creating a program that takes advantage of the topology created and allows us to apply criteria and modify the attack graph will be challenging.
- **Computational power.** Large organizations with natural Active Directory environments contain an extensive network of computers, groups, and users, thus involving many nodes in the attack graph. Applying the algorithms to these networks will affect much computational power, slowing down the work if we cannot guarantee sufficient computational power.

1.4 Methodology and rigour

In this section, we will define the methodology, monitoring tools, and validation methods we will use to visualize the detailed workflow and performance of the project.

1.4.1 Methodology

KanBan is the methodology we have considered adopting, which provides a visual representation of our process flow, allowing us to be more efficient. With the KanBan methodology, we will control the tasks through a division by phases until their completion. This division of graphic cards will be completed through the following online board 1.1.

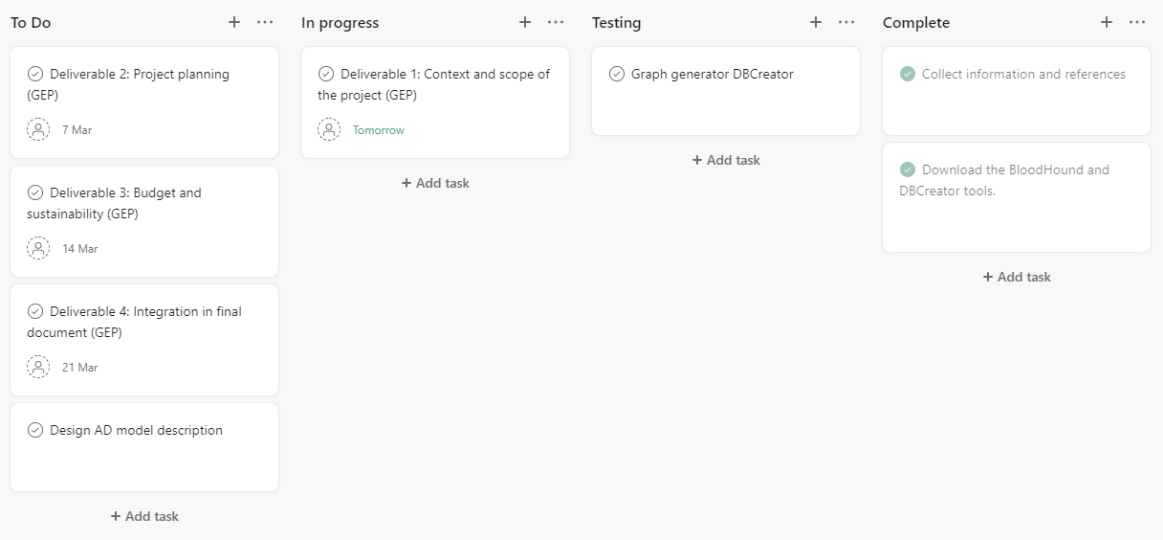

Fig. 1.1 Kanban board using Asana

We have used the web application Asana to create the methodology because it is very flexible and easy to use. As shown in the figure, we have created a table with different fields. Each field contains the tasks to be done, the tasks in progress, the tests, and the completed assignments. Also, for each card, we can define a description of the task, write comments and even create a list of subtasks.

1.4.2 Monitoring tools and validation

We will use a GitHub repository as a storage and control tool. We will create a *main* branch that will include tested and updated code and a *development* branch that will consist of all the code under development or testing. We will perform several tests to check the correct

functioning of the implemented code and algorithms. In creating the Active Directory attack graph, we will only consider three types of edges: *AdminTo*, *MemberOf*, *HasSession*. These three types of edges are the default in BloodHound. The optimal parameters of the Active Directory environment shall be selected using some model validation technique, such as cross-validation. And each experiment is performed five times, and some results will be averaged.

Lastly, we have created a Gmail space with the supervisor where we can easily communicate via chat, pass files, and plan meeting times. We will meet with the tutor every two weeks to discuss the project status and check that everything is OK. If we have a problem, we will create an extraordinary meeting session to deal with it.

1.5 Work delay

For work reasons, finishing the project for the first reading shift has not been possible. The project has been postponed for a couple of months. So last month's planning for the first delivery has been moved to mid-August for the completion of the project in the second available delivery.

Chapter 2

Project planning

This project started in early February 2022 and is planned to be completed by June. This thesis will have an approximate working time of 610 hours, spread over 125 days. So I have to distribute the hours devoted to each task to ensure the project is completed successfully. The time spent per week will be around 25-30 hours, although this may vary due to external factors.

2.1 Task definition

In this section, we will explain how the project is scheduled. It will be divided into five group tasks: project planning, theoretical, practical implementation, experimentation/analysis, and project documentation.

Project planning (T1) is a significant step in carrying out a project. It helps us establish each activity's priority and better control the time to execute a project with the desired quality and success. This part is divided into five parts:

- **Context and scope of the project (T1.1):** It defines the scope of the project in the context of its study. It also specifies the objectives, the relevance of the area, and how the project is developed.
- **Temporal planning (T1.2):** It plans the work's entire execution, describing the phases, resources, and requirements needed.
- **Budget and sustainability (T1.3):** It is essential to keep track of the costs and the impact of the project development. This phase describes the budget, economic viability, and environmental sustainability.

- **Final project definition (T1.4):** Create a document grouping all previously performed tasks, modifying those wrong parts.
- **Meetings (T1.5):** We will meet with the tutor every two weeks to discuss the project's status and evolution.

Before starting with the project's development, a fundamental task is the **research** of previous studies. Search for applications, papers, and articles on the internet and cite them in our work.

In the **theoretical part (T2)**, we will focus on describing the Active Directory model used and the realization of the proposed algorithms for blocking edges of an attack graph.

- **Model description (T2.1):** Describe the Active Directory attack graph, define the parameter list, and demonstrate the algorithmic complexity of the problem.
- **Algorithms edge-blockable (T2.2):** The idea is to propose five or more shortest path edge interdiction algorithms (*greedy, FPT, heuristic techniques ...*) for defending Active Directory. We will perform this task on the environment version where all edges are lockable with unit-length edges.
- **Algorithms, not all edge-blockable (T2.3):** Same as in the previous task, but not all edges are blockable this time.
- **Algorithms, not all edge-blockable with costs (T2.4):** Finally, we will do the last algorithms in an environment where not all edges are blockable and edges have different weights.

The **practical part (T3)** is the most important, as this project needs to test the algorithms implemented in the theoretical part on different attack graphs.

- **Program skeleton (T3.1):** Program everything necessary so that all that remains is to add the functions that execute the algorithms proposed in the theoretical practice. It will be required to program the extraction of the graph's topology created with DBCreator, program the shortest path from one point to another in an attack graph, and more.
- **Program algorithms edge-blockable (T3.2):** Program algorithms where all edges are blockable with unit-length edges.

- **Program algorithms are not all edges-blockable (T3.3):** Program algorithms where not all edges are blockable with unit-length edges.
- **Program algorithms not all edges-blockable with costs (T3.4):** Program algorithms where not all edges are blockable, and edges have different weights.

The next stage will **experiment, analyze, and collect data (T4)** from the created algorithms. We will compare the results with the different versions on different attack graphs and conclude.

During the project's entire development, it will be **documented (T5.1)** in parallel. And finally, we will have to prepare for the **oral defense (T5.2)** for the project presentation.

ID	Name	Time(h)	Dependencies	Resources
T0	Research	**60**		PC, articles, papers, books
T1	Project planning	**70**		
T1.1	Context and scope	24		PC, Overleaf, Atenea
T1.2	Temporal planning	10	T1.1	PC, Overleaf, Atenea, Asana, GanttProject
T1.3	Budget and sustainability	10	T1.2	PC, Overleaf, calculator
T1.4	Final project definition	18	T1.1, T1.2, T1.3	PC, Overleaf
T1.5	Meetings	8		PC, Google Meet
T2	Theoretical part	**140**		
T2.1	Model description	20		PC, articles, books
T2.2	Algorithms edge-blockable	40	T2.1	PC, articles, books
T2.3	Algorithms not all edge-blockable	40	T2.2	PC, articles, books
T2.4	Algorithms not all edge-blockable with costs	40	T2.3	PC, articles, books
T3	Practical implementation	**180**		
T3.1	Program skeleton	50		PC, Python, git, VSCode
T3.2	Program alg. edge-blockable	40	T2.2, T3.1	PC, Python, git, VSCode
T3.3	Program alg. not all edge-blockable	40	T2.3, T3.1	PC, Python, git, VSCode
T3.4	Program alg. not all edge-blockable with costs	50	T2.4, T3.1	PC, Python, git, VSCode
T4	Experimentation and analysis	**30**		
T4.1	Experimenting with algorithms	10	T3	PC, attack graphs created
T4.2	Obtain and compare results	20	T4.2	PC, results obtained
T5	Project documentation	**110**		
T5.1	Documentation	80		PC, Overleaf, project resources
T5.2	Oral defense preparation	30	T5.1	PC, Overleaf, results obtained
Total		**590**		

Table 2.1 Summary of the tasks to be performed

2.2 Resources

We will use the following human and material resources to realize this project.

2.2.1 Human resources

Three human resources are involved in this project. Firstly, the researcher is in charge of developing the whole project. His role will be to plan, define concepts, conduct an

experimental study, analyze, and document it. On the other hand, the supervisor supervises and guides the researcher to stay on track in the project's development. Finally, the GEP tutor will be in control of correcting the researcher's project management during the first month of the project.

2.2.2 Material resources

Research projects are based on previous studies; therefore, searching for knowledge on the internet, books or articles will be necessary. In addition, we will need the following hardware and software resources to realize the whole project.

Software resources

- **Overleaf**. We decided to use LaTeX as it is a clean, neat, and tidy text format widely used in academia to communicate and publish scientific documents in many fields, including mathematics, computer science, and engineering. We will use the online text editor Overleaf, as it does not require installation, and you can share the document.
- **Atenea**. We will use it to obtain material to help manage the project and communicate and have feedback with the GEP tutor. It is also the site where deliverables are uploaded.
- **Visual Studio Code**. We will use Microsoft's Visual Studio Code IDE to program the experimental part for its efficiency and versatility.
- **Github**. We will use a GitHub repository as a storage and control tool due to its accessibility.
- **BloodHound/DBCreator**.We will use BloodHound in the experimental part to observe the topology created with the DBCreator tool.

Hardware resources

Our hardware specs are Intel® Core™ i7-8700K 3.7GHz, 32GB of RAM, and a GPU GTX 1080 8GB computer desktop.

2.3 Risk management

During the project's development, obstacles and risks may appear that could delay the realization of the project. The potential risks and barriers have been introduced previously. This section will look at the degree of risk of each problem that may arise, how they may affect the project, and what alternative tasks we could use to mitigate these setbacks.

2.3.1 Deadline of the project

One of the problems with planning before starting the project is that we may not have distributed the hours spent on each task correctly and may need more than planned. Hence, it can cause chaos in the project plan and delay completing tasks.

- **Impact:** Medium.
- **Solution:** Adapt the project plan to the new situation and recalculate the hours needed for each task. We will create new planning at a later point in the project. If the new plan has been unsuccessful for any external reason, we can continue solving the problem by applying more daily hours. If things get too complicated and we need more time, we can use an extension to deliver the project later (*a few more months*), but that is not the plan.

2.3.2 Bad decision to describe AD attack graphs parameters

Getting it wrong to decide on the model description parameters that describe an Active Directory environment to do the experiments can be a big problem as they are the project's foundation.

- **Impact:** Medium.
- **Solution:** We can apply the same list of parameters from the article [18] that inspired this paper as a solution. We ensure that we create a model suitable for Active Directory environments. We could also apply the parameters used in other articles on blocking the most vital edge problems style.

2.3.3 Create a program for the experiment

The experiments of this project will be carried out with synthetic graphs generated with DBCreator, from which we will extract the topology to carry out the study. In case of not taking advantage of this tool, it can be a substantial project delay.

- **Impact:** High.
- **Solution:** Suppose we cannot take advantage of DBCreator. In that case, we will have to create our network generator from scratch, making it as similar as possible to an Active Directory attack graph. This process will delay the experimental part of the project.

2.3.4 Computational power

There are vast Active Directory environments with an extensive network of computers, groups, and users, thus involving many nodes in the attack graph. This project will attempt to create pretty large networks, and applying relatively complex algorithms to these graphs can require a lot of computing power.

- **Impact:** Low.
- **Solution:** We will exercise two solutions; the first would be to look for online alternatives such as Google Collaborative that will run the code for as long as it takes. And the second option would be to reduce the number of nodes in the graph to make it smaller so that the algorithm can finish in a reasonable time.

2.4 Gantt chart

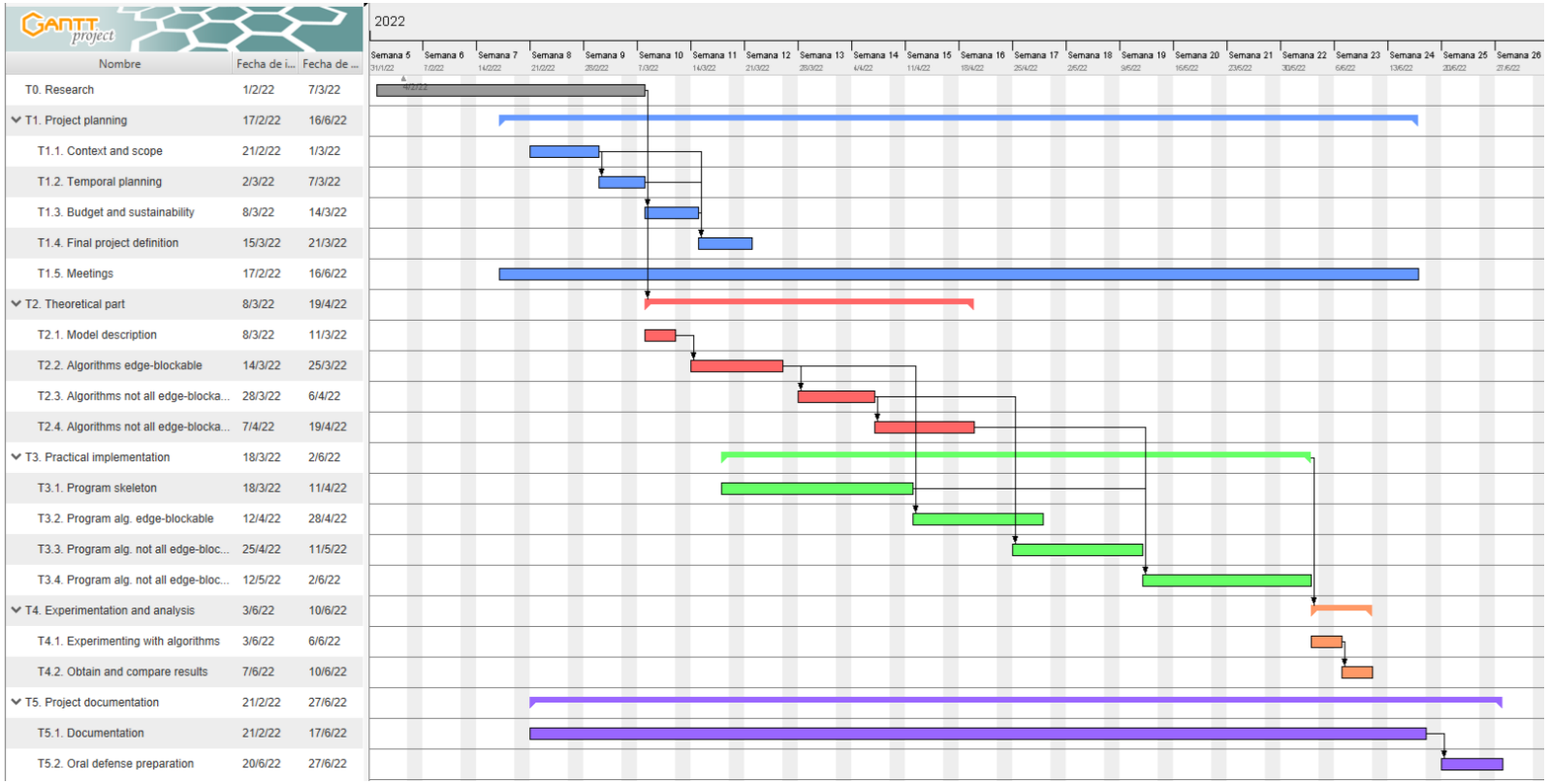

Fig. 2.1 Gantt chart

Chapter 3

Budget and sustainability

This chapter will calculate the costs of all elements used in the planning. We will have human, hardware, software, and indirect costs. Moreover, we will consider management control mechanisms to control deviations that may arise in the project due to unforeseen events. Finally, we will answer a series of questions on the project's sustainability.

3.1 Budget

3.1.1 Identification of costs

To identify the project's costs, we first have to calculate the cost of each task defined in Section 2.1. The cost of each labor is calculated by adding the worker's worth. Each worker's salary is calculated by multiplying their hourly wage by the number of hours performing an activity. Four staff roles will carry out this project, each hourly cost. The first role is the **Project Manager**, who will be in charge of planning, executing, and monitoring the project and making all the decisions to achieve each phase's objectives. The supervisor, the GEP tutor, and I will take this role's reins. The next role is the **Researcher**, who will study edge blocking for defending Active Directory style attack graphs and propose algorithms to solve it and its computational complexity. I will play this role and the following ones. Once the researcher has the algorithms, the **Programmer** will have to program them and see that they work correctly. Finally, there is a role in charge of all the project documentation. I will also be a **Technical Writer**.

We must also remember that material resources, such as software, hardware, and workplace resources, must be amortized.

3.1.2 Cost estimates

In this section, we will calculate the approximate cost of the project. In Table 3.1, we can see a complete summary of the different expenses calculated.

Name	Cost(€)	Comments
Tasks		
T0.1 - Research	1,140.00	Researcher, 60 hours
T1.1 - Context and scope	576.00	Project Manager, 24 hours
T1.2 - Temporal planning	240.00	Project Manager, 10 hours
T1.3 - Budget and sustainability	240.00	Project Manager, 10 hours
T1.4 - Final project definition	432.00	Project Manager, 18 hours
T1.5 - Meetings	192.00	Project Manager, 8 hours
T2.1 - Model description	380.00	Researcher, 20 hours
T2.2 - Algorithms edge-blockable	760.00	Researcher, 40 hours
T2.3 - Algorithms not all edge-blockable	760.00	Researcher, 40 hours
T2.4 - Algorithms not all edge-blockable with costs	760.00	Researcher, 40 hours
T3.1 - Program skeleton	800.00	Programmer, 50 hours
T3.2 - Program alg. edge-blockable	640.00	Programmer, 40 hours
T3.3 - Program alg. not all edge-blockable	640.00	Programmer, 40 hours
T3.4 - Program alg. not all edge-blockable with costs	800.00	Programmer, 50 hours
T4.1 - Experimenting with algorithms	160.00	Programmer, 10 hours
T4.2 - Obtain and compare results	380.00	Researcher, 20 hours
T5.1 - Documentation	1,520.00	Technical Writer, 80 hours
Total CPA (Costs per activity)	**10,420.00**	
Software		
Visual Studio Code	0.00	—
Github	0.00	—
Overleaf Software	0.00	—
GanttProject	0.00	—
BloodHound	0.00	—
DBCreator	0.00	—
Hardware		
Desktop Computer	283.64	590 hours. Resource price: 1,923€
PC Peripherals	209.75	590 hours. Resource price: 711€
Workplace		
Electricity	40.41	0.3233€ per day. Project duration of 125 days
Internet	66.66	80€ per month. Project duration of 5 months
Total CG (General Costs)	**600.46**	
Contingency Margin	1,322.46	Inclusion of a 12% contingency margin
Incidents		
Deadline of the project	178.50	Cost: All roles, 30 hours. Risk 35%
Bad decision to describe AD attack graphs	38.00	Cost: Researcher, 10 hours. Risk 20%
Program creation	288.00	Cost: Programmer, 40 hours. Risk 45%
Computational power	8.00	Cost: Cluster use, 5 hours. Risk 10%
Total incidents	**512.50**	
TOTAL	**12,855.42**	

Table 3.1 Budget structure

Human resources budget

As described above, there are five roles during the project: Project Manager, Researcher, Programmer, and Technical Writer. The salaries have been taken from the GlassDoor [19] page, which provides the average salary for the different jobs in a year. The hourly rate has been calculated based on 1888 hours worked in a year. To define the overall personnel cost, we have to multiply the cost per hour by each employee's number of project hours. In Table 3.2, we see the cost of recruitment, known as Costs per Activity (CPA).

