



# 6CCS3PRJ Final Year Project

## Generating Trees Arising from Two-Player Combinatorial Games

Final Project Report

Author: Berke Muftuoglu

Supervisor: Hubert Chen

Student ID: k20053893

April 4, 2023

## **Abstract**

The concept of gaming and game design has been prevalent in human society for centuries. With the passage of time, the nature and complexity of games have evolved significantly. In the early '00s, a new and exciting development arose known as "General Game Playing" (GGP). GGP was developed to generalize the concept of gaming by defining a range of essential concepts including players, legal moves, and terminal moves.

As the design and development of games progressed, there was an increasing need to have more concrete representations of games. One of the most effective solutions to this problem was the use of visual aids such as pictures and graphs. These visual representations proved to be an ideal way to show the various sequences of transitions between game states.

This report discusses and implements the visualization of combinatorial game trees using directed graphs. The result of this work will provide a clear picture of the current state of a game and the possible transitions or moves from that state. Users will be able to define their own games and view a tree-like visualization of the moves made by each player. Additionally, they will be able to download a text version (XML) of this tree.

### **Originality Avowal**

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Berke Muftuoglu

April 4, 2023

## **Acknowledgements**

I would like to thank my supervisor Hubert Chen for the efforts he put into this project. His guidance and support for this project have played a crucial role during the process. Also, I would like to thank Jason M Barnes(jazzyb) for implementing a GDL parser that lies at the heart of all of the computations necessary for parsing and Markus Partheymueller(parthy) for his providing games in GDL format.

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Motivation . . . . .	3
1.2	Scope . . . . .	4
1.3	Report Structure . . . . .	4
<b>2</b>	<b>Background</b>	<b>5</b>
2.1	Combinatorial Game Theory (CGT) . . . . .	5
2.2	General Game Playing (GGP) . . . . .	6
2.3	Conceptualization . . . . .	7
2.4	Game Description Language (GDL) . . . . .	9
2.5	Compilers . . . . .	14
2.6	Graph Theory . . . . .	17
2.7	Literature and Existing Work . . . . .	19
<b>3</b>	<b>Design</b>	<b>21</b>
3.1	Overview . . . . .	21
3.2	User Interface . . . . .	22
3.3	System Architecture . . . . .	23
<b>4</b>	<b>Requirements &amp; Specification</b>	<b>28</b>
4.1	User Requirements . . . . .	28
4.2	System Requirements . . . . .	29
4.3	Specifications . . . . .	29
<b>5</b>	<b>Implementation &amp; Testing</b>	<b>31</b>
5.1	Parsing GDL Files . . . . .	31
5.2	Tree Generation . . . . .	40
5.3	Graphical User Interface(GUI) . . . . .	44
5.4	Tree Visualisations . . . . .	45
5.5	External Resources and Libraries . . . . .	50
5.6	Testing . . . . .	51
<b>6</b>	<b>Legal, Social, Ethical and Professional Issues</b>	<b>54</b>
6.1	Legal Issues . . . . .	54
6.2	Social Issues . . . . .	55
6.3	Ethical Issues . . . . .	55

6.4	Professional Issues . . . . .	55
6.5	Code of Conduct & Code of Good Practice issued by the British Computer Society	55
<b>7</b>	<b>Evaluation</b>	<b>57</b>
7.1	Limitations . . . . .	57
7.2	User Interface . . . . .	57
7.3	Imported Code . . . . .	58
7.4	Engine . . . . .	58
7.5	Overall . . . . .	59
<b>8</b>	<b>Conclusion and Future Work</b>	<b>60</b>
8.1	Conclusion . . . . .	60
8.2	Future Work . . . . .	61
	Bibliography . . . . .	64
<b>A</b>	<b>Extra Information</b>	<b>65</b>
A.1	List of Games . . . . .	65
A.2	List of Reserved Keyword . . . . .	67
A.3	Snapshot of the Database . . . . .	68
<b>B</b>	<b>User Guide</b>	<b>73</b>
B.1	Setting up the environment . . . . .	73
B.2	Compiling . . . . .	74
<b>C</b>	<b>Source Code</b>	<b>77</b>

# Chapter 1

## Introduction

The study of games and game design has increased due to advancements in technology making games more diverse, spanning different cultures and platforms. Thus, it is now being studied in various fields such as Computer science, Mathematics, Artificial Intelligence; and even fields like Sociology, Psychology, and Education. This project will focus on the fields of Computer Science and Mathematics. It will focus on how two-player combinatorial games, which are games where the outcome is determined solely by the players' strategic choices and not influenced by chance or luck, are designed and generalized how they are played. Particularly, it will explore this visualization of such games, in the form of trees. The outcome will be an interactive visualization as the user will be able to engage with the tree facilitating their understanding of the game in depth.

