Dissertation Type: enterprise

DEPARTMENT OF COMPUTER SCIENCE

Attack Tree Teaching Tool

James Wickenden

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

Thursday 20$^{\text{th}}$ May, 2021

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of MEng in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

This project fits within the scope of ethics application 97842, as reviewed by my supervisor, Matthew Edwards.

James Wickenden, Thursday 20th May, 2021

Contents

Executive Summary

Abstract

In a 1998 paper, Bruce Schneier proposed a method of modelling threats against a system by creating a diagram of ways in which somebody could theoretically attack that system. Schneier called these diagrams *attack trees*, and today they are used in one of the most common threat modelling methodologies. Several commercial tools for building attack trees exist, which primarily allow businesses to perform detailed risk analysis on complex systems. Other less powerful open-source tools also exist.

For the sake of learning about threat modelling and risk assessment, however, there is a lack of accessible, easy to pick up tools that allow users to build and experiment with attack trees. This paper presents Atree, a web tool built for this purpose. Atree provides value to students through accessibility of the tool and its features; it also supports real-time collaborative work between a group of students, allowing them to design, build, and modify attack trees as a group. Atree is lightweight, currently without an external database, and can therefore be deployed quickly and easily for complement lectures and labs, or be hosted locally to support offline learning. In either case, trees can then be downloaded and uploaded to allow for work across sessions.

Summary of Achievements

- I spent two weeks comparing and extensively reviewing 14 different threat modelling and diagramming tools to develop a list of criteria.
- I built an online website tool using the Node.JS framework and building sourced from the powerful mxgraph diagramming library. In total, this tool consists of over 2400 lines of source code across the JS stack.
- I implemented real-time client communicative development using the socket.io library to interact between clients and a central server.
- I spent two weeks researching the CRDT algorithm for distributed computing, and contrasting it with my own approach.
- I assessed the performance of this tool by testing it with users who had experience creating attack trees, as well as those unfamiliar with diagramming tools.

Supporting Technologies

Several third-party resources and libraries were used over the course of this project, which are outlined here.

- The Node.JS framework was used in conjunction with Express.JS to provide a http server instance.
  https://nodejs.org/en/
  https://expressjs.com/
- The *mxgraph 4.2.2* library was used in its JavaScript implementation to handle storing and rendering the tree model.
  https://jgraph.github.io/mxgraph/
- I used the *socket.io 4.0* library to allow for client-server messaging.
  https://socket.io/
- I deployed and tested live versions of my tool on a Heroku instance with Git integration for build testing.
  https://www.heroku.com/home

Acknowledgements

In the course of this project I have received a great deal of support and assistance. I would like to thank my supervisor, Dr Matthew Edwards, for his knowledge and guidance at every stage.

I would also like to thank my sister Alice for proofreading and writing advice, and to my friends for their constant support.

Chapter 1

Contextual Background

1.1 Threat Models

The early days of the internet and of shared computing were built on the notion of free exchange of information and ideas. Floppy disks with home-made software for tinkering with spreadsheets were passed around between programmers. In the 1980s, Bulletin Board Systems, or BBSes, sprung up to allow users to share what they had made. The internet was not designed with security in mind. Already by this point, malicious and curious users had noted some of the security vulnerabilities in the complex computing systems that were emerging, and it became clear that a system of academic trust would not prevent an attack.

This provided the motivation for a new field of study for the engineers and scientists maintaining systems, which explored the process of decomposing a complex computer system and attempted to determine ways through which an attacker might attempt to exploit any vulnerabilities. As far back as the 1960s, students sharing computing time on a mainframe at MIT had printed out the shared list of passwords and had logged on as their peers, giving themselves more hours of processing [21]. They left notes on each others files with witty greetings; a more ill-natured attacker could easily have caused a much greater level of damage.

The idea of modelling attackers and how they might interact with systems was refined over the next few years, and in 1994 a graph-based 'threat tree' model was formally proposed [1]. This took the form of a decision tree, looking at a component in an IT system, and decomposing ways it could be exploited. While threat modelling can be applied to any complex system, such as how an attacker might try to break into a building, the IT sector has seen several key models developed; two of these will be briefly considered to gain a frame of reference for the current threat modelling process.

1.1.1 STRIDE

The STRIDE threat model was developed by Microsoft, and is the most widely-used threat model as of today. It can be decomposed into modelling against six main threat categories: Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, and Elevation of privilege. Each category of threat is considered individually by constructing a Data-Flow Diagram (DFD) of the system, then considering how that threat affects every component in the system and the relationships between them.

STRIDE is a methodology and a thought process rather than a tool, which is seen in the case of many threat models. Threat modelling requires careful and open thinking both about how every component in a complex system works together, and how an attacker might attempt to exploit any single point of weakness in that system. As such, diagramming is a vital tool for classical threat modelling, as we see in STRIDE's Data-Flow Diagram and the earlier Threat Tree Decision Diagram. Diagrams and graphs allow us to represent abstract systems and the flow of information through them in a human-readable way.

1.1.2 PASTA

The Process for Attack Simulation and Threat Analysis (PASTA) threat model is a methodology similar to STRIDE. There are seven stages of threat analysis, from identifying aspects of the current system through to developing countermeasures to possible attacks [19].

Figure 1.1: Stages of the PASTA threat model. From [14]

The first three stages involve defining things including the system's goals, its actors and users, and its use cases. These are a preliminary set of stages undertaken in order to gain the information required to carry out a thorough enough analysis, although (as in STRIDE) it also includes building Data-Flow Diagrams.

In the next three stages, the system is examined, starting with Threat Analysis. This determines probabilistic attack scenarios, and looks at threat intelligence. Following this, vulnerability and weakness analyses are performed. These consider knowledge of existing vulnerabilities and issues, and look at design flaws found by considering possible use and abuse cases. Threat trees are also built in correlation with these existing vulnerabilities. The last modelling stage here is Attack Modelling, where a tree is developed to analyse the attack surface and assess the risk of possible exploits.

The final stage is for Risk and Impact Analysis, where the gained knowledge from each stage is summarised. The potential impact to the business is quantified, and strategies for mitigating the greatest risks are identified.

Overall, PASTA is a methodology that puts great focus on how the attacker is thinking, as well as on the potential risk of being attacked.

1.2 Attack Trees

This risk-centric threat modelling methodology of considering the perspective and abilities of the attacker, as used in stage 6 of the PASTA model, comes from a line of research that occurred under the National Security Agency (NSA) during the 1990s, around the same time that the threat tree model was being realised.

This work culminated in a paper by Bruce Schneier which proposed a methodology of modelling adversaries and vulnerabilities of a system before considering the system's security properties [3]. Accompanying this rationale is the introduction of a tree-based model designed 'to enumerate and weigh different attacks against a system.' Schneier called these graphs 'Attack Trees', and constructed them by attempting the replicate the work of an attacker trying to determine weak points in a system. The root node of the attack tree is the objective of the adversary. To form child nodes, the system engineer

decomposes the node into its life cycle, or components. This process is repeated iteratively; thus, the leaves of this tree are all vulnerabilities in the system.

Although the concepts presented by Schneier here are hugely influential in threat modelling today and foundational in this project, the definition for attack trees given above is somewhat rudimentary. As Schneier himself explains, one potential complication is that the root node could either be the attacker's goal or 'the system component that prompted the analysis' ([3], Section 4.0), which contradicts the key focus on the adversary and their goals above the current system.

Definition of Attack Trees

In a paper which appeared in the following year, Schneier built upon his concept and provided a more concrete definition of an attack tree [15]. The root node was now seen as being always the attacker's goal, implying a change in focus to take the attacker's perspective in identifying vulnerabilities. Working with this definition, we can now describe a list of criteria for an attack tree; it is this definition which will be referenced and implemented in this project.

- An attack tree is a model of how a system might be attacked. It considers characteristics of a potential attacker and aims to highlight vulnerable areas of a system. An attacker may be financially motivated, skilled, motivated, and already with internal company access - or simply a bored student.
- The root node of an attack tree should be the attacker's main goal. For a computer system, this could be to gain unauthorised root access, or to prevent access to a website.
- Children of nodes in the tree are ways of achieving their parent attack. They are goals themselves. For example, to prevent access to a website, an attacker might organise a distributed denial-of-service (DDoS) attack to flood them with requests and overwhelm the server, or to remotely gain access to the server and shut it down internally.
- Leaf nodes in the tree are atomic attacks. Given the level of depth required for an attack tree, these could be somewhat basic, as you could decompose child attacks to a great (but possibly unnecessary) extent. For example, 'bribe website admin' could be considered a leaf attack in gaining root access to a server.
- Attacks with two or more children are either considered **And** nodes or **Or** nodes.
  - **And** nodes imply that the children together form a set of steps to achieve that goal; they must all be carried out.
  - **Or** nodes imply that the children are alternative ways of achieving that goal; any one could be a potential attack method, and only one needs to be carried out for the goal to be achieved.
- Values can be assigned to leaf nodes. For example, leaf attacks can be designated a difficulty for the attacker of Easy, Medium, or Difficult. Some other typical values include:
  - The cost of carrying out that attack
  - Whether the attack requires any special equipment to be carried out
  - The probability of that attack- this could be further simplified to whether the attack is possible or impossible
  - The time taken for that attack
  - Probability of success for a given attack

  We will refer to these values as being *attributes* of the tree from here. Each leaf node is manually assigned a value for each attribute. Attributes can also take from a range of domains- some are boolean (true/false), some are numerical, some lie in the unit interval (between 0 and 1).
- Attributes propagate up the tree to the root goal. This is how we gain information from assigning attribute values to nodes. Each node is assigned a value for its attributes based on its children's values, and this flows up to the root. This is most clearly seen in the example below in Figure 1.2.
- **And** and **Or** nodes have different rules for how they combine their children's attribute values according to the semantic difference between the two. For example:

– An attack that is an 'Or' node would only require one of its children to have a difficulty level of 'Easy' for the node itself to be designated 'Easy' in difficulty.

– An attack that is an 'And' node would have its cost as the sum of all of the costs of its children.

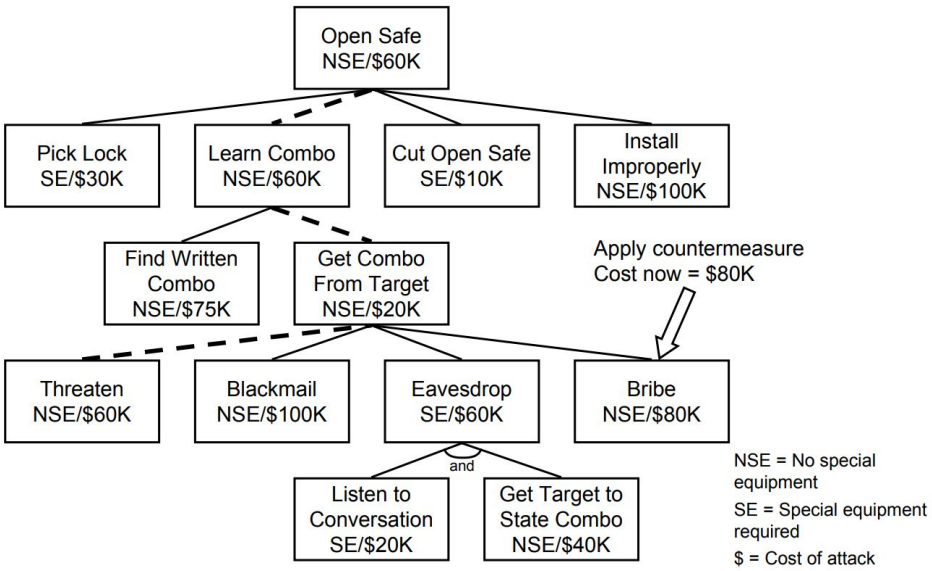


Figure 1.2: An example attack tree with two attributes. From [15]

Figure 1.2 neatly shows the value of attack trees as a way of representing potential weaknesses of a system. It does this concisely without losing usefulness. It is easily comprehended by humans, yet still information dense and containing several sets of useful values. From looking at the above graph, it can be seen that an attacker looking to open our safe could do so with no special equipment at an estimated cost of $60k, should they have access to the target. A safe containing more than that amount of money would therefore provide motive to a financially motivated attacker, and countermeasures should be taken if that is the case.

This introduction has served to demonstrate that attack trees are incredibly useful models for threat modelling and modelling the effects of a countermeasure, and therefore that learning how to develop and interpret one is a valuable skill.

It is in facilitating the teaching of this skill that this project is based, through the presentation of Atree, a web tool allowing for the fast creation and modification of these attack trees, which has been designed explicitly for students learning about attack trees and threat modelling to interact with. It allows users to collaboratively work together in real-time in groups; it is platform independent, lightweight, and encourages experimenting with attribute values and propagation to rapidly build a natural understanding of the involved mechanisms.

1.3 Why is this tool important?

Developing Atree was motivated by a lack of similar software for creating attack trees in an educational environment. The industry demand for threat modelling is sufficient that several commercial products exist, however these are largely inaccessible to students. The currently available applications for attack tree development and live collaborative group work are investigated below, demonstrating a gap in utility between the two, and even for a student working by themselves. This is shown in Table 1.1, which presents the results of researching and experimenting with a wide range of existing software.

The intention of this project is to create a tool from which students will benefit, and with which they can more easily learn about threat modelling with attack trees.

It is worth noting that real-world threat modelling is a specialised process. It must comply with frameworks and regulations such as GDPR. Companies that may have experts in security *may* use threat modelling tools to assist their work, such as LINDDUN [9]. In some applications this might involve creating attack trees.

1.3.1 Existing Attack Tree Tools

Table 1.1: Comparison of Existing Attack Tree Software

Software Name	Free?	Dedicated Attack Tree Support?	Collaborative?
Isograph AttackTree+	No	Yes	No
dia-attacktree	Yes	No	No
drawio-threadmodeling	Yes	Yes	No
Ent Attack tree visualiser	Yes	Yes	No
MIT CSAIL Tool	No	Yes	No
Amenaza SecurITree	No	Yes	No
SeaMonster	Yes	Yes	No
AttackTreeMonkey	Yes	No	No
ADTool	Yes	Yes	No
Tutorialspoint Whiteboard	Yes	No	Yes
Creately	Yes	No	Yes

Following this surveying of existing software, there are two standout options for students trying to learn how to build and modify attack trees, and both have significant drawbacks.

ADTool is a free, open-source threat modelling tool developed by Piotr Kordy at the University of Luxembourg [8]. Developed as a Java form application, it offers both a textual and graphical view of an attack tree, and supports a wide set of attributes. Each attribute has a domain, rules for how they should be combined by And/Or nodes, and a default value.

On the other hand, the user interface (UI) is unintuitive, the textual view is barely human readable and hard to use, and adding attributes creates a new tab with a dedicated render of the graph for its values. Propagating values up the tree or looking at and editing two attributes at once is not possible. Furthermore, there is no web support; it exists solely as a local application instance. An example of using ADTool can be seen in Figure 1.3.

Tutorialspoint Whiteboard is an online, in-browser whiteboard tool. Users can create and join groups with a unique ID, and start drawing and creating simple shapes and text in a live session. The in-browser room joining, and ease of using the whiteboard as a group makes this an excellent brainstorming tool for discussing ideas. While you could use it to draw a simple tree, adding attributes and modelling a complex system and countermeasures quickly becomes messy and the graph is rendered unreadable. It does not work effectively as an attack tree creating tool.

Thus there is a deficiency in the existing software that calls for a tool to be created which can combine the strengths of these two applications, whilst successfully mitigating the areas in which they do not meet the requirements of an educational support tool.

1.4 Notable Challenges

At the outset of the project, there were several key issues that I considered to be particularly notable. One such challenge seemed to be developing a tool that could render a tree in a consistently human-readable way; researching a library which could shoulder this functionality would have to be a priority. Furthermore, storing attributes and the internal representation of the tree would require some consideration. Attributes can take a range of values and domains, and would have to be presented as such. Nodes would have to contain a label, this attribute metadata, as well as an And/Or value.

The other main facet of the tool would be real-time collaborative editing, which again would hopefully be implemented with the support of an existing library, handling communication between clients. Sending huge amounts of requests to a central server would not be a desirable solution.

Lastly, designing a human-readable layout was an issue I had with several of the tools I used when experimenting with software, in particular ADTools and SeaMonster. I wanted to allow users to view a graph-based and textual view of their attack tree, as well as the tree's attributes, to enable editing of such attributes, and to include a host of auxiliary features.

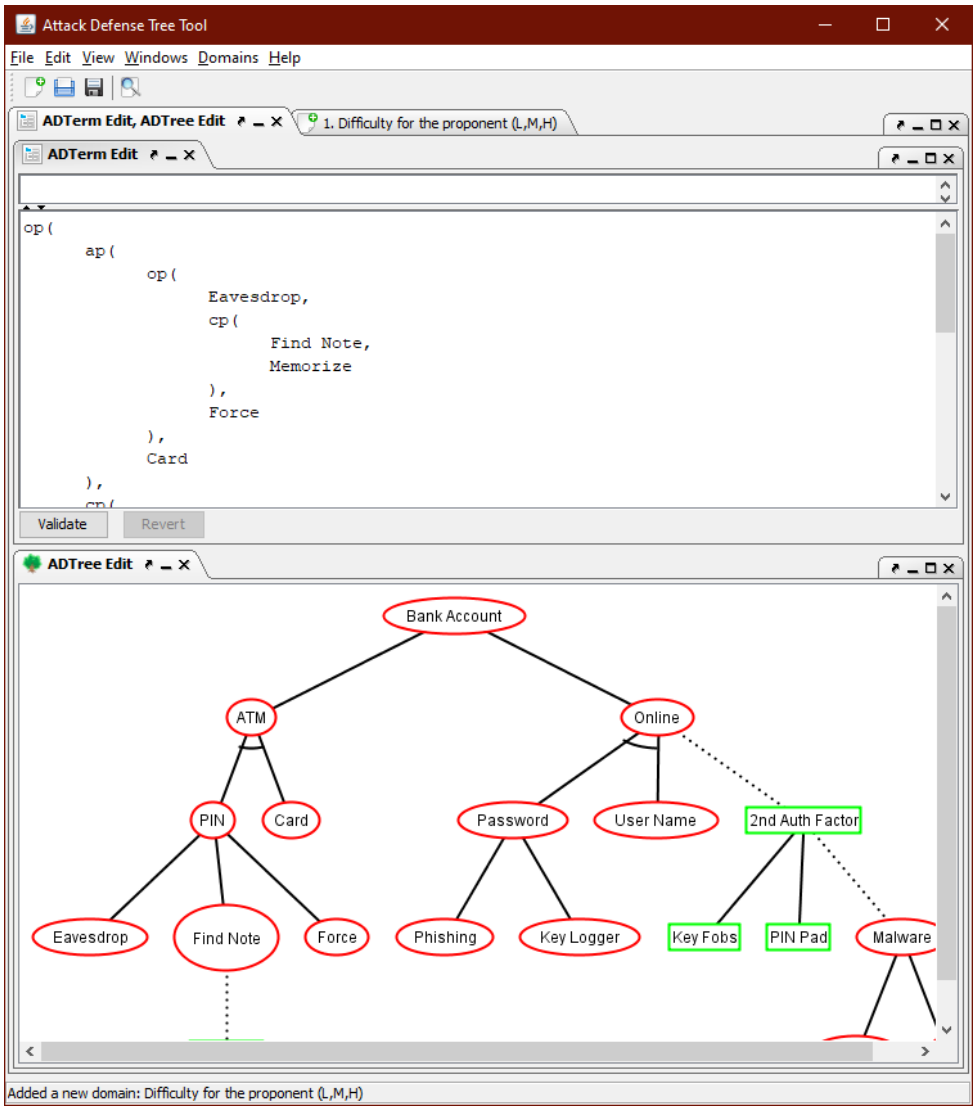


Figure 1.3: The ADTools GUI

1.5 Summary of Aims and Objectives

The goals of this project are to:

1. Research and evaluate graphing libraries and to design a complementary internal representation for storing attack trees.