Role	**Cost(€)/h**	**Project hours**	**Cost(€)**
Project Manager	24.00	70	1,680
Researcher	19.00	220	4,180
Programmer	16.00	190	3,040
Technical Writer	19.00	80	1,520
CPA cost			**10,420**

Table 3.2 The total cost of human resources

General Costs

We consider hardware, software, and workplace resources to calculate the General Costs (CG). A part of the economic cost will be given by the amortization of the resources used in the project. We will carry out the project on a desktop computer with some peripherals. We will calculate the project's depreciation, considering that the project uses utterly free software tools. We must believe that we will work approximately 4 hours daily for 125 days. The formula we will apply to calculate the amortization is as follows 3.1:

$$Amortization = Resource\ price \times \frac{1}{Lifespan} \times \frac{1}{Work\ days} \times \frac{1}{Hours\ per\ day} \times Hours\ used \tag{3.1}$$

Hardware	**Price(€)**	**Lifespan (years)**	**Amortization(€)**
Desktop Computer	1,923.00	8	283.64
PC Peripherals	711.00	4	209.75
Total cost			**493.39**

Table 3.3 Amortization costs for the hardware resources

To finalize the CG costs, we must count the electricity and internet. An essential factor also considered is the transport cost, but we will carry out this project online, so there will be no transport cost.

- **Electric cost:** Currently, the cost of electricity is relatively high. The actual price per kWh is 0.3233€ [20]. The computer desktop uses an average of 250 Watt hours *(including peripherals)*. Assuming that the computer is on for four hours a day, the total electricity cost of the project will be $(0.3233€/kWh) \times 250Wh \times 125days \times 4h$. Total = 40.41€
- **Internet cost:** The internet rate costs 80€ per month. Considering that the project lasts five months and the working hours per day are four, the internet cost is $5months \times (80€/month) \times (4h/24h)$. Total = 66.66€

Adding all previous costs, the estimated generic cost for this project is available in Table 3.4.

Concept	**Cost(€)**
Amortization	493.39
Electric cost	40.41
Internet cost	66.66
CG cost	**600.46**

Table 3.4 The general cost of the project

Contingency

The contingency budget is an amount of money we will include to cover potential and unforeseen events. To the total costs of the items described above, we will add a contingency margin of 12%. So the total cost ascends to $(CPA + CG) \times 1.12 = 12342.92€$.

Incidents costs

The last thing to consider is the cost of implementing alternative plans in case of unforeseen events during the project. Section 1.3.3 lists all possible risks that may appear. So, we will have to spend more on salary costs in case of delays. Table 3.5 below shows the total cost of resolving these incidents by risk probability.

Incident	Estimated cost(€)	Risk(%)	Cost(€)
Deadline of the project (30 hours)	510.00	35	178.50
Bad decision to describe AD attack graphs (10 hours)	190.00	20	38.00
Program creation (40 hours)	640.00	45	288.00
Computational power (5 hours)	80.00	10	8.00
Total cost			**512.50**

Table 3.5 Incidental cost of the project

Final budget

The project's total budget with the calculations made in previous sections is shown in Table 3.6.

Concept	Cost(€)
CPA cost	10,420.00
CG cost	600.464
Contingency	1,322.46
Incidents cost	512.50
Total	**12,855.42**

Table 3.6 Total project cost

3.1.3 Management control

In this section, we will report the procedures to control the budget. To prevent deviations in the budget during the project development, we will calculate the actual cost assumed at the end of each task, adding the additional resources used and the possible unforeseen events that may have arisen. We will obtain the difference between the actual and estimated resources consumed by following the formulas below to calculate the deviation.

$$\text{Cost deviation} = (C_e - C_r) \times H_r \tag{3.2}$$

$$\text{Efficiency deviation} = (H_e - H_r) \times C_e \tag{3.3}$$

- C_e = Estimated cost
- H_e = Estimated hours
- C_r = Real cost
- H_r = Real hours

3.2 Sustainability

3.2.1 Self-assessment

After completing the sustainability self-assessment survey of EDINSOST, I realized that I have a vague idea of most sustainability concepts. It is difficult to answer questions related to environmental experiences from previous projects, as this project is the first one I have considered.

On the one hand, I am aware of the social and ecological effects produced by the development of the technology industry and why we should strive to make it sustainable. Every project impacts society, so this influence must be measured and considered from the outset. We should thoroughly analyze all possible factors of the project and its impact. However, I do not have enough knowledge to accurately measure the result of a project economically, socially, and environmentally.

I must point out that I did not know I had to consider many indicators when carrying out a sustainable project. Making these indicators at the beginning of a project can benefit the impact as we will know what sustainable impact we are facing and how we could improve it or solve it in case of problems.

As far as the economic field is concerned, I have a general idea of assessing the viability of a project and how this field is directly related to the environmental and social impact. Although there were concepts like amortization that I had to remember, I think it is essential to take them into account to understand the project's impact. Finally, I have delved a little into the world of sustainability. It has made me reflect on the importance of taking sustainability into account when planning and organizing a large-scale project from the point of view of the three different dimensions.

3.2.2 Economic dimension

Regarding PPP: Reflection on the cost you have estimated for the completion of the project:

Reflection on the estimated costs is explained in detail in Section 3.1 of the document. In this project, we have evaluated the price of human and material resources, including contingency and incidental cost estimates for the entire project.

Regarding Useful Life: How are currently solved economic issues (costs...) related to the problem you want to address (state of the art)?

The lifetime costs of the project are the least significant in the life cycle and are difficult to estimate. As an algorithmic research project studying blocking edges of attack graphs, they will not undergo substantial changes to solve it. Therefore, the only cost will be the maintenance and updates of the created program during its lifetime. One way to reduce the economic cost of the project would be to reuse programs for the generation of Active Directory attacking graphs, such as DBCreator. By doing this, we will reduce programming time, which will reduce the total economic cost. Another essential detail is to build the algorithms as efficiently as possible to reduce computation time and thus reduce energy consumption, which reduces the financial cost of electricity.

Regarding Useful Life: How will your solution improve economic issues (costs...) concerning other existing solutions?

By proposing algorithms to identify high-risk edges in practical Active Directory environments, we will achieve better performance against competitors by leveraging the cost of resources for those AD relationships potentially at risk, thus better-distributing costs across the different edges.

3.2.3 Environmental dimension

Regarding PPP: Have you estimated the project's environmental impact?

We have not fully estimated the environmental impact of the project. This is because this project does not waste material resources; the only environmental impact is electricity consumption for the whole project.

Regarding PPP: Did you plan to minimize its impact, for example, by reusing resources?

The only environmental impact that influences us is electric consumption. To minimize the effect, we aim to build efficient algorithms and create not very large graphs, thus reducing the time needed and, therefore, the environmental footprint. We prioritize computational efficiency to reduce overall energy consumption, and we use a single computer, minimizing consumption compared to more energy-demanding computers.

Regarding Useful Life: How is currently solved the problem that you want to address (state of the art)? How will your solution improve the environment concerning other existing solutions?

As mentioned above, we will identify high-risk edges in AD environments. Our solution will help IT administrators find the weak points in the network and thus better allocate resources to strengthen them and decrease the ecological footprint.

3.2.4 Social dimension

Regarding PPP: What do you think you will achieve -in terms of personal growth- from doing this project?

This project will allow me to introduce myself to research and present it with rigor. It will also allow me to properly organize and plan tasks and avoid minor scale issues. Finally, I consider it an excellent starting point for future projects. It has taught me to assess the possible repercussions of the measures taken during the project and the sustainable aspects of a project.

Regarding Useful Life: How is currently solved the problem that you want to address (state of the art)? How will your solution improve the quality of life (social dimension) concerning other existing solutions? Is there a real need for the project?

From a social point of view, we approach this project similarly to other cutting-edge projects in this field: collaborative research. Our contribution will be of great use to society as a whole as it allows us to understand the most vital relationships in an Active Directory environment. This study and its experiments will improve security in this type of network, which will even serve similar studies in the future.

Chapter 4

Formal definition and preliminaries

4.1 Preliminaries

This section introduces notation, definitions, and properties of graph theory used throughout the document to better use this work.

4.1.1 Basic definitions of graph theory

Definition 4.1.1 (Graph)**.** A *graph* $G = (V, E)$ consists of a finite, non-empty vertex set V and an edge set E of unordered 2-element subsets of V.

Elements of V are called *vertices* (or nodes), and elements of E are called *edges* (or links). If $e = \{u, v\}$ is an edge, then u and v are *ad jacent* to each other, and u and v are each incident to the edge e. To abbreviate the notation, we would write $e = uv$ instead of $e = \{u, v\}$.

The *order* of a graph is the number of vertices of G, i.e. the cardinal of V denoted by $|V(G)|$; the *size* of G is its number of edges denoted by $|V(E)|$.

Example. Given the graph $G = (V, E)$, where

$$V = \{1,2,3,4,5,6\} \text{ and } E = \{\{1,2\},\{1,3\},\{2,3\},\{2,4\},\{2,5\},\{3,4\},\{4,5\}\},$$

the order of G is 6, and its size is 7. As far as adjacencies are concerned, it is observed that, for example, 1 and 2 are adjacent, but 1 and 4 are not. These relations can be seen immediately if the graph is represented by a drawing where the vertices are points and the edges are lines joining vertices. The above graph G is depicted in Figure 4.1.

Definition 4.1.2 (Digraph)**.** A *digraph* $G = (V, E)$ is a finite, non-empty vertex set V and an edge set E of ordered 2-element subsets of V.

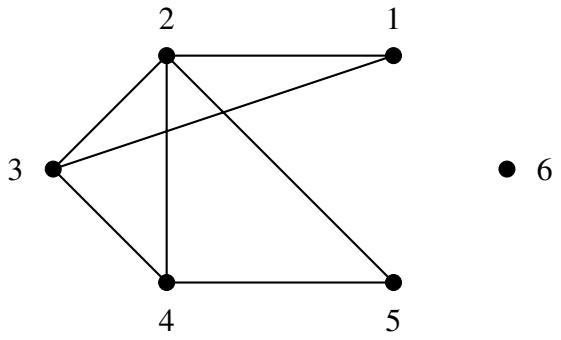

Fig. 4.1 Graphical representation of a graph.

In a *directed graph* (digraph), elements of V are called *vertices* (or nodes), and E elements are called *arcs* or *edges*. Note that in the case that $uv \in E(G)$ does not imply $vu \in E(G)$ how it was for the graph G above.

Definition 4.1.3 (Subgraph)**.** A graph $G' = (V', E')$ is a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$.

Definition 4.1.4 (Induced subgraph)**.** If $G = (V, E)$ is a graph and S is a subset of V, the *subgraph induced* by S, denoted by $< S >$, is the graph $< S >= (S, E')$, where the elements of E' are the edges of G joining the vertices of S, i.e.

$$E' = \{uv \in E \mid u \in S \text{ and } v \in S\}$$

In case $S = V - \{v\}$, the induced subgraph, denoted by $G - v$, is obtained by deleting from G the vertex v and the edges incident to this vertex. On the other hand, the *suppression* of an edge e of a graph $G = (V, E)$ gives rise, by definition, to the subgraph $G - e = (V, E - \{e\})$, i.e., edge e is deleted but not the incident vertices of this edge.
An important concept is a node's neighbors since there are no edges without neighbors.

Definition 4.1.5 (Neighborhood)**.** For an undirected graph $G = (V, E)$, the *neighbourhood* $N_G(u)$ of a vertex $v \in V$ is its set of all neighbors of v, i.e., $N_G(v) = \{u \mid \{u, v\} \in E(G)\}$. For a directed graph we use $N_G^+\{v\}$ to indicate the set of out-neighbors and $N_G^-\{v\}$ to indicate the set of in-neighbors of v.

Definition 4.1.6 (Degree)**.** The *degree* of a vertex $v \in V$ in a graph $G = (V, E)$, denoted by $d_G(v)$, is the size of the neighbourhood $|N_G(v)|$.

Definition 4.1.7 (Path)**.** A $u - v$ *path* of length l of graph $G = (V, E)$ is a sequence of vertices $u_0, u_1 ... u_{l-1}, u_l$, with $u_0 = u$, $u_l = v$ and $u_{i-1}u_i \in E$ for $i = 1, 2, ...l$, i.e. each pair of

consecutive vertices are adjacent. Note that paths are not unique. There may exist multiple paths between two nodes. We denote it as $P_G(u,v)$.

We must clearly understand the following concepts of distances and shortest paths as they are the mainstay of the work.

Definition 4.1.8 (Distance)**.** The *distance* between two vertices u and v of $G=(V,E)$, denoted by $d_G(u,v)$, is the minimum of the lengths of the $u-v$ paths of G.

Definition 4.1.9 (Shortest Path)**.** A *shortest path* $SP_G(s,d)$ from s to d in $G=(V,E)$, in a weighted graph is defined as a $s-d$ path which minimizes the sum of the weights of the edges along the $s-d$ path. In an unweighted graph, it is the minimum length of $s-d$ paths of G. In this case, $|SP_G(s,d)| = d_G(s,d)$.

Definition 4.1.10 (Cycle)**.** In a directed graph, a *cycle* is a path that starts and ends at the same vertex. A *simple cycle* is a cycle $v_1v_2...v_lv_1$ that has l different $v_1,v_2,...,v_l$ vertices.

Definition 4.1.11 (Trees and forests)**.** An undirected graph with no cycles is called a *forest*. A connected forest is called a *tree*. A directed graph is a tree (or forest). If all edges are converted to undirected edges, it is an undirected tree (or forest).

Definition 4.1.12 (Spanning Tree)**.** If G is a connected graph on n vertices, a *spanning tree* for G is a subgraph of G that is a tree on n vertices.

Theorem 4.1.1. Every connected graph contains a spanning tree.

Proof. By induction on the number of edges.

- If G' is connected and $|E(G)|=0$, it is a single vertex, so G is already a spanning tree.
- $|E(G)| \geq 1$ edges. If G has no cycles, it is its spanning tree. If G has cycles, remove one edge from each cycle. The resulting graph G' is still connected and cycles free, containing all the vertices of G. This is also a spanning tree for G.

□

Definition 4.1.13 (Shortest Path Tree)**.** Given a graph G, the *shortest path tree* rooted at vertex u is a spanning tree T of G, such that the path distance from root u to any other vertex v in T is the shortest path distance from u to v in G.

Definition 4.1.14 (Cut Edge)**.** Let G be a connected graph. A *cut edge* (bridge) is an edge e that $G-e$ results in a disconnected graph. Therefore two or more graphs are formed.

4.1.2 Basic algorithms

This subsection will detail basic graph algorithms, which will help us carry out the practice.

Breadth First Search (BFS)

A breadth-first search (BFS) is a search algorithm that traverses the nodes of a graph, starting with a root node[1] and explored all the neighbors of that node. Then, for each neighboring node, its adjacent neighbors are examined, and so on until the entire graph (nodes that share the same connected component as the root node) is traversed.

The implementation of the breadth-first search algorithm in pseudocode is shown below Algorithm 1. We initialize the list of visited nodes to false and the distance vector as infinite. Then we traverse the graph nodes starting with the neighbors and then the child nodes. Each time we visit a node, that node is set to true in the visited list, and the distance is updated. Since we are doing a breadth-first traversal, the data structure used is a queue since it processes the nodes that first arrive at the queue.

Algorithm 1 Breadth-First Search's algorithm

BFS(G,s)
Input: A graph $G=(V,E)$ and a node s of G
Output: d is a vector of size $|V|$ whose components are the distances from node s to all graph vertices.

```
1:  for each node v ∈ V(G) do
2:      visited[v] ← false
3:      d[v] ← ∞
4:  visited[s] ← true
5:  d[s] ← 0
6:  Q := a queue data structure, initialized with v
7:  while Q is not empty do
8:      u ← remove node from the front of Q
9:      for each v adjacent to u do
10:         if not visited[v] then
11:             visited[v] ← true
12:             d[v] ← d[u] + 1
13:             insert v to the end of Q
14: return d
```

This algorithm takes $O(V)$ time to initialize the distance and predecessor for each node. We discover all its neighbors for each node by traversing its adjacency list, so each node is

[1] In the case of a non-tree, we chose some node as a root

enqueued at most once. Since we explore the edges incident on a vertex only when we visit from it, each edge is examined at most twice, one for each of the vertices it's on. Thus, BFS spends $O(|V|+|E|)$ time.

Dijkstra

Dijkstra's algorithm is a well-known graph search algorithm that solves the single-source shortest path problem for a graph with non-negative edge path weight. This algorithm is very similar to BFS, but Dijkstra uses a priority queue instead of a queue. Thus, when visiting the unvisited neighbors of a node, we insert it into the priority queue using the distance from the edge. The priority queue will first extract the node with the least distance from the source. The implementation of Dijkstra's algorithm in pseudocode is shown below Algorithm 2.

Dijkstra finds the path with the lowest cost (i.e., the shortest path) between one node and every other node in the graph. The algorithm can also use for finding the costs of the shortest paths from a node to a destination node. Once the algorithm finds its way to the destination node, the algorithm stops.

Algorithm 2 Dijkstra's algorithm

DIJKSTRA$(G, source, target)$
Input: A graph $G = (V, E)$, a *source* node and a *target* node of G
Output: variable d as $SP_G(source, target)$

```
1:  Q := a priority queue, initialized with s with priority 0
2:  visited := empty set of node
3:  while Q is not empty do
4:      (n, d) ← extract (node,distance) from Q
5:      if n = target then
6:          return d
7:      add n to visited
8:      for each arc from node n to some node n' with weight w do
9:          if n' not in visited then
10:             insert n' with priority d + w in Q
11: raise NodeNotFound
```

We can implement Dijkstra's algorithm more efficiently by storing the graph in adjacency lists and using a search tree, binary heap, or a Fibonacci heap as a priority queue to extract the minimum efficiently. This will produce a running time of $O(|E|+|V|\log|V|)$.

Chapter 5

Problem formulation

5.1 Model description

An Active Directory attack graph is defined on a directed graph $G = (V, E)$ with $|V| = n$ vertices and $|E| = m$ edges, with a positive weight $w(e)$ associated with each edge $e \in E$. The graph contains a set of *entry nodes* where the attacker will start the attack. In our model, we study a case where the entry nodes are the nodes the attacker has already compromised (probably from phishing attack victims), so he can initiate the attack from any of those nodes.

Let s be the total number of entry nodes. Given a entry node s_i and a destination node d, a *shortest path attack* $SP_G(s_i, d)$ from s_i to d in G is defined as a path which minimizes the sum of the weights[1] of the edges along the path from s_i to d. We consider only one destination node[2] as the Domain Controller. The distance between two vertices $u, v \in V(G)$ (i.e., the length of $SP_G(u, v)$) is denoted by $d_G(u, v)$.

In this problem, the defender has a limited budget for blocking edges to protect themselves from attacks and delay the attacker's movement. They must decide the b edges to secure, where b is the defensive budget. We use B to denote the set of edges blocked. Once the edges are blocked, the graph is denoted as $G - B$, where $G - B = (V, E \setminus B)$. As mentioned in Section 1.1.1, attackers know which edges have been blocked by defenders, so they perform a best-response attack. This best-response attack consists of the *shortest path attack* to the DC, denoted by $SP_{G-B}(s_i, DC)$. Thus, the stage environment is as follows:

Scenario: *The attackers have compromised the entry nodes*. The attacker can start the attack at any of the entry nodes. Therefore, knowing that the attacker will perform a shortest

[1] We work with weighted ($w(e) \geq 0$) and unweighted graphs ($w(e) = 1$).