### 1.1 Motivation

Games have been evolving as humanity is advancing. From games like Royal Game of Ur [9], a game that belongs to Mesopotamian times, to the current century's video games the concept of playing games has been around for a very long time. The field of General Game Playing (GGP) seeks to conceptualize gaming by breaking it down into smaller components. As games started getting more common and more complex, the need for visual representations for such games grew as well. Visual aids such as graphs are proven [21] to be effective and a strong tool for representing games allowing clear transition and progression of games. This project delivers a solution for this need using directed graphs as a means of visualizing combinatorial games. The resulting trees will have a clear representation of the current state and possible

transitioning states from the current state. This project will not only enable users to generate trees from existing games but also allow them to define their own games. Hopefully, this project will contribute to its research field and provide valuable insights into the tree generation of combinatorial games.

## 1.2 Scope

The scope of this project is to generate mathematical trees for combinatorial games that adhere to two-player combinatorial game specifications and are expressed in Game Descriptive Language (GDL) format. In addition to the games already established within the system, users should be able to define their own games as long as they comply with the requirements. It should be mentioned that while the conversion of the game to a tree structure (this process is also called treeification) falls under the purview of this project, the parsing and interpretation of GDL files are not and they are adapted from other resources.

With regards to visualizing the generated trees, the focus of this project is to make them available in the graphML format. While a JPG image can be used to illustrate the appearance of the graphML file, it is not the primary format for visualizing the generated trees.

## 1.3 Report Structure

The report will start with a Background overview of necessary topics that need to be covered and explore the previous researches that had similar intents to this project. It will then move onto the Design chapter where the detailed outline of how the software was designed. The following chapter is Requirements and Specifications which clearly defines strict borders for the project and achieves this via listing the requirements and specifications clearly. Furthermore, it will explore how the Design chapter was implemented diving into technical explanations by going in-depth into every major part of the software. Moreover, it discusses how the testing of the software will be performed. After describing how the software was designed and implemented the next chapter makes a brief evaluation of the legal, social, ethical, and professional aspects of the software. It then moves on to evaluate the software as a whole covering its limitations and how it evaluates parts of the software. Finally, the last chapter is the Conclusion and Future Work section where the end-products meaning to its relative research areas have been discussed followed by future possible works that can be worked on. At the end of the report, three appendices have extra information, installation guidelines, and source code respectively.



## Chapter 2

# Background

This chapter will cover the necessary fundamental information necessary for understanding the theoretical and practical topics covered in the project.

It will first cover the theory of games used in the project by explaining Combinatorial Game Theory, it will then approach games in terms of how they are played by going in-depth into General Game Playing. Once the reader understands what games are and how they are played, the report will discuss the importance of Conceptualization which will depict how a game can be considered a problem and how it can be solved. Then the report will continue to build upon the concept of GGP and document how the language of GDL serves a crucial role in defining the games that the trees are generated upon. A critical part of the project is how these GDL files are being processed, which is why this section will cover how compilers work and how they are relevant to this project. Finally, it will explain the concept of Graphs (the result of compilers) and how they are used in generating trees. The section will be finalized with the outline of the literature work used in the project and explore the already existing work.

### 2.1 Combinatorial Game Theory (CGT)

Combinatorial games are defined as a subset of mathematical games which have a finite set of moves and predetermined winning conditions. The key fact of these games is there are no external factors, no hidden information, and no chance is involved meaning that the result of the game is solely dependent on the sequence of the moves that the players are making. There is often a winner, which is the last player in normal games however in misère games, the last player is the loser [5]. Some examples include tictactoe, nim, chess, and go. In formal

literature, these games are studied under Combinatorial Game Theory (CGT) which is a part of Economics, Mathematics, and Computer Science. It focuses on analyzing the structure of games and developing methods to solve these games. The main reason for this type of game to choose is there is deterministic and finite which makes it easier to generate a tree. This theory is limited to the study of developing these games. To generate a tree, a more general concept of "how" to play these games is required.