2. Write a parser to take the internal attack tree representation and convert it to a human-readable and compact list-based format.

3. Build a tool with these components that allows users to build attack trees in real-time collaborative editing sessions. This tool should be lightweight, allow users to build complex attack trees that satisfy all of the criteria defined in the Definition of Attack Trees, and usable by students without much technical knowledge.

4. Collect and analyse feedback from my tool to objectively analyse areas requiring improvements to usability.

Chapter 2

Technical Background

2.1 Graph Theory

To understand attack trees, it is important to first understand what defines a *tree*, and more generally, a *graph*. Graph Theory is a field of mathematics that is complex and incredibly powerful, but an introductory level of understanding is sufficient to describe the structure of an attack tree.

A graph is an abstract mathematical object, defined as a tuple (or pair) of sets (V,E). To fully understand how a graph is constructed, a set must therefore be first defined.

2.1.1 Sets

A *set* is an object from its own related field of mathematics, Set Theory. It will be sufficient here to simply define a set as a collection of something where each element is of the same sort.

For example, in a set of pennies, each coin could have a different denomination or currency- but they are all still pennies. This set is given a name, for example $\mathcal{X}$. In notation, $\mathcal{X}$ could therefore be defined as follows:

$$\mathcal{X} = \{1p, 2p, 10p\}$$

A set contains some number $\mathbf{n} \in \mathbb{N}_0$ elements. A set containing no elements is the case of the special empty set, represented by the symbol ϕ.

If one set, $\mathcal{A}$, contains every element in another set, $\mathcal{B}$, then $\mathcal{B}$ is a subset of $\mathcal{A}$.

$$\mathcal{B} \subseteq \mathcal{A}$$

Also important to consider are strict subsets. $\mathcal{B}$ is a strict subset of $\mathcal{A}$ iff $\mathcal{A}$ contains every element in $\mathcal{B}$ and more.

$$\mathcal{B} \subset \mathcal{A}$$

Finally, an element $\mathbf{a}$ can be said to belong to a set $\mathcal{A}$ if the set contains that element. This is written as:

$$\mathbf{a} \in \mathcal{A}$$

2.1.2 Graphs

Now sets have been defined, a basic graph (V,E) can be constructed.

- V is a set of vertices, also called nodes. These vertices could be anything; consider the case where vertices are integers, for simplicity.
- E is the set of edges. Each edge is a tuple of vertices: the two vertices connected by that edge. E is defined at its most basic as

$$E \subseteq \{(x, y) | x, y \in V\}$$

This states that E is the subset of all the possible tuples of vertices in V, including edges that go from a node to itself. x and y are two vertices in V.

A graph can be drawn by first drawing each vertex and its value, and then drawing lines between them to represent edges in the graph. For example, for the graph $\mathcal{G}_1 = (V_1, E_1)$:

$$V_1 = \{10, 20, 30, 40, 50\}$$
$$E_1 = \{(10, 20), (30, 40), (10, 40), (40, 50)\}$$

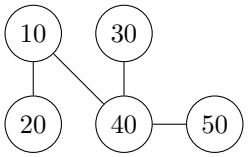


Figure 2.1: Graph $\mathcal{G}_1$

This is the simplest type of graph. To define a tree, variations on graphs should be first considered, which typically come in the form of restrictions on how edges are applied and viewed, and which nodes they can connect.

Directed graphs make a simple addition to the definition of edges. Instead of just linking two vertices, x and y, a directed, one-way link from x to y is created. x is referred to as the *source* vertex, and y is the *target* vertex. Unless otherwise restricted, a vertex could have a directed edge pointing to itself.

Weighted graphs introduce values to each edge in the graph. Each edge is now a triple (x, y, w), and joins the vertices x and y with an edge of weight w. A weighted graph may also be directed.

Connected graphs are graphs where a series of edges from any one vertex in the graph to any other vertex can be traced. A disconnected graph is the opposite. A disconnected graph can be decomposed into multiple connected graphs, the most basic of which is a graph with one vertex and no edges.

Simple graphs are graphs with no parallel edges, and no loops. Parallel edges are any two edges (x,y) that link the same two vertices; edges $(x_1, y_1), (x_2, y_2) \in$ E for which $x_1 = x_2$ and $y_1 = y_2$. Loops are edges that connect a vertex to itself; that is, edges for which $x = y$. Figure 2.1 is an example of a simple graph.

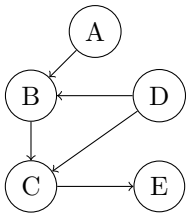
Figure 2.2: Left: parallel edges. Right: edge loops

Complete graphs are graphs where every vertex in the graph has one edge to every other vertex, excluding itself. These may also be directed or undirected. By this definition, complete graphs are therefore also simple.

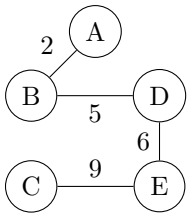
2.1.3 Trees

At last, a *Tree* can be formally defined - a graph with the structural constraints that will be used for building valid attack trees. A tree is defined to be a **connected, directed, unweighted, simple, acyclic graph**.

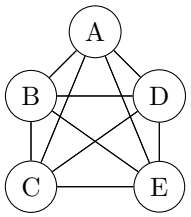
A cycle is simply a subset of edges that forms a closed region within a graph. An example of a cycle can be seen in Figure 2.3a, in the subset of edges between vertices B, D, and C. The term *acyclic* refers to a graph that does not contain any cycles. Note that traditional trees in graph theory are undirected;



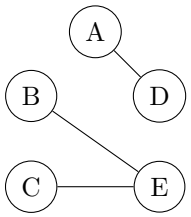
(a) Directed graph



(b) Weighted graph



(c) Complete graph



(d) Disconnected graph

Figure 2.3: Different Graph types

however, attack trees in particular are a form of Directed Acyclic Graph, or DAG [17]. This is a further restriction on trees to contain no *directed* cycles.

The structure of such a tree has one 'root' node. Nodes can have any number of child nodes. A node m is a child of node n if there exists an edge $(n, m) \in$ E with source n and target m. Every node in a tree is a child of some other node with the singular exception of the root node. Nodes in a tree with no children are called 'leaf' nodes.

A *subtree* is the tree equivalent of a subset. It is created by choosing any node in a tree and taking only the tree with that node as its root. A subtree will have its set of vertices and edges as non-strict subsets of the trees vertex and edge sets. In other words, for a tree (V, E), and subtree (V_{sub}, E_{sub}):

$$V_{sub} \subseteq V$$

$$E_{sub} \subseteq E$$

In Figure 2.4, taking the subtree at the node D would give a tree with V = {D, F, H, I}, and E = {(D,F), (F,H), (F,I)}.

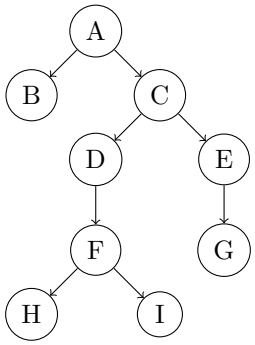


Figure 2.4: Example DAG

2.1.4 Viewing a tree

It is important to remember that a tree, or any graph, is just the tuple of sets (V,E) as defined earlier. The way that this information is presented can vary greatly, and drawing a tree as in Figure 2.4 is only one way of visualising them. Another common way of representing a graph is as a list (as in Figure 2.5), where each node is written on a line of its own and given an index based on the depth and child number of that node. The *depth* of a node in a tree is the number of edges from the root to that node. This is similar to the tree *height*, which is the maximum depth across all leaf nodes.

1. A
1.1 B
1.2 C
1.2.1. D
1.2.1.1. F
1.2.1.1.1. H
1.2.1.1.2. I
1.2.2. E
1.2.2.1. G

Figure 2.5: Example DAG from Figure 2.4 written as a list.

Both representations have their benefits and drawbacks. Drawing nodes and connecting them with edge lines (which from here will be referred to as the *graph view* of the tree) is an excellent visual aid for following the overall flow of a graph. This will be especially powerful when considering attributes in the next section. On the other hand, it is less space-efficient, and can cause the graph to be *more* difficult to read, especially when dealing with large quantities of nodes.

Writing the graph as a depth-first list as in Figure 2.5 will from here be referred to as the *list view* for a graph. This is far more compact than a graph view, and can easily represent graphs with a huge number of nodes and trees of a great height. On the other hand, the visual aid is lost; reading the indices of complex graphs quickly becomes difficult.

Both the list and graph view have their merits and both are applicable to attack trees; Schneier himself presents a complex attack tree in his second paper in the form of a list, and ADTools uses an editable list view to build the graphical view, complete with syntax highlighting. Therefore, both views should ideally be implemented in any attack tree tool.

Overall, knowing how DAGs (and therefore attack trees) are represented, and how they arise from graph and set theory, gives a solid foundation to build from.

- It allows for understanding the decomposition of trees when parsing them, and for constructing more effective attribute propagation algorithms.
- Any graphing library or framework to be considered must support a DAG implementation, and some way of either validating trees as such, or building an interface where only valid DAGs can be created.
- Finally, consider transmission of attack trees between clients. To do so effectively, only the minimal amount of data necessary should be transmitted- converting a tree to and from such a structure, or even only transmitting information about parts of the tree in updates, is made easier by knowing how a DAG is composed.

2.2 Attributes and Propagation

Attributes are the key factor separating attack trees from an annotated DAG. Each node in the tree will contain a value for each attribute the tree currently maintains. Only leaf nodes are manually assigned values for attributes. For non-leaf nodes, attribute values are given by the propagation of attributes up the tree. Different attributes can have different rules by which they are propagated; whether a node is an And or Or node also fundamentally changes how that node's attribute values are calculated. This is all best demonstrated with simple examples.

Consider an attack tree with one attribute, the *cost* for the attacker to carry out the goal described by that node. The user wants to get an estimate for the cost to the attacker of the root goal, to understand if a financially motivated attacker poses a threat to our system. Cost can be assumed to be a positive numerical value. The And and Or rules are first derived from rationally analysing what those different types of nodes mean in the context of that attribute.

To help in doing this, it is useful to consider how Schneier describes the attacker's thought process in the 1998 NSA paper:

> "A rational adversary follows the path of least resistance; that is, he chooses an attack that maximizes his expected return on investment, given his budget constraints of resources (money, expertise, manpower, and time), access, and risk."

- **And** nodes mean that an attacker must carry out every child attack to achieve that node's goal. Therefore, the cost of an And node is the *sum* of it's children's costs.
- **Or** nodes mean that an attacker can carry out any one of the child attacks; they are different ways of achieving that node's goal. Considering Schneier's quote above, it can be assumed that the attacker would attack using the cheapest method available, given that each method has the same chances of success, same level of ease to carry out, etc. However, these can all be modelled by other attributes! The cost of an Or node is the *minimum* of all it's children's costs.

Knowing these rules, cost can be propagated up a tree to the root.

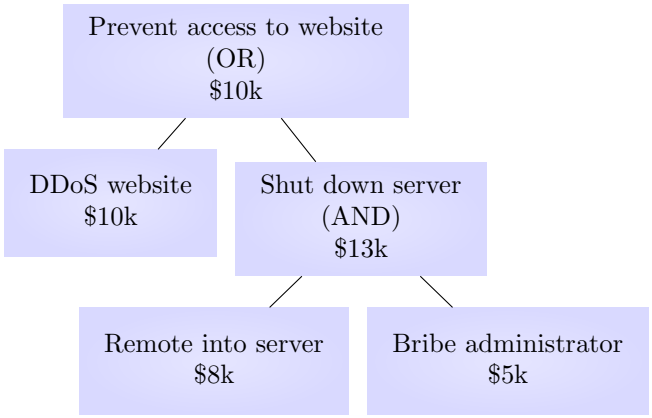


Figure 2.6: Example attack tree showing propagation of attack cost

While cost can be easily represented as a positive number, different attributes can take from completely different domains; some may be numerical, while others may be boolean, or take from a list of enumerated options. Some of these options are given in Table 2.1. Note that in this table, the values x represents the cumulative value for the node's attribute, and y represents each child value iteratively.

Table 2.1: Common attributes and their combination rules

Attribute	Domain	And Rule	Or Rule
Cost of attack	$\mathbb{N}_0$	sum(x,y)	min(x,y)
Requires special equipment	{True/False}	$x \vee y$	$x \wedge y$
Possible for attacker	{True/False}	$x \wedge y$	$x \vee y$
Probability of attack	[0..1]	x * y	x + y - (x*y)
Difficulty for attacker	{Easy/Medium/Hard}	Hardest(x,y)	Easiest(x,y)

An attack tree with two or more attributes propagates them all together. This can be seen in Schneier's example tree in Figure 1.2, which has two attributes; a label denoting whether the attack requires special equipment or not, and the cost of the attack. This tree has been adjusted to show the cheapest possible attack requiring no special equipment at the root. As well as this, Schneier's diagram also represents a tree after the application of a countermeasure that has affected the cost of bribing a target. Editing attributes then causes a propagation of new attribute values up the tree to the root. This flow of information allows for easily visualising the effects of changes to a system for risk analysis and threat mitigation.

There are more modifications to a tree that cause a revised propagation of attribute values up the tree:

- Adding new children to the tree must cause a recalculation of values for the node the child is appended to. While a theoretical attack tree has its values manually given at the time of creation, a more easily implementable option may be to assign default values for each attribute, and propagate those.

- Deleting a node on a tree has the same effect. When deleting a node, two approaches could be taken. Either one still requires recalculation of the attributes for the parent of the deleted node.
  - First, one could define deleting a node to mean removing the entire subtree with that node as its root.
  - The other way to interpret deleting a node would be to set all of its children to be siblings of that node, and then to remove the node. This would preserve the subtree structure beneath the removed element.
- Revising a node's type from an And node to an Or node, or visa versa, requires that node to be recalculated, and that value propagated up the tree.

Note: *sibling* nodes are those which share the same parent. That is, if there are edges $(x_1, y_1), (x_2, y_2) \in$ E for which $x_1 = x_2$ and $y_1 \neq y_2$, then it is written that y_1 and y_2 are siblings.

Since propagation in attack trees will therefore be frequent and could require recalculation of many variables, the propagation function will be structured roughly as in Algorithm 2.1. This is recursive, since the depth of the node that propagation begins at is unknown.

```
Function Propagate_Attribute(node, attribute):
    cumulative_value = attribute→default_value
    foreach child ∈ (node→children) do
        f = attribute→rule
        cumulative_value = f(cumulative_value, child→value, node→AND_OR)
    end
    node→value = cumulative_value
    if node ≠ root then
        Propagate_Attribute(node→parent, attribute) ;
    end
End Function
```

Algorithm 2.1: Attribute Propagation

2.3 Real-time Collaborative Editing

Since this project involves developing a tool that allows users to work in groups, there will need to be some way of sending and receiving an attack tree and its associated attribute metadata over a network. The implementation and supporting library used will depend on the framework and language used, but how updates are transmitted over a network can be done in several different ways. The simplest way to do this would be to transmit the entire tree, either the full internal model or by parsing it into some other representation first.

Secondly, each type of modification, such as those listed in 2.2, could have its own message structure built and applied by the recipient. This method means that the entire tree would not have to be sent to the server with each update, only the update itself. However, this method would require an implementation and message signature for each type of modification. Furthermore, it creates the possibility of the tree becoming desynchronised between different versions. Two users might send different or conflicting updates to the tree at the same time, and how the server updates that tree might lead to inconsistencies, which could be difficult to identify and which might reduce the usability of the tool or render correct computation of attributes impossible. This would not be an issue if the whole tree is transmitted with every modification.

Transmitting modifications to the tree is a form of Operational Transfer (OT). There is another way synchronicity could be ensured between tree versions: by implementing a CRDT datastructure.

2.3.1 CRDT vs OT

Conflict-free Replicated Data Types, or CRDTs, are datastructures that implement an alternative approach to the OT method. OTs transform index positions to ensure convergence, and require a 'central source of truth' to store the master copy of the datastructure. CRDTs, on the other hand, update datastructures with mathematical models that change the state of the datastructure [7]. Since state is sent

instead of operations, no central source of truth is required, and inconsistencies in models are guaranteed to be able to be mathematically resolved.

A CRDT tree datastructure exists in the form of Treedoc [13]. This implements two operations, Add and Delete, which occur on a user's local replica of the tree. The state of the tree is represented by a set of (*atom, PosID*) tuples. Atoms are elements in the datastructure - for example nodes in the tree, or edges. Each one has an associated unique position identifier, which is the PosID. Since PosIDs are unique, add and delete operations can happen in any order.

The Treedoc solution is elegant and mathematically sound, but the CRDT state can only grow in size over time. This is because the addition and deletion operations are the inverse of each other. Addition followed by deletion (of the same node) leaves the tree as it was. In CRDT these become part of the state for the tree, and are called *tombstones*. Performing 'garbage collection' on these tombstones can mitigate the issue, but may reach the point at which the solution to a problem is made unnecessarily over-complicated. As well as this, the Treedoc implementation would have to be extended to represent the different modifications that attack trees represent.

All variations on transmitting updates can be considered up to the point of implementation, as all should be feasible and have their own merits. For CRDT, a suitable library should be preferred due to the intricacy of implementation.

This chapter has so far considered how attack trees and their attributes are constructed and stored from the ground up, and ways in which real-time editing between users can be achieved. In the final section, graphing libraries and communication libraries are discussed and compared, as the prelude to designing and implementing the tool.

2.4 Supporting Technologies

2.4.1 Graphing Libraries

A range of libraries for storing, manipulating, and rendering graphs were tested across a range of languages, in order to determine the strengths and weaknesses of each, and to help identify early what features would be needed from any library used. Furthermore, having libraries at disposal in a range of languages would grant later freedom in working with the language that best suited the needs of the tool.