[2] There is usually more than one node with high privileges (Domain Admins). We merge all the administration nodes into a single node and call it Domain Controller.

path attack, the defense will focus on finding b blockable edges $\{e_1, e_2, ..., e_b\} = B$ whose blocking is such that the sum of the weights of $SP_{G-B}(s_i, DC)$ is maximum. These edges are recognized as the *b most vital edges*.

We experiment with three distinct graph characteristics. First, we focus on studying a graph with no weights ($w(e) = 1$). In the second, we add a property to the edges: there will be a set of edges that can be *blockable* (others cannot) to protect against AD attacks, denoted by $E_b \subseteq E$. And finally, the third type of graph is with weights($w(e) \geq 0$), and not all edges are blockable.

We are interested in the scenario where the goal is to block some edges so that the shortest $s-t$ path gets longer in the resulting attack graph. These problems are motivated by obvious applications in investigating robustness and critical infrastructure in network design.

The following figure 5.1 is a simple representation of the problem network. Nodes A and B are the entry nodes, and node K is the Domain Controller. The attacker's objective will be to get from one of the red nodes to the green node as quickly and efficiently as possible.

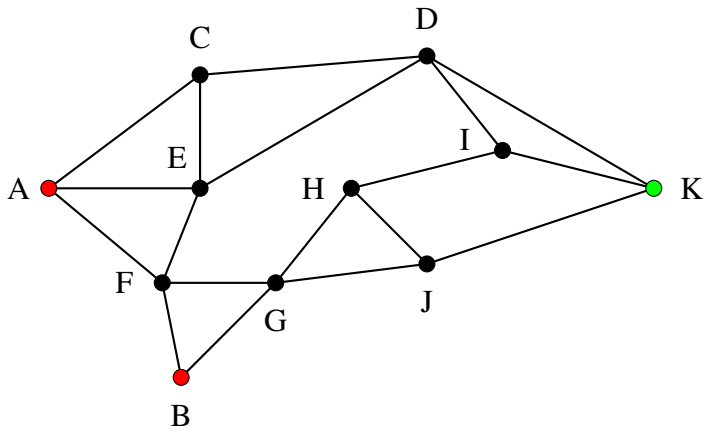

Fig. 5.1 Simple graphical representation of the problem.

5.2 The complexity of the problem

In this section, we prove that finding the most vital edges (MVEs) problem is NP-hard[3] with any weight, even when $w(e) = 1$, by showing that the decisional problem is NP-complete. Baruch Schieber et al. [9] proved this proof in 1995. We will name the corresponding problem D-MVEs (decision most vital edges). D-MVE is:

[3] A problem X is NP-hard if there is an NP-complete problem Y, such that Y is reducible to X in polynomial time.

INPUT: A directed graph $G = (V,E)$, with a positive $w(e)$ associated with each edge $e \in E$; entry node s and a target DC where $s, DC \in V$; budget $b \geq 0$ and a threshold $T \geq 0$.

OUTPUT: Yes, if there are b edges whose blocking is such that the sum of the weights of $SP_G(s,DC)$ is at least T; no otherwise.

It is apparent that if there were a polynomial algorithm to solve the MVEs problem, this would imply the existence of a polynomial algorithm for the D-MVEs problem. Next, we show that the D-MVEs is NP-Complete by reducing the decisional vertex cover problem (D-VC). D-VC is known to be NP-Complete [21], and it is often used in computational complexity theory as a starting point for NP-hardness proofs:

INPUT: A graph $G = (V,E)$, and a positive integer k.

OUTPUT Yes, if G have a vertex cover of size at most k, i.e., is there a subset $U \subseteq V$ with $|U| \leq k$ such that for each edge $(u,v) \in E$ at least one of u and v belongs to U; no otherwise.

The challenge is to prove D-MVEs $\leq_p$ D-VC. The proof proceeds as follows. Given a instance of D-MVEs, we associate a graph $G' = (V',E')$. Each graph node of G' is part of one of the "*parts*" the graph will contain. Each *part* of the graph consists of 2 parallel paths comprised of 5 edges with the same endpoints. Figure 5.2 shows the graph we want to form to reduce the problem. Where the *parts* explained above are these kinds of decagons[4] found in the graph G' (in the example image, 2 *parts* are observed). The first *part* corresponds to node i and the second *part* to node j. We will call these "*parts*" decagon/decagons for a better understanding. We should also note that each decagon's end (or rightmost) node is the same as the next decagon's first (or leftmost) node.

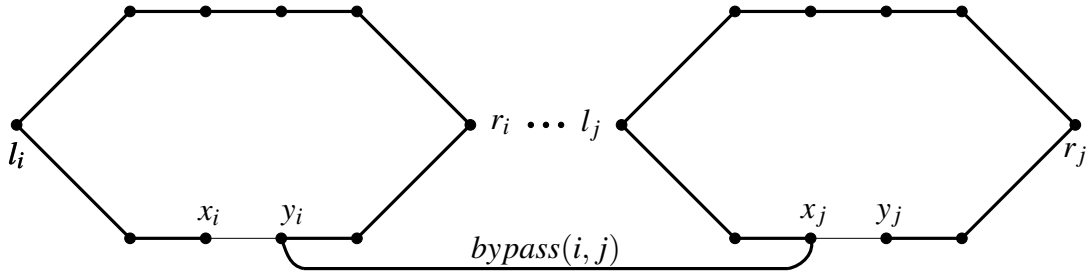

Fig. 5.2 The reduction

It is worth mentioning that the initial graph $G = (V,E)$, contains $|V| = n$ nodes labeled $1,2,...,n$. For each edge $(i,j) \in E$, $i < j$, we have added to the graph G' a path/edge of

[4] In geometry, a decagon is a ten-sided polygon.

weight $5(j-i)-2$ from node y_i to node x_j. (See Figure 5.2). This alternative path we have just specified will be called *bypass* (i, j).

In this graph, for each decagon i, only the edge (x_i, y_i) is blockable (the thinnest lines in Figure 5.2). More characteristics of the entry of the graph G': node s is the first node of the first decagon (the leftmost entry node of the graph), and node DC is the last node (r_n) (the rightmost node of the graph). The parameter b is set to be k, and the threshold T is set to be $5n$.

We prove the following property about graph G', which will be helpful later:

Lemma 5.2.1. Let G'' be a subgraph of G' obtained by removing some blockable/removable edges, and let $SP_{G''}(s, DC)$ be the shortest path. Then all the edges in $SP_{G''}(s, DC)$ are traversed from left to right.

Proof. Let (i, j), $i < j$, be a *bypass* in G'. Baruch Schieber et al. [9] found the following observations from the alternative *bypass* (i, j) path of length $5(j-i)-2$.

Observation 1. The path from r_i to x_j uses only the upper edges from decagons $i+1$ to $j-1$, and when it reaches the decagon j, it uses the lower edges $l_j, ..., x_j$; total path length is $5(j-i)-3$. Any other path between these two nodes and using the *bypass* (i, j), its length is at least $5(j-i)$.

Observation 2. The path from y_i to l_j uses only the lower edges in the decagon i, and then from r_i to l_j uses only the upper edges; therefore its path length is $5(j-i)-3$. Any other path between these two nodes and using the *bypass* (i, j), its length is at least $5(j-i)$.

□

Back to the reduction.

Lemma 5.2.2. There exists a vertex cover for G of size at most $k = b$ if and only if there are b edges in G' whose removal increases the length of $SP_{G'}(s, DC)$ to $5n$.

Proof. $\Leftarrow$) Now assume a vertex cover on the graph G of size at most $k = b$. For each decagon of the graph G', we block the blockable edge (x, y). Then, once the n edges are removed (one for each decagon), it is easy to observe that $SP_{G'}(s, DC)$ results in $5n$. For the sake of proving, let us assume the contrary and exists a $SP_{G'}(s, DC) < 5n$.

Bypasses had to be used at least once for the shortest path to be smaller than $5n$. Suppose the bypass (i, j), since either i or j are in the vertex cover for G, we blocked the blockable edge in decagon i or the blockable edge in gadget j, or we blocked the two edges. As a result, the $SP_{G'}(s, DC)$ must use some *bypass* (i, j), but it finds that the path is blocked, and has to

pull back or go from right to left from x_j to l_j or from r_i to y_i. So it contradicts the previously described lemma 5.2.1.

$\Rightarrow$) Now suppose that there are b edges in the graph G', such that blocking them does $SP_{G'}(s,DC) = 5n$. Recall that we can only block edges of the style (x_i, y_i) and that they each belong to a decagon. We can claim that the set of nodes $U \subseteq G$ corresponding to these decagons is a vertex cover of G. Now let us assume the contrary, given the edge (i, j) with nodes i and j, both i and j are not in U. This would imply that in the graph G', neither (x_i, y_i) nor (x_j, y_j) edges are blocked in their respective decagons. Not blocking these edges causes the shortest path to use the uppermost portions of all the decagons, except decagon i and j that use the *bypass* (i, j), therefore the $SP_{G'}(s,DC)$ is of length $5n-1$; a contradiction to our assumption. □

5.3 Most vital edge problem

This study starts by studying the case where the budget equals one, i.e., we can block only one edge. In the past, researchers have studied the problem of finding an edge in a graph such that removing it (in our case, blocking it) from the graph results in the greatest increase in the shortest path length between two given nodes s and t. Such an edge is called the *most vital edge*; there is only one edge in the graph, and it must belong to the path $SP_G(s,t)$. The following simple observation is straightforward.

Observation 3. The most vital edge concerning the shortest $s-t$ path of a graph G must lie on the shortest $s-t$ path.

Strictly speaking, the problem is to find an edge e (*most vital edge*) of the shortest path, such that $d_{G-e}(s,t) \geq d_{G-e'}(s,t)$ for every edge e' of the graph G, where $G-e = (V, E-e)$.

These problems are essential in network applications to identify sections of a network that need to be protected, such as our defensive problem model in Active Directory environments. Let's consider the following conflict situation in a two-point communication network: There is a defender who seeks to find the weakest road in the network to reinforce it against possible attacks. While on the other side, an attacker seeks to destroy the road resulting in the most significant increase of the shortest path distance through the network between two given points. This situation is an example of possible network application problems different from the one we have formulated. This type of problem, where the goal of increasing the shortest path distance is sought, will be referred to as the MVE (the most vital edge) problem for brevity and ease of reference.

One thing to bear in mind is that if graph G contains bridges (i.e. is not *2-edge connected*) and for some edge e in $SP_G(s,t)$ does not have an alternative path. With Tarjan's first algorithm for finding bridges, we can solve the MVE problem with linear complexity $O(m)$ [22]. That is, if s and t are not 2-edges connected, then any edge whose removal results in the separation of s from t is a single most vital edge of G.

Chapter 6

Algorithms

In this chapter, the different algorithms implemented will be presented. We will begin by describing two algorithms for resolving the most vital edge, which will become part of a greedy algorithm, and we will finish by implementing an FPT algorithm. In our scenario, the attackers have compromised specific entry nodes and can decide which node to start attacking. The defender aims to block the edges to maximize the shortest path. You should note that this chapter does not explain the local search and heuristics algorithms implemented, as these have their explanation in chapter 7.

6.1 Solving the MVE problem

We will propose different ways of calculating the MVE problem in the following. It is evident that the most vital edge of the graph belongs to each shortest path. We use this observation to develop algorithms, some more efficient than others, to solve this problem.

6.1.1 Basic sequential algorithm

First, we will start with a basic algorithm, the typical algorithm that runs through our heads. We begin by calculating the shortest path from the entry nodes to the Domain Controller. If there is more than one entry node $s > 1$, we add an auxiliary node s_0 to the attack graph and connect s_0 to each entry node with a zero-weight edge. Now we can calculate $SP_G(s_0, DC)$ by applying Dijkstra's algorithm in $O(m + n\log n)$ time in case we deal with a weighted graph [1] or by using a modified Breadth-first search in $O(m+n)$ time otherwise.

[1] In our model, there are no negative weights in the attack graph.

After obtaining the shortest attack path, we block (remove at the graph level) sequentially all edges $e = (u, v)$ along $SP_G(s_0, DC)$ and computing at each step $SP_{G-e}(s_0, DC)$, we are left with that edge e_i such that $SP_{G-e_i}(s_0, DC)$ is maximum. This way of solving the problem is quite expensive in general terms, but the maximum attack path length in Active Directory graphs is relatively short [2]. Applying Dijkstra for $O(n)$ edges in $SP_G(s_0, DC)$, the algorithm gives a total amount of time of $O(nm + n^2 \log n)$. In the following algorithms, we will improve this time.

6.1.2 A more efficient solution

Let us now solve the MVE problem more refinedly, avoiding computing Dijkstra for each edge of the shortest path. This algorithm was discovered by Malik, Mittal, and Gupta [23]. In this paper, we explain the algorithm and perform our implementation. As in the previous algorithm, if there is more than one entry node, we use the single auxiliary node s_0 as the source. Let $SP_G(s_0, DC)$ be the shortest path from the auxiliary entry node s_0 to the Domain Controller DC at $G = (V, E)$. We calculate the single-source shortest path trees from s_0 and DC to all the nodes, denoted as $ST_G(s_0)$ and $ST_G(DC)$, respectively. With the time complexity of $O(m + n \log n)$, we can calculate the shortest paths tree [24]. As mentioned above, the most vital edge with the described properties must belong to the set of edges of the shortest $s - p$ path. The following result was identified in [23].

Observation 4. An edge (u, v) is on a shortest $s - t$ path if and only if

$$d(s,t) = d(s,u) + w(u,v) + d(t,v) = \min_{(x,y) \in E} \{d(s,x) + w(x,y) + d(t,y)\}$$

If any edge $(u, v) \in SP_G(s_0, DC)$, with u closer to s_0 than v, is cut from ST, then two subtrees are created. Let the first set of the subtree $M_{s_0}(u)$ denote the set of nodes reachable in $ST_G(s_0)$ from s_0 without passing through the edge (u, v) and let $N_{s_0}(u) = V - M_{s_0}(u)$ be the remaining nodes. By removing the edge e, we notice that the distance from s_0 to the other $M_{s_0}(u)$ nodes does not change, while the distance from s_0 to the other $N_{s_0}(u)$ nodes may increase because of removing e.

An erroneous claim made in [23] was corrected by Bar Noy et al. [9]. The corrected claim is as follows:

[2] There are short paths because it comes from the idea that all people are six or fewer social connections away from each other: *Six degrees of separation*.

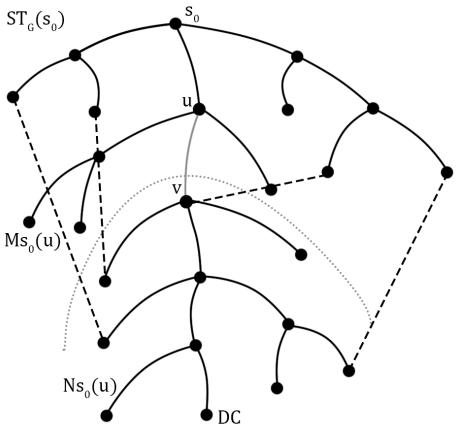

Fig. 6.1 $M_{s_0}(u)$ and $N_{s_0}(u)$

Observation 5. Let $ST(s_0)$ the single-source shortest path tree rooted at s_0 and let $SP_{ST}(s_0, DC)$ the shortest path in $ST(s_0)$. If some edge $(u,v) \in SP_{ST}(s_0, DC)$ is removed from $ST(s_0)$, dividing the node set V into $M_{s_0}(u)$ and $N_{s_0}(u)$ such that $s_0 \in M_{s_0}(u)$ and $DC \in N_{s_0}(u)$, then there exists shortest paths from all other nodes in $N_{s_0}(u)$ to DC that do not use the edge (u,v).

Proof. By contradiction. We consider a node $i \in N_{s_0}(u)$, and we have stated that u cannot be on the path $SP_{ST}(i, DC)$. Suppose u does lie on that path. Then there are shortest paths from u to i and from u to DC paths in $ST(s_0)$. Since $DC, i \in N_{s_0}(u)$, it turns out that these paths use the edge (u,v). Hence $SP_{ST}(i, DC)$ is not simple, which contradicts the assumption that it is a shortest path. □

Let the partition of V into $M_{s_0}(u)$ and $N_{s_0}(u)$ defines a cut in $G-e$, and

$$Q_{s_0}(u) = \{(x,y) \in E - (u,v) \mid x \in M_{s_0}(u) \wedge y \in N_{s_0}(u)\} \tag{6.1}$$

is the set of *crossing edges*[3]. Since $Q_{s_0}(u)$ is a (s_0, DC)-cut, it is clear that the shortest path $SP_{G-e}(s_0, DC)$ must have an edge in the crossing edges. Thus, such edge and the length of the shortest $s_0 - DC$ path in $G-e$ can be identified using the previous observations by computing

$$d_{G-e}(s_0, DC) = \min_{(x,y) \in Q_{s_0}(u)} \{d_{G-e}(s_0, x) + w(x,y) + d_{G-e}(y, DC)\} \tag{6.2}$$

[3] A *crossing edge* is an edge from a node u to a node v such that the subtrees rooted at u and v are distinct.

By having $ST_G(s_0)$ and $ST_G(DC)$ calculated, we can obtain all the terms of the above expression in $O(1)$ time. In fact $d_{G-e}(s_0,x) = d_G(s_0,x)$, since $x \in M_{s_0}(u)$, therefore, from the shortest tree path $ST_G(s_0)$, we can get the distance in time $O(1)$. Then, $w(x,y)$ is the weight of the edge with vertices x and y, the obtaining is immediate. Finally, we have the term $d_{G-e}(y,DC)$, which can be extracted from $ST_G(DC)$, as it complies:

Lemma 6.1.1. Let $(x,y) \in Q_{s_0}(u)$. Then, we have that $y \in M_{DC}(u)$.

Proof. By contradiction. Suppose $y \notin M_{DC}(u)$, i.e., $y \in N_{DC}(u)$. □

If we compute the formula 6.2 for every $(u,v) \in SP(s_0,DC)$, we can spot the most vital edge. Finding the most vital edge concerning the $s_0 - DC$ path will thus require $O(m|P|)$ time. Using Fibonacci heaps, Malik reduced this to $O(m+nlogn)$.

Algorithm 3 Most vital edge algorithm

MVE(G)
Input: Attack graph $G = (V,E)$
Output: The edge that, when removed, results in the greatest increase in the $SP(s_0,DC)$

```
1: Compute ST_G(s_0) and ST_G(DC)
2: Initialize dist = 0; lst(i) = ∞, i ∈ E
3: for blockable edge e in SP(s_0,DC) do
4:     for each (u,v) ∈ E do
5:         if lst(v) > d_{G-e}(s_0,u) + w(u,v) + d_{G-e}(DC,v) then
6:             if lst(v) = ∞ then
7:                 HEAP_INSERT (v, d_{G-e}(s_0,u) + w(u,v) + d_{G-e}(DC,v))
8:             else
9:                 HEAP_DECREASE (v, d_{G-e}(s_0,u) + w(u,v) + d_{G-e}(DC,v))
10:    if HEAP_FINDMIN > dist then
11:        dist = HEAP_FINDMIN
12:        edge = e
13:    HEAP_DELETE(e), ∀i ∈ N_i \ N_{i-1}
   return edge
```

6.2 A Greedy-based algorithm implementation

Once the most vital edge is obtained, given the budget b, we will repeat the MVE algorithm b times. We will modify the graph each iteration by removing the most vital edge for a more straightforward resolution.

Algorithm 4 Greedy Maximize the Shortest Path

GREEDYSP(G,b)

1: **repeat**
2: $vital\_edge := MVE(G)$
3: remove $vital\_edge$ to G
4: decrease the budget b by one unit
5: Record $vital\_edge$ in B
6: **until** no more budget b
7: **return** $SP_{G-B}(s_i, DC)$

Observation 6. GREEDYSP is the optimal algorithm if graph G is exactly a tree.