## 2.2 General Game Playing (GGP)

GGP is a subfield of AI that aims to develop agents that can play any game without being specifically programmed to do so. The scope of this project doesn't contain developing an agent however, it is important to understand to core concepts of such agents like understanding "how" any game is played given the axioms. To put that into perspective the system will be acting like GGP agent because it needs to interpret the game and act accordingly the only difference is it doesn't look for an outcome but rather displays all the possible moves. The system is only interested in the understanding part of the game and doesn't develop a strategy to play it.

The book by Michael Genesereth (known to be the creator of GGP) of Stanford University and Michael Thielscher [11] clearly explains the concept to its deepest but this chapter does not discuss the entire concept rather it will discuss the first few chapters. In their book the authors suggest that every game could be reduced and explained using states and transitions can be demonstrated as edges in their book they call it "state graph" in figure 2.1. Later the report will discuss this in section 2.6 mathematical shape in depth.

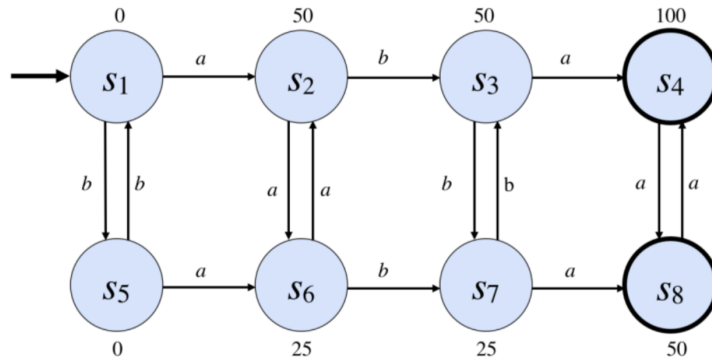


Figure 2.1: A state graph source: [11]

This figure 2.1 depicts a single-player game where the only possible moves are "a" and "b" in different stages of the game which has different effects on the score that you obtain. For

instance, starting from the initial state, S1, if the player makes the move "a" it gains 50 points as it goes to the state S2 whereas if the player makes the move "b" it goes to state S5 and it doesn't affect the score of the player.

It is also important to note that this figure 2.1 assumes a very simple cause-and-effect scenario where if an action is taken, it has a certain outcome (if a player is in s1 and does the action it will receive 50 points and go to state S2) however in the real world applications sometimes calculating an outcome isn't that simple. Imagine a game of football when a player kicks the ball, what would be the outcome? The possible outcomes include the ball landing in a 100-meter radius in which there are an uncountable amount of possibilities. Thus, it becomes hard incredibly hard to "play" these games and put them in graphs like in figure 2.1. So if anyone would want to analyze the game of football, they have to quantify the possible outcomes and possibly ignore some factors and outcomes while doing so. That process is called Conceptualization and its use will be explained in the next section.

## 2.3 Conceptualization

Conceptualization is defined as "the act or process of forming an idea or principle in your mind". In this project, GDL is used to conceptualize the idea of games by defining them using certain parameters.

The main reason for this is to limit the amount of uncertainty in the world and define small environments with simple axioms. Creating an environment is like creating a smaller world where rules of science are scrapped and redefined from scratch. In this little world, new paradigms and axioms can be defined and it is only limited to one's imagination. A simple example might be a speed of a falling object where a lot of factors like friction are almost neglected or omitted. This is in fact simplifying a problem into a smaller solvable environment. Here is a visual figure that will make a clear demonstration of what is described:

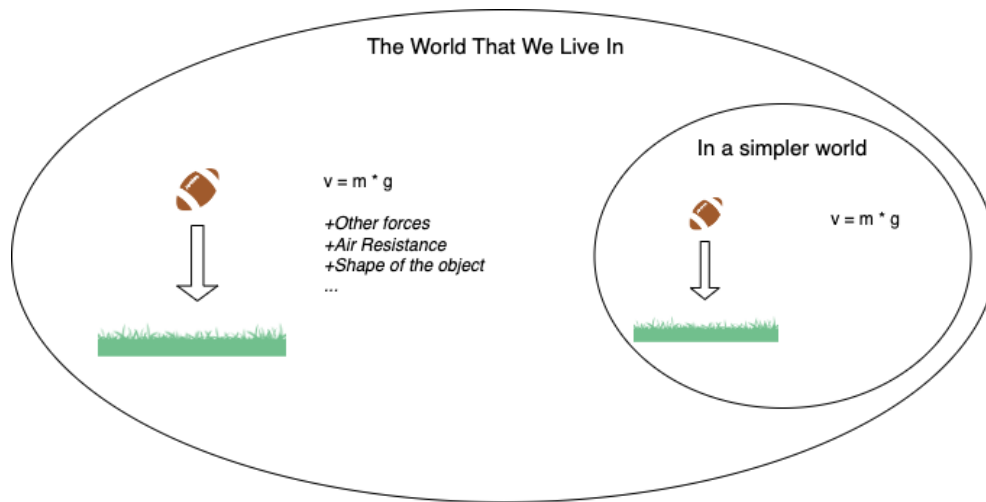


Figure 2.2: Visual Representation of Conceptualisation

As seen from figure 2.2 in the real world we live in calculating the precise speed of a football dropping to the ground is merely impossible. First of all, there is always an assumption to make the gravitational constant  $g$  fixed to a number like 9.84 however this will cause a precision error during calculation. Even though this number is fixed to a certain value, other factors such as other forces(magnetic fields, electric currents), air resistance, the shape of the object and so much more factors need to be calculated in order to solve this problem which makes it impossible to solve. That is why this problem is solved within an environment where all other forces and factors are neglected. In the "simple world", there is a single axiom stating, "An object will fall at a speed at mass times the gravitational force of the earth" where in the real world the same rule doesn't stand as there are other forces acting on the football that cannot be neglected. This is exactly the aim of conceptualizing something so that the problem can become solvable in the simpler world like in figure 2.2. That simple world is the concept that problem solvers use. In this project, every game will be put in such a world so that they become solvable.

Looking from a different perspective, the games described in section 2.1 can be seen as a problem just like in figure 2.2. Instead of asking the question "What is the velocity of the ball given the formula?", in a game this question can be "How does the game change given the rules and the current move that is being made?". Therefore, every computation that is done in this application is a "problem". Through the use of Conceptualization, the problem in the application can be simplified and thus become solvable. The use of GDL allows the user to define their problems in a solvable space by simply defining facts of their own and assuming

nothing else is true.

## 2.4 Game Description Language (GDL)

Game Description Language (GDL) is a high-level formal language intended for representing video games clearly and understandably. GDL has been applied in numerous other contexts since it was first created with the goal of enabling artificial intelligence (AI) systems to play games. Fundamentally it aims to conceptualize games which are discussed in section 2.3.

It is a very important part of the project as this is semantics used to define games. By compiling the GDL files, the program can be pinpointed to a game state, compute possible moves and extract the rules of the game. Knowing where the game is at and how to make a legal move is enough to play and treeify the game. These next states are then formed into a tree.

First-order logic serves as the foundation for GDL's concise and intuitive syntax. The two fundamental building blocks of the language are terms and formulae. Players, pieces, and places are just a few examples of the game's objects that are represented by terms. The beginning state, legal moves, and win conditions are all represented by formulas in the game's rules. All formulas start with the symbol `<=` whereas all terms are simple expressions that are easily human-readable. Below demonstrates a sample GDL game:

```

(role player1)
(role player2)
(init (alive player1) (alive player2))
(<= (legal ?p (move ?x ?y))
    (true (alive ?p))
    (true (at ?p ?x ?y)))
(<= (next (at ?p ?x ?y))
    (does ?p (move ?x ?y)))
(<= (goal ?p 100)
    (true (at ?p treasure)))
(<= (goal ?p 0)
    (true (alive ?p))
    (not (true (at treasure)))))
(<= terminal
    (true (at treasure)))

```

### 2.4.1 Structure

Usually, GDL games follow this convention, in this order, while defining a game:

1. **Role declaration:** The GDL description starts with declaring the roles(or the players) of the game.

```

(role player1)
(role player2)

```

2. **Initial state:** The roles are followed by some initialization of the game. This can be setting blank spaces in tictactoe or it can set the players as alive like in the example above.

```

(init (alive player1) (alive player2))