Treelib

The first library experimented with was a Python 2/3 library, Treelib [5]. Treelib supports tree creation and manipulation, and higher-level functions including taking subtrees, performing lambda functions on nodes, and saving trees to files.

I spent a few hours implementing a custom wrapper for treelib in the context of attack trees, and was able to take advantage of a useful functionality that treelib implements. Trees can be built with nodes of custom python objects through manipulating the `node.data()` payload variable. This data object could contain attack tree node information, such as And/Or and attribute metadata, allowing for a full implementation, although this was not done in during experimentation. The treelib attack tree wrapper is available in Appendix A.

However, there were drawbacks; firstly, accessing nodes through the `node.data()` call is verbose and complicates traversal. The much more significant drawback is that treelib is a purely textual library. Trees can be printed with the `tree.show()` function, as seen in Figure 2.7. This list view provides a good reference for how to represent a tree textually, but it means that any graph view would have to be implemented manually, complete with a GUI, as part of the wrapper. The treelib documentation does mention a plugin with support for a python graph visualiser, but also mentions that it is deprecated.

MindFusion

MindFusion ASP.NET NetDiagram is a mature commercial diagramming library that supports a range of platforms and tools [10]. It supports manipulating nodes and high level tree functions in a .NET Core WinForms environment, providing a strong UI toolset especially in contrast to treelib. Features such as XML parsing are supported, allowing for parsing and saving trees as static files, which is a useful criteria for a teaching tool. Nodes are created as MindFusion `ShapeNode` objects, a complex class that also contains rendering metadata such as rectangle bounds and node hashes.

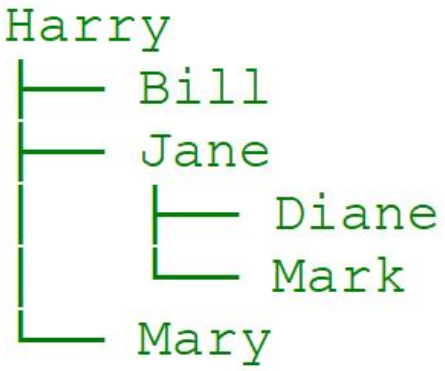


Figure 2.7: An example treelib tree drawn with the `tree.show()` function

MindFusion is implemented in C# with ASP.NET for web connectivity, however as a commercial library it requires a subscription for full access; only local graphing support is available at a free tier. Paid libraries will not be considered for the sake of this tool as it compromises the tool's availability; this includes the JavaScript library GoJS, which would otherwise be an excellent fit for building interactive attack trees with HTML5.

MxGraph

MxGraph 4.2.2 [12] is a mature, open-source diagramming library implemented in C# with ASP.NET, Java, and JavaScript. It has been built into a tool that can represent a massive range of graphs and models, from Organisation Charts to apartment floor plans, and as such it has a somewhat daunting API and steep learning curve, which is at its simplest in the JS implementation.

The Java implementation is done through the library JGraphX. Web support could then be combined by running a servlet such as via the Spring framework, although it is designed for high-performance development on a local desktop application.

The JavaScript implementation is contained entirely within a few lone files, and is designed specifically for web tooling, working natively in all major browsers by executing in a HTML5 container. It is incredibly easy to set up and experiment with the mxgraph JS API, and it is designed to reflect this more lightweight rationale over the Java API. The key disadvantage is rendering speed, which is sacrificed in graphs with node counts that carry into the hundreds, even in current browsers.

The C# implementation, while containing the same complex feature-set as Java, is also web-focused. However, it requires publishing ASP.NET servers with Microsoft Internet Information Services (IIS). Having to do so would reduce the tool's offline accessibility for a student wanting to locally build and run the tool, so there would have to be a continuous server with database support.

Overall, mxgraph is an incredibly powerful and mature library, supported in a series of powerful languages with web support, and each could feasibly be used to build an attack tree tool.

Own implementation

The final option is to pick the most suitable language and build an attack tree model, renderer, and GUI support all from scratch. The key advantage to this is the ability to tailor an implementation to the attack tree criteria list. The bloated feature list that mxgraph suffers from at times is not an issue, and having full control over how a class is implemented allows for a more natural understanding.

The downsides to doing so come from the time investment required to implement and test to a point of stability, which could easily stretch into weeks of valuable development time. Rendering a graph naturally with any number of children and any height is a difficult task, to make nodes readable and not overlap, and to add handlers for necessary events, etc; performance will also likely be sacrificed compared to a graphing library with a decade of iterative improvements. As such, this should only be considered a worthwhile option if no suitable communication libraries exist in an mxgraph-supported language.

```
var io = require('socket.io')();
io.on('connection', (socket) => {
  io.to(socket.id).emit('msg', 'Hello,
      World!');
};
```

Listing 2.1: Serverside socket.io example

```
var socket = io();
socket.on('msg', function (response) {
  // response == 'Hello, World!'
  alert(response);
};
```

Listing 2.2: Clientside socket.io example

Figure 2.8: socket.io example

2.4.2 Communication Libraries

Two communication libraries were considered in detail, *socket.io 4.0* and *y-js*. Both of these are JavaScript libraries; while socket implements client-server messaging, y-js is built specifically to provide a CRDT implementation.

socket.io

socket.io is a library that "enables real-time, bidirectional and event-based communication between the browser and the server." [16] It can handle a range of events, including client connection and disconnection, as well as sending objects to clients and the server, with event listeners for receiving specific messages on each end. These objects that socket.io sends can be primitives, such as integers or strings; or they can be more complex objects, such as lists and JSON objects, allowing for a surprising amount of complexity in what can be sent between clients. socket.io also works in conjunction with Node.JS libraries such as Express.JS.

An example server-side and client-side socket messaging system could be as is seen in Listings 2.1 and 2.2. When a client connects, a socket connection is established, and the `io.on('connection')` handler is called, causing socket to emit a message with the name `'msg'` and the contents `'Hello, World!'` to the id of the socket passed as a parameter by the connection event. The client catches this message in the `socket.on('msg')` listener, and shows the response to the client as an alert. In short, when a user connects to the page, a popup saying `Hello, World!` will appear. These messaging events work the same in both directions, and are extremely powerful and far more convenient than typical `/GET` and `/POST` requests.

Socket could support real-time collaborative tree editing as it gives control over which socket connection ids the server can message. Storing these ids in a database or KV-store will allow for holding groups. Furthermore, messaging JSON objects could be a powerful way to transmit the tree or its modifications, for any one of the three approaches discussed in Section 2.3. For a purely CRDT implementation, the other JS library option is y-js.

y-js

y-js is a library for implementing the CRDT algorithm that is built on top of socket.io. It has many existing implementations in real-time collaborative work, supporting 'offline editing, version snapshots, undo/redo and shared cursors'. While it is a powerful tool, y-js is implemented most often with, and is built with a focus on, rich text editors. Each text editor has a custom implementation that must be custom defined for each use case, meaning in that order to use it with a tree, this custom implementation must also be constructed.

For example, the Quill rich text editor has a suite of y-js *bindings*, which binds Quill to a y-js text type called *ytext*: `const binding = new QuillBinding(quill, ytext)`. y-js also requires a provider to be chosen to exchange document updates with other peers. The alternative to this is to implement your own communication protocol, which could be done off socket.io.

Overall, y-js is an effective CRDT implementation, but it is restricted by the complexity of creating a binding and provider to handle state changes and transmit them. The benefit of doing so is the ability to not require a central version of the tree, which is why y-js naturally allows for offline editing and versioning. The ability to implement undo/redo is also a powerful feature for an editor that should be strongly considered.

Chapter 3

Project Execution

3.1 Technology Stack Selection

At the outset of development, libraries were evaluated according the needs of developing an attack tree editor with support for collaborative work. After looking at example projects for each library, and experimenting with some in more detail (as with Treelib), it was eventually decided that JavaScript's Node.JS framework would be able to support development, and mxgraph 4.2.2 was a library that offered everything that would be required for tree development. Live collaborative editing would be implemented after a local model version was successfully built.

A Jira project and Github repository were also set up before any model was developed to assist an agile methodology. Week-long sprints in Jira ensured that working versions of the tool could consistently be released for testing and getting a feel for usability, and Jira's project phases assisted in working to a schedule over a greater time-frame.

3.1.1 Language Choice

JavaScript was chosen due to its excellent web integration, as it can be written for both client and server operations. This followed from deciding to develop Atree as an online web tool rather than a standalone application with web access. An in-browser tool is far more accessible, and can be developed and tested to work across platforms easily. Since JavaScript is a scripting language, it is also faster to work in than C# or Java, meaning experimenting with features and functionality would impose less of a time cost, encouraging developing flexibly according to changing criteria.

Working with Java to take advantage of the full mxgraph implementation was also considered, since this could also be integrated with a web framework. However, JGraphX is designed for Java standalone applications. The extra work required in setting up a servlet with JGraphX integrated, as well as still having to write JavaScript for any clientside functionality, resulting in opting for the more lightweight choice.

3.1.2 Framework Choice

Running a web server with JavaScript was done using the Node.JS runtime framework, which allowed for working with JS libraries. This was used extensively throughout development to experiment with functionality and features. One such library used was Express.JS; using Express, a local server could be easily run to develop and test features quickly.

Also considered was the option of developing with front-end frameworks such as Angular, React, or Vue; such frameworks are favoured currently by many front-end developers as they offer fast performance, complex templating, etc. These were rejected in favour of more straightforward development without having to rely on integration or learning a complex front-end framework, which are designed for far more complex applications.

Also set up was a Heroku server connected to the Github repository. Builds could be pushed to this Heroku instance at the end of sprints, to test functionality outside of a local server. Heroku has excellent support for JavaScript and Node.JS, and the Git integration created a smooth deployment pipeline.

3.2 UI Design

After establishing a development environment, designing the tool in accordance to its target list of features was the next step. There was a set of elements that needed to be built into the webpage:

- An interactive graph view
- An interactive list view
- A way of editing and viewing node attributes
- A toolbar or dropdown for accessing file and edit options, such as importing and exporting trees
- Some way of joining, creating, and seeing groups

Implementing a toolbar or dropdown functionality would be relatively straightforward based off standardised designs that most users are familiar with in related tools. Joining and creating groups could be merged into this functionality given a solid enough design. This left three separate items which would need to be presented to the user in an easily recognisable manner. Sketching out some UI mock-ups resulted in a set of designs that fell into two main categories. One possible solution was to have two windows side by side containing the graph view and list view, with an adjustable slider between the two. Dragging the slider along its axis would resize the two windows accordingly. The other would be to have each view as dedicated tabs in a single window. Figures 3.1 and 3.2 show these contrasting design ideas and how the early UI designs were laid out.

With a slider, the user would be able to view both views at the same time, to watch them update simultaneously, and to control the dominant view by scaling it to take up more space in the panel. Using tabs, more space can be given to other important features without cluttering the screen (for example dealing with attributes).

How attributes would be handled was not fully realised at this stage. Some designs considered a dedicated window, others a context-specific panel that would pop up when needed. These ideas can be seen in the two UI sketches; one shows an attributes tab on the left, the other would have both a global toolbar and an attribute navigator within the graph view.

Continuing the rationale that resulted in using Jira to develop in sprints and experiment with different concepts, the initial plan was to first implement both with tabs and a slider, and then to evaluate and choose from the two. The next consideration was therefore how to design the editable graph view and how this would correspond to an internal model.

3.2.1 Graph View Interactivity

In building an interactive editor for an attack tree, the main operations to support would be adding children, deleting nodes, and editing the label and attributes of existing nodes. The UI sketch in Figure 3.2 shows a drag-and-drop system for building a tree with different node types; here, this would be a typical attack node, a pair of And and Or nodes given a corresponding internal tree representation, and a node for labelling the tree with external comments. Edges would be constructed by joining nodes, and there would be a large degree of freedom in how the user arranged nodes in the tree. A user could design the layout to be as readable as possible for that tree. This is the approach taken by the diagramming application draw.io, which gives the entire left side of the screen to a drag-and-drop node menu. Deleting nodes and editing attributes would have to be done by selecting nodes and then having context-sensitive operations that would act on what the user had selected. This created a blend of ways of interacting with the tree, which is less desirable as users would have to try several different things to understand all the functionality available to them.

Alternatively, the Org Chart builder in the mxgraph examples implements a completely different method of building a tree. Each node in the tree is given overlay buttons with functionality to add children and delete the subtree at that node. These are context-specific actions, and the context is given naturally by clicking on the button overlaid on that node. This creates an intuitive interaction with the graph. Editing the Org Chart node labels is done by double-clicking nodes, which is the same for many mxgraph implementations, including draw.io. As such, this would be an intuitive way of interacting with a diagram for those who have used similar tools before, but it may not be the case for users without previous experience using diagramming tools.

The Org Chart interactive model was chosen as a reference implementation for several reasons. Firstly, the internal representation of And/Or nodes would likely be stored as node metadata, rather than as a

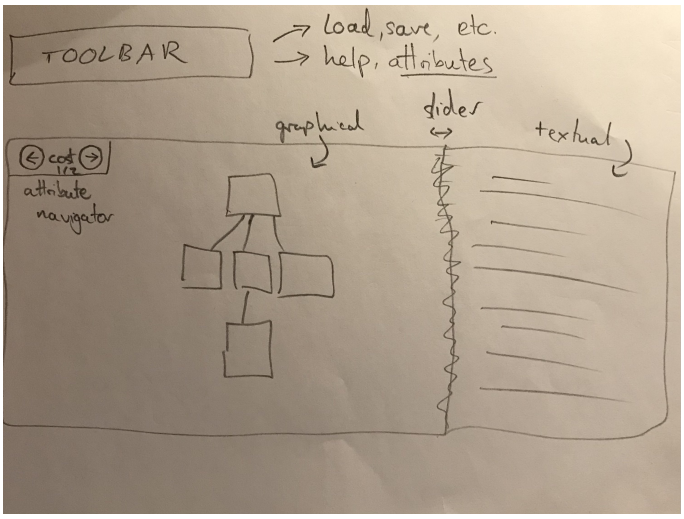


Figure 3.1: Slider-based mock-up

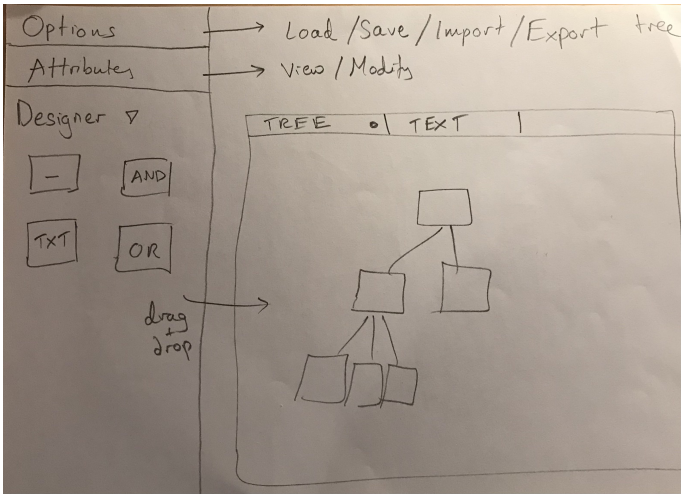


Figure 3.2: Tab-based mock-up

dedicated node. The rationale behind this is discussed in more detail below in Section 3.2.2. With this information as metadata, there would not need to be And/Or nodes in a drag-and-drop menu, and the only remaining options would be an attack node and a comment box (and maybe possibly an edge object too). At this point, there is little need for such a menu, and this interface becomes needlessly difficult to interact with. draw.io makes effective use of such a layout because of the large quantity of available shapes to choose from, and because it acts as a more freeform diagramming tool, where shapes can be rearranged and pulled around, and edges connected at multiple points.

The Org Chart editor, on the other hand, offers no control over node placement. When adding and removing nodes, the tree automatically adjusts itself to accommodate the new layout in a human-readable way, taking advantage of one of the mxgraph tree layouts, which are discussed in more detail later. Nodes can still be reshaped to fit the contents better. Balanced and unbalanced trees can both be cleanly represented. Figures 3.3 and 3.4 show the example attack tree from earlier recreated in draw.io with the drag-and-drop menu and with the mxgraph Org Chart demo, respectively. The mxgraph node overlays can be seen, offering options to add children and delete subtrees at each node. In the draw.io window, the left side shows the menu and its wide range of shape nodes and objects, which an attack tree does not require.

3.2.2 Representing And/Or nodes

How And and Or nodes would be represented and stored within the tree was also considered ahead of implementation. Namely, should such nodes be stored as individual nodes within the tree, or should they be part of a node's metadata, stored alongside the attribute values? Furthermore, how would

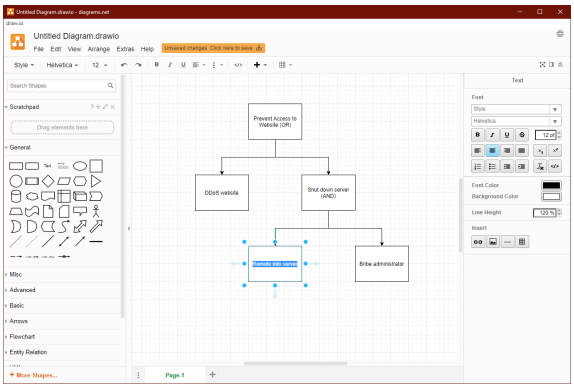


Figure 3.3: draw.io example tree

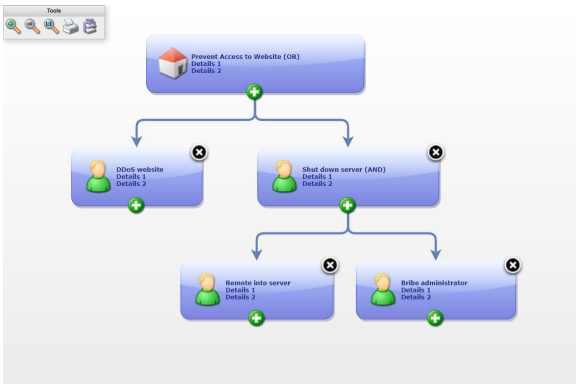


Figure 3.4: mxgraph example Org Chart tree

corresponding edges be represented in a way that clearly conveyed this?

Having And/Or information stored within the node's metadata would lead to a less complex overall tree, with more complex individual nodes as a trade-off, and visa versa. Dedicated nodes raise the number of nodes in the tree, but other nodes don't need to store and represent this data, which could make reading the tree easier. The example attack tree with dedicated nodes is shown in Figure 3.5.