Proof. This is straightforward to see. Because when you block an edge, it occurs on different branches of the tree, and they are independent of each other. Given the edge $e_1, e_2 \in G$, if the path e_1 to DC passes trough e_2, then GREEDYSP will always block e_2. □

Another property, easy to observe, is that problem MVEs is not equivalent to sequentially performing the MVE algorithm. In fact, the most vital edge may not be amongst the b most vital edges for $b > 1$.

We can make a stronger statement with the example graph Fig.6.2 proposed in [23].

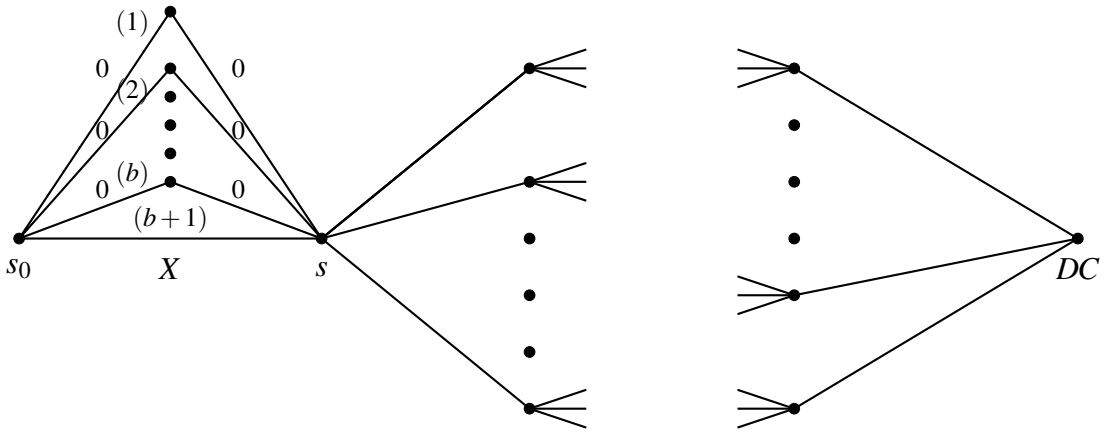

Fig. 6.2 Example graph, where the MVE is not found among the b most vital edges for $b > 1$.

Suppose the graph G is the one in Figure 6.2. We have a blocking budget of $b-1$ edges with s_0 as the entry node and DC as the Domain Controller. The result will be the same starting from node s, since from s_0 to s the distance will be 0. Therefore, edges $1,2,...,b+1$ will not be vital edges. On the other hand, the edges $1,2,...,b$ are the b most vital edges when X is highly large. So with that example, we have seen that the most vital edge may not be among the b most vital edges when $b > 1$.

Furthermore, the greedy algorithm does not work well with substitutable block-worthy edges. To make it more visible, let's assume the following graph 6.3:

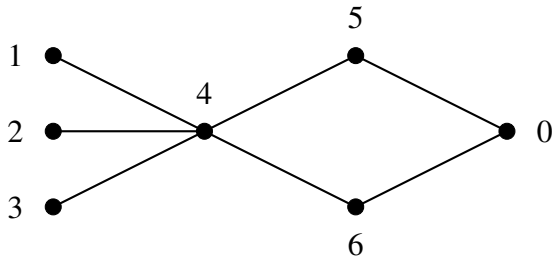

Fig. 6.3 Example of a graph with substitutable block-worthy edges.

Let nodes $1,2,3$ be the entry nodes, and node 0 be the Domain Controller. If the budget b is 2, it is easy to see that the optimal defense would be to block edges $\{4,5\}$ and $\{4,6\}$, or $\{5,0\}$ and $\{6,0\}$. However the greedy MVE algorithm would block two edges from the set $\{\{1,4\},\{2,4\},\{3,4\}\}$. The algorithm would block none of the previously mentioned optimal blocking edges. The most optimal solution of this graph is ∞ (we cannot reach the *DC*). While the Greedy algorithm has not improved the defense, it still has the same shortest path weight as before we ran the algorithm.

6.3 An FPT-based algorithm implementation

In this section, we briefly review some fundamental concepts of parameterized complexity. We implement an FPT algorithm for our problem.

6.3.1 Parameterized complexity

The idea behind fixed-parameter tractability is that starting from an NP-hard problem, we try to separate the algorithmic complexity into two pieces. One part depends purely on the size of the input (in our case, the size of the graph G), and the other depends only on some parameter of the problem (in our case, the b budget). One part scales polynomially to the input size but can exponentially scale the problem from a parameter. More technically, an instance of a parameterizable problem is a pair (I,k) where I is the main part of the problem input, and k is the parameter. An FPT algorithm is an algorithm that decides any instances (I,k) in time $f(k)|I|^{O(1)}$ where f is a computable function solely depending on k and $|I|$ denotes the size of I.

Let's look at an example of an algorithm that belongs to this class and is considered one of the best-studied problems in parameterized complexity, the P-VERTEX COVER algorithm that requires $O(|E(G)|2^k)$ time. Let's look at the algorithm that solves the parameterized vertex coverage problem:

Algorithm 5 p-Vertex Cover

P-VC(G,k)
Input: a graph G and a positive integer k (*parameter*)
Output: does G contain a vertex cover of size at most k?

1: **if** $|E(G)| = 0$ **then**
2: **return** true
3: **if** $k = 0$ **then**
4: **return** false
5: Select and edge $e = u, v \in E(G)$
6: **return** P-VC$(G-u,k-1)$ or P-VC$(G-v,k-1)$

We stop a tree branch if we reach a node labeled with (G',k') such that either $k' = 0$ or G' has no edges. It is easy to see that this bounded search tree[4] algorithm is correct and decides P-VC in time $O(|2^k|n)$ for graphs with n vertices.

6.3.2 Solving the problem with an FPT algorithm

We now explain the proposed FPT algorithm (MVEsFPT). This exhaustive algorithm solves all possible configurations blocking b edges along graph G, maximizing the attacker's shortest path. As with the previous algorithms, if there is more than one entry node $s > 1$, we add an auxiliary non-blockable node s_0 to the attack graph and connect s_0 to each entry node with a zero-weight edge. We repeatedly compute $SP(s_0, DC)$ to generate all combinations. After blocking (removing) an edge, we reduce the budget to 1. When the budget is 0, we return the shortest path (as well as the blocked edges). When the budget is 1, we compute the shortest path in linear time, and this cost is distributed over d combinations (since there are at most d edges of the shortest path to block). When the budget is 2, the cost is distributed in d^2 combinations and so on, successively d^b. Therefore, having the number of combinations d^b, the algorithm's complexity would be $O(d^b(m + n\log n))$ where d is $|SP_G(s_0, DC)|$. Below is the code in pseudocode:

[4]The idea of the bounded search tree technique is to bound the search tree's degree of branching and depth by the parameter k.

Algorithm 6 Fixed-parameter tractable algorithm

MVEsFPT(G,b)
Input: Attack graph G, budget b
Output: The $SP(s_0,DC)$ of the most significant increase occurs in eliminating b edges.

1: $p = SP_G(s_0,DC)$
2: **if** $b = 0$ **then**
3: **return** p
4: **for** blockable edge e in p **do**
5: Get G' by removing e
6: Record MVEsFPT$(G',b-1)$
7: Revert back to G
8: **return** max(recorded value)

Chapter 7

Heuristics

7.1 Introduction to local search optimization

Local search is a heuristic method for solving computationally difficult optimization problems. The basic principle of local search is to find a better solution from a given initial solution. Using what we will call operators, the algorithm moves from solution to solution in the space of candidate solutions by applying local changes. Each solution evaluates several solutions (from *heuristic function*) and uses the most suitable move to take the step to the next solution. This procedure is repeated until an optimal solution or a time limit is reached. This project uses one heuristic method to address the problem.

7.1.1 Hill Climbing

The Hill Climbing algorithm is the most straightforward local search algorithm used. The problem is that it is somewhat optimistic and assumes that there is a relatively direct path to local optima. This does not have to be the case when we face challenging problems. More specific strategies are needed, managing to explore more nodes and knowing how to exploit the information that is located along the way.

The hHill Climbing algorithm starts with an initial solution to the problem and tries to improve it by making local transformations. The algorithm moves between neighboring processors using the operators and then evaluates the generated solution from that move. If the change is positive, the movement is accepted and moves to the new configuration. Otherwise, the old configuration is kept. This process is repeated until no more change occurs, i.e., a local minimum has been found, and there is no more improvement. The algorithm can be pictured as follows.

Algorithm 7 Steepest Ascent Hill Climbing

HILL CLIMBING

1: generate an initial configuration S_0 ($S \leftarrow S_0$)
2: **repeat**
3: $S' \leftarrow$ calculate a neighboring configuration by a local move in S
4: **if** cost(S') $<$ cost(S) **then**
5: $S \leftarrow S'$
6: **until** there is no better move

7.1.2 Simulated Annealing

The algorithm is similar to Hill Climbing, except instead of picking the best move, it chooses a random action most of the time. The Simulated Annealing algorithm works as follows.

The algorithm performs a total number of local random transformations to reach the temperature equilibrium. The temperature value decreases by a certain amount, starting from an initial temperature and reaching zero in the last phase. This algorithm will be necessary to choose the most suitable initial temperature and the best way to decrease it.

This algorithm can adapt very well to combinatorial optimization problems and escape from an ample search space in which many local optima surround the optimum. Simulated Annealing offers a way to overcome Hill Climbing's major drawback, but the computational time is longer. The algorithm can be pictured as follows.

Algorithm 8 Simulated Annealing

SIMULATED ANNEALING

1: generate an initial configuration S_0 ($S \leftarrow S_0$)
2: **while** the temperature is not zero **do**
3: **for** a preset number of iterations **do**
4: generate and compute a random neighboring configuration S'
5: $\Delta E \leftarrow$ cost(S') $-$ cost(S)
6: **if** $\Delta E > 0$ **then**
7: $S' \leftarrow S$
8: **else**
9: with probability $e^{\Delta E/T}$: $S' \leftarrow S$
10: temperature decrease

Figure 7.1 shows how Simulated Annealing would behave compared to Hill Climbing. As we can see, Hill Climbing gets stuck at the first local minimum (or maximum), while Simulated Annealing can explore more solution space and is more likely to find a good

solution. But this only is possible if we determine well the values of the parameters required by this algorithm. These parameters vary depending on the problem.

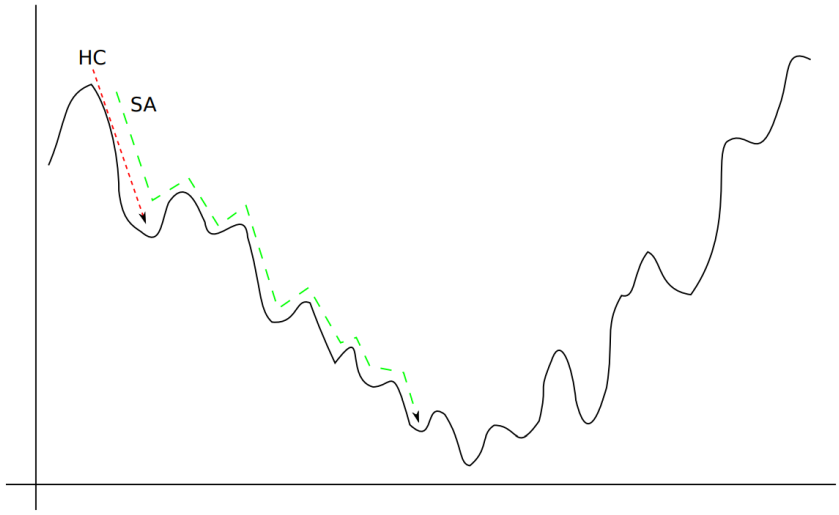

Fig. 7.1 Simulated Annealing vs. Hill Climbing strategy

7.2 Representation of the problem as a local search problem

We will use local search algorithms to address the problem in the following. We have had to think about how to represent the states, which operators to use to generate the successor states, the initial solution generators, and the quality functions that will measure how good a solution is and allow us to guide our search. We will look at how the problem behaves using *Hill Climbing* and *Simulated Annealing*. We will then describe how we have designed the elements necessary to perform a local search adapted to the problem we want to solve.

Problem state

First, we need to know how to implement and represent a state so that it is comfortable enough to work with, create operators, generate an initial solution, and facilitate the calculation of the heuristic function. So, the state is represented by the attack graph class. It allows us to access the edges, blockable edges, nodes, entry nodes, and required information.

Operators and successor generating function

Once we have the representation of the states, we choose which operators to use. Since we start from a solution graph, the attack graph contains b blocked edges where b is the

defensive budget. The reader should notice that implementing a simple operator "*block_edge*" would not be valid since the algorithm would give indications to block all the edges of the graph when it would not be possible as the budget limits it. Moreover, we would go from a solution state to a non-solution state in one move since we would have $b+1$ blocked edges. Something similar would happen with implementing an operator unblocking an edge "*unblock_edge*". The local algorithm would not use this operator as it would give solutions equal to or worse than the previous solution. We are interested in blocking b fair edges.

So, the most logical operator we implement is the operator: *move_edge*. Its function is a combination of the two operators dictated above. Starting from a blocked edge, it unblocks it and blocks a different edge. When applying these operators, we have to verify that these movements are valid, that is, only block blockable edges and never exceed the b blocking edges.

Heuristic function

We will now discuss the heuristic function implemented to guide the algorithm toward the objective of the proposed problem. Our problem focuses on blocking b edges to maximize the sum of weights of the shortest attack path from the entry nodes to the Domain Controller. Therefore, the heuristic function in this scenario is $SP_{G-B}(s_i, DC)$. This heuristic is valid as it will guide the algorithm toward the best route to maximize the shortest attack path. As previously mentioned, the cost of this heuristic function is $O(m+n\log n)$.

Initial state generation

For Hill climbing and Simulated Annealing to work, it has to start with a random solution to our problem. From there, it can generate neighboring solutions and begin the optimization process. The two very similar strategies implemented for initial state generation are:

- *generate_init_solution_1*: Our situation described in the model is based on blocking one of the edges of the shortest attack path. This first initial solution calculates the shortest path to the DC in time complexity of $O(m+n\log n)$ and randomly blocks one of the blockable edges of the path. The edges that remain blocked are randomly distributed throughout the attack graph.

- *generate_init_solution_2*: This proposed initial solution is simply random. It randomly blocks b blockable edges of the attack graph.

7.3 Genetic Algorithm

This section explains our latest heuristic algorithm, named Genetic Algorithm. Genetic algorithms are based on the analogy of natural selection as a mechanism of adaptation of living beings. This analogy comes from the idea that living beings adapt to their environments thanks to characteristics inherited from their progenitors, thus creating more competent generations to survive. As will be seen later, this concept of genetics and natural selection can be translated into a genetic algorithm. As in the previous algorithms, it will help us to find a near-optimal solution to problems where a polynomial solution is infeasible.

Genetic algorithms are more complex than the previous ones and require more parameters for their configuration, so good experimentation of the parameters is necessary. Here are some procedures to be performed:

- Decide the number of individuals of the initial generation.
- Have a coherent codification of the solutions.
- As in Hill Climbing and Simulated Annealing, create a function that measures the quality of the solution. This function is called the *fitness score*.
- Perform operators that combine solutions to obtain new solutions (*crossover*) and perform slight variations to the individual's genetic code (*mutation*).

Each of these elements we implemented is detailed in the following section.

7.4 Representation of the problem as a genetic algorithm

We use genetic algorithms to address the problem in the following. We must correctly implement the above requirements for the algorithm to work. Next, we detail the steps taken to operate the genetic algorithm.

Codification

The solutions, which are the individuals in the population, are represented in a particular way so we can combine them. Individuals are usually coded as bit strings; by analogy, this coding is called a gene. Each bit or group of bits in the line encodes a characteristic of the solution. The good thing about using a bit string is that the operators and modifications of the individuals are elementary to implement.

However, not constantly encoding the individuals in a binary way is the most appropriate; our algorithm does not use it. Binary encoding of individuals for our problem could be as follows:

Given the attack graph $G = (V,E)$ with $|V| = n$ nodes and $|E| = m$ edges, we create a list of edges e where $e[i] = 0$ or 1 for all $i \in 0,1,2,...,m-1$; Those edges that are at 1 mean that they are blocked, otherwise they are not blocked; It is obvious that $sum(e) = b$ should be satisfied, where b is the budget.

This binary representation could be valid, but it will not be helpful since we work with many edges, most of which will be zero. So at the time of combining and mutating them (passing some edge from non-blocked to blocked or vice versa), it is very likely to generate non-solution generations.

Our individuals contain information only about the blocked edges, i.e., a list of b blocked edges. This encoding makes it easier for us to operate, and we store less memory since every individual has only b edges, where b is usually a relatively low number.

Operators

Two types of operators in these genetic algorithms cause different possible solutions: crossover and mutation. Crossover operators are applied to a pair of individuals to create their successors. Their effect is to exchange part of the coding information between individuals. There are multiple ways to combine these pairs; the simplest is to choose a random point in the coding and exchange the two individuals' bits (or edges in our case) from that point. For a better understanding, the following Figure 7.2 shows a possible crossover for the famous N queens problem with binary codification:

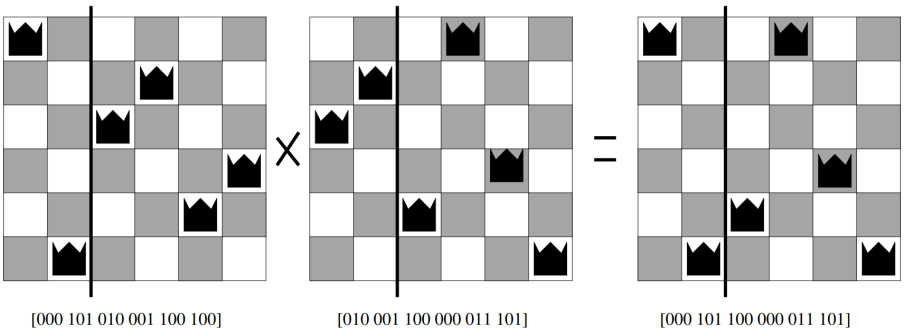

Fig. 7.2 Crossover example for the N queens problem.

In our algorithm, we use two methods. The first method is the same method used in the N queens example, choose a random point from the list of blocked edges and swap the halves

formed from that randomly chosen point. An example of this crossover is shown below (Figures 7.3 and 7.4). The other crossover is purely random. We take the edges according to two solutions, mix them, and distribute them to the following two new generation individuals.

With just these operators, it can happen that in the next generation, an individual inherits the same edge from their parents, thus creating what we call overlapping. When we detect an overlap, the technique we use to prevent it is that the individual inherits all the edges of one of its parents.

Figure 7.3 and 7.4 show an example of a crossover when $b = 4$. In figure 7.3, when a one-point crossover occurs between two individuals $p1$ and $p2$, offspring, $c1$ and $c2$, are generated with codification inherited from the parent $p1$ and codification inherited from the parent $p2$. Figure 7.4 shows the example that the offspring $c1$ is generated by inheriting m_a and m_b from $p1$ and inheriting m_k and m_a from $p2$, it is observed that the factor m_a overlaps. As a result, the offspring $c1$ inherits all edges directly from $p1$ to avoid the overlap.