```

3. **Legal moves:** Once the game is initialized, the way of making legal moves are defined using formulae, it is defined via the keyword `legal`.

```
(<= (legal ?p (move ?x ?y))
      (true (alive ?p))
      (true (at ?p ?x ?y)))
```

4. **Next state:** In this stage the game needs to know what happens if a new legal move has been made. In other words, the consequence of an action should be defined in the description of the game. In the game above, **next** read as follows: "If a player (?p) makes a move to a new position (?x, ?y), then in the next state of the game, the position of the game piece (?p) will be at the new location (?x, ?y)"

```
(<= (next (at ?p ?x ?y))
      (does ?p (move ?x ?y)))
```

5. **Goal conditions:** This defines the winning condition of the game. In the game above the two goal statements read as follows: "If a player finds the treasure, they win with a high score." and "If a player stays alive and nobody finds the treasure, they win with a low score."

```
(<= (goal ?p 100)
      (true (at ?p treasure)))
(<= (goal ?p 0)
      (true (alive ?p))
      (not (true (at treasure)))))
```

6. **Terminal Conditions:** Finally the terminating conditions, the GDL file also includes a formula where it specifies a state where the game ends. In the game above, the terminal condition is treasure being at any player.

```
(<= terminal
      (true (at treasure)))
```

## 2.4.2 Keywords

Below you may find the keywords in this language that are mentioned above but not for the game above to give a more clear understanding of

**ROLE** <role\_name>

```
(role player1)
```

```
(role player2)
```

Every game has two players, so every GDL-defined game should have exactly two players for this system to work.

**INIT** <proposition>

```
(init (control player1))
```

```
(init (cell 1 1 b))
```

INIT initializes the first game state. For instance, in a simple tictactoe game, the player that controls the x starts the game and the space at index (1, 1) is initialized as blank.

**LEGAL** <role><move >

```
(<= (legal player1 (mark ?x ?y))
```

```
  (true (cell ?x ?y b)))
```

LEGAL defines the legal moves of the GDL games. The first parameter is the player and the second parameter is the legal move that the player can make. Right below the legal keyword, there are conditions that this move has to satisfy. This would translate to: It is legal that player1 can make the move "mark" on x and y if and only if the cell which has the coordinates x and y is tagged b(blank).

**NEXT** <state>

```
(<= (next (control xplayer))
```

```
  (true (control oplayer)))
```

```
(<= (next (control oplayer))
```

```
  (true (control xplayer)))
```

NEXT defines the next state of the game. Above you can see the example which is taken from a tictactoe game. By using next the game knows what state to proceed to, for instance, xplayer



will take control after oplayer takes control and vice versa.

**GOAL** <role><value>

```
(<= (goal ?p 100)
      (has_sum15 ?p))
```

GOAL is used to define the winning conditions of the game. Essentially this means that the player ?p is awarded 100 points which is a common convention in GDL. 100 is usually the most points that you can achieve and once a player reaches 100 points the game ends. In this specific example taken from the sum15 game, a player is awarded 100 points(wins the game) if the condition has\_sum15 ?player holds true. In this context, has\_sum15 means the player with the sum of their three numbers adding up to 3 wins the game.

**TERMINAL** <state >

```
(<= terminal
      (has_sum15 ?p))
```

```
(<= terminal
      (not open))
```

TERMINAL is used to determine the conditions when the game ends, once these states are met the game is terminated. Building upon the example in the GOAL section, there are only two terminal conditions: if ?p player has reached the sum of 15 or if there are no more numbers left to play.

### 2.4.3 Preset Games

The system offers a range of games already defined within the system. Although some of these games are simple, they will help with the understanding of GDL works and they will also clarify some of the rules of these games. Also, note that these games are adapted from an online repository at <https://github.com/parthy/ggp>. Every game has two players and players take turns unless said otherwise. Please refer to Appendix A.1 part of the report to see the full list.

#### 2.4.4 Summary

GDL lies at the heart of the software as the only form of input that the software accepts is GDL. For a more formal description and detailed explanation please refer to Stanford's website: <http://logic.stanford.edu/ggp/notes/gdl.html>. Writing a GDL game can be challenging so this guide could be helpful for anyone who wants to learn more in-depth about GDL. Additionally, you may find the list of keywords typically in a GDL game in the Appendix A.2.