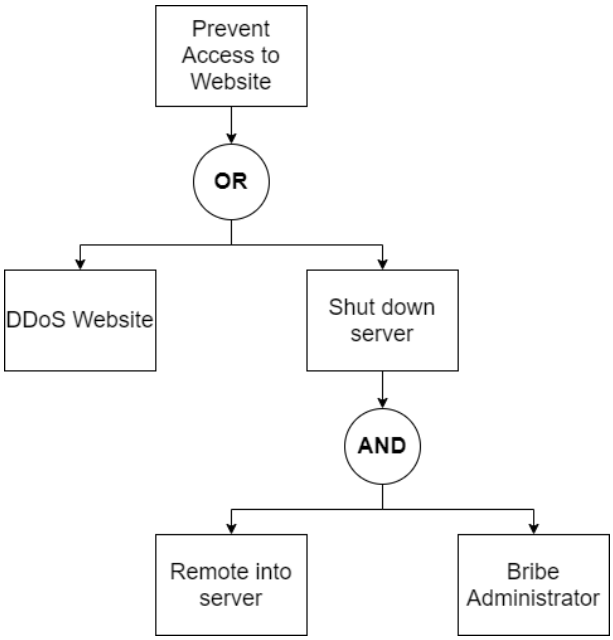


Figure 3.5: The example tree with dedicated And/Or nodes

Node type can also be represented by how the edges are styled. ADTools draws an arc between the edges to the two outer children for And nodes and nothing for Or nodes, as can be seen in Figure 1.3. Without knowing what they represent beforehand, this is not helpful to someone viewing the tree. Schneier's example tree does the same, although the arc is at least labelled 'and'.

Other ways of styling edges were considered - primarily, dashed edges or coloured edges. These were rejected for the sake of visibility and accessibility- dashed lines can be hard to differentiate from regular lines on a smaller graph, and coloured lines could be difficult for colourblind users to differentiate, in addition to ceasing to carry information if the tree is presented in grayscale. Edges would therefore best be represented identically, and And/Or information shown as part of the nodes themselves.

The implementation of storing And/Or values as part of node metadata was chosen over dedicated nodes after seeing the overlaid graphics implemented on the mxgraph Org Chart tree. Having an overlay showing this information would combine the visibility of dedicated nodes with the simplicity of using node metadata.

3.3 Building an Embedded Tree Editor

At this stage, the internal representation and graph view of the attack tree could finally be constructed together by looking into the different tree models that mxgraph provides and selecting the most appropriate option. Attack tree functionality could then be built onto this model. mxgraph renders its trees within a container, so the initial contents of the tool was simply this container and the rendered tree within.

3.3.1 mxgraph Tree Patterns

To instantiate an mxgraph graph, the command `var graph = new mxGraph(container);` is used. The `graph` object contains the tree, its vertices and edges, and all associated metadata and functions. One such functionality offered by mxgraph is the ability to bind a layout to the graph, which contains a set of rules telling the renderer how to display and update the graph when modified. Each layout is designed to fulfil a different purpose, and there are several that could be effectively implemented for an attack tree. The documentation for layouts is found at https://jgraph.github.io/mxgraph/docs/js-api/files/layout/mxGraphLayout-js.html.

Some layouts could be discounted easily, such as `mxCircleLayout` for rendering graphs with nodes forming a circle, and `mxPartitionLayout`, for partitioning a space into smaller and smaller cells. This left a list of layouts to be considered more thoroughly:

- `mxGraphLayout` is the base implementation for layouts that others build off. It contains functions for moving individual nodes to points. With this layout, making the tree readable and with nodes that flow down a tree and expand to give room for siblings would be left up to a manual implementation.
- `mxRadialTreeLayout` is a layout for creating radial trees, which display nodes radiating outwards from a central root node. This way of representing trees has not been discussed yet, and could be applied to drawing attack trees.
- `mxFastOrganicLayout` implements a fast organic layout algorithm. Nodes do not need to be manually positioned for a clean graph without overlaps to be drawn, but there are no constraints on drawing nodes in a tree, and as a result it may produce trees that are hard to follow.
- `mxHierarchicalLayout` implements a hierarchical graph. This is an excellent candidate for easily drawing trees in a way that a reader will be able to follow, with a root node at the top and subsequent levels of children below. An example of this layout with the Java JGraphX implementation can be seen in Figure 3.6.
- `mxCompactTreeLayout` is the other layout focused on tree editing. It supports the dynamic reshaping of the whole tree on adding and removing nodes in a way that produces a compact but human-readable tree, using an algorithm presented by S. Moen in 1990 [11]. Moving nodes is discouraged as it contradicts this compact displaying algorithm.
- `mxCompositeLayout` allows for composition of layouts in a tree. Two layouts are first defined then composed in a parent layout. Doing so allows for building far more complex trees than would otherwise be possible without implementing a custom layout.

The layouts most strongly considered and experimented with were the `mxHierarchicalLayout` and `mxCompactTreeLayout`, due to their well-suited implementation and available functions. While initial work was done with the hierarchical layout, the compact tree eventually proved a far better fit once overlay functions were added for modifying the tree, and this was used for the rest of development. This did cause issues in later development due to the way this layout handles node parenting, which is discussed further on in Section 3.3.4.

Radial trees were not used as there was no guarantee that a clean, readable tree editor could be built using one. The time investment to attempt doing so could be costly, and result in creating attack trees that are still not as easily readable as a hierarchical tree.

3.3.2 Compact Tree Implementation

The `graph` object was created and assigned a `mxCompactTreeLayout` with some parameters for specifying distances between nodes and levels, and for rendering edges as straight diagonal lines. A root node is

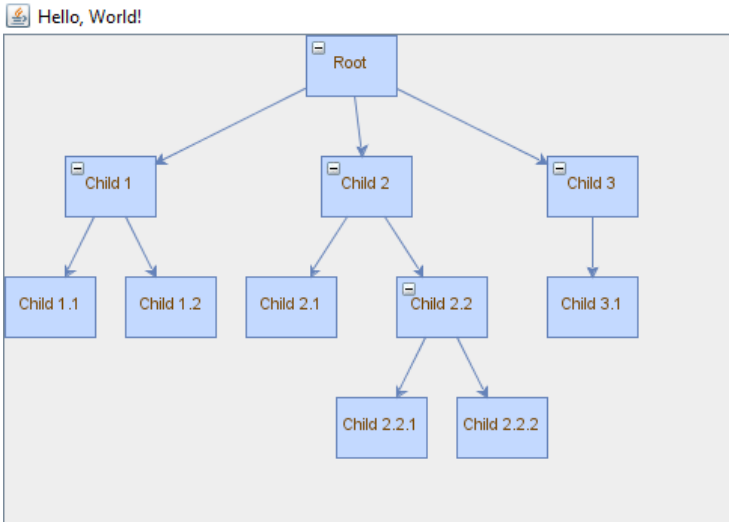


Figure 3.6: JGraphX example hierarchical tree. From [4]

statically added as a starting point to build off. Edges are set to be unselectable, and cells are set to be unable to be moved, to enforce the compact tree structure. The `graph.panningHandler` is also enabled, which supports 'panning' around a tree by clicking and dragging.

The most primitive tree editing functions are adding children and deleting subtrees. When nodes are created, they are given overlay buttons which, when clicked, perform these operations; however, this was not implemented in the first stages of development. Instead, a *context-sensitive menu* was used.

Context-Sensitive Menu Interaction

This is the menu that appears upon right clicking, showing available options. mxgraph provides functionality to capture right-click events inside the graph container, and to show a menu with different options depending on what was right clicked, which could be nodes, edges, or neither- hence the term 'context-sensitive'. The snippet in Listing 3.1 shows how this was done. The `CreateContextMenu()` function takes in the cell parameter: this is the node that was right clicked on (or is null, if no cell was selected). The method looks at this cell and calls `menu.addItem()` to build a context-sensitive menu based on this. Menu items include a function that is called when that menu item is clicked; this function is passed the cell object.

The end result of this is that a cell can be right clicked and an option to add a child to that node selected. The chosen node is then available to build off.

```
1 mxEvent.disableContextMenu(container);
2 graph.popupMenuHandler.factoryMethod = function (menu, cell, evt) {
3     return CreateContextMenu(graph, menu, cell, evt);
4 };
```

Listing 3.1: Context menu overriding with mxgraph

Adding Children

The first tree editing operation is `AddChild(graph, cell)`, called by the context menu (and later, the overlay's click handler and list view's add child handler). Thanks to the choice of compact tree layout, adding children and having them be rendered below their parent and not overlapping with their siblings is easy. mxgraph updates its model within a `try..finally` clause and a pair of functions, namely `graph.getModel().beginUpdate()` and `graph.getModel().endUpdate()`. In this try clause, the graph default parent is used as the the parent vertex for the new cell, given by `graph.getDefaultParent()`.

The default parent is the 'master node' for a graph; it is not rendered and should not be interacted with by the user. To add the new child to its parent cell, an edge is then inserted into the graph connecting the two vertices, and on ending the update the graph is refreshed and the new cell is drawn. Having the parent of new cells be the default parent is a quirk of the way that the compact tree layout

is implemented. The default node is the parent of each cell, and this cannot be modified later, or nodes will not render as part of the tree.

Other functionality is performed in the AddChild method relating to constructing a new child. This includes setting up attributes, adding overlays to the new cell, and setting up the 'geometry' of the cell, which handles properties such as the height and width.

Deleting Subtrees

The other key tree editing operation is `DeleteSubtree(graph, cell)`. Earlier in Section 2.2, the two approaches to doing this were discussed; namely, deleting the entire subtree at a cell, or setting each child of the cell to be siblings instead, preserving the structure beneath. The prior implementation was chosen since it felt a more natural approach to tree interaction. This was done by traversing the tree from the current cell with an mxgraph traversal function, building a list of all the cells in the subtree, and then calling another mxgraph function to remove those cells. This is seen in Listing 3.2. Note the first line, which is an extra check to ensure that the root node is not deleted; if this happens, there would be no more nodes to build off. However, this option should never be presented to the user, as this check is also made in generating the context sensitive menu options, and when generating the overlays for a cell.

```
1 if (cell.getId == 'root') return;
2
3 // Gets the subtree from cell downwards
4 var cells = [];
5 graph.traverse(cell, true, function (vertex) {
6     cells.push(vertex);
7 });
8
9 graph.removeCells(cells);
```

Listing 3.2: Deleting a subtree at a node

Once again, there is other functionality in this method not shown here: attributes must be recalculated in the parent of the deleted cell, and a check is made to see if the parent's And/Or overlay should be removed.

At this point, a user can load the page, and in a container a tree will be created with one root node. The user can add children to the tree and delete subtrees, and they can pan around the tree. They can also zoom in and out with context menu options that call more inbuilt mxgraph functions. Nodes in the tree are complex `mxCell` objects, but the only data the user can interact with is the `value` property, which is just a string containing the label that node displays. The compact tree layout by default supports double-clicking nodes to change their label.

The next step was to convert this tree into an attack tree, by adding support for storing attributes and And/Or values.

3.3.3 Implementing Node Metadata

Storing attributes is a problem that requires a predetermined approach to prevent a solution from becoming over-complicated and ineffective. Each node in the tree (or at the very least, each leaf in the tree) would have to somehow store and access a value for each attribute. Furthermore, these attributes could take different domains, as described earlier. Some might be numeric, some boolean, etc.

Initially, this was going to be done by storing a complex JSON object that would allow for nodes on the tree to access the values associated with that node ID. These IDs are automatically generated by mxgraph as ascending values, with the exception of the root, which has the ID *root*.

However, after implementing a static demonstration version, it became clear that this was not the best solution. Storing the attribute data in another object spreads out where the tree is stored, and another datastructure would be needed anyway to hold the attribute metadata: the propagation rules, default values, domains, etc. Fortunately, the *mxCell* documentation at https://jgraph.github.io/mxgraph/docs/js-api/files/model/mxCell-js.html notes a way of implementing custom attributes in the exact manner required by storing cell payload values as XML nodes instead of strings. These XML nodes can contain any number of values in a K-V datastructure, and these values can be from different domains.

Doing this simply requires creating a global mxgraph XML document. The document can then be referred to in creating a new XML element, which is then referred to when creating a new node in the

tree. The new node is now an XML node, and attributes can be read from and written to with the `cell.setAttribute()` and `cell.getAttribute()` commands.

Other Attributes

Using XML provides a very effective solution to the problem of storing attributes in the tree for each node. As well as attack tree attributes, two other properties would have to be assigned a value in the node XML at the time of node creation: the And/Or value and the cell's label.

- And/Or values could be held as attributes in this format, solving the issue of how to internally store this property. The attribute was given the name `nodetype`, and when new cells are created they are given the node type of 'Or' as a default value.
- Cell Labels had to be also represented with this since the XML element now replaced where it was once stored. The `label` attribute was also given a default value, but extra work had to be done to utilise it in displaying the graph. Fortunately, the mxgraph documentation provides a function that overrides the inbuilt `graph.convertValueToString` function called when rendering, although this was later replaced with a custom redefinition of the `graph.cellRenderer.getLabelValue` function to also display attribute values on nodes.

3.3.4 Connecting Attributes

Attribute Dictionary

The other half of storing attributes was in the use of a JSON dictionary, `attributes`, which stored information about the attributes currently being represented by the attack tree. Since attribute names had to be unique for the XML nodes to reference their value, names were also used as keys for the attribute dictionary. The value at each index was an object containing all the properties of an attribute; its name, domain, default values, and rules. This metadata changed several times over the course of development as attributes were refactored, but originally a pair of maximum and minimum values were also included. They were removed for being too focused on numeric attributes, although restricting the range of values was conceptually used in later introducing a 'unit interval' domain.

Storing an attribute's rules - functions for how they combine And and Or nodes - allowed for easily using the same propagation algorithm for all attributes and for both types of node. This algorithm was the final element required for attributes in a minimal attack tree editor.

Implementing Propagation

Propagation of attributes was implemented using Algorithm 2.1 as a reference. To call this method, two test attributes, cost and probability, were statically added to the tree when creating it. Options were added to the context-sensitive menu to edit these values for a leaf cell. For cells with 2 or more children, a menu option to toggle the cell between And and Or values was also added. Both of these methods, as well as adding and deleting nodes, would then call the function to recalculate attribute values for that cell, and propagate any changes up the tree.

Upon implementing this, there was an issue with how propagation occurred up the tree. Finding a cell's parent is simple with the mxCell `getParent()` function. However, the `mxCompactTreeLayout`, as mentioned earlier, adds new nodes to the tree as children of the graph's *default parent*, then connects the new node to it's tree parent with an edge. Thus, calling the `getParent()` function will not return the cell's parent in the tree; in fact, nowhere in the cell is the actual parent referenced. Trying to redefine a cell's parent after creation causes it to then not render as part of the tree.

So, a workaround was needed. The only element connecting the two cells is the edge between them. Edges in mxgraph are technically just a type of vertex. Because of this, every vertex in a graph has a pair of pointers to other nodes, called *source* and *terminal*. These are meant for edges to store the source node and the terminal node, and are not meant to be used or defined in cells- but they can be. When a new cell is added, the *source* value for that cell is set to the parent cell in the tree. This only works because a cell in this tree only ever has one parent, with the exception of the root node, which has no parent.

The other half of the workaround is a way to generate a list of all the children belonging to a cell. The same method does not apply, since multiple nodes cannot be stored in the *terminal* property. Therefore, the function `GetChildren(cell)` is defined. mxCells do store a dynamically updated list of all the edges

connected to that cell. This list is iterated through, and each edge is checked to see if it has its source as the cell in question. If so, the terminal cell of that edge must be a child. A list of children can therefore be generated, albeit with some extra difficulty.

Combining the elements of creating XML nodes, setting the source value, adding overlays, and defining and propagating attributes can be seen in the `AddChild(graph, cell)` function, which is included in Appendix B. The result of this was that the attack tree could now store, propagate and represent attributes. Creating new attributes was not implemented, as it required building a complex interface, and the UI design for doing so had not been fully realised at this point. Instead, a parser was next designed and fed into building a textual representation of the tree: the list view.

3.4 Textual Representation

3.4.1 Generating the List View

Parsing the graph was done by recursively building a list of strings, where each element in the list is a string holding information about one node: the label, each attribute name and value, and if necessary, the And/Or value. Also included is the 'path' to that node from the root. This is a series of numbers, where each number indicates which index of child that node is. For example, a path of "1.2.3.1." means the node located by going to the root node, then the second child of the root, then the third child of that node, then the first child of that node. Note that this applies to every cell in the tree, not just leaves. The construction of this path was the driving factor in designing this function to be recursive, since when parsing children this path can be re-used and built up, saving extra calculation. The parsing of the tree is done in a 'depth-first' traversal. This means that the algorithm traverses the whole tree from the root, going as far as possible into each branch, before backtracking the minimal distance before doing the same again. This builds up a list order in a manner as seen in the example list view, Figure 2.5.

After generating this list, it is passed to a separate container for displaying the list view, which initially was displayed alongside the graph view in a manner similar to the slider UI design. The `Load_textual_graph(cells_list, graph)` function takes this information and builds a HTML list of the tree, with each successive `li` item displaying the string for one corresponding node. By adding some CSS, alternating nodes are coloured slightly differently to make reading the list easier. Figure 3.7 shows this list view and its equivalent graph view. Note that each successive layer of child cells is indented to the same level, allowing easier visualisation of how the tree is formed.

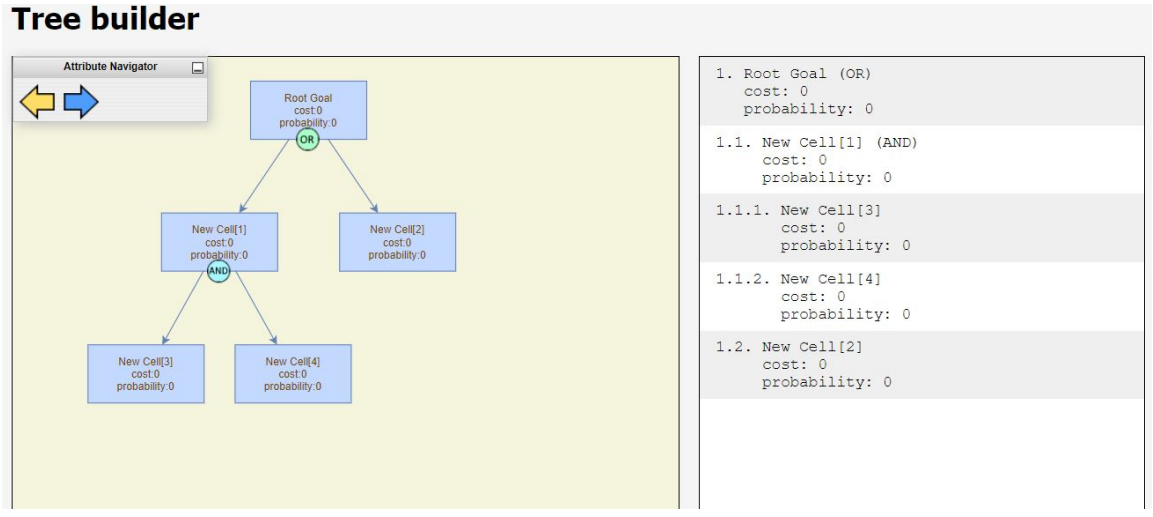


Figure 3.7: A graph and corresponding list view

3.4.2 Editing the Tree from the List

Now the list view accurately represented the mxgraph tree, but one of the aims of the project was to allow a user to fully design an attack tree from either perspective. This meant that within the list view, functionality had to be added to allow for adding, deleting, and editing nodes.