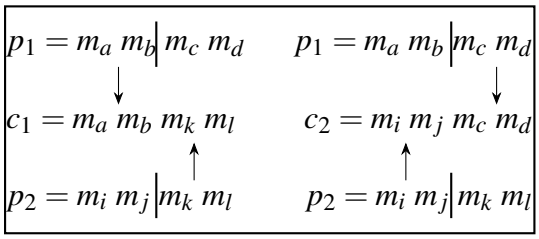

Fig. 7.3 Crossover without overlap

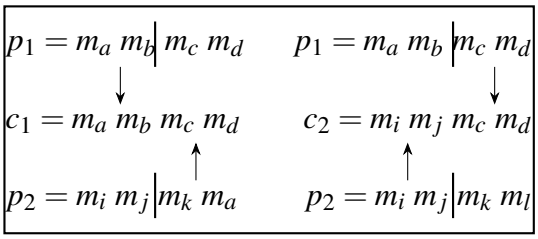

Fig. 7.4 Crossover with overlap

On the other hand, mutation operators are applied to a single individual and consist of randomly changing one of its factors. In this study, the overlap with the mutation is avoided since, at the moment of changing an edge for another blockable edge, this list of blockable edges does not contain the edges contained in the individual itself, so an overlap can't occur.

These operators have an associated application probability, a parameter of the algorithm. The crossover probability indicates when a chosen pair of individuals exchange their information to create the next generation, the successor also will have a possibility of mutation. In general, the crossover probability is much higher than the mutation probability. If the crossover probability is very high, the next generation will have a large population. This situation can be good because the exploration is done faster, but it tends to cause the patterns not to stabilize appropriately.

On the other hand, if the probability is very low, the solution exploration will be slower and may cause a delay in convergence. Something similar happens with mutation. Therefore, the choice of these parameters is fundamental for the correct functioning of the algorithm.

Fitness Function

As stated in the previous section, the fitness function evaluates how close a given individual is to the optimum solution of the problem. As during the last local search algorithms, it is a heuristic that determines how fit a solution is. Given the attack graph $G = (V, E)$ with $|V| = n$ nodes and $|E| = m$ edges, entry node s_i and destination node DC; B denotes the set of edges blocked. The fitness function used in this study is:

$$\max \mid SP_G(s_i, DC) - SP_{G-B}(s_i, DC) \mid \tag{7.1}$$

With objective function 7.1, we can expect that the greater the fitness value is, the more vital it is.

More details to bear in mind

The genetic algorithm must start with an initial population. We call the population the set of solutions obtained at a specific time. Suppose, at a given time, the population is too large. In that case, the cost per iteration can be prohibitive, so it is advisable to have a parameter that controls the maximum number of individuals (there can be no more than x individuals). In addition, if the number of individuals is too small, we will be prone not to obtain reasonable solutions, as we will not have visited enough solution space.

At each iteration of the algorithm, individuals pair up and give rise to new individuals, thus forming what we call a generation. When developing the algorithm, we must choose how we decide how the individuals are matched. There are many ways, in our study, we gather the individuals with the best solution, pass them to the next generation, and kill the rest. This method represents natural selection, where the fittest are more likely to reproduce, and the less fit may not produce offspring.

The steps performed by the genetic algorithm are summarized below.

1. We create an initial random population.
2. From the fitness function, we chose the N fittest individuals.
3. We pair the individuals, and for each pair:
   (a) We apply the crossover operator with a probability P_{cross}, and two new individuals appear in the next generation; otherwise, the original pair passes to the next generation.
   (b) With a probability P_{mutat}, the individuals that have been crossed are mutated.

4. We replace the current population with the new generation.
5. We go to point two until the population has converged or several generations have passed.

Chapter 8

Experimentation

8.1 Development specifications

This chapter details the experiments performed for each of the algorithms. All these experiments have been performed on a desktop computer. The hardware specs are an Intel® Core™ i7-8700K at 3.7GHz, 32GB of RAM, and an 8GB GTX 1080 GPU, with Windows 11 as the operating system. All algorithms are written in Python.

We performed all the experiments on the topology of Active Directory attack graphs generated by DBCreator. This BloodHound generator uses different types of edges; in our case, we only consider the three default edge types: MEMBEROF, ADMINTO, HASSESSION.

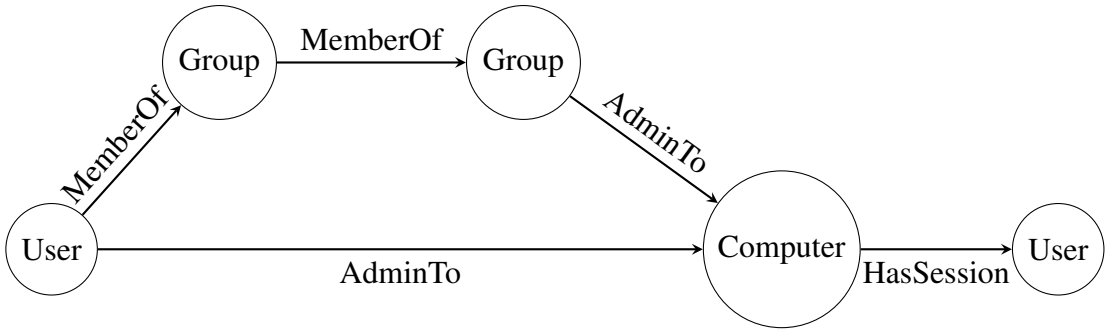

Fig. 8.1 The BloodHound attack graph design

8.2 Parameter and function selection

In this section, we determine which parameters and functions of the local search algorithms give the best results in order to compare the algorithms with the rest later. All choices occur

in the third version, i.e., we experiment with weighted attack graphs, in which some edges are non-blockable.

All the subsections of this section are realized in an attack graph with 605 nodes[1] and 2316 edges, where 500 edges are not blockable. The graph contains 7 entry nodes s and a budget b of 12 edges to block. All tables with the justification for the choice of parameters are included in the annexes. For the tables in the annexes, the same selection criteria are used as described in the following sub-sections.

8.2.1 Justification of Simulated Annealing parameters

Initial solution generator

In this first experiment, we must determine the best initial solution generation between the proposed two similar functions 7.2. We hypothesize that the initial solution generators are equal to or better than the other. The method by which the experiment will be performed is as follows: We will use the Hill Climbing algorithm. We will run ten tests for each initial function, where the entry nodes and blockable edges will vary, but the attack graph will be the same. We will measure the $SP_{G-B}(s, DC)$ length of the final solution for comparison.

Test	$\lvert SP\rvert$**ini**$_1$	$\lvert SP\rvert$**ini**$_2$	**Time[s]**$_1$	**Time[s]**$_2$
1	34	34	17.4958	17.4806
2	∞	∞	6.6918	7.4329
3	62	62	16.4502	16.3228
4	45	29	7.2480	1.5162
5	74	74	10.4523	10.7929
6	∞	∞	11.1669	11.5445
7	42	42	10.6115	10.1371
8	31	31	12.9074	12.8229
9	44	44	20.9352	24.1442
10	52	52	16.3839	16.2595

Table 8.1 Comparison of two initial generators solution of Hill Climbing

The attached table 8.1 shows the information from the above experiment. As we can see, practically all the results are the same, so there is almost no difference between the two initial generators in this case. It makes sense since the two initial generators start randomly, leading

[1] Out of 9000 nodes (users, groups, computers), only 605 nodes can reach the Domain Controller.

to poor results. However, in Test 4 (marked in red), we see a discrepancy; it turns out that function 1 has better results than function 2. If we recall, generator one blocks first any edge of the shortest path and blocks $b-1$ random edges, while generator two is purely random; it blocks b random edges. This leads us to think that in init solution 1, we have avoided a possible local maximum by securing an edge of the shortest path. This is because we found at least two shortest paths after the initial generator 2. So we cannot further improve the solution with the move operator, thus finding a local maximum. However, with function 1, we have eliminated one of these shortest paths (leaving one remaining). Then, with the move operator, we managed to block the remaining shortest path, improving the solution. As for the time, no appreciable difference is found between the two functions. From the results obtained, as they are very similar to the results and time, we have decided to stay with the first initial solution generator for the subsequent experiments.

Number of iterations

Simulated Annealing, unlike Hill Climbing, performs a fixed number of iterations before returning a result. Being N the number of iterations passed as a parameter to the algorithm, we must determine an appropriate number of iterations. We will experiment with the same graph by applying a different number of iterations, repeating it 5 times.

It is intuitive to think that too few iterations will not allow the shortest path result to converge to a local maximum. At the same time, too many iterations will perform redundant work by not improving the solution for the optimal number of iterations.

Keeping the same attack graph and with a temperature of 150, we have tried successive numbers of iterations. We have obtained the following results, consistent with our initial intuition of how they would vary the solution.

N	$\|SP\|_{\mathbf{min}}$	$\|SP\|_{\mathbf{avg}}$	$\|SP\|_{\mathbf{max}}$	$\mathbf{Time[s]}_{\mathbf{avg}}$
500	4	21	32	0.1156
2500	4	35	38	0.5441
5000	21	38	38	2.2580
10000	38	67	70	4.2928
15000	38	50	78	9.4559
20000	38	91	∞	8.9661
25000	116	∞	∞	10.1561
50000	∞	∞	∞	17.6046

Table 8.2 Determination of the number of iterations in Simulated Annealing

In table 8.2, we note that we achieved the best result once with 20,000 iterations, where we even managed to block the attacker's path to the Domain Controller. With 500 iterations, due to the size of the graph, we observe that the result is much worse. Moreover, the result has not improved after the initial generator. This is because the algorithm does not have enough iterations to converge. Whereas with 50,000 iterations, there is no difference in the maximum result (no further improvement is possible) compared to 20,000 iterations, and the time taken is twice as long. But with 50,000 iterations, we have achieved the best result in all tests. Since with 25,000 iterations, we almost achieve the optimum in all tests, the minimum result is excellent, and it is 50% faster than with 50,000 iterations. We have chosen $N = 25,000$ for the next experiments.

Temperature

As mentioned above, Simulated annealing attempts to solve the problem using a physical analogy. Where we find an energy function that measures the solution and a control parameter called temperature, which allows us to control the algorithm's operation; depending on the temperature and the difference in quality between the current candidate and the successor, it will choose the behavior of the successors. The higher the temperature, the higher the probability of creating a worse successor, but unlike Hill Climbing, this will allow us to jump from local optimal.

As the iterations pass, the temperature value will decrease by a certain amount, which is called the cooling strategy. It starts from an initial temperature and goes to zero in the last phase. When the temperature is shallow, Simulated Annealing becomes a Hill Climbing. Keeping the same graph from the previous experiment, and with 20.000 iterations to observe more variety of results, we have tried different temperatures. We have obtained the following results:

Temperature	$\lvert SP\rvert_{\textbf{min}}$	$\lvert SP\rvert_{\textbf{avg}}$	$\lvert SP\rvert_{\textbf{max}}$	**Time[s]$_{\textbf{avg}}$**
1	38	82	99	11.3516
25	39	78	∞	10.7649
50	39	144	∞	10.2808
100	74	79	∞	11.8815
150	70	75	79	11.5023
200	78	78	∞	9.7433
300	38	101	∞	9.5038

Table 8.3 Temperature determination in Simulated Annealing

At least once, we have reached the optimum result at different temperatures, except at temperatures 1 and 150. It is observed that the worst effects are due to low temperatures. It makes sense because they work as Hill Climbing[2]. However, with temperature 300, the algorithm jumps from one worst solution to another in the solution space, causing tests with bad results. These results are very variable (especially if your initial solution is random), so it does not change much in the long run to choose an average temperature. Therefore, we selected 200 as the temperature since we observed the best results.

8.2.2 Justification of genetic algorithm parameters

Crossover function

We have defined crossover as a function responsible for exchanging coding information between individuals. In our case, we have implemented two crossover types, a one-point crossover and a random crossover. For this experimentation, we start with 1000 initial individuals, capped at 10,000 individuals, sufficiently large to have valuable results. We have created one hundred generations of individuals, of which there is a 70% probability that individuals will exchange their information and a 20% probability that an individual will mutate.

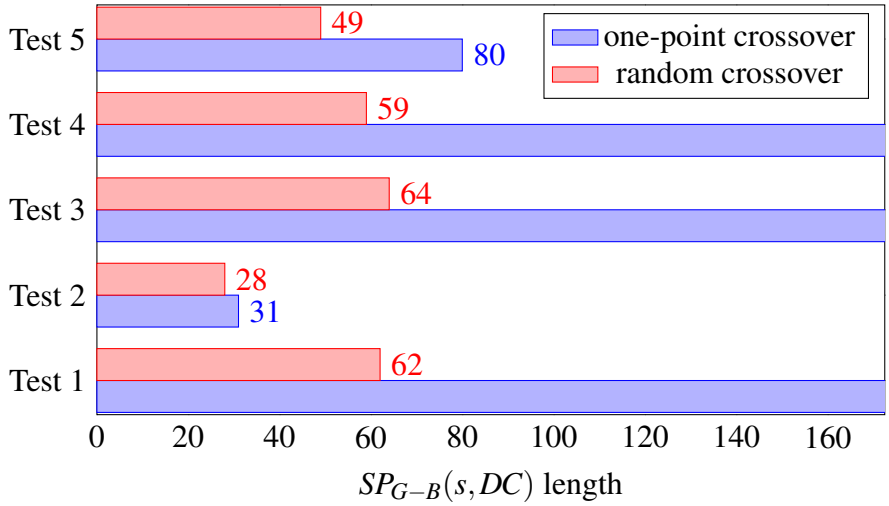

Table 8.4 Comparison of crossover operators

The following table 8.4 shows the difference in results for each crossing, where if the frame cuts the bar, it means it is infinite. Five tests have been used on the same graph,

[2]Simulated Annealing is considered to have better results than Hill Climbing.

varying the entry nodes and the non-blocking edges. As we can see in all tests, the one-point function gives better results than a completely random solution. The cause of this behavior is that individuals tend to have the most vital edges at the beginning of their genes. So in the creation of successors, the random function is more likely to cause overlapping, which makes the generations very similar to their ancestors and does not improve the solutions as fast as the one-point crossover. We have decided to keep the one-point crossover function for the subsequent experiments from these results.

Initial population

In this second genetic algorithm experiment, we must decide the initial number of individuals. The maximum number of individuals is 5,000, and there is a maximum of 100 generations, 70% crossover, and 20% mutation.

$N_{\mathbf{ini}}$	$\|SP\|$	**Time[s]**
50	62	500.6845
100	78	486.3106
250	69	470.4661
500	67	484.9114
1000	67	480.5552

Table 8.5 Determination of the initial population of the genetic algorithm

Looking at the table, we find 100 as the best number of initial individuals. Therefore, the remaining experimentation is performed with an initial population of 100.

Maximum population

Once the initial population is determined, we must decide the maximum number of individuals. The population size is an essential parameter in genetic algorithms since it influences the search space's ability to search for solutions. Having many individuals gives us a greater probability of obtaining the optimal solution. However, an excessive population can delay the algorithm. In our case, in each generation, we have to calculate the shortest path produced by each individual, and having a large population makes the algorithm more expensive. So it is essential to find a large population number to give good results and small enough not to take too long.

N	$\|SP\|$	**Time[s]**
500	62	50.0259
1000	67	97.4221
2000	67	192.8298
3000	67	289.2561
5000	64	475.4167

Table 8.6 Determination of the maximum population of the genetic algorithm

The best results have been obtained with 1000, 2000, and 3000 maximum populations (See Fig. 8.6). Looking at the time taken to complete the algorithm, a better time is achieved with a lower maximum population. So we decided to take 1000 as the maximum population.

Number of generations

We now turn to determine how many iterations are necessary to get as close as possible to the optimal solution. Increasing the generation number can improve the final result. But many generations may slow down the algorithm and cause many generations without improvement. The algorithm has been executed five times (test1,test2..,test5) in the same attack graph with the parameters that have been collected.

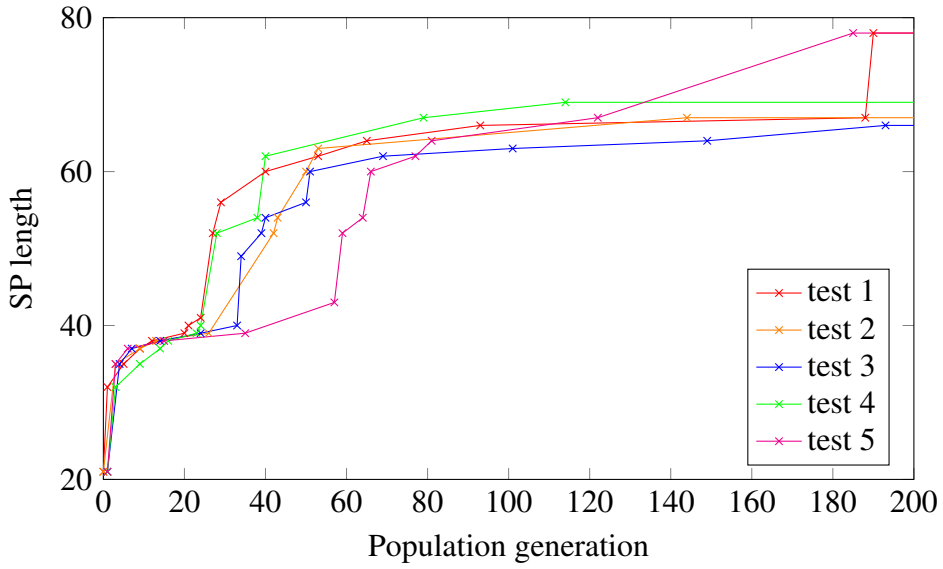

Table 8.7 Determination of the number iterations of the genetic algorithm

The table 8.7 shows how the solution evolves through the generations for the different tests. In the first generations, many new solutions appear, but as the generations go by,

the population converges, and it becomes more difficult to find better solutions. After 200 generations, no better solutions have been found in a reasonable time, so it has been decided that there will be a maximum of 200 generations/iterations.

Mutation rate

The mutation rate is usually applied in genetic algorithms because the algorithm needs to explore parts of the search space that we could not reach by combining the initial solutions. This mutation rate is usually relatively low; otherwise, it may cause the exploration to fail to converge by altering the solutions. Experiments have been conducted with percentages of less than 50 percent.

Mutation rate	$\|SP\|$	**Time[s]**
0.1	∞	46.4807
0.2	78	180.2610
0.3	52	212.4913
0.4	39	236.6045

Table 8.8 Determination of mutation rate

Table 8.9 shows how this is fulfilled, and the mutation percentage must be low. We will remain as 10% mutation.

Crossover rate

We now experiment with different percentages of combining genetic information from two parents to generate new offspring. Usually, the crossover probability is much higher than the mutation probability. However, suppose the crossover probability is very high. In that case, each generation will have many new individuals, which may make the exploration larger and faster. Still, depending on the problem, it may also make the solutions unable to stabilize. If the probability is minimal, the exploration will be slower and may take time to converge. In contrast to the mutation rate, we will study percentages above 50 percent.

We have found the optimal solution for all tests with different crossover percentages. So the choice of the rate will be based on the time spent. Finally, we chose 0.7 as the crossover rate.

Crossover rate	$\|SP\|$	Time[s]
0.6	∞	78.2471
0.7	∞	59.8753
0.8	∞	66.7604
0.9	∞	59.9824

Table 8.9 Determination of crossover rate

8.3 Experimentation of the proposed algorithms

This section focuses on comparing the results of the algorithms developed during the project. It will focus on studying three Active Directory attack graphs generated by DBCreator. For the smallest of the three graphs, we set up 1500 computers, and the resulting graph contains a total of 4505 nodes (users + computers + groups, etc.) and 19014 edges. The second attack graph includes a capacity of 9006 nodes and 47408 edges. Finally, the most significant attack graph we experiment with has a total of 30006 nodes and 188393 edges. All these attack graphs are then simplified to only those nodes that can reach the Domain Controller.

The section is subdivided into three subsections. The first part compares the algorithms with the three different graphs but with no edge weights, and all edges are blockable. In the second subsection, we compare the weighted graphs' algorithms; all edges are blockable. Finally, in the last part, the algorithms are compared to the graphs with weights, and not all edges are blockable. All of our experiments are repeated 5 times.

8.3.1 Experimentation with a small AD attack graph

This first experiment compares the different algorithms studied on relatively small Active Directory attack graphs. The simplified graph contains 280 nodes, 850 edges, 5 random entry nodes, and a budget b of 10 edges to block. For graphs where all edges are not blockable, we have blocked 150 edges. The blockable edges and the entry nodes are not changed in the five repetitions.