The system will have a set of preset games that are available to be played which are explained in the upcoming section however the system will also need to work with user-inputted GDL files. Therefore the system inputs the GDL file it needs to be understood. The system needs a mechanism to understand the game and how it is played. The section 2.5 will introduce the concept of the compiler which is how the system understands the game.

### 2.5 Compilers

Compilers are a whole other field of study under Computer Science. For the sake of this project, it is necessary to at least have a working understanding of how compilers work because the project has a process that is adapted from compilers in it. Compiling is the process of translating software source code written in one programming language(usually a higher level) into another language(usually a lower level). Simply, it is a way of converting the code that was written by humans and convert it into another form. The terms "high level" and "low level" refer to how close the languages are to a machine-level binary code. The reason behind this process is usually to translate code down to a level that the machines can understand.

This definition states that a piece of code(in this case the GDL code), should be translated into another language; even though there is no translation involved, the code is still being processed as if it is being compiled. That is why is important for the context of this project to know what the term compiling refers to.

Compiler implementations may vary depending on the type of compiler, this section will discuss general compilers and mention the necessary background for the compiler used in the implementation, it will discuss technical details of the compiler in-depth in section 5.1.

Compilers usually consist of three essential parts: a lexer, a parser, and code generation [20]. Once a file starts to get compiled it goes to the lexer where it tokenizes a series of words and whitespaces. Next, the parser parses these tokens to output an abstract syntax tree (AST).

This abstract syntax tree can be run directly by an interpreter or it can be compiled into a lower-level language. In this project, an engine just like an interpreter was used.

### 2.5.1 Interpreter-like Engine

The software doesn't actually adapt an entire compiler, technically this process isn't technically even compiling. Interpreting actually refers to executing the result line by line which is why it gets the name "Interpreter-like". The implementation actually does something different to better suit this application. Essentially, the code is lexed and parsed but not interpreted instead the AST is directly translated into a directed graph.

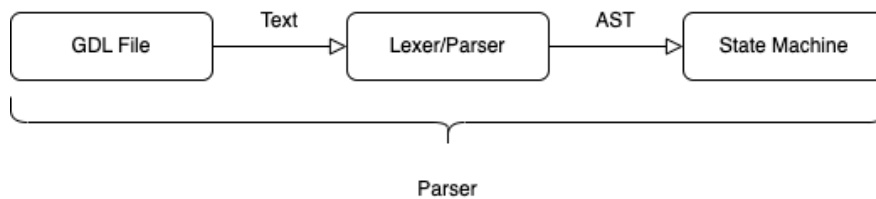


Figure 2.3: Flow diagram of GDL interpretation in the system

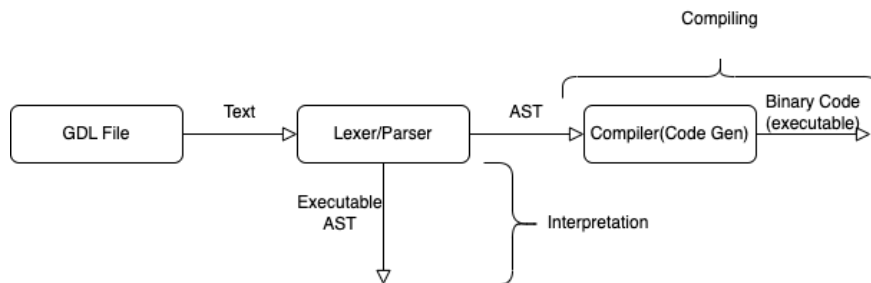


Figure 2.4: How the process would look in an actual compiler/interpreter

As seen from figure 2.3 and from figure 2.4 GDL file is directly fed into a state machine, so it doesn't need to run what parser outputs which is why it gets the title "interpreter-like" because the results from the parser are not interpreted per sé but it is transformed into a different data structure called a state machine. In a normal interpreter or a compiler, the process always ends up with an executable whereas in the system it is wrapped into a state machine however, it is clear that their origin is very much the same. It is also important to highlight the importance of state machines and how they work in the next section.

### 2.5.2 State Machines

State Machines are studied under the name Finite-state machines or more commonly known as Finite-State Automata is a mathematical structure that is used to represent a set of rules and transitions. These can be anything from theoretical maths like regular expressions to real-life examples like turnstiles.