To implement this, each HTML list element was given an `onclick` handler that, when clicked, would toggle showing a set of buttons for these three operations. Adding a child would call the `AddChild()` method defined earlier, and deleting the subtree would do the same for its corresponding method. For editing a node, however, a more complex function was built, called `TurnListIntoEditableForm(li, cell, graph)`. This function wipes the content of the list element and replaces it with a form for updating the cell's label, And/Or value, and each attribute value. Only leaf nodes can have attribute values edited in this form. On submitting the form, every value is validated, including attributes being within domain boundaries and labels being non-empty strings. Then, the cell is updated, and changes are propagated up the tree. Figure 3.8 shows both the selected operations for a cell and the editing form for another. Note that the root node has the 'Delete Subtree' option greyed out and disabled, to prevent deleting the root.

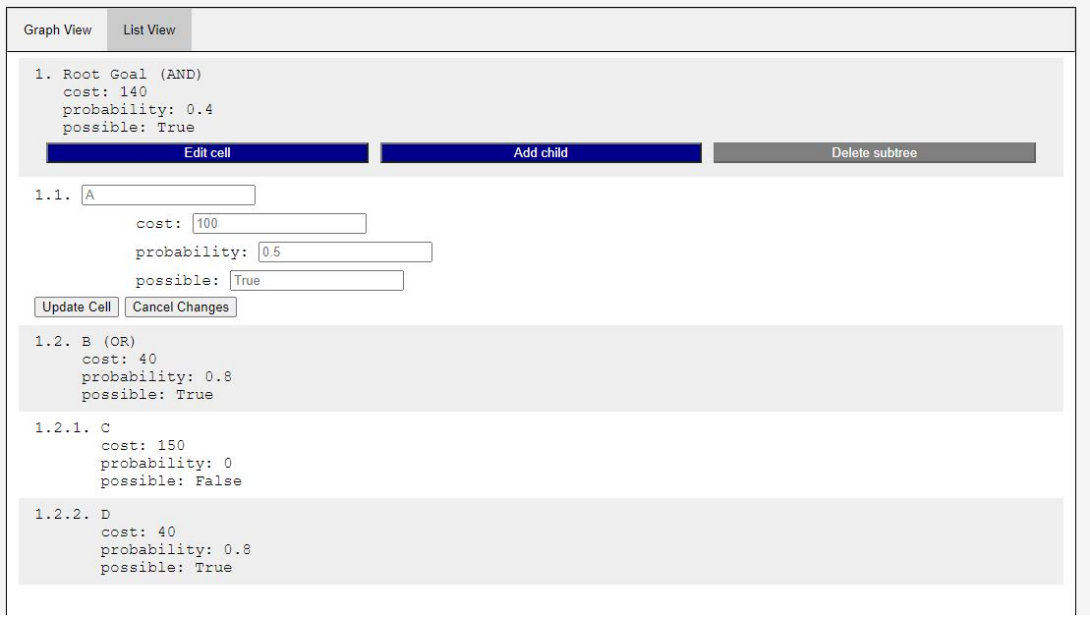


Figure 3.8: The list view, showing selecting nodes and editing nodes

This implementation worked smoothly with the corresponding graph view, with both updating each other in real-time. However, the source code for manually creating the HTML lists and forms was verbose and could benefit from refactoring or templating. With the two views working together and allowing for building a primitive attack tree in a private session, the next step was to develop the other main feature of the tool - collaborative editing.

3.5 Client Communication

At this point, the choice between socket.io and y-js for an initial implementation was made. Since y-js is built off socket, and is considerably more fine-tuned and requiring a far greater amount of work, socket.io was used to transmit the graph to a central server, which would then distribute the updated graph out to clients. This was in line with the agile methodology, which aims for a build-able product at the end of every sprint, something that implementing CRDT datatypes with y-js could feasibly be a far cry from.

3.5.1 socket.io

Setting up a socket.io messaging demo was simple; socket.io 4.0 is available as a Node.JS package, and the clientside version is loaded by running a JS script on the socket website. Therefore, a function `EmitTree()` was constructed, which would take the graph and the attributes and send them to the server. This was called from a range of functions that modified the graph in some way, such as all those that cause a recalculation and propagation of attributes. Other operations causing a socket emission included updating a node's label, adding/removing attributes, and loading trees.

As mentioned earlier, socket.io supports sending complex and composable datatypes, such as lists and objects. Therefore, the first attempt at transmitting the tree to the server built an object containing

simply the entire `graph` object and the entire `attributes` object. The server would then update its stored model and forward it to other users. There were two issues with this implementation. Firstly, object functions cannot be sent in this manner; socket cuts them out of the payload in a pre-processing step. This meant the attribute rules were not transmitted. To fix this, the attribute rules were first converted to a string and this was sent, then the receiver would convert these rules back into functions with the JavaScript `new Function()` constructor. However, this was a rudimentary fix that would require reworking.

The other issue caused by this approach was that the mxgraph `graph` object is complex, composed of many objects, values and functions. In particular, cells have attributes that store edges that connect to them, and edges have attributes that store the cells they connect to. This is a feedback loop of pointers. Socket's pre-processing attempts to parse objects it sends: this loop then causes a recursion error. Combined with the vast amount of unnecessary mxgraph metadata values being sent, there was a strong need for using a custom-defined tree for transmission. The list parser was considered, but it was written to parse the tree into strings, and parsing back again would take a greater amount of work than simply creating a minimal JSON tree object and unpacking it.

```
1  // Given a graph, parse it into a JSON object.
2  function GetTreeData(graph) {
3      var tree_data = {};
4      var cells = [];
5      TraverseTree(graph, function (vertex) {
6          var cell = {};
7          cell.data = {};
8          var vertex_Attributes = Array.prototype.slice.call(vertex.value.attributes);
9          for (var i = 0; i < vertex_Attributes.length; i++) {
10             cell.data[vertex_Attributes[i].nodeName] = vertex_Attributes[i].nodeValue;
11         }
12
13         cell.id = vertex.id;
14         cell.parent = null;
15         if (vertex.source != null) cell.parent = vertex.source.id;
16         cells.push(cell);
17     });
18     tree_data.cells = cells;
19     tree_data.attributes = attributes;
20     return tree_data;
21 };
```

Listing 3.3: Parsing an attack tree for socket.io

Listing 3.3 shows the final version of the function for parsing the socket.io emission object, called by `EmitTree()`. Note that the `attributes` object is global. The `GetTreeData()` method traverses the tree and performs a function on each vertex. The function constructs a JavaScript object with the data needed to rebuild each cell: the id, it's parent, and the data from the XML map (which includes the label and nodetype). Converting the XML data to a clean JS object itself is done in lines 8-11; the complexity here comes from mxgraph storing the XML data as a `NamedNodeMap`, which must be converted into an array of objects. This array is then iterated through, and the attribute names and values are pulled out and stored as a K-V pair in the data object. Finally, the cell object is pushed to the list of cells. The final `tree_data` object contains the cells list and the attributes object, which is the minimal information needed to represent the whole graph. An example `tree_data` output from parsing a simple tree with one attribute is seen in Figure 3.9.

3.5.2 Group Editing

With transmission working, groups of users could then be implemented serverside. The aim was to allow users to create groups with an identifying code. Other users could then use that code to join the group and seamlessly join that editing session, loading the same tree and gaining equal and full editing permissions.

Rejecting Redis

The server up until this point had just been serving HTML pages with Express.JS and handling socket events. Groups were originally intended to be stored on a permanent Redis instance, so the Node.JS package was installed and a Redis client was set up, and worked both locally and on the Heroku instance. However, developing with Redis led to issues with the project becoming more complex than called for,

```
{
  "cells": [
    {
      "data": {
        "label": "Root Goal",
        "nodetype": "AND",
        "cost": "100"
      },
      "id": "root",
      "parent": null
    },
    {
      "data": {
        "nodetype": "OR",
        "label": "Attack A",
        "cost": "100"
      },
      "id": "9",
      "parent": "root"
    }
  ],
  "attributes": {
    "cost": {
      "name": "cost",
      "desc": "expected cost of the attack",
      "domain": "POSITIVE_INTEGER",
      "default_val": 0,
      "display": true
    }
  }
}
```

Figure 3.9: Example socket.io `tree_data` payload

when a temporary storage solution would be sufficient for development, and permanent support could be added at a later date. So, Redis was scrapped and the new model instead would use a dictionary on the server with group codes as keys and an object containing the group data as a value.

Members

Socket events were set up for creating and joining groups. When receiving a socket event, the server has access to an id that uniquely identifies that socket instance. This same id can then be emitted to at any point. So, when a client requests to create a group, they send the server a proposed group 'key'; a user-defined string to identify that group. The following steps are taken by the server:

- Is the key already in use by another group? If so, return a `KEY_IN_USE` acknowledgement. Otherwise, create the group.
- Create a `group_data` object to store everything about the group. This is assigned two components:
  - A list of member socket ids, `members`. This at first contains only the socket id of the group creator.
  - The attack tree object in the form as seen in Figure 3.9, `tree_data`. At the time of creation, this contains only a flag indicating the tree object is uninitialised.
- Add the user to the `clients` dictionary. This is a dictionary with id keys and group code values, and is used to lookup the group a client belongs to quickly and easily.

- Return an `OK` acknowledgement to confirm the group creation.

When joining a group, a similar procedure is followed:

- Is the key not in use by a group? If so, return a `KEY_NOT_FOUND` acknowledgement. Otherwise, join the group.
- Push the joiner socket id to the list of members.
- Add the socket id and group key to the `clients` dictionary.
- Emit to the joiner an `OK` acknowledgement to confirm joining the group, and also the `tree_data` master copy in a separate emission.

Receiving an acknowledgement from the server even if a key is rejected is important for creating a responsive tool that tells the user what the result of their action was. If they tried to join a group with the wrong key, they should get an error message specifying this.

When a user disconnects, socket catches the event. The `clients` dictionary is used to find the group key of the group that user was in. The leaver can then be removed from the list of group members. Furthermore, a group that is reduced to zero members is also deleted to prevent a server run for an extended session from becoming bloated.

Graph Updating: Execution Flow

Putting these components together ensured that when a group member performs a modification on their local version of the tree that warrants an update, the `EmitTree()` function is called. Group keys are stored for users on the browser's `sessionStorage`, which itself acts like a dictionary for holding values in a browsing session. The tree is parsed into the `tree_data` object, to which the group key is then appended. This object is sent to the server via socket.

The server catches this message, and forwards it to the handler for a tree update event. The group key is validated, and the server's 'master copy' of the tree is updated to the new one. The server then iterates through the members and sends each one the new tree.

Back on the clientside, each member receives a socket message containing this tree. They call a function to completely rebuild the tree from scratch with the new version, and also update the attributes to the new version. Remaking the tree from scratch is another trade-off; it saves having to find any changes to the tree and try and surgically update only that section, but it leads to a decent amount of unnecessary rebuilding of the same components time and time again.

3.6 Attribute Navigating

With tree editing working, the list view updating dynamically, and collaborative editing and groups implemented, the final section to be implemented was an attribute control window. Several key attribute functions still needed to be implemented:

- Viewing a list of attributes
- Deleting attributes
- Creating/ editing attributes
- Editing attribute values in the graph view
- Controlling what attributes are displayed in the graph

This final feature was originally implemented with a `mxToolbar` instance, which allows for a toolbar inside the mxgraph container. However, it offers very little room to build more complex features; for example, no vertical elements can be stacked, and no text inside the toolbar. A custom implementation was attempted, but mxgraph does not render containers within it's graph container very well, if at all. As a result, this functionality would be best added in its own window alongside the graph view. It was initially implemented as a tab in the same space as the list view, but eventually the list view was moved to the left and the two ways of viewing the attack tree became tabs sharing the same area on the screen, with a dedicated space for the attribute navigator on the right.

3.6.1 Attribute Wizard and Listing

The attribute navigator was divided into three spaces: the top would contain a list of attributes currently associated with the tree, each with checkboxes to toggle rendering that attribute's value in the graph view. Each attribute in this list also has a button to remove that attribute from the tree and hovering the mouse over the attribute displays the attribute description in a tooltip, if there is one. There is also a button to add a new attribute, which instantiates the next window: a form for creating new attributes. The bottom window would contain the attribute properties of the currently selected cell, with a form for updating them in an identical manner to that used in the list view attribute editor.

The final version of the attribute editor is shown in Figure 3.10.

3.6.2 Refactoring Attributes

Before an attribute creation wizard could be created, attributes first had to be refactored to be more usable. Having users define their own rules could be done with having users define their own JavaScript, but trying to prevent exploits from this would be difficult. The easiest solution would be to not have users define their own rules, but instead to select a domain from a predefined static list, meaning that each domain would then have its own rules. This also solved the problem of functions not being able to be sent over socket.io without messily converting to strings and then evaluating them (which, in retrospect, would have allowed malicious users to send rules and run custom code on a group member's machine).

All attribute values would now be internally stored as a float. The tree would interpret and render attributes differently depending on their domain; for example, the `TRUE_FALSE` domain displays and accepts only true/false values, but internally stores values as 1/0. The `UNIT_INTERVAL` domain stores a value between 0 and 1 inclusive, allowing for implementing probabilities, and is set to reject values outside of this range. This refactor was a compromise; it reduced the potential of truly custom attributes, but made the implementation feasible within the scope of the project. Overall it allowed custom attributes to be effectively implemented in some form.

3.7 Final Steps

At this point, a considerable deal of time was spent on bug-fixing and styling the graph. Small features were added to improve usability, such as an `mxKeyHandler` object to catch pressing the 'Delete' key and to delete the selected subtree. Larger, more important features were also added, however:

Uploading and Downloading was a key feature intended from the beginning to support the goal of developing an educational tool. The original intention was to either download/upload trees as XML or YAML, and mxgraph does in fact have support for downloading a graph as pretty XML. However, this does not include the attribute metadata, although it does include cell geometry, graph metadata, edge data, etc. Thankfully, the socket parser has already been written to reduce an attack tree to its minimal form as a JSON object. The same code could be re-used for getting a file to download and for reading and loading an uploaded file.

Loading an example tree was also implemented for the sake of testing and modifying the tree. This sends a `GET` request to the server for the example tree, and treats the response JSON as an uploaded file. The result is easily being able to open a session and load a tree complete with attributes, values, and nodes to experiment with.

A Help and About page were both added to increase the usability of the program. For the help page, each functionality and use-case of the tool was considered and a set of instructions on how to use it was written, together with images. Despite this, these pages were simple and rudimentary.

With this, the development cycle effectively came to a close; the Heroku server was updated to run the most recent build, which supports group and private sessions, graph and list view editing, attribute creation, and viewing and propagation of attack trees, with the ability to export a tree into a JSON file, then import it at a later date. Despite this, there were many areas where compromises had been made, or where a design decision had been made, either by choice, or through being forced to. Choices had been made in the cases of using socket.io over y-js, or a temporary group storage over Redis. A forced approach was seen with mxgraph not allowing rendering containers within a graph container, which forced

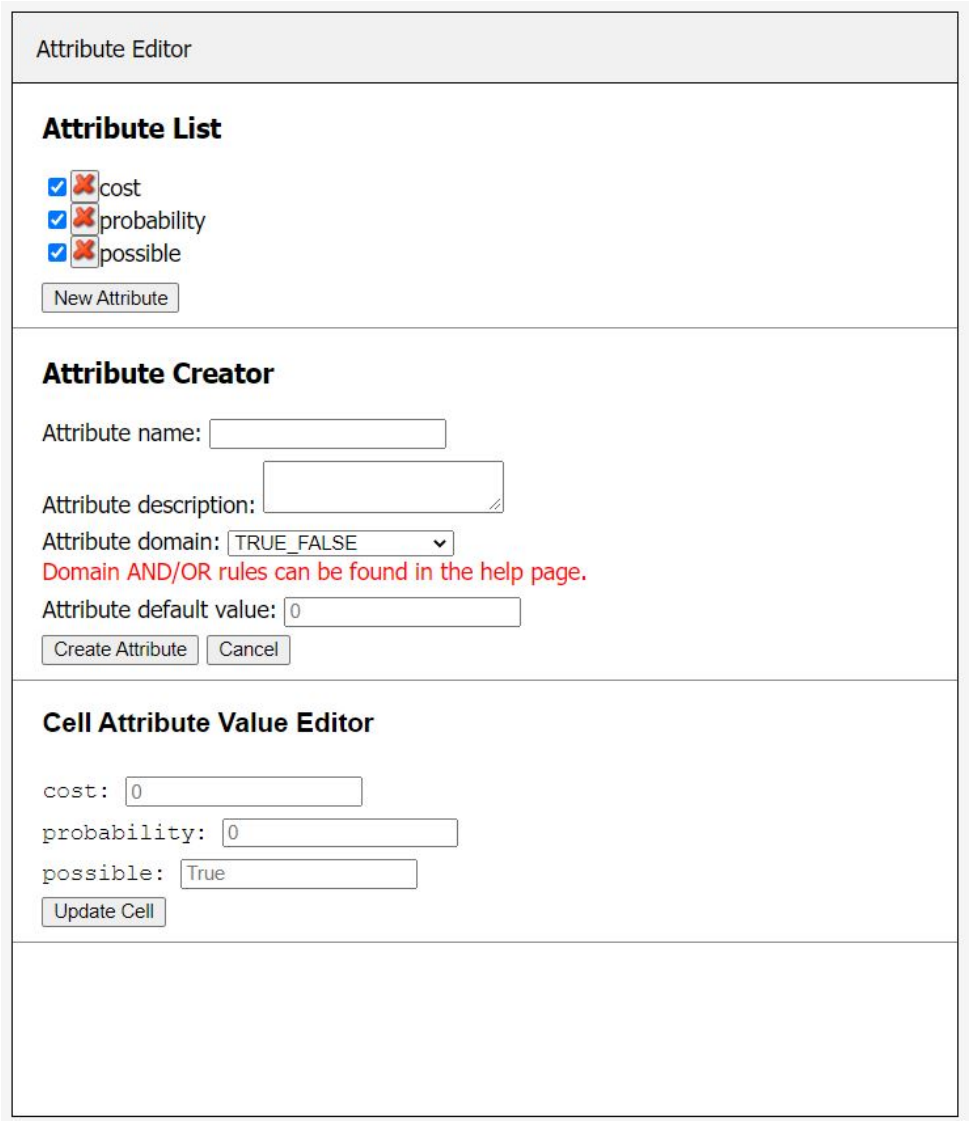


Figure 3.10: The Attribute Editor in full

the development of the Attribute Editor window. There were many elements of the design that could be reflected upon, or evaluated in user-testing.