Table 8.10. Version 1: Note that FPT ALGORITHM is optimal. #Opt shows how many times the algorithm produces the optimal results (among 5 trials). We observe that the greedy algorithms consistently achieved the optimal result with the most accessible attack graph we will experiment with. This result makes us understand that the AD graph formed does not contain many cycles, does not have substitutable block-worthy edges, or it may be that the graph is quite tree-like. Both algorithms are high-speed.

Algorithm	$\|SP\|_{\min}$	$\|SP\|_{\text{avg}}$	$\|SP\|_{\max}$	Time[s]	#Opt
GREEDY ALGORITHM	∞	∞	∞	0.0180	5
COMPLEX GREEDY	∞	∞	∞	0.1766	5
FPT ALGORITHM	∞	∞	∞	343.9228	5
HILL CLIMBING	4	4	4	0.6919	0
SIMULATED ANNEALING	4	7	∞	3.3642	2
GENETIC ALGORITHM	7	∞	∞	46.4945	3

Table 8.10 The small AD attack graph results where all edges are blockable and contain unweighted edges.

Regarding the local algorithms, HILL CLIMBING gets the worst result. This is because, being a unit-length edges graph and in AD there are many relations, we find many shortest paths. This causes the plateaus to abound since the values of the neighboring nodes to the current one have equal values. Avoiding these problems requires extending the search beyond the neighbors to obtain enough information to route the search. An alternative is SIMULATED ANNEALING which, if we look at the results, has managed to avoid plateaus and find better results. The algorithm has found two times the optimum. However, the time spent was five times longer, but it compensates for the results obtained. On the other hand, the GENETIC ALGORITHM had outstanding results (3 times the optimum), but the time taken was quite long.

Algorithm	$\|SP\|_{\min}$	$\|SP\|_{\text{avg}}$	$\|SP\|_{\max}$	Time[s]	#Opt
GREEDY ALGORITHM	6	∞	∞	0.02552	4
COMPLEX GREEDY	8	∞	∞	0.1727	4
FPT ALGORITHM	∞	∞	∞	408.9983	5
HILL CLIMBING	5	5	∞	1.9712	1
SIMULATED ANNEALING	5	∞	∞	4.2326	4
GENETIC ALGORITHM	10	∞	∞	9.9265	4

Table 8.11 The small AD attack graph results where not all edges are blockable and contain unweighted edges.

Table 8.11. Version 2: For this second version, where non-blocking edges have been added. No algorithm (except for FPT, which is always optimal) has been able to give the optimum in all repetitions. It is observed that the COMPLEX GREEDY algorithm has achieved a slightly better result than the GREEDY ALGORITHM. This is because, in the shortest path, there can be more than one vital edge, and depending on which edge we block, we can influence the result in the long run. But both algorithms still work very well for these types of graphs. On the other hand, there is an improvement in HILL CLIMBING. This version has been able to

handle better with the plateaus, that there are edges that are not blockable has helped and, in a test, has managed to reach the optimum (probably from a very good initial solution). SIMULATED ANNEALING and the GENETIC ALGORITHM have managed to improve their results, especially the time of the GENETIC ALGORITHM, where if we see in the annexes has been imposed less number of generations and therefore less computational work.

Algorithm	$\|SP\|_{\min}$	$\|SP\|_{\text{avg}}$	$\|SP\|_{\max}$	Time[s]	#Opt
GREEDY ALGORITHM	21	60	∞	0.0259	1
COMPLEX GREEDY	21	53	60	0.1890	0
FPT ALGORITHM	∞	∞	∞	352.7984	5
HILL CLIMBING	21	21	21	2.3243	0
SIMULATED ANNEALING	21	42	54	3.4919	0
GENETIC ALGORITHM	54	54	54	119.2117	0

Table 8.12 The small attack graph AD results where not all edges are blockable and contain weighted edges $(w(e) \leq 15)$.

Table 8.12. Version 3:: When applying weights to the edges, it is observed that fewer algorithms have reached the optimum in this case; only the GREEDY ALGORITHM (apart from the FPT) has achieved it. Between the two greedy algorithms, the results continue to be very similar. HILL CLIMBING is still stuck at local maxima, and as expected, SIMULATED ANNEALING gets better results in some tests. On the other hand, the GENETIC ALGORITHM remains constant with its near-optimal result. However, the parameters chosen for this algorithm (maximum population and number of generations) have caused a significant increase in time, unlike the two previous experiments.

8.3.2 Experimentation with a medium AD attack graph

This second experiment compares the different algorithms studied on relatively medium Active Directory attack graphs. The simplified graph contains 605 nodes, 2316 edges, 7 random entry nodes, and a budget b of 15 edges to block. For graphs where all edges are not blockable, we have blocked 500 edges. The blockable edges and the entry nodes are not changed in the five repetitions. In this section, the FPT algorithm has only been repeated once. The algorithm takes much longer to execute as the size and budget have increased. The result will always be optimal since it is an FPT algorithm, and we will only look at the time spent.

Table 8.13. Version 1: We start with a new attack graph more than twice as large as the previous experiment, so we would expect the times to be longer. In this first version, we find

Algorithm	$\|SP\|_{min}$	$\|SP\|_{avg}$	$\|SP\|_{max}$	Time[s]	#Opt
GREEDY ALGORITHM	∞	∞	∞	0.0661	5
COMPLEX GREEDY	14	14	14	1.6637	0
FPT ALGORITHM	∞	∞	∞	80334.3641	5
HILL CLIMBING	4	4	4	5.1561	0
SIMULATED ANNEALING	4	8	8	4.9655	0
GENETIC ALGORITHM	8	8	10	177.4144	0

Table 8.13 The medium AD attack graph results where all edges are blockable and contain unweighted edges.

an oddity; the GREEDY ALGORITHM consistently achieves the optimum, while the other more complex version never reaches the optimum. As explained above, choosing the shortest path may contain different vital edges. In this case, the COMPLEX GREEDY usually blocks an edge that conditions it and makes it unable to reach the optimum. Even so, it achieves an excellent result, almost optimal. As the budget increases and the attack graph is bigger than the previous one, the time of the FPT algorithm scales exponentially. It has gone from running in a few minutes in experiment one to taking hours to run for experiment two.

On the other hand, with the local algorithms, something similar to what happened in version one of the previous experiment happens. HILL CLIMBING does not improve because it encounters many plateaus and cannot improve the result. SIMULATED ANNEALING achieves a better outcome without ever reaching the optimum. The GENETIC ALGORITHM gets the best heuristics results; however, it takes much execution time compared to the other algorithms.

Algorithm	$\|SP\|_{min}$	$\|SP\|_{avg}$	$\|SP\|_{max}$	Time[s]	#Opt
GREEDY ALGORITHM	∞	∞	∞	0.0548	5
COMPLEX GREEDY	10	∞	∞	1.0286	4
FPT ALGORITHM	∞	∞	∞	78744.3228	5
HILL CLIMBING	5	8	10	17.6444	0
SIMULATED ANNEALING	5	10	∞	14.3545	1
GENETIC ALGORITHM	14	∞	∞	120.5418	4

Table 8.14 The medium AD attack graph results where not all edges are blockable and contain unweighted edges.

Table 8.14. Version 2: In this second version of the attack graph. The greedy algorithms have reached the optimum, unlike the previous version, as there are non-blocking edges; the COMPLEX GREEDY algorithm has not been so conditioned. HILL CLIMBING has improved the result without reaching the optimum, but it is still below SIMULATED ANNEALING

in impact and time. Surprisingly the GENETIC ALGORITHM has almost always got the optimum except in one which has achieved an outstanding result. Also, the time is less than in the previous version; however, the time is still high.

Algorithm	$\|SP\|_{\min}$	$\|SP\|_{\text{avg}}$	$\|SP\|_{\max}$	Time[s]	#Opt
GREEDY ALGORITHM	35	78	∞	0.0602	1
COMPLEX GREEDY	68	74	74	1.2823	0
FPT ALGORITHM	∞	∞	∞	90913.3806	5
HILL CLIMBING	37	37	38	17.4699	0
SIMULATED ANNEALING	67	∞	∞	12.5043	4
GENETIC ALGORITHM	∞	∞	∞	68.0134	5

Table 8.15 The medium attack graph AD results where not all edges are blockable and contain weighted edges ($w(e) \leq 15$).

Table 8.15. Version 3:: In this version, surprisingly, the only algorithm that has achieved the optimum (apart from the FPT) is the GENETIC ALGORITHM, even though the time has improved with respect to previous experiments. It turns out that heuristic algorithms perform better than greedy algorithms in this setting. SIMULATED ANNEALING has achieved four times the optimum; however, HILL CLIMBING is still stuck at local maxima, so it does not reach the optimum. Unlike all previous experiments, the greedy algorithms have not achieved a good result, but their time is still very good.

8.3.3 Experimentation with a big AD attack graph

This last experiment compares the different algorithms studied on relatively big Active Directory attack graphs. The simplified graph contains 1808 nodes, 6390 edges, 12 random entry nodes, and a budget b of 20 edges to block. For graphs where all edges are not blockable, we have blocked 650 edges. The blockable edges and the entry nodes are not changed in the five repetitions. We can no longer afford to run the FPT ALGORITHM in this experiment.

Table 8.16. Version 1: With a large AD attack graph, we have expanded the number of edges to block to 20, so we can no longer run the FPT ALGORITHM. The algorithm grows exponentially by the budget, and we can see that in the time of experiment one (b = 10) compared to experiment two ($b = 15$), there is almost a day difference in time.

In this third experiment, we observe a behavior opposite to version 2 of the second experiment in the greedy algorithms. In this case, the COMPLEX GREEDY algorithm achieves

Algorithm	$\|SP\|_{\min}$	$\|SP\|_{\text{avg}}$	$\|SP\|_{\max}$	Time[s]	#Opt
GREEDY ALGORITHM	14	14	14	0.6987	0
COMPLEX GREEDY	∞	∞	∞	8.5967	5
FPT ALGORITHM	-	-	-	-	-
HILL CLIMBING	5	5	5	206.3225	0
SIMULATED ANNEALING	5	5	5	27.9632	0
GENETIC ALGORITHM	7	8	9	653.0331	0

Table 8.16 The big AD attack graph results where all edges are blockable and contain unweighted edges.

the optimum in the five repetitions. In contrast, the GREEDY ALGORITHM remains at the gates of the optimum.

On the other hand, HILL CLIMBING continues not to give the expected results; in this case, even SIMULATED ANNEALING has not been able to surpass the results of HILL CLIMBING. This result is because we have a solution space with many plateaus, so it isn't easy for local algorithms to find good local maxima. However, SIMULATED ANNEALING achieves a much better time than HILL CLIMBING since SIMULATED ANNEALING is performed for a low finite number of iterations. The GENETIC ALGORITHM does not achieve bad results, but the execution time is still prohibitive.

Algorithm	$\|SP\|_{\min}$	$\|SP\|_{\text{avg}}$	$\|SP\|_{\max}$	Time[s]	#Opt
GREEDY ALGORITHM	12	14	14	0.4973	0
COMPLEX GREEDY	12	∞	∞	8.8598	3
FPT ALGORITHM	-	-	-	-	-
HILL CLIMBING	5	5	7	185.2132	0
SIMULATED ANNEALING	5	5	5	8.2025	0
GENETIC ALGORITHM	8	8	9	674.9348	0

Table 8.17 The big AD attack graph results where not all edges are blockable and contain unweighted edges.

Table 8.17. Version 2: The complex greedy algorithm achieves the best results in this second version as in the previous version. However, it is observed that the complex algorithm takes much longer to execute when this algorithm should be somewhat faster in theory. This may be due to two causes: As an AD graph contains few cycles and concise paths, this graph benefits the GREEDY ALGORITHM more. The other cause could be an inappropriate data structure or poor implementation of the Fibonacci heap.

On the other hand, surprisingly, HILL CLIMBING has performed better in this version than in all iterations of SIMULATED ANNEALING. But the time is still much longer. Per-

haps applying more repetitions in SIMULATED ANNEALING would have achieved better results than HILL CLIMBING. The GENETIC ALGORITHM is still the third-best-performing algorithm, but the time is still bad.

Algorithm	$\|SP\|_{\min}$	$\|SP\|_{\text{avg}}$	$\|SP\|_{\max}$	Time[s]	#Win
GREEDY ALGORITHM	78	82	82	0.3714	4
COMPLEX GREEDY	78	82	82	4.4829	4
FPT ALGORITHM	-	-	-	-	-
HILL CLIMBING	53	53	53	272.9882	0
SIMULATED ANNEALING	53	61	63	53.0645	0
GENETIC ALGORITHM	78	82	82	687.1644	4

Table 8.18 The big attack graph AD results where not all edges are blockable and contain weighted edges $(w(e) \leq 15)$.

Table 8.18. Version 3::In the last experiment, it is observed that no algorithm has succeeded in blocking b edges such that there is no path $SP_{G-B}(s_i, DC)$. Therefore, as we do not know the optimal result since it is unfeasible to calculate it with FPT, it has been decided to change #Opt by #Win, which indicates the number of times that the highest result of the experiment has been reached. This version shows that GREEDY and GENETIC ALGORITHM have reached the highest mark in four repetitions. The ratio of times is very similar to those of previous experiments (greedy high-speed, genetic algorithm very slow). SIMULATED ANNEALING finally achieved better results and time than HILL CLIMBING in this experiment.

Chapter 9

Final words

9.1 Conclusion

We studied edge blocking for defending Active Directory-style attack graphs—specifically, different versions for solving the problem of finding the b most vital edges.

First of all, we studied the problem theoretically. We observed that the problem is computationally difficult. To begin, we investigated the problem of blocking only the most vital edge of the attack graph, where two algorithms have been formulated. In the end, we proposed six algorithms to solve the b most vital edges problem.

Secondly, we made an experimental study of the problem for different sizes of AD attack graphs. For each attack graph size, three different versions have been experimented with. In the first version, the attack graph was unweighted. In the second version, there were edges we could not block; in the last version, not all edges were blockable in a weighted attack graph.

The FPT ALGORITHM consistently achieves the optimal result but can only be applied when the budget is small. The greedy algorithms have been found to give the best results in all versions. This event is because Active Directory graphs are very similar to a tree graph, and almost no substitutable blockable-worthy edges are found (See 6.3). However, the GREEDY ALGORITHM managed to run the algorithm with less time than the COMPLEX ALGORITHM when we thought the complex would be faster according to the theoretical part. This discrepancy may be due to two reasons: As an AD graph contains few cycles and concise paths, this graph benefits the GREEDY ALGORITHM more. The other most likely cause is that the data could not be adequately structured and may have been due to poor implementation of the Fibonacci heap.

On the other hand, heuristic algorithms have not been as good as expected. In the HILL CLIMBING algorithm in versions one and two, being unweighted graphs, many short paths are found, which causes a lot of plateau in the solution space generating the algorithm to stay in not very good local maxima. However, as expected, SIMULATED ANNEALING has been shown to perform better than HILL CLIMBING and shorter times. Finally, the GENETIC ALGORITHM has performed exceptionally well in all the experiments. The problem is that it has required a lot of population and generations, causing the execution time to be very high.

9.2 Future work

We could extend this work by adding more algorithms, new model description strategies, and more descriptive parameters. We could also generate a graphical interface such as BloodHound, where we can view the attack graph with its shortest paths and most vital edges. A future option would be to add these algorithms to the new open-source tool called BlueHound, which helps blue teams identify security issues (vulnerabilities) and uses attack graphs like BloodHound.

Although this work was oriented to the study of Active Directory graphs, it would be interesting to observe the behavior of the algorithms in completely different graphs, where perhaps, greedy algorithms would not perform so well. Another possible study would be that each edge contains a blocking cost, so there will be expensive edges to block, and the defender will have to think better about which edges to block. Perhaps blocking a high-cost edge gets a better result than securing five lower-cost edges.

Finally, this version has studied the problem starting from a series of input nodes compromised by the attacker. From this information, the defender blocks the most vital edges to maximize the shortest path of the attacker. Another study version could be that the attacker has not yet compromised any node, but the defender knows that there are potentially vulnerable computers or users (entry nodes). At this point, the defender decides to block a series of edges to prevent the minimum damage if an attacker manages to infiltrate these entry nodes. In this situation of uncertainty, the smartest thing to do would be to block those edges that minimize the attacker's success, a more focused study on average. An example of such a study is the one done in the paper by Mingyu Go et al. [18].

References

[1] John Lambert. Defenders think in lists. Attackers think in graphs. As long as this is true, attackers win. https://docs.microsoft.com/es-es/archive/blogs/johnla/defenders-think-in-lists-attackers-think-in-graphs-as-long-as-this-is-true-attackers-win.

[2] Active Directory Domain Overview. https://docs.microsoft.com/en-us/windows-server/identity/ad-ds/get-started/virtual-dc/active-directory-domain-services-overview.

[3] Per Christensson. Active Directory Definition. https://techterms.com/definition/active_directory, 2017.

[4] Rohan Vazarkar, Will Schroeder, and Andrew Robbins. Bloodhound: Six degrees of domain admin. *BloodHoundAD*, 2016. https://github.com/BloodHoundAD/BloodHound.

[5] BloodHound. SharpHound. *Collectors*. https://github.com/BloodHoundAD/SharpHound.

[6] BloodHound. DBCreator. *BloodHound-Tools*. https://github.com/BloodHoundAD/BloodHound-Tools/tree/master/DBCreator.

[7] John Dunagan, Alice X Zheng, and Daniel R Simon. Heat-ray: Combating identity snowball attacks using machinelearning, combinatorial optimization and attack graphs. *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 305–320, 2009.

[8] Zero Networks. BlueHound. 2022. https://github.com/zeronetworks/BlueHound.

[9] Baruch Schieber, Amotz Bar-Noy, and Samir Khuller. The complexity of finding most vital arcs and nodes. *Technical Reports from UMIACS*, 1995.

[10] André, Niedermeier Rolf Bazgan Cristina, and Nichterlein. A refined complexity analysis of finding the most vital edges for undirected shortest paths. *Algorithms and Complexity*, pages 47–60, 2015.

[11] Enrico Nardelli, Guido Proietti, and Peter Widmayer. Finding the most vital node of a shortest path. *Theoretical Computer Science*, 296:167–177, 2003.

[12] Georg Baier, Thomas Erlebach, Alexander Hall, Ekkehard Köhler, Petr Kolman, Ondřej Pangrác, Heiko Schilling, and Martin Skutella. Length-bounded cuts and flows. *ACM Trans. Algorithms*, 7, 2010.

[13] Pavel Dvořák and Dušan Knop. Parameterized complexity of length-bounded cuts and multi-cuts. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9076:441–452, 2015.

[14] Petr A. Golovach and Dimitrios M. Thilikos. Paths of bounded length and their cuts: Parameterized complexity and algorithms. *Discrete Optimization*, 8:72–86, 2011.

[15] Yash Raj Guo Jiong and Shrestha. Parameterized complexity of edge interdiction problems. *Computing and Combinatorics*, pages 166–178, 2014.

[16] Michael O Ball, Bruce L Golden, and Rakesh V Vohra. Finding the most vital arcs in a network. *Operations Research Letters*, 8:73–76, 1989.

[17] Cristina Bazgan, Till Fluschnik, André Nichterlein, Rolf Niedermeier, and Maximilian Stahlberg. A more fine-grained complexity analysis of finding the most vital edges for undirected shortest paths. *Networks*, 73:23–37, 2019.

[18] Mingyu Guo, Jialiang Li, Aneta Neumann, Frank Neumann, and Hung Nguyen. Practical fixed-parameter algorithms for defending active directory style attack graphs. *arXiv*, 2021.

[19] Glassdoor. https://www.glassdoor.es/index.htm.

[20] Tarifaluzhora. https://tarifaluzhora.es/.