Turnstile mechanism is very straightforward, once it is unlocked you have to tap it to unlock the turnstile once it is unlocked it tapping again will not change the state of the turnstile. This can be represented in a state machine like the figure below:

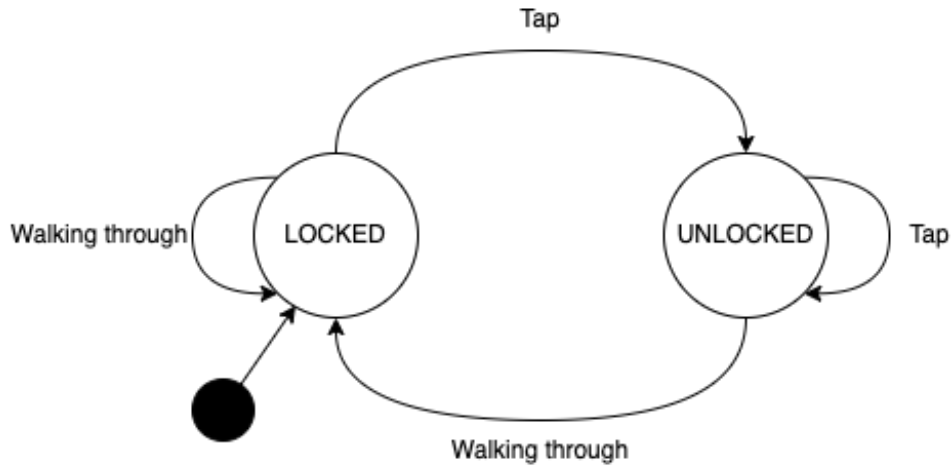


Figure 2.5: London Tube Turnstile State Machine

The figure describes a directed graph representation of a simplified version of the London tube’s turnstile algorithm. This mechanism is used to make the game playable and presentable. By using this structure, all states from any given state can be inferred, given the move the next state could be found, and much more. Therefore, the actual mechanism heavily relies on this structure. However, the mechanism standalone isn’t enough, the implementation requires a ”brain” to tell which states it can go to, and this is done via a datalog database.

### 2.5.3 Datalog Database

An entire game can be reduced into a format called GDL then it will be parsed and interpreted as trees, all there is left to do is what nodes are going to be added to this tree. A datalog is a logical programming language (a subset of Prolog) that allows the manipulation and organization of data. Although there is no official documentation, there are a few resources online [19] to fully understand the syntax of Datalog.

The term datalog database refers to a logical database system that stores the logical axioms and allows its users to make logical queries about the database. All axioms are given to the database, the logical database uses a series of computations to build relations between objects, and rules specify how these can be combined and inferred. Its design allows it to handle more complex data that cannot be handled by SQL.

The system will have a Datalog database to store all the information about the game and constantly make queries to the database to make decisions. A sample datalog database would be like this:

```
Likes(john , apples ).  
Likes(john , bananas ).  
Likes(jane , oranges ).  
Likes(bob , apples ).  
Likes(bob , oranges ).
```

These are facts like: "John likes apples" and so on. If a query like the following is made, it will return true since that fact is true:

```
?- Likes(john, apples).
```

Although there isn't a single design of Datalog databases, the design of the datalog database mentioned above is similar to the one used in the system.

## 2.6 Graph Theory

Finally, it is important to explain the outcome of this compiling process and what is produced at the end of it. The project will benefit from a structure called a "tree" (also mentioned in the title) which is formally defined as a "graph" in Computer Science. Figure 2.1 demonstrates what is referred to as a "state-graph" which is categorized as a graph. In this project, a sub-category of graphs which is directed graphs are going to be used to generate a tree.

A graph is a mathematical structure that has vertices and edges where each edge has two vertices [6]. This project will utilize these structures. Usually, edges are represented as lines, and vertices are represented as discs.