Chapter 4

Critical Evaluation

An interactive tool will always be open to critique and evaluation. Some is objective; one library may be slower or more unstable than another, or likely to become deprecated sooner. Other critique, particularly in designing the UI, is more subjective. The use of graph representations in their own tabs is an example of this, or hiding and revealing HTML based off what node in the tree is currently selected.

In order to perform a thorough evaluation, every facet must be analysed as objectively as possible. In the case of some decisions, this will mean providing sufficient rationale to choose one method over another. For evaluating the tool's usability, it will be important to gather quantitative data as well as qualitative feedback, and to evaluate with reference to this.

4.1 Design Evaluation

In this section, the design of the tool is evaluated; the choice to develop with JavaScript in combination with Node.JS, the choice of libraries used, and a reflection on the final UI design. Each had an impact on the development of the project and the final outcome, and each required making a set of decisions informed by the content of previous chapters.

4.1.1 Choice of Language

Implementing Atree in JavaScript was influenced by a desire to work in one language for both serverside and clientside development, and to work in a lightweight agile manner (in this case, by choosing scripting languages). For the server-side, another scripting language that could have been used was Python, which would also have allowed for the use of a great range of libraries.

Overall, the choice to use JavaScript greatly benefited development more than it hindered development. Framework support for JavaScript made hosting a server easy, whereas with a language such as Java, even with the Spring Framework, web development is relatively complex. Defining and passing functions dynamically was a functionality that proved extremely useful in development, for overriding mxgraph's functions, and creating callback handlers.

JavaScript allows for asynchronous execution with promises and async functions. This is especially useful in clientside development, where the page loads asynchronously and makes requests the server that could be acknowledged at any time. With socket, reliance on `GET` and `POST` requests was avoided, and there became little need for asynchronous functions. Callbacks in form responses were written as part of the form in order to share the local scope. Scripts that need to be executed after the page has loaded are done using `onload()` parameters in the HTML body. As a result, there are no JavaScript `async/await` functions in the Atree code. This is something that could be improved on, as modern JavaScript development favours this pattern due to allowing for better performance, as execution no longer halts on expensive functions.

Dynamic and Static Typing

An issue that using JavaScript semi-frequently caused came from it being a dynamically typed scripting language. Dynamic typing means that the interpreter assigns a type to variables at runtime, rather than the programmer assigning types in source code. This makes developing faster and simpler, which is

important in a scripting language; compare creating a list in C# with `List<int> example_list = new List<int>();`, against in JavaScript with `var example_list = [];`.

The downside to doing this is that the programmer loses control over variable typing. This affected the development of Atree at several points. In particular, when moving values around from being stored in different places. While socket can handle sending integer values in the objects it emits, mxgraph's XML nodes can only store strings. When pulling values out for attribute propagation, they had to be individually parsed as floats every time. This made implementing different attribute domains far more difficult; a domain-specific function would be needed to parse the string attribute into the correct type before propagation, which would have required transmitting and evaluating another function string with socket. Validation of attributes was made more complex for the same reason.

Furthermore, static typing helps write more stable programs. In most cases, adding two conflicting types should create an error and notify the programmer- even Python doesn't let you print a string and an integer in the same line without first casting it. Static typing allows the programmer to ensure that the correct data is being passed to functions, and that the results of functions are of the correct type. This need for static typing was great enough that it led to the development of TypeScript [18], which is a language that builds on JavaScript but with the addition of static typing to all variables. TypeScript can be run in most contemporary JavaScript frameworks, including Node.JS. Since TypeScript compiles to JavaScript, it can also run clientside. As a result, TypeScript would likely have better suited the project and its needs, although this would not have had a notable effect on performance; it would simply assist in more robust development.

Choice of Framework

The use of the Node.JS framework also proved to be a benefit more than a hindrance when developing. Having access to packages with ease and including them in the project with the project's `package.json` file encouraged experimenting with libraries in development. This was done when briefly experimenting between Koa and Express.JS as a web framework; Express was eventually chosen due to better integration with socket, as the serverside socket object can be instantiated using the http object that Express creates.

Node.JS also runs excellently on Heroku, having extensive Heroku documentation on running and configuring a Node project running on their instances. While other frameworks such as Deno can be run on Heroku, Node offered everything required for building and running the tool locally and on a permanent instance. Any other framework choice would largely be down to personal preference.

Running with Heroku was used as with one server instance, all development and hosting would be guaranteed to be within the free tier. The most viable alternatives, AWS Elastic Beanstalk and Google App Engine, have free quotas that could have been exceeded accidentally. Both these options also have their own documentation for running and configuring Node.JS applications, at https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/create_deploy_nodejs.html and https://cloud.google.com/appengine/docs/standard/nodejs correspondingly. The downside to Heroku comes from having to spin up an instance after making a long time without making a request, although in a teaching session this is not a factor that would be a large concern.

4.1.2 Evaluating mxgraph

Choosing to build a tree with mxgraph was a strongly premeditated choice, as discussed in the Technical Background. mxgraph with JavaScript had all of the functionality required and more. Being able to override the suite of functions, including those for rendering and refreshing, made customising the library to the tool's needs clean and stable. In particular, being able to store an XML payload node for storing attributes was incredibly valuable and made implementation far simpler. Similar functionality was seen in some other graph libraries such as Treelib, which allowed for a custom data value, but the mxgraph implementation required no custom attribute class to be defined within the node, or a wrapper over the tree to access it cleanly.

Overlays also presented a clean and customisable way of interacting with the graph in an intuitive manner that was well-suited to the project. Other context-sensitive interactivity, such as the right-click menu, was useful but had to be discovered before it could be used. Discovering the functionality of overlay options is given by their easily recognisable icons, and also in mousing over them and seeing the cursor change to a pointer.

Finally, an inbuilt graph panning system was a feature that also worked well in a tool for exploring and experimenting with attack trees. Other libraries, such as GoJS, also support panning; this feature helped to improve the usability of the graph view.

Compact Tree vs Hierarchical Layouts

One decision that had to be made in the early stages of development was the choice of mxgraph layout, which would expose different methods for rendering the graph and building a tree. This was narrowed down to the `mxCompactTreeLayout` against the `mxHierarchicalLayout`. Both are applicable to rendering a DAG, and both could have been a feasible choice for designing an attack tree. The former was chosen as editing the tree would then automatically trigger a fast algorithm for recalculating the tree layout [11]. Adding children and deleting subtrees are accommodated for well. The downside, as discussed previously, came from how all nodes in the tree are parented to a special 'parent node' in each mxgraph model, which is an immutable reference point. Since nodes are only connected by edges, traversal for the key propagation functionality had to be done by building a custom set of functions to get a node's parent and children. Part of this is done by assigning the node's parent to that node's `source` value, which is intended for use by edges and should not be used as a storage value. This could have been worked around in the same manner as the function to generate a list of children by iterating through a node's edges; however, the `source` value version requires no iteration and should be faster as a result, which motivated this decision.

4.1.3 Evaluating UI design

The final UI design is seen in Figures 4.1 and 4.2. The former shows the main page and acts as an entry point to the tool. From here users can create a private session that only they can work in, or they can create or join a group to work collaboratively. Also included are links to the help and about pages. The help page in particular was added when presenting users with some initial demonstrations to gather feedback, and there were common questions about interacting with the tool and wanting a way to see the scope of options available to the user.

The latter figure shows the editor. On the left is a window for viewing the tree, either in the graph view tab or list view tab. On the right is the attribute editor, with two of its three elements active. At the top, the group code can be seen to send to others, and also the options dropdown for extra functionality such as importing and exporting, as well as also containing links to open the help and about pages. These open in new browser tabs when clicked, so that the user does not lose their progress if editing a complex tree.

Evaluating these two UI designs is difficult. To do so without consulting or surveying any other users is just presenting one's own opinions, which is subjective and contributes little to a thorough UI evaluation. Analysis of the UI is key in evaluating the tool overall, since an interactive learning tool is only as good as it is usable and accessible. Complex and powerful functionality that a user cannot navigate through to using provides no benefit over an easier-to-use tool that does not have this functionality.

Therefore, a short but comprehensive survey was designed to analyse every key facet and functionality of the tool. The construction and thinking of this survey and its questions, as well as the results and their interpretation, is discussed in Section 4.3.

4.2 Implementation Evaluation

Implementation of this project led to another set of decisions, and also resulted in a set of issues with the final version that highlight areas requiring revision or another approach. These are discussed in this section. In particular, this section considers in detail the decision to not store group data in a persistent decoupled Key-Value store such as Redis, and the decision to parse and send the whole tree over socket in place of implementing OT or CRDT algorithms. Failure cases in these topics are also looked at, as well as briefly considering static implemented forms for attribute creation and updating, and better ways of handling this.

4.2.1 Key-Value Store Removal

As of the end of the development, a native JavaScript dictionary was used for storing group keys and their associated group data. Another dictionary linked socket ids to group keys for removing disconnected clients from their group. These dictionaries are part of the server process runtime. They are local only to that server instance, and are lost when the server is closed. Originally, a permanent Key-Value server, primarily Redis, was intended to be implemented for handle this functionality. Such a store is not local to process runtime and retains data after shutdown, allowing for extended group sessions. The downside

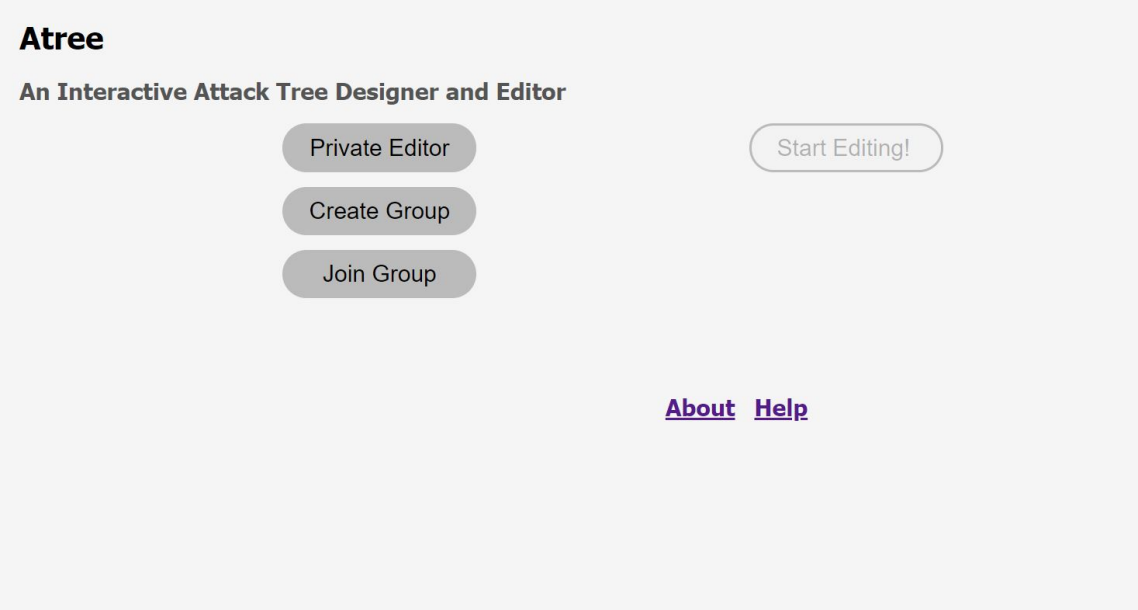


Figure 4.1: ATree Main Page

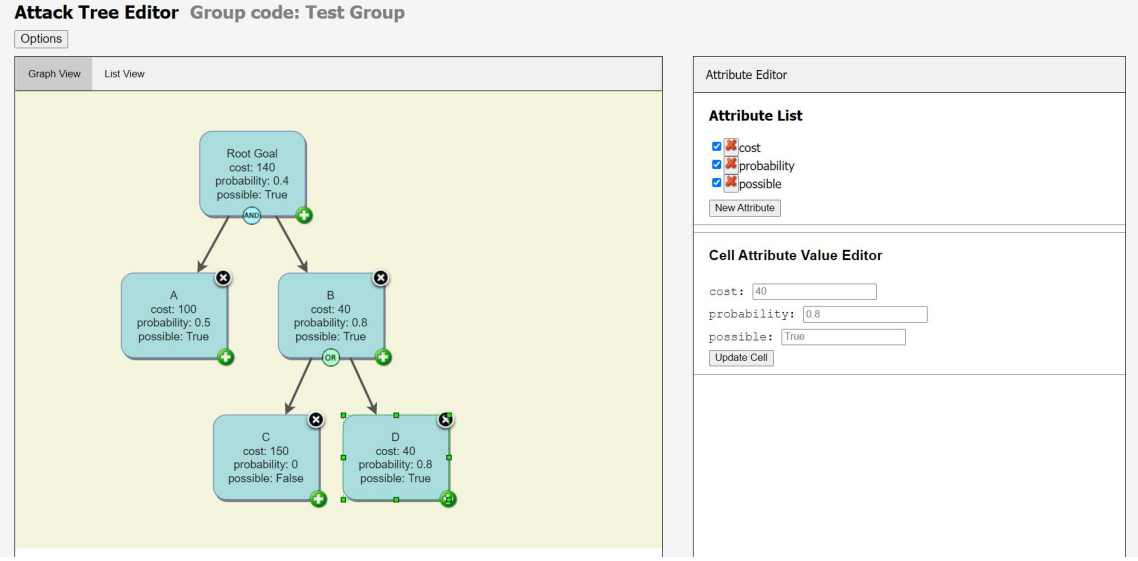


Figure 4.2: ATree Editor Page

to doing so is latency in server requests and access, although this is not why Redis was rejected once implemented with Node.

Redis was rejected as it contradicted the premise of a lightweight, accessible application that a student could host locally as well as a teacher hosting a web session for students to collaboratively access. Having to download and host a local Redis server as well imposes a significant extra requirement for users. This could be avoided by hosting a dedicated full-time Redis server that the application connects to, and in this case students would be able to host private servers and still work with peers collaboratively. This is an interesting premise, and one worth considering as a future extension, but hosting and handling interactions with a full-time Redis instance was decidedly over-complicating the ability to quickly and easily access group data. The implemented architecture is seen in Figure 4.3, with this potential dedicated Redis alternative shown in Figure 4.4.

4.2.2 Tree parsing y-js Rejection

Also considered in implementation was the algorithm for sharing updates to the tree with group members. Live collaboration was a feature not initially considered at the outset of development, and arose after the

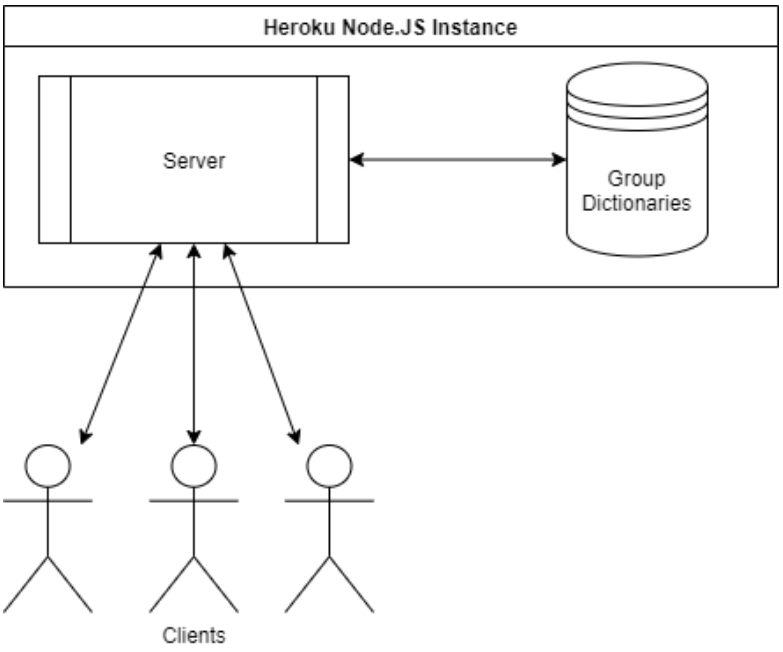


Figure 4.3: Architecture diagram following Redis removal

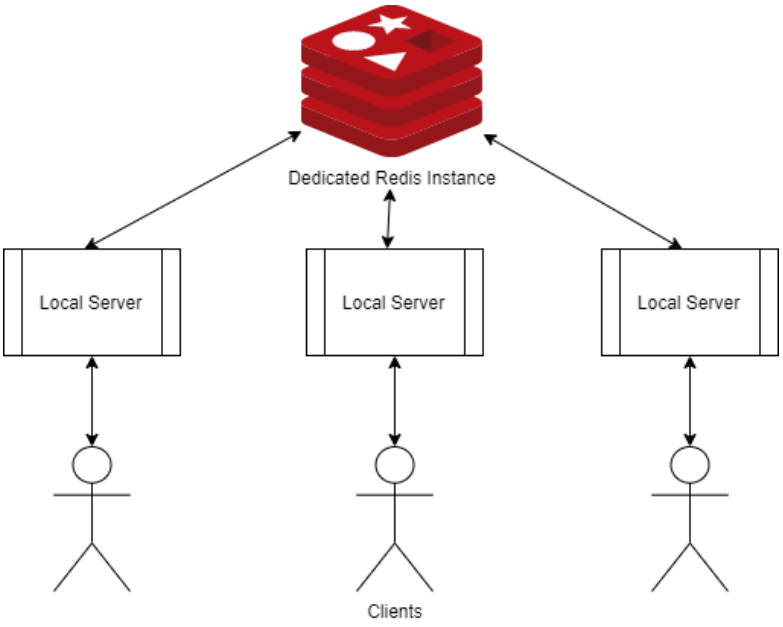


Figure 4.4: Potential Alternative Architecture with dedicated Redis server

local model was underway. The chosen implementation was simplified as a result to ensure a working product, with the entire tree being parsed and sent to update the server's master copy, which is then sent to all clients in the group. Each client then rebuilds their tree *from scratch* with the new version. The implemented algorithm is reasonably sound; working only with a whole tree means that errors in trying to update only a section of the tree are a non-issue. Consider an implementation that sends an update to the server only detailing the change made; the node that is acted on, and the operation made to that node. Client A sends an update that deletes a node, and at the same time client B sends an update that appends a child to the same node. The server executes these updates in turn, and cannot perform Client B's update without compromising the tree's structure. Trying to implement this would be hard to debug.

The Operational Transfer (OT) algorithm works around this by modifying the received operation according to state of the sent tree, and CRDT works by sending state to other clients directly, as with y-js. The implemented algorithm is not efficient, and with OT the same functionality could be achieved with less computation.

Tree Rebuilding

In particular, mxgraph specifically mentions that the JavaScript version can become slow as tree size grows, and rebuilding the entire tree with every received update might exacerbate this slowness. If working in a group making lots of updates, this reloading time could be significant, implying that a switch to OT would benefit the user experience, since this rebuilding could be replaced with local tree updates.