[21] D.S. Johnson M. Garey. Computers and intractability: A guide to the theory of np-completeness. *W.H: Freeman, San Francisco, CA*, 1979.

[22] R. Endre Tarjan. A note on finding the bridges of a graph. *Information Processing Letters*, 2:160–161, 4 1974.

[23] K Malik, A K Mittal, and S K Gupta. The k most vital arcs in the shortest path problem. *Operations Research Letters*, 8:223–227, 1989.

[24] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, jul 1987.

[25] Neo4j. Neo4j Documentation. 2007. https://neo4j.com/docs/.

Appendix A

Handle with DBCreator data

DBCreator

DBCreator consists of a python script that generates random data simulating an Active Directory environment. It is often used for testing BloodHound features and analysis. DBCreator is the tool that has been used to create an attack graph topology for the investigation of this project. Once we have started and authenticated our database with neo4j, we proceed to run DBcreator.exe in our terminal. Figure A.1 shows the commands available to create the attack graph.

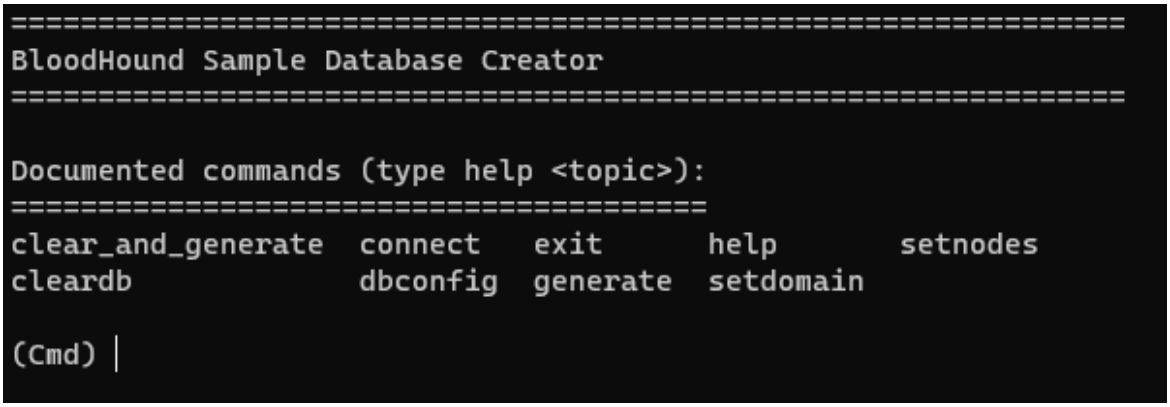

Fig. A.1 DBCreator console display.

Commands

Here is a brief description of each command: `dbconfig` - DB configuration command. Set the credentials and URL for the database we want to connect. `connect` - Connect to the

established database. `setnodes` - Set the number of nodes to generate. `setdomain` - Set the domain name. `cleardb` - Clears the database. `generate` - Once we have specified how we want the graph, it generates random data in the database.

Import the attack graph into our program

Once we have the data created by DBCreator in our database, we proceed to import it into our program. The figure A.2 below shows a small example of an Active Directory environment created by DBCreator visualized with neo4j. Users are shown in pink, computers in green, groups in orange, and the Domain Controller in blue.

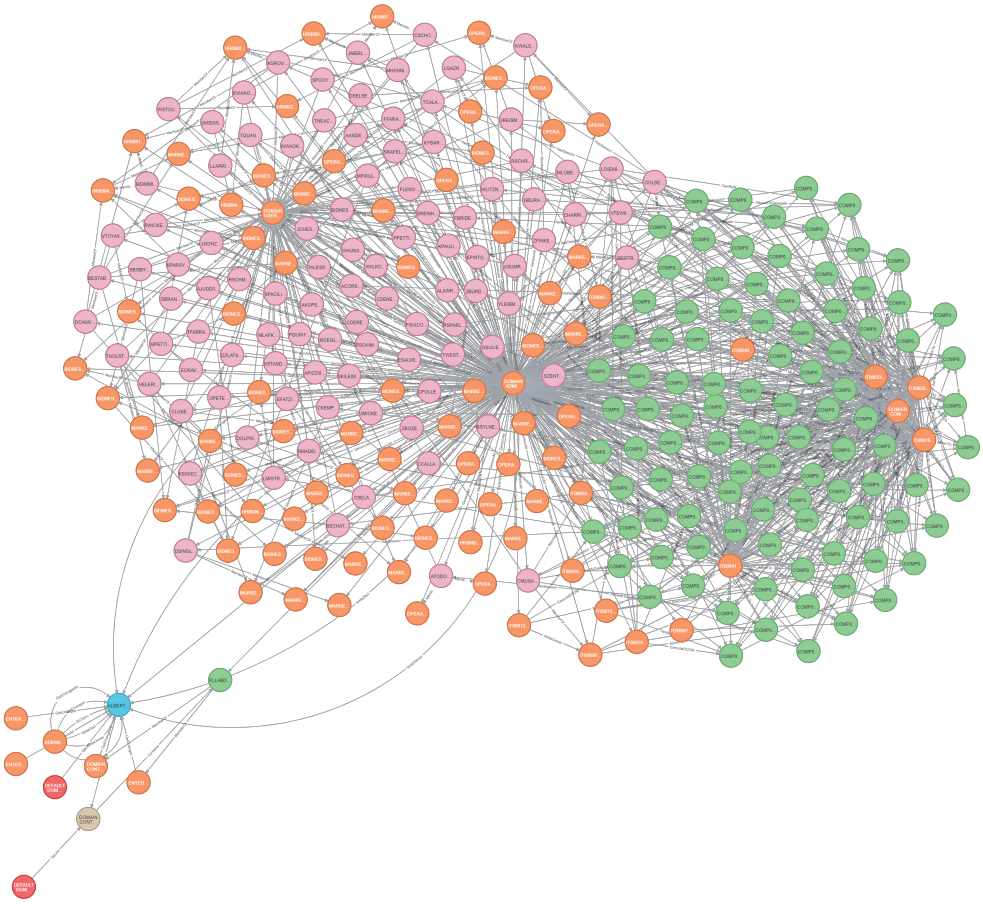

Fig. A.2 DBCreator graph representation using neo4j.

The neo4j database has a console available where we can use command consoles. Using a query that returns all the contents of the graph, we observe that the result contains the following structure for the nodes and relations:
The nodes are stored as follows; here is an example:

```
<Node id=987 labels=frozenset({'Base', 'Computer'})
    properties={'name': 'COMP00018.ALBERT.TFG', '
    operatingsystem': 'Windows Server 2008', 'objectid': 'S
    -1-5-21-883232822-274137685-4173207997-1017', 'enabled':
    True}>) type='AdminTo' properties={}>
```

We see that this node is a computer that contains a 2008 windows server and has privileges concerning other nodes since it is of type AdminTo.
On the other hand, the relationships have this form:

```
<Relationship id=3438 nodes= ...
```

The ellipses contain a list of the nodes that creates this relationship.

Python script

To finish with the import of the attack graph, we have created a python script to make the graph adapt to our program. We should note that the entire project uses the Python graph package, NetworkX. From the neo4j documentation [25], the script first connects to the database, then executes a query to return all the data, and iterating; we filter the properties of the nodes we are interested in (id, type, DC?) and the relationships between nodes. Once collected all the information, we will only keep the edges of type: ADMINTO, MEMBEROF, HASSESSION since these edges are the default of BloodHound. Once the attack graph is complete, we will optimize it by eliminating all those nodes that do not exist a path to the Domain Controller. Also, if there is more than one administrative node, we merge them into the DC.

Appendix B

Parameter selection tables

This appendix shows all the parameter justification tables for all the studied attack graphs and their respective versions. For a better understanding of the criteria for selecting parameters, see chapter 8. The selected parameter is in bold.

Graph of 280 nodes and 850 edges

This section justifies the choice of parameters for an attack graph of 280 nodes, 850 edges, 5 entry nodes, and a budget b of 10 edges to block. For versions 2 and 3: 150 edges are not blockable. For version 3: edges contain a weight with a maximum of 15.

Version 1: All edges are blockable, and edges have no weight.

N	$\|SP\|_{\mathbf{min}}$	$\|SP\|_{\mathbf{avg}}$	$\|SP\|_{\mathbf{max}}$	$\mathbf{Time[s]}_{\mathbf{avg}}$
500	2	4	4	0.0160
2500	5	6	∞	0.6491
5000	2	8	∞	0.4408
10000	37	38	∞	1.1779
15000	2	∞	∞	0.5301
20000	4	∞	∞	1.3711
25000	**8**	∞	∞	**3.2596**
50000	8	∞	∞	8.2317

Table B.1 Determination of the number of iterations in Simulated Annealing

Temperature	$\|SP\|_{\mathbf{min}}$	$\|SP\|_{\mathbf{avg}}$	$\|SP\|_{\mathbf{max}}$	**Time[s]$_{\mathbf{avg}}$**
1	4	4	∞	5.4769
25	6	7	∞	6.6080
50	4	6	7	5.4181
100	∞	∞	∞	**5.4181**
150	7	∞	∞	5.7589
200	7	7	∞	5.4474
300	6	7	∞	6.1994

Table B.2 Temperature determination in Simulated Annealing

Table B.3 Initial and maximum population of the genetic algorithm

(a)

$N_{\mathbf{ini}}$	$\|SP\|$	**Time[s]**
50	∞	**32.2186**
100	∞	54.4510
250	∞	94.2197
500	∞	39.2583
1000	∞	70.6713

(b)

N	$\|SP\|$	**Time[s]**
250	9	12.7621
500	∞	**15.3575**
1000	∞	32.9339
2500	∞	20.6026
5000	∞	53.0855

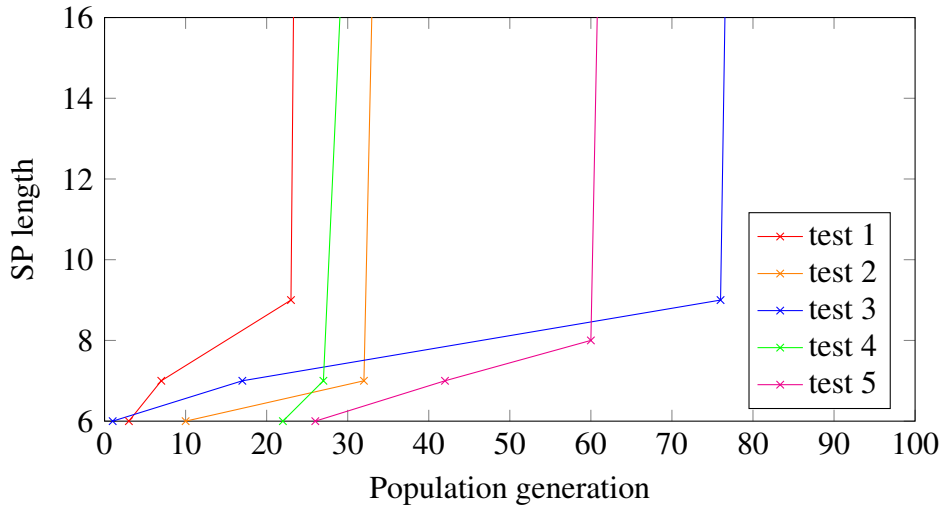

Table B.4 Determination of the number of iterations of the genetic algorithm

Table B.5 Mutation and crossover rate

(a)

Mutation rate	$\|SP\|$	**Time[s]**
0.1	∞	15.8069
0.2	∞	**6.5533**
0.3	∞	9.3660
0.4	7	21.7615

(b)

Crossover rate	$\|SP\|$	**Time[s]**
0.6	∞	14.2889
0.7	∞	**7.7961**
0.8	∞	8.1753
0.9	∞	14.4782

Version 2: Not all edges are blockable, and edges have no weight

N	$\|SP\|_{\mathbf{min}}$	$\|SP\|_{\mathbf{avg}}$	$\|SP\|_{\mathbf{max}}$	**Time[s]$_{\mathbf{avg}}$**
500	2	2	4	0.0529
2500	2	4	5	0.2733
5000	2	5	8	0.9194
10000	2	5	∞	1.8810
15000	2	5	∞	2.3910
20000	2	8	∞	3.8959
25000	**4**	∞	∞	**5.0160**
50000	5	∞	∞	12.2027

Table B.6 Determination of the number of iterations in Simulated Annealing

Temperature	$\|SP\|_{\mathbf{min}}$	$\|SP\|_{\mathbf{avg}}$	$\|SP\|_{\mathbf{max}}$	**Time[s]$_{\mathbf{avg}}$**
1	5	∞	∞	1.4366
25	5	∞	∞	3.4478
50	2	∞	7	2.8665
100	∞	∞	∞	**2.9566**
150	∞	∞	∞	4.7969
200	∞	∞	∞	4.3452
300	2	∞	∞	3.8729

Table B.7 Temperature determination in Simulated Annealing

Table B.8 Initial and maximum population of the genetic algorithm

(a)

$N_{\mathbf{ini}}$	$\|SP\|$	**Time[s]**
50	∞	36.4552
100	∞	59.5460
250	∞	41.0704
500	∞	**28.8872**
1000	∞	57.4960

(b)

N	$\|SP\|$	**Time[s]**
500	∞	**4.3472**
1000	∞	9.7241
2000	∞	30.065
3000	∞	32.1817
5000	∞	60.3846

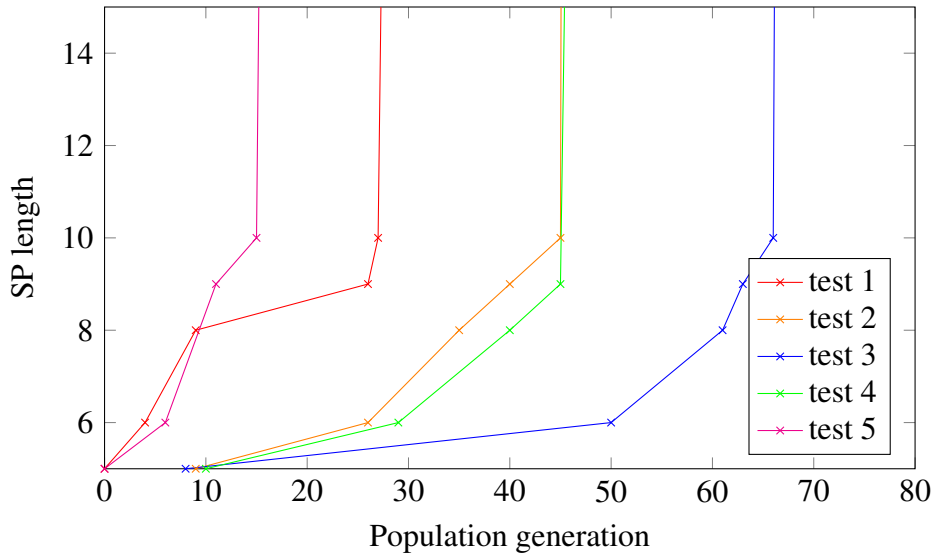

Table B.9 Determination of the number of iterations of the genetic algorithm

Table B.10 Mutation and crossover rate

(a)

Mutation rate	$\|SP\|$	**Time[s]**
0.1	∞	8.4325
0.2	∞	**7.6395**
0.3	9	14.1891
0.4	6	14.6700

(b)

Crossover rate	$\|SP\|$	**Time[s]**
0.6	∞	6.1945
0.7	∞	**5.3535**
0.8	∞	5.6043
0.9	∞	9.3402

Version 3: Not all edges are blockable, and edges contain weight

N	$\|SP\|_{\text{min}}$	$\|SP\|_{\text{avg}}$	$\|SP\|_{\text{max}}$	**Time[s]$_{\text{avg}}$**
500	18	21	35	0.1397
2500	14	41	∞	0.3921
5000	30	60	∞	0.6232
10000	45	∞	∞	1.1882
15000	**59**	∞	∞	**1.4543**
20000	41	∞	∞	2.2273
25000	41	∞	∞	3.0122
50000	59	∞	∞	8.0023

Table B.11 Determination of the number of iterations in Simulated Annealing

Temperature	$\|SP\|_{\text{min}}$	$\|SP\|_{\text{avg}}$	$\|SP\|_{\text{max}}$	**Time[s]$_{\text{avg}}$**
1	21	21	54	4.9098
25	21	50	54	3.9941
50	21	30	54	3.7174
100	21	21	54	4.5541
150	21	37	54	3.9920
200	**37**	**54**	**54**	**3.1957**
300	21	42	53	3.7892

Table B.12 Temperature determination in Simulated Annealing

Table B.13 Initial and maximum population of the genetic algorithm

(a)

N_{ini}	$\|SP\|$	**Time[s]**
50	37	202.9619
100	**42**	**203.4066**
250	37	211.4207
500	37	217.2348
1000	37	195.9877

(b)

N	$\|SP\|$	**Time[s]**
1000	37	41.6799
2000	**49**	**82.0874**
3000	37	125.6158
5000	42	193.3331
10000	42	380.3287

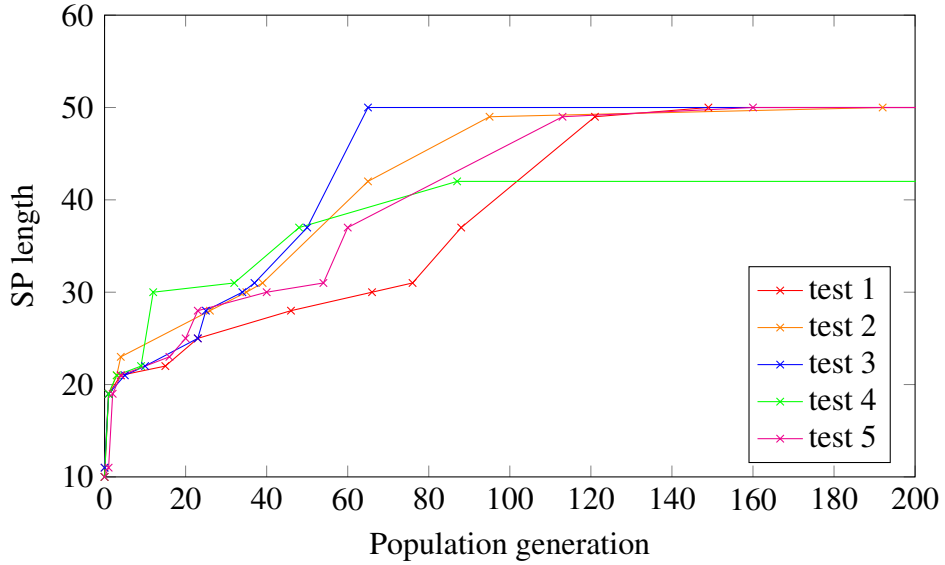

Table B.14 Determination of the number of iterations of the genetic algorithm

Table B.15 Mutation and crossover rate

(a)

Mutation rate	$\|SP\|$	**Time[s]**
0.1	**54**	**140.2869**
0.2	50	168.0182
0.3	30	226.9280
0.4	22	233.5391

(b)

Crossover rate	$\|SP\|$	**Time[s]**
0.6	**54**	**142.6651**
0.7	54	143.1758
0.8	54	147.0654
0.9	54	154.7707

Graph of 605 nodes and 2315 edges

This section justifies the parameters for an attack graph of 605 nodes, 2315 edges, 7 entry nodes, and a budget b of 15 edges to block. For version 2: 500 edges are not blockable. For version 3: see chapter 8.