To test this, a script was created to add some number of children to the root node, and functionality was added to print the time taken to perform the tree rebuilding procedure. Trees of varying sizes were updated, and the time taken for another group member to rebuild the tree was noted (as a best-of-three average). Also noted was the size of the `tree_data` object stored as the server's 'master copy'. The results of this experiment can be seen in Table 4.1.

Table 4.1: Comparison of tree size with time taken to receive an update

Number of Tree Nodes	Average Time Taken (ms)	`tree_data` size (KB, 2dp)
10	57	0.87
20	110	1.71
30	158	2.55
40	202	3.39
50	260	4.25
60	311	5.27
70	370	5.90
80	445	6.76
90	496	7.62
100	562	8.48

The results of this table shows a more significant time cost in rebuilding the tree on each update than anticipated. While a tree with a node count in the latter half of the table would be outside the expected use-case, even a tree with 20 nodes regularly took over 100 milliseconds to rebuild the tree on each update, with larger trees taking over half a second to load. This implies that the user experience for a group working actively on a complex tree would be affected by this delay, and as such it would have been a better option to invest in integrating OT or CRDT for more efficient tree sharing.

Getting object sizes was done with the `object-sizeof` Node package: https://www.npmjs.com/package/object-sizeof. The object sizes were a less significant issue; even complex graphs with multiple attributes struggled to break 10KB, which alone would not be cause enough to refactor to an OT datastructure.

For these figures, is worth noting that the script used adds all new children to the root node, creating a short but incredibly wide graph, which may slow down mxgraph more than anticipated as it tries to render nodes so far outside of the container. These figures were also gathered on a locally hosted server, although the timed section does not include any requests to or responses from the server. The variation in timings for the same graph was small, but occasionally would spike to a much greater time, presumably coinciding with other machine or browser execution; these values were discarded as outliers.

4.2.3 Static Forms

Finally, the implementation of forms, while implemented smoothly and functionally work well, involved generating the forms dynamically with JavaScript. Doing so gave a high level of control over form elements, and consolidated form creation, handling and processing within one file. On the other hand, doing so is incredibly verbose, and ballooned these files by hundreds of lines. Styling more complex forms dynamically also became difficult, and eventually doing so was rejected in favour of a simple interactive form. There are two main alternatives to this approach:

1. Implement forms in HTML, and hide/reveal the forms according to when they are needed. Statically implementing forms like this makes them easier to style and is far more compact, but is effectively a different way of doing the same thing. As such, it is more of a design preference as to which is used.

2. Implement forms with a library in a front-end framework. Front-end frameworks had originally been rejected due to not considering them necessary for the tool's requirements; however, styling,

designing and handling forms would have greatly benefited from an appropriate library. A redesign of the tool would likely include pivoting to a framework like React and a library like Formio [6].

4.3 Surveying and Feedback

The motivation for surveying as a means of evaluating the tool's usefulness has been given in Section 4.1.3. In this section, the survey's design is described in detail, followed by a thorough interpretation of the collected results, and the information learned from these results in the context of designing a teaching tool.

4.3.1 Survey Design

Surveys were chosen as the source for gathering feedback on the tool. They can be conducted at scale, anonymously, and allow users to provide first-hand written responses as well as statistical data. The other highly applicable feedback method would be in-person interviews. While these would allow for even more detailed responses and an insight into how users try to interact with the tool, the number of easily attainable responses is limited, and asking others to conduct interviews complicates the process.

Effective User Surveys

To design a useful user survey, it is first important to ask what information it is you want to know. In this case, "evaluating the effectiveness of a user tool" is too vague to immediately start forming questions. What makes a tool effective? In this case, there is a suite of features- groups, tree editing, attributes, etc; each with their own interfaces, functions, and interactions. An effective teaching tool is one that makes these features easy to learn and navigate, with visible, accessible, and intuitive controls.

Constructing the survey was assisted by looking at existing work analysing the usability of threat modelling tools. One paper in particular analyses many of the existing tools looked at in the first chapter, and asks three main research questions: *Are the tools accurate? Is time being saved by using threat modelling tools? Are the tools user friendly?* ([20], Section 3). The final question was the key focus of this survey, but considering a time element would also be a good metric to analyse.

The aims of the survey are to establish which parts of the tool work well, and which are hard to use. So, a list of questions were designed asking the user how difficult they felt it was to use certain parts of the program. To understand how easy to use and navigate the tool as a whole is, an overall question is also useful. Finally, the help page was constructed to assist with providing an explanation of the tool's utility, so some feedback on the help page would also be helpful.

Creating a List of Tasks

To answer these questions, a user would have to first have interacted with the tool and each of its features. Asking a user to 'spend some time using the tool' does not guarantee that the user will do so, especially if a tool is hard to access. So, it became clear that a set of tasks would have to be constructed alongside the questionnaire that the user would first complete. These tasks would engage every main function of the tool. In defining this list of tasks, it became clear that importing an example tree would allow the user to work off and make writing questions far easier. It is easier to ask a user to and, edit and delete specific nodes than to vaguely gesture to their tree they and tell them to do so somewhere. This example tree was added as an option to the help menu for this reason. After some work, the set of user tasks was finalised as follows:

1. Go to `https://desolate-dawn-31882.herokuapp.com`
2. Create a Group editing session, using the group code of your choosing.
3. Load the example JSON tree.
4. Delete the node with the label 'C'.
5. Add a new child to the root goal.
6. The new node should have the label 'New Cell'. Change it to 'E'.

7. Change the root goal from an AND node to an OR node. Some of its attributes should update to reflect this.

8. Remove the 'cost' attribute from the tree.

9. Add a new attribute, 'difficulty', which should have its domain between 1 and 0 (the unit interval). It should have a default value of 0.5.

These tasks cover all of the basic functionality offered by the tool; group creating, tree editing, attributes, and And/Or nodes. Each of these tasks can then be referred to in the question section, to get more specific feedback on what parts of the tool are difficult to use.

Extra Tasks

The set of tasks also involved two other steps. Firstly, the user should record with a stopwatch the time taken to complete the set of tasks. This would provide extra numerical data about the tool's usability as a whole. To make doing so as accessible as possible, a link to an online stopwatch was included at https://www.online-stopwatch.com/. Extra overhead to completing the test was not ideal; however, it was reasoned that implementing a timer within the tool would itself be an extra feature. Its presence could clutter the UI, making the tool harder to use, and modifying the tool to create a special testing version is dangerous; should the two deviate too much, the results become useless.

The other extra task related to assigning participants a view. Two views had been implemented for viewing an attack tree, and both afforded the user full functionality. Without specifying, there would be no way of knowing which view the user had been referring to - what if adding nodes is trivial for users in the graph view, but unintuitive and difficult for users in the list view? So, it was reasoned that users should be told to use one view only when starting the tasks. The tool did not support this option, and the Google Form used did not allow for splitting questions between participants. The solution was to ask users to flip a coin, with heads assigning them to the graph view, and tails to the list view. Again, a link to an online coin flipper was included at https://flipsimu.com/. With this, the two views could be compared and evaluated.

Creating a List of Questions

Once the tasks are completed, the user would then be prompted to ask a series of questions about their experience. Following up immediately with questions is vital to capture the initial response to the tool. Collecting both quantitative and qualitative feedback is also important. Quantitative feedback, with higher numbers of participants, gives concrete statistical data to evaluate elements of the tool. Qualitative feedback is optional, and was used for users to voice their opinions on the tool and their experience with it. In particular, it is useful to prompt for feedback on 'critical incidents'; for example, if something went wrong while the user was completing the tasks. People typically have better memories for nonstandard situations than normal ones, which makes this effective for gaining useful feedback on edge cases and opinions.

The questions were also designed carefully. Questions should not be too long, ambiguous, leading, or double-barrelled (i.e. *what did you like about X, what did you dislike about Y?*). For the quantitative area-specific tasks, the Likert scale was used; a set of 5 options, from 'very easy' to 'very difficult'. The final list of questions was as follows:

1. Were you working with the Graph View or the List View?

2. How long did you spend completing the given tasks? If you didn't finish, write N/A

3. How easy/difficult did you find starting a new group session?

4. How easy/difficult did you find loading the example tree?

5. How easy/difficult did you find deleting a node and adding a new node?

6. How easy/difficult did you find changing a nodes' label?

7. How easy/difficult did you find adding a new attribute?

8. How easy/difficult was navigating the tool overall?

9. Did you have to refer to the help page to complete the tasks?
10. Did you encounter any particular difficulties or issues while completing the tasks?
11. Any further comments/ suggestions about using the tool?

The final two questions were optional long answers; all other questions were mandatory, and prompted a response from a set of answers. The survey was anonymous, and designed to be short enough to encourage engagement, while also giving a rich set of feedback across the whole tool.

4.3.2 Survey Results

The survey was given to a cohort of students who had studied attack trees, as well as to other Computer Science and some Engineering students. The aim was both to maximise the number of responses, and also to get responses from users with a range of levels of experience with attack trees and diagramming tools. In total, there were 19 responses. Of these, eleven flipped tails and used the list view, and eight flipped heads and used the graph view. This reflects that users were not assigned in a perfect split between the two views, although the 58/42 split is good enough for comparisons.

Time Taken

The time taken to complete the tasks is continuous data. A histogram could be used to represent this cleanly, but it is also desirable to contrast the graph and list view times. Histograms with clustered columns were generated, but were messy and hard to read. A better solution is to present a pair of box and whisker plots. This allows for comparing the median time taken for each view easily, as well as the spread of times.

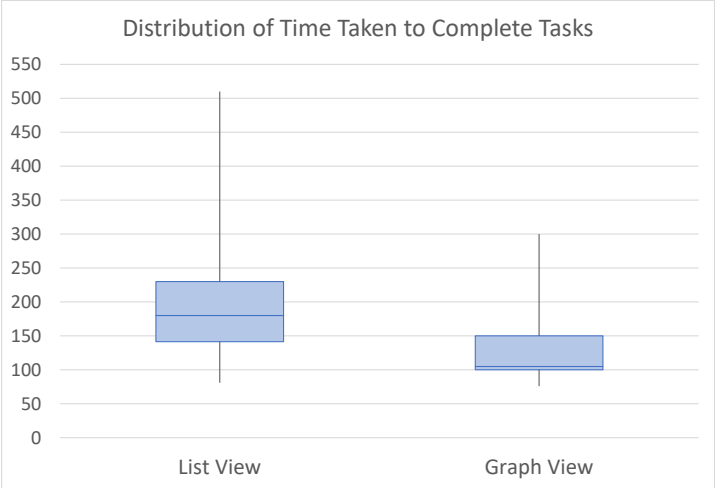


Figure 4.5: Box plots comparing list and graph view user times in seconds

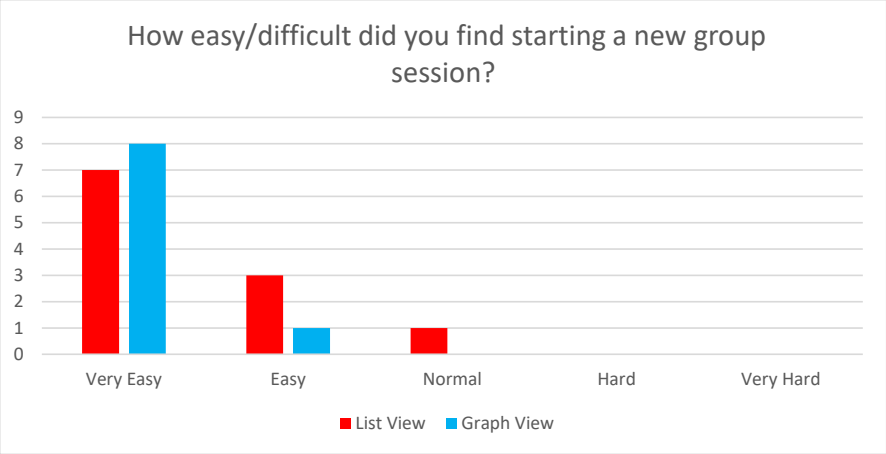
These box plots are shown in Figure 4.5. The median time for the graph view lies outside of the list view box, and the graph view upper quartile is very close to the list view lower quartile. This implies a *likely* significant difference between the two views (since the two boxes overlap slightly), that users using only the graph view took less time to complete the set of tasks than those using the list view.

This could be due to the visual overlay buttons the graph view presents that instantly make it clear for users to navigate adding and deleting nodes. A user in the list view has to click the relevant item then navigate the set of options that reveals, which are not visible to the user without this interaction. However, this is somewhat contradictory to the results in Figure 4.6c, which shows that users had similar difficulties in adding and deleting nodes. Difficulty in navigating the list view may come from elsewhere.

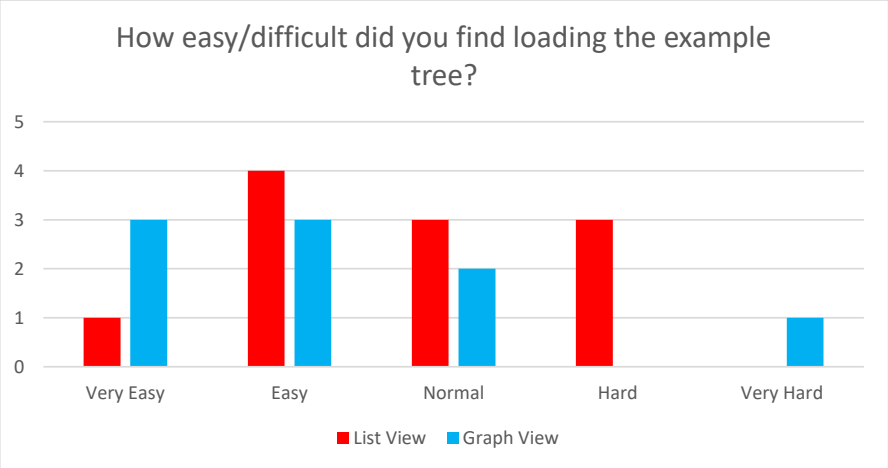
The difference in results could be due to another reason. The default view is of the graph whenever loading the tool. Therefore, a user who has flipped tails was instructed in the tasks to click the 'List View'

tab as soon as loading the tool. This is an extra step for users and can not be ignored as potentially taking users extra time to locate and load the correct view. Further testing would be required with loading users into the correct view automatically to draw any stronger conclusions.

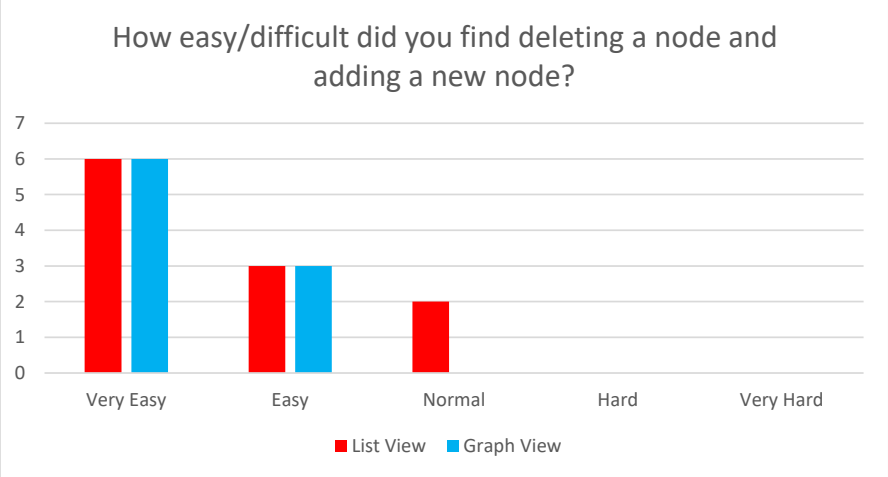
Easy and Difficult Components



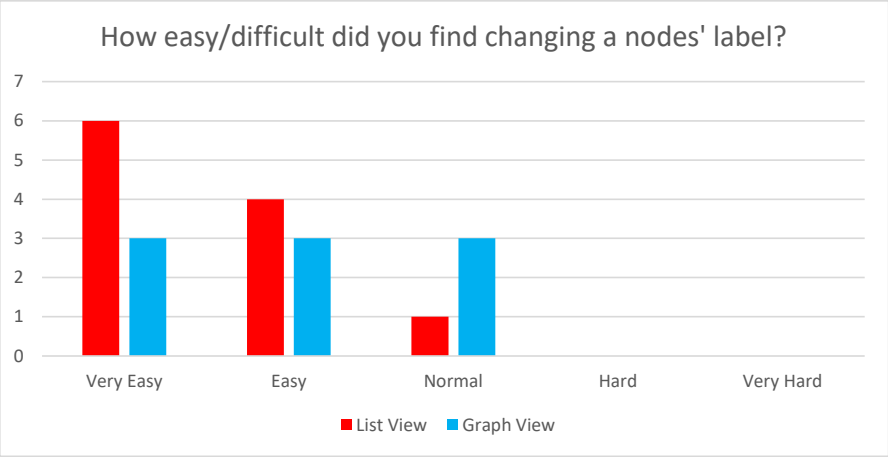
(a) Difficulty of starting a new group session



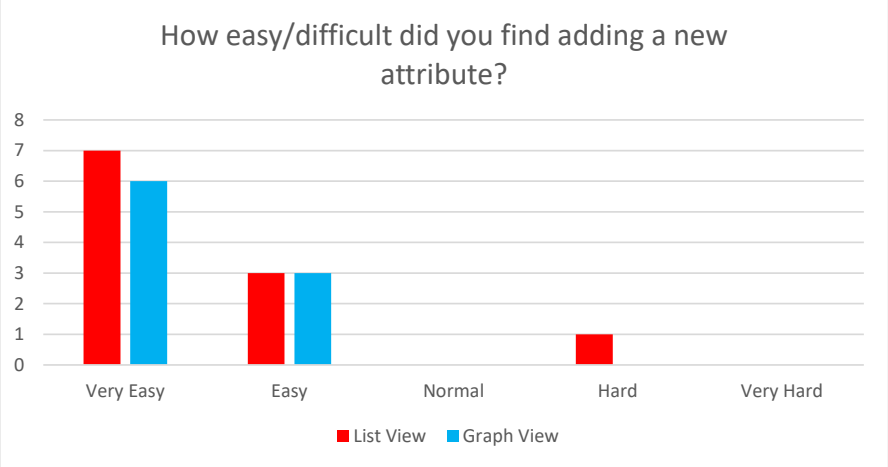
(b) Difficulty of loading the example tree



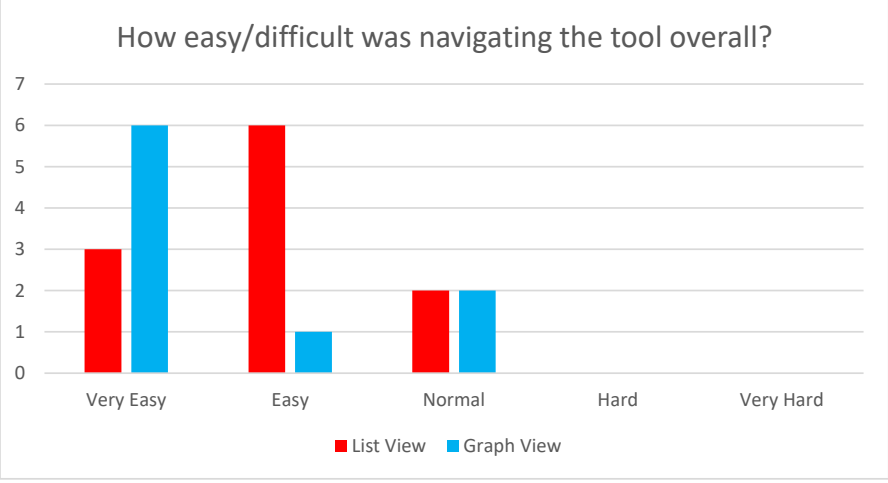
(c) Difficulty of adding and deleting nodes



(d) Difficulty of changing node labels



(e) Difficulty of adding new attributes



(f) Difficulty of navigating the tool overall

Figure 4.6: Survey Likert scale results

Figure 4.6 shows a collection of clustered column graphs with the survey results for the Likert scale questions. The graph and list views are again compared to investigate any differences between the two, as well as visualising the spread of results as a whole.

- Figure 4.6a shows how much difficulty users had creating a new group session and loading in. This is unrelated to the type of view a user is using, so a significant difference in list and graph results should not be expected, which is seen in the results. Doing so was received as remarkably easy on a whole, possibly due to the simple UI the main page presents users with, which prevents confusion from trying to navigate a cluttered page. Such a design does therefore not need to be altered significantly.

- Figure 4.6b was one of the most interesting results due to the spread of responses. Once again, loading the example tree is done the same way regardless of view, and no significance between the two is observed as expected. However, nearly half of all users found it less than 'Easy' to load the example tree. The reason for this is likely due to lacking discoverability, as loading the example tree is hidden within the 'Options' menu, where users may not have expected it to be. Overall, the data strongly implies that a more intuitive way of accessing the example tree is needed.

- Figures 4.6c and 4.6e similarly show that users found it fairly easy to add/delete nodes in the tree and to add attributes, regardless of the view used. Adding attributes was again something that would roughly be expected, as the attribute navigator used to view and add new attributes is visible on the right side of the screen at all times, and is interacted with irrespective of view. One user in Figure 4.6e did have some difficulty adding attributes, which is discussed in more detail in the 'Written Responses' in Section 4.3.2.

  The results in Figure 4.6c of how users found adding and deleting nodes, however, was more surprising. Users found doing so mostly very easy or easy, with the exception of two users interacting with the list view. Some revisions to make the list view easier to use may be required, but the graph view's overlay buttons work very well as they are.

- Figure 4.6d shows that most users found changing a node's label to be very easy or easy, but that a set of them found doing so to be somewhat difficult. This is almost entirely comprised of users interacting with the graph view. While the list view label is edited via the form to edit a node's values, graph view labels are only able to be edited by double-clicking the node and typing in there. This was not immediately obvious to all users, as discussed in the 'Written Responses' in Section 4.3.2. These results imply that the graph view may benefit from a more intuitive way of editing a node label.

- Finally, the graph in Figure 4.6f shows the results of asking users to rate their overall experience with the tool. The majority of users working with the graph view responded with 'Very Easy', while the majority working with the list view responded with 'Easy'. This seems to imply that the graph view was slightly easier to use than the list view, which is in line with the results of Figure 4.5. This may be simply due to graphical trees being easier to read than as a list, or due to the list view being complex and unintuitive to use. To further study this, in-person user interviews studying how people interact with the list view could be carried out. This would allow for seeing what mistakes users make and what they look for when trying to complete tasks, which could highlight weaknesses in the interface.

Referring to Help

Towards the end of the survey, users were asked whether they had to look up help at some point during the tasks. The results of this are shown in Table 4.2. Both views show that a small proportion of users had to look up help using the tool at some point, with a slightly higher proportion of those using the graph view having to do so. These low figures could be for several reasons:

- The tool is easy enough to use that the majority of users simply did not need to access help.

- More users required help at some point, but could not work out how to access it. It has already been shown that users struggled to load the example page, which is also located in the 'Options' dropdown menu, so this is feasible.

- More users did use help at some point, but opted to respond that they had not. This is unlikely as the test is anonymous, meaning people should be more likely to answer truthfully.
- More users did need help and opened the help page, but were unable to locate guidance in what they were trying to do. The help page was written as a long stream of text and images grouped into sections, and has little structure other than skimming the entire page and looking for relevant content. In fact, one user gave a response to the final question that "*The help page looks complicated*", giving some credence to this theory.

Overall, a better way of accessing help may help make the tool more usable. The main page has a link to load help as an always-visible footer, which could be a better approach to providing a link to find help.

Table 4.2: Comparison of users requiring the help page according to view

View	Percentage of users needing help
Graph	33
List	27

Written Responses

The final two questions in the survey gave a chance for responding with qualitative long responses. Respondents were encouraged to talk about any particular difficulties they had while completing the tasks, as well as an open-ended question for commenting on their experience with the tool. Doing so gave some incredibly valuable insights into the reasoning behind the quantitative data gathered, as well as how the tool was perceived as a whole.

Changing graph view labels was something respondents using the graph view scored as being more difficult to do. Looking at the written responses, two comments stand out that give insight as to why this is the case:

> "It was a _tiny_ bit jarring to have to double click the nodes to change the name - with everything being on the right I expected it to be listed as a cell attribute. Not a major issue but it would be nice to have both the option to edit the name via double clicking and also in the cell attribute value editor."

This first comment shows that double-clicking labels, which is an interaction not used anywhere else in the tool, lead to some confusion and was not immediately obvious to the user. The suggested change, however, is moving the label to the cell value attribute editor. Labels are stored in XML as attributes, and users may semantically group them as such. This theory is reinforced by the second comment:

> "At one point there is a message which says that only leaf nodes can have their attributes changed. However the root node is not a leaf node, and yet I was able to change its name (as the instructions required). I guess that a node's name is not considered an attribute as far propagating changes is concerned, but it might be worth clarifying the message."

This user was assigned to the list view. Editing a node in the list view form allows for editing And/Or values and labels on each node, but attribute values can only be edited on leaves. They also assumed labels to count as an attribute. Therefore, moving node labels to the attribute editor may improve usability by making the tool act more in-line with the user's internal conceptual model.

Creating attributes was found difficult by one user who gave an explanation of what they had trouble with:

> "I struggled to find where to input the 0 - 1 interval for difficulty when adding an attribute."

The '0 - 1 interval' refers to how the question asks users to create a new attribute with "*a domain between 0 and 1 (the unit interval)*". This domain in the Attribute Creator form is accessed in a dropdown list, and is the option called `UNIT_INTERVAL`. These names directly come from the internal names for identifying each domain. The difficulty in using this form likely came from this unintuitive naming, which has no description for domains and could easily be confusing to a user. Adding in-line help and comments to this form could assist users in creating attributes more easily to represent what they want.

4.3.3 Survey Reflection

As a whole, this survey worked very well for gathering an informative set of data and feedback. A combination of qualitative and quantitative questions gave both valuable opinions and information, with the statistical data to solidify these points. The use of the Likert scale helped in writing non-leading questions, and dividing the surveyed populace between the two views also allowed for comparisons to be drawn between them at the same time.

However, there were some issues. Some users did not give times accurate to the second (i.e. responding with '3 minutes'), which made accurate graphing difficult. This is because the question was too vague; it only prompted with the respondent to give a time in a written answer, giving no instructions on how precise the time should be or in what form.

As well as this, a more detailed set of questions could have been asked. The given questions were kept brief in order to encourage more responses. The written responses proved valuable enough that conducting in-person interviews and studies with a more in-depth set of questions may have presented strong results.

Chapter 5

Conclusion

5.1 Contributions and Achievements

The motivation behind this project was a lack of available tools for students to build attack trees. While some options exist, such as the standalone application ADTools or SeaMonster, these do not allow for students to work together. Online whiteboards do support this collaborative group work style, but lack the tools to experiment with rapidly building and modifying attack trees.

This project has presented Atree, a tool that combines these two concepts in a web application. Students can work alone or in groups and build trees graphically in real-time. The server can also be hosted locally allowing for fully offline working.

Two different ways of displaying trees have been implemented. A user can choose the view which best suits their preferences and the needs of the current tree. Attack trees are built with attributes, which can be created with some degree of customisation. These attributes are propagated dynamically up trees according to changes, neatly highlighting at the root node those values. Users can download and upload the trees they have built, allowing for the possibility of working with the tool in a learning environment off a specially-designed tree, and for saving and submitting a student's work.

Aside from the final tool itself, this project demonstrates the potential of the two backbone JavaScript libraries from which the tool was built: mxgraph 4.2.2 and socket.io 4.0.

- **mxgraph** has a wealth of utilities that make it applicable for many different applications other than just tree creation. The tool is lightweight and (relatively) user-friendly. In particular, the fast tree algorithm in-built in the `mxCompactTreeLayout` was hugely beneficial in allowing users to build their own trees and render them graphically.

- **socket.io** allows for compact, two-way client-server communication of both primitive types and composable JavaScript objects such as dictionaries and lists. This made creating a system for working in groups far more approachable, and the possibilities presented by the socket API apply to almost any JavaScript program involving client-server communication.

Atree is hosted on a Heroku instance with Git integration to allow for continuous deployment. The instance was used in testing the tool's usability with students from the University of Bristol, and the results of this surveying have highlighted areas which require potential reworking, as well as demonstrating that as a whole the tool is user-friendly and easy to work with.

5.2 Project Status

The aims and objectives, as outlined in Section 1.5, have all been met to some degree. Goal 1 involved researching graphing libraries, which was done by experimenting with a variety of libraries in several languages, which is detailed in Section 2.4.1. It was this process that resulted in selecting mxgraph. An internal representation of the tree was done through storing attributes in the `mxCell` node as XML data, which was a less premeditated design choice but paired the implementation well.

Goal 2 was to write a parser to handle this tree and convert it to a list-based format, which was also fully implemented. A second parser to create a minimal format of the entire tree was also written, that compresses even large complex trees to only a few kilobytes. This is documented in Chapter 4. This chapter also describes the process of completing goal 3.

Finally, goal 4 was to evaluate and analyse this tool in the context of evaluating a threat-modelling tool's usability. This was done successfully, and some useful conclusions could be drawn from the collected results. However, what was not evaluated was the tool's effectiveness in actual threat-modelling, as described in existing literature analysing the effectiveness of threat modelling tools ([20], Chapter 3).

Some features are not working as of this final version. For example, one survey respondent in the written section detailed a set of steps in the list view where changing And/Or nodes do not propagate, implying that there is a functional disconnect in the two views. Surprisingly, towards the very end of development the private sessions only stopped updating the list view and attributes (neither of which directly require a socket connection to work). This has not been fixed, and was temporarily worked around by adding buttons to manually update these elements, due to time constraints.

Atree consists of one web application as a Node.JS project, and is available for download from: https://github.com/James-Wickenden/Attack-Tree-Tool/releases/tag/1.0

5.3 Future Plans

5.3.1 Additional Threat Models

Before the added criteria of online collaborative work, an original potential goal was to then extend the model into representing Attack-Defence trees [2], which is the model used in ADTools. This would make for a more complex tool, able to model not just attacks but also possible defenses against them, and the effects those defenses would have on the propagation of attributes. In the real world, attack trees are used as a stage in the much broader threat modelling process, in methodologies like PASTA. Other methodologies such as LINDDUN (https://www.linddun.org) use Threat Trees for this modelling stage. As such, more threat modelling diagrams could be implemented with mxgraph under the same tool to add functionality and integration with different methodologies. Although this would increase the tool's complexity, doing so would also make it applicable to broader teaching.

5.3.2 Reworks and Extra Functionality

In terms of work within the current scope of the tool, Table 4.1 strongly implies a rework to sending and receiving tree updates would be beneficial to performance. Working on a y-js implementation and binding could be a significant time investment for no visible difference in functionality, but would make for an interesting challenge requiring further research.

The groups feature works as it currently is, but there is no way of looking at a current group as a client. How many members are in the group? Is there a cap on members? Who is in the group? What modifications are members making? How can members communicate within the tool? Implementing a solution to these issues could greatly improve the experience of working in a group, and would therefore make the tool much more useful for students trying to learn together. Many of these design elements could take inspiration from current collaborative technologies, such as Google Docs, or the Online Whiteboard examined in Section 1.3.1.

Bibliography

[1] E.G. Amoroso. *Fundamentals of computer security technology.* Prentice-Hall, 1994.

[2] S. Radomirović B. Kordy, S. Mauw and P. Schweitzer. *Foundations of attack–defense trees. In International Workshop on Formal Aspects in Security and Trust.* Springer, 2010.

[3] B. Schneier C. Salter, O.S. Saydjari and J. Wallner. *Toward a secure system engineering methodology.* NSA, 1998.

[4] cchampion. https://stackoverflow.com/questions/47972193/how-to-create-a-hierarchical-tree-with-collapsible-nodes-in-jgraphx.

[5] ceasar0301. https://github.com/caesar0301/treelib.

[6] Formio. https://github.com/formio/react/tree/1.x.

[7] Kevin Jahns. https://github.com/yjs/yjs#yjs-crdt-algorithm.

[8] Piotr Kordy. https://satoss.uni.lu/members/piotr/adtool/.

[9] LINDDUN. https://people.cs.kuleuven.be/ kim.wuyts/linddun/linddun.pdf.

[10] MindFusion. https://tinyurl.com/3w95hcjw.

[11] S. Moen. Drawing dynamic trees. *IEEE Software*, 7(4):21–28, 1990.

[12] mxGraph. https://jgraph.github.io/mxgraph/.

[13] M. Shapiro N. Preguiça, J.M. Marquès and M. Letia. *A commutative replicated data type for cooperative editing.* IEEE International Conference on Distributed Computing Systems, 2009.

[14] P. O'Riordan T.P. Scanlon N. Shevchenko, T.A. Chick and C. Woody. *Threat modeling: a summary of available methods.* Carnegie Mellon University Software Engineering Institute Pittsburgh United States, 2018.

[15] B. Schneier. Attack trees. *Dr. Dobb's journal*, 24(12):21–29, 1999.

[16] socket.io. https://socket.io/docs/v4.

[17] K. Thulasiraman and M.N. Swamy. *Graphs: theory and algorithms, p.118.* John Wiley & Sons., 2011.

[18] TypeScript. https://www.typescriptlang.org/.

[19] T. UcedaVelez and M.M. Morana. *Risk Centric Threat Modeling: Process for Attack Simulation and Threat Analysis.* John Wiley and Sons, Inc., 2015.

[20] L. Verheyden. *Effectiveness of threat modelling tools.* Doctoral dissertation, Master thesis, 2018.

[21] D.C. Walden and T. Van Vleck eds. *The compatible time sharing system (1961-1973): Fiftieth anniversary commemorative overview.* IEEE Computer Society, 2011.

Appendix A

Treelib Attack Tree wrapper

```
1  from treelib import Node, Tree
2  import copy
3
4  class AttackNode:
5      def __init__(self, nodeType, cost):
6          self.cost = cost
7          self.nodeType = nodeType
8
9  class AttackTree:
10     def __init__(self, rootGoal):
11         self.tree = Tree()
12         self.no_nodes = 0
13         self.AddNode(rootGoal, None, "ROOT")
14
15     def UpdateCost(self, node):
16         if node.is_root(): return
17         parent = self.tree.parent(node.identifier)
18         f = min
19         if parent.data.nodeType == "AND":  f = sum
20
21         child_costs = [self.tree[child.identifier].data.cost for child in self.tree.
               children(parent.identifier)]
22         parent.data.cost = f(child_costs)
23         self.UpdateCost(parent)
24
25     def AddNode(self, node, parent_id, nodeType="LEAF", cost=100):
26         self.tree.create_node(node, self.no_nodes, parent=parent_id, data=AttackNode(
               nodeType, cost))
27         self.UpdateCost(self.tree[self.no_nodes])
28         self.no_nodes += 1
29
30     def show(self):
31         print("TREE:")
32         tmpTree = copy.deepcopy(self.tree)
33         for node in tmpTree.all_nodes():
34             node.tag += " " + str(node.data.cost) + " " + node.data.nodeType
35         tmpTree.show()
36
37 attacktree = AttackTree("Open safe")
38 attacktree.AddNode("Pick lock", 0, cost=20)
39 attacktree.AddNode("Learn combo", 0, nodeType="OR")
40 attacktree.AddNode("Eavesdrop", 2, nodeType="AND")
41 attacktree.AddNode("Listen to conversation", 3, cost=100)
42 attacktree.AddNode("Get target to give combo", 3, cost=50)
43 attacktree.AddNode("Bribe", 2, cost=75)
44 attacktree.show()
45
46 print("Updating root nodeType")
47 attacktree.tree[attacktree.tree.root].data.nodeType = "AND"
48 attacktree.tree[1].data.cost = 40
49 attacktree.UpdateCost(attacktree.tree[1])
50 attacktree.show()
```

Listing A.1: Python Attack Tree prototype

Appendix B

Adding Children to the mxgraph Tree

```
1  // Create a new leaf node with cell as its parent node
2  function AddChild(graph, cell) {
3      var parent = graph.getDefaultParent();
4
5      graph.getModel().beginUpdate();
6      try {
7          var xmlnode = doc.createElement('cell');
8          xmlnode.setAttribute('nodetype', 'OR');
9
10         var newnode = graph.insertVertex(parent, null, xmlnode);
11         var geometry = graph.getModel().getGeometry(newnode);
12
13         // Updates the geometry of the vertex with the preferred size computed in the
               graph
14         var size = graph.getPreferredSizeForCell(newnode);
15         geometry.width = size.width;
16         geometry.height = size.height;
17
18         // Adds the edge between the existing cell and the new vertex
19         var edge = graph.insertEdge(parent, null, '', cell, newnode);
20         newnode.setTerminal(cell, true);
21
22         // If needed, add a graphical AND/OR overlay to the parent
23         if (GetChildren(cell).length > 1) Add_AND_OR_Overlay(graph, cell);
24         AddOverlays(graph, newnode);
25
26         // Any tree attributes need to be added
27         // the new child will be a leaf by definition, so we can assign default values
28         for (var key in attributes) {
29             var attr = attributes[key];
30             xmlnode.setAttribute(attr.name, attr.default_val);
31             cell.setAttribute(attr.name, attr.default_val);
32             PropagateChangeUpTree(graph, cell, attr);
33         }
34         xmlnode.setAttribute('label', 'New Cell');
35     }
36     finally {
37         graph.getModel().endUpdate();
38     }
39     EmitTree(graph);
40 };
```

Listing B.1: Atree `AddChild()` function