Version 1: All edges are blockable, and edges have no weight

N	$\|SP\|_{\text{min}}$	$\|SP\|_{\text{avg}}$	$\|SP\|_{\text{max}}$	**Time[s]$_{\text{avg}}$**
500	2	2	2	0.0309
2500	2	2	4	0.4169
5000	2	4	4	0.8971
10000	4	4	4	1.8443
15000	4	4	8	5.4017
20000	4	4	8	6.1443
25000	**4**	**8**	**8**	**10.1561**
50000	4	4	10	24.6075

Table B.16 Determination of the number of iterations in Simulated Annealing

Temperature	$\|SP\|_{\text{min}}$	$\|SP\|_{\text{avg}}$	$\|SP\|_{\text{max}}$	**Time[s]$_{\text{avg}}$**
1	4	4	8	5.4559
25	4	4	7	4.1886
50	4	4	8	5.2759
100	**4**	**8**	**8**	**8.8079**
150	4	4	8	8.9531
200	4	4	4	3.3487
300	8	8	8	4.5077

Table B.17 Temperature determination in Simulated Annealing

Table B.18 Initial and maximum population of the genetic algorithm

(a)

N_{ini}	$\|SP\|$	**Time[s]**
50	8	503.7324
100	8	523.4037
250	**10**	**522.8672**
500	8	516.6230
1000	8	497.0724

(b)

N	$\|SP\|$	**Time[s]**
1000	**8**	**90.9157**
2000	8	187.7226
3000	8	294.3626
5000	8	453.8934
10000	8	953.5680

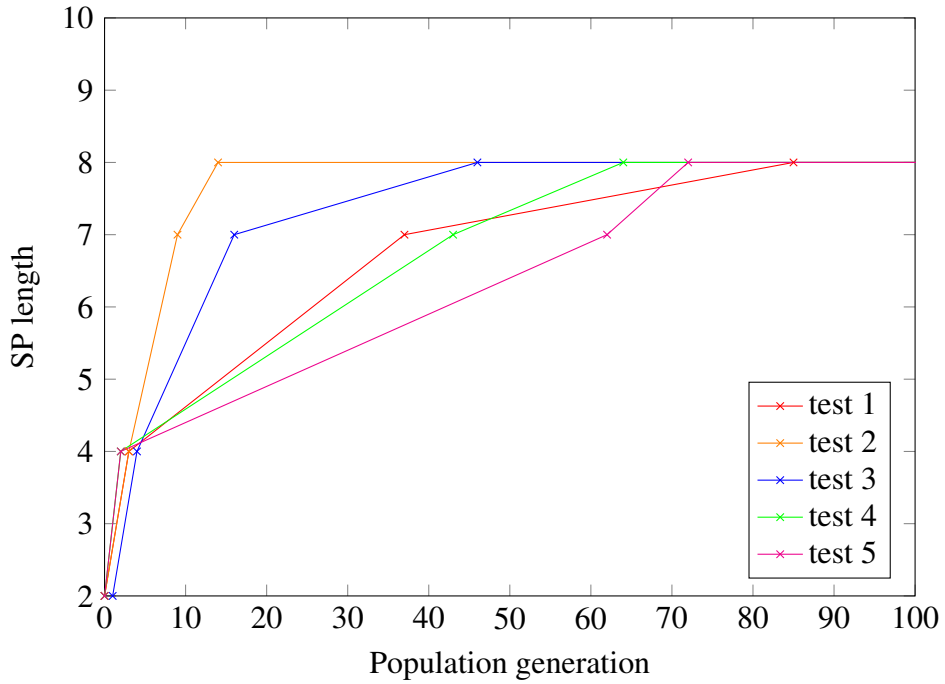

Table B.19 Determination of the number of iterations of the genetic algorithm

Table B.20 Mutation and crossover rate

(a)

Mutation rate	$\|SP\|$	**Time[s]**
0.1	**8**	**75.1658**
0.2	8	96.7004
0.3	8	104.8896
0.4	7	125.1525

(b)

Crossover rate	$\|SP\|$	**Time[s]**
0.6	**8**	**77.9593**
0.7	8	102.9991
0.8	8	112.3581
0.9	7	131.1609

Version 2: Not all edges are blockable, and edges have no weight

N	$\|SP\|_{\text{min}}$	$\|SP\|_{\text{avg}}$	$\|SP\|_{\text{max}}$	**Time[s]$_{\text{avg}}$**
500	2	3	4	0.2026
2500	3	3	5	0.3289
5000	3	4	5	1.7863
10000	3	7	9	4.7788
15000	3	8	8	8.5310
20000	3	8	9	6.1443
25000	**6**	**8**	**8**	**14.1488**
50000	4	7	∞	20.1665

Table B.21 Determination of the number of iterations in Simulated Annealing

Temperature	$\|SP\|_{\text{min}}$	$\|SP\|_{\text{avg}}$	$\|SP\|_{\text{max}}$	**Time[s]$_{\text{avg}}$**
1	1	2	5	0.0261
25	2	2	5	0.6602
50	5	8	10	10.6105
100	5	5	∞	13.5580
150	5	10	11	12.8404
200	**5**	**10**	∞	**12.5966**
300	5	8	∞	12.4427

Table B.22 Temperature determination in Simulated Annealing

Table B.23 Initial and maximum population of the genetic algorithm

(a)

N_{ini}	$\|SP\|$	**Time[s]**
50	12	460.5475
100	**12**	**454.1463**
250	10	464.1610
500	12	455.6477
1000	11	497.0426

(b)

N	$\|SP\|$	**Time[s]**
1000	**11**	**93.2333**
2000	9	190.3044
3000	11	272.7374
5000	12	462.0874
10000	11	894.9110

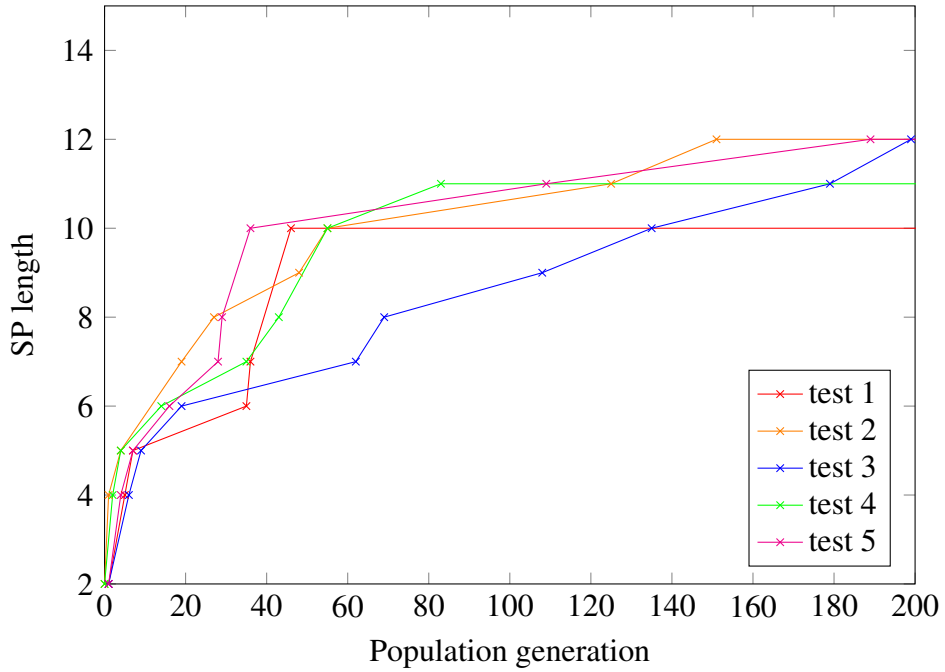

Table B.24 Determination of the number of iterations of the genetic algorithm

Table B.25 Mutation and crossover rate

(a)

Mutation rate	$\|SP\|$	**Time[s]**
0.1	∞	**44.1547**
0.2	11	166.7264
0.3	8	218.5639
0.4	6	234.2360

(b)

Crossover rate	$\|SP\|$	**Time[s]**
0.6	∞	**99.5938**
0.7	∞	140.6951
0.8	10	166.7659
0.9	∞	133.0485

Graph of 1808 nodes and 6390 edges

This section justifies the choice of parameters for an attack graph of 1808 nodes, 6390 edges, 12 entry nodes, and a budget b of 20 edges to block. For versions 2 and 3: 650 edges are not blockable. For version 3: edges contain a weight with a maximum of 15.

Version 1: All edges are blockable, and edges have no weight

N	$\|SP\|_{\text{min}}$	$\|SP\|_{\text{avg}}$	$\|SP\|_{\text{max}}$	**Time[s]$_{\text{avg}}$**
500	3	5	5	0.4169
2500	3	5	5	0.9134
5000	4	5	7	3.5465
10000	4	7	7	28.4919
15000	**5**	**7**	**7**	**42.4495**
20000	5	7	7	51.2079
25000	5	7	7	77.6580
50000	5	7	7	153.0311

Table B.26 Determination of the number of iterations in Simulated Annealing

Temperature	$\|SP\|_{\text{min}}$	$\|SP\|_{\text{avg}}$	$\|SP\|_{\text{max}}$	**Time[s]$_{\text{avg}}$**
1	7	2	5	0.0261
25	2	2	5	0.6602
50	5	8	10	10.6105
100	5	5	∞	13.5580
150	5	10	11	12.8404
200	5	10	11	12.8404
300	5	8	∞	12.4427

Table B.27 Temperature determination in Simulated Annealing

Table B.28 Initial and maximum population of the genetic algorithm

(a)

N_{ini}	$\|SP\|$	**Time[s]**
50	**8**	**1863.3750**
100	8	1904.2818
250	8	1900.9318
500	8	1930.7742
1000	8	1931.2976

(b)

N	$\|SP\|$	**Time[s]**
1000	**8**	**186.1749**
2000	7	376.7624
3000	8	762.1881
5000	8	1132.3904
10000	8	1878.8279

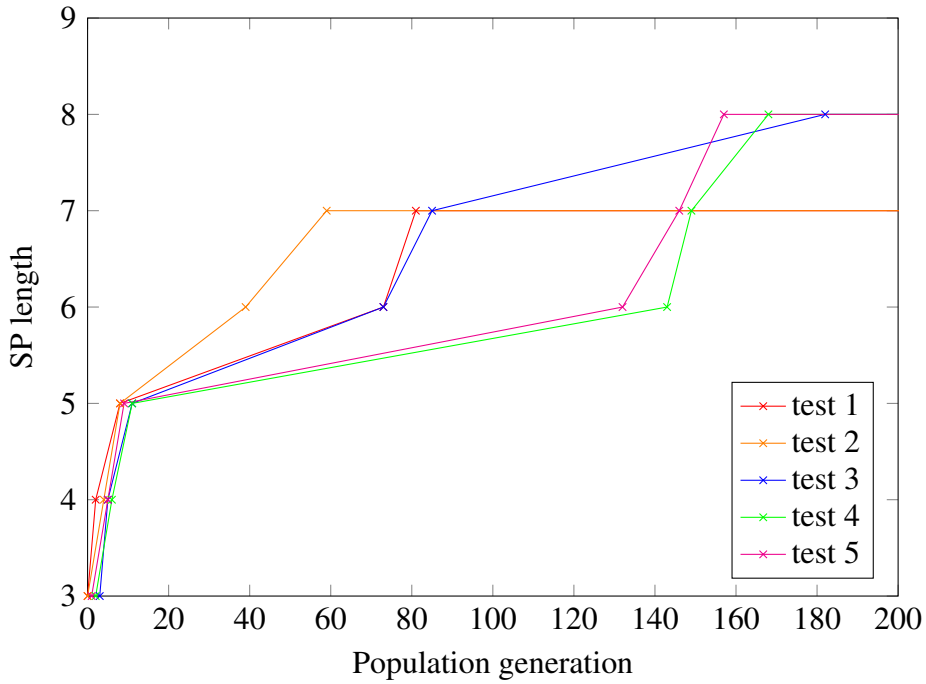

Table B.29 Determination of the number of iterations of the genetic algorithm

Table B.30 Mutation and crossover rate

(a)

Mutation rate	$\|SP\|$	**Time[s]**
0.1	**8**	**305.0469**
0.2	7	377.8842
0.3	6	454.2839
0.4	6	535.5353

(b)

Crossover rate	$\|SP\|$	**Time[s]**
0.6	**9**	**321.5169**
0.7	7	320.7297
0.8	6	293.3425
0.9	8	348.3359

Version 2: Not all edges are blockable, and edges have no weight

N	$\|SP\|_{\text{min}}$	$\|SP\|_{\text{avg}}$	$\|SP\|_{\text{max}}$	**Time[s]$_{\text{avg}}$**
500	3	4	4	0.1874
2500	3	4	4	0.4991
5000	3	4	5	0.7874
10000	4	5	5	7.9814
15000	**5**	**5**	**7**	**19.3729**
20000	4	5	7	49.1790
25000	4	5	7	62.4890
50000	4	7	7	125.6599

Table B.31 Determination of the number of iterations in Simulated Annealing

Temperature	$\|SP\|_{\text{min}}$	$\|SP\|_{\text{avg}}$	$\|SP\|_{\text{max}}$	**Time[s]$_{\text{avg}}$**
1	5	5	5	61.9058
25	5	5	7	65.5495
50	4	5	7	63.0062
100	5	5	5	62.3408
150	5	5	5	61.8128
200	5	5	5	61.0004
300	**5**	**7**	**7**	**60.5375**

Table B.32 Temperature determination in Simulated Annealing

Table B.33 Initial and maximum population of the genetic algorithm

(a)

N_{ini}	$\|SP\|$	**Time[s]**
50	8	1805.3773
100	**8**	**1802.6214**
250	8	1840.4259
500	8	1828.4445
1000	8	1841.0294

(b)

N	$\|SP\|$	**Time[s]**
1000	7	380.4275
2000	**8**	**798.3553**
3000	8	1194.0691
5000	8	1981.8827
10000	8	3908.6475

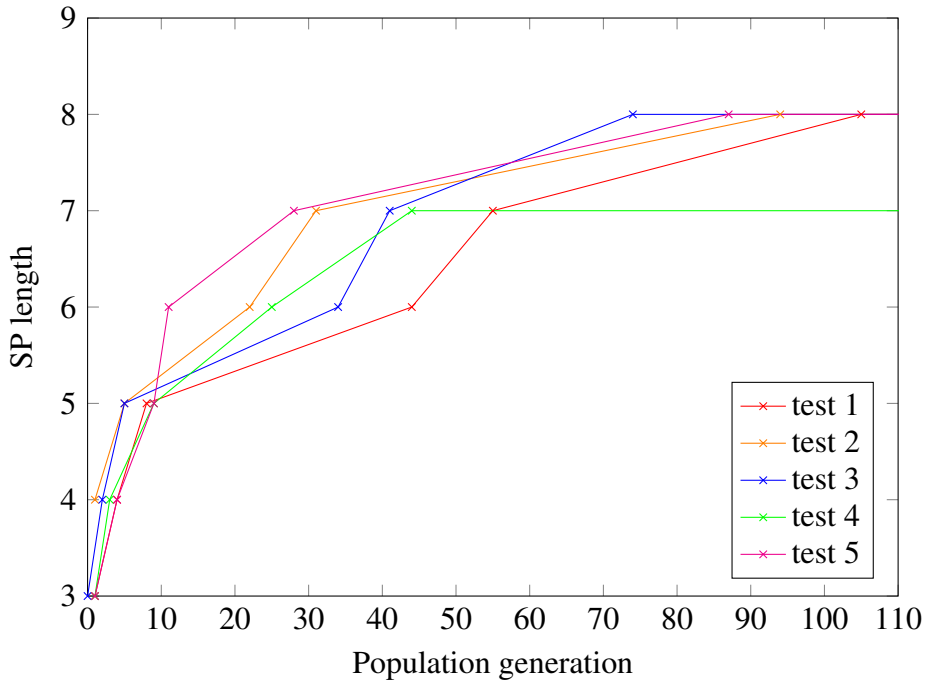

Table B.34 Determination of the number of iterations of the genetic algorithm

Table B.35 Mutation and crossover rate

(a)

Mutation rate	$\|SP\|$	**Time[s]**
0.1	**8**	**765.2727**
0.2	8	862.1863
0.3	7	1048.2778
0.4	6	1269.8225

(b)

Crossover rate	$\|SP\|$	**Time[s]**
0.6	**9**	**321.5169**
0.7	7	320.7297
0.8	6	293.3425
0.9	8	348.3359

Version 3: Not all edges are blockable, and edges contain weight

N	$\|SP\|_{\text{min}}$	$\|SP\|_{\text{avg}}$	$\|SP\|_{\text{max}}$	**Time[s]$_{\text{avg}}$**
500	9	27	42	1.2135
2500	9	31	42	2.9261
5000	9	37	45	8.3453
10000	9	42	43	16.8698
15000	25	41	51	23.1075
20000	21	53	57	49.7552
25000	**37**	**55**	**58**	**61.6041**
50000	40	55	55	120.3975

Table B.36 Determination of the number of iterations in Simulated Annealing

Temperature	$\|SP\|_{\text{min}}$	$\|SP\|_{\text{avg}}$	$\|SP\|_{\text{max}}$	**Time[s]$_{\text{avg}}$**
1	43	53	58	73.8345
25	51	57	63	72.7002
50	55	58	63	78.2895
100	50	58	63	73.2655
150	53	53	63	74.8848
200	**53**	**63**	**63**	**61.7822**
300	51	53	56	71.7075

Table B.37 Temperature determination in Simulated Annealing

Table B.38 Initial and maximum population of the genetic algorithm

(a)

N_{ini}	$\|SP\|$	**Time[s]**
50	53	2494.8181
100	53	2553.8511
250	**58**	**3263.9850**
500	57	3466.9833
1000	57	2492.9392

(b)

N	$\|SP\|$	**Time[s]**
1000	53	459.9254
2000	**56**	**908.8167**
3000	53	1356.3812
5000	53	2150.8860
10000	56	3812.8331

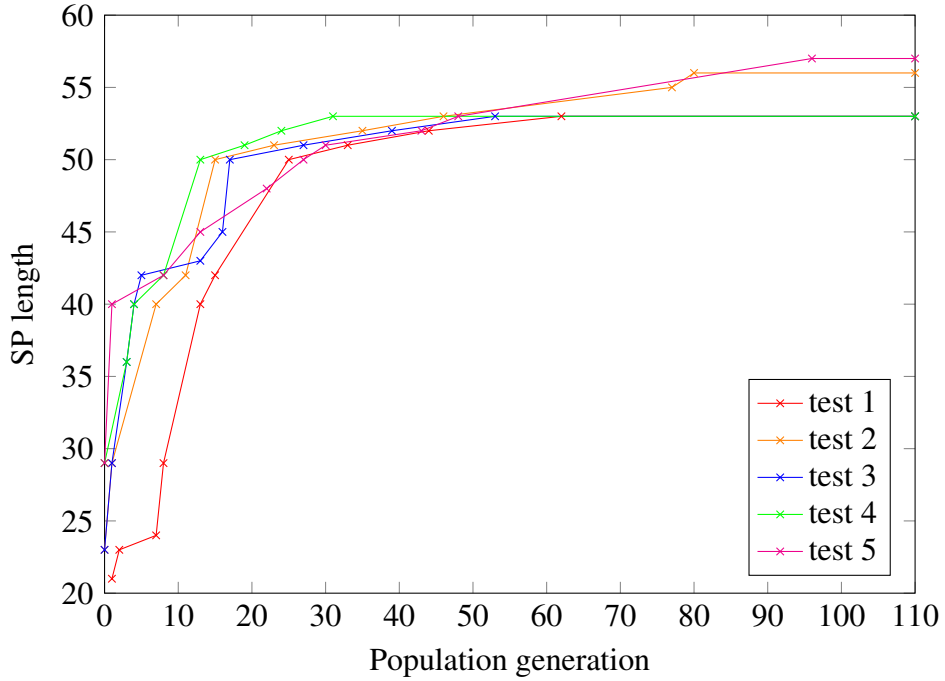

Table B.39 Determination of the number of iterations of the genetic algorithm

Table B.40 Mutation and crossover rate

(a)

Mutation rate	$\|SP\|$	**Time[s]**
0.1	**64**	**627.1817**
0.2	53	845.1675
0.3	51	1050.7880
0.4	45	1284.3395

(b)

Crossover rate	$\|SP\|$	**Time[s]**
0.6	**63**	**734.9426**
0.7	63	779.6364
0.8	63	812.8000
0.9	56	833.7995