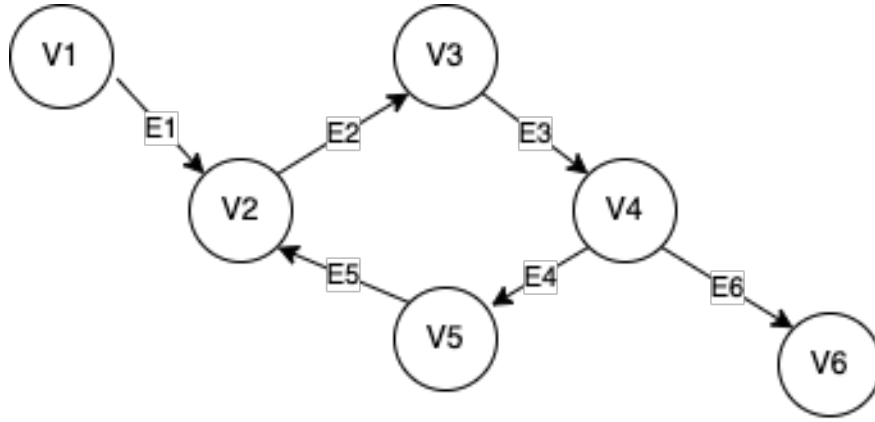


Figure 2.6: A graph with 6 vertices and 6 edges

Each disc is going to represent a "state" in the game whereas an edge will represent a move. For instance, from game state V1 if an E1 move is played the game will be at state V2. This is often written as  $(V1, E1, V2)$  in formal literature. As mentioned earlier, this project will use a really specific type of directed graph which is referred to as a "tree".

### 2.6.1 Tree

Graphs have been categorized into groups depending on their attributes. This report is going to focus on one specific type of graph type which is trees. Trees are referred to as acyclic-connected graphs. The term acyclic refers to without any cycles meaning that there is no path that any vertex can reach to itself the figure 2.7. In the figure 2.6 vertices  $(V2, V3, V4, V5)$  and edges  $(E2, E3, E4, E5)$  form a cycle; such shapes and paths are not allowed in a tree.

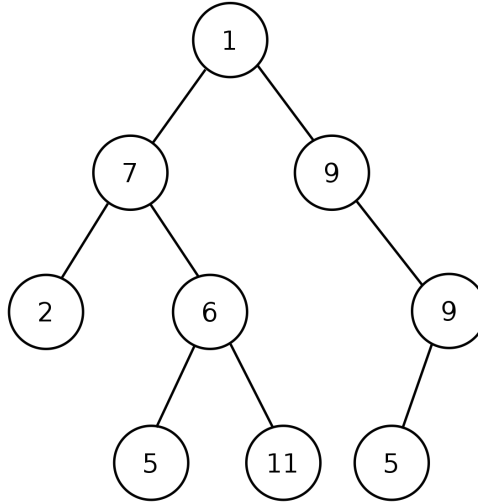


Figure 2.7: A tree

A connected graph means that there exists a path from any vertex to any other vertex. Formally it is defined as follows: “A graph  $X = (V, E)$  is said to be connected if, for any  $x = y \in V$ , there exists an  $(x, y)$ -walk.” [6] Combining these two traits, a tree would look like figure 2.7 where there are no cycles, and each node (previously mentioned as vertex) is reachable from any node. Each node has an attribute called degree (or arity) referring to the number of edges attached to it. A node with a degree of one is called a leaf. In this project, trees are going to be used to demonstrate the states of combinatorial games.

## 2.7 Literature and Existing Work

### 2.7.1 Literature

The system will benefit from work already published in the field. For instance, it will use GDL to make the concept of games more generic. There is research published [15] that extended the GDL language and helps use this more efficiently. GDL needs to be read and understood so that the system can work with all rules, states, and players.

A compiler for GDL using Ocaml was written in 2011 [18] and another compiler for GDL is proposed in 2013 [14]. In their doctoral thesis [13], Kowalski developed his own version which enables faster computation of game states than other known approaches. In 2019, an ERC-funded research project hosted by Maastricht University named Ludeme project designed

tens of games that abled users to generate trees and outperform most GDL reasoners [17]. Generating a graphML file from a python code is done via an external library called NetworkX [2] developed by NetworkX developers(an open-source project). Other than these research and developments there hasn't been any published research on the internet that is directly related to the topic of this report.

### **2.7.2 Existing Work**

Ludii Portal -part of the Ludeme Project- [1] is a similar application to the system in this project. It has been getting funding for over 5 years and it grew to a great scale, much greater than this project. It has a lot of predefined games that can be selected by the user and allows them to generate trees from the given state. It also lets the user define their own game and lets them interact with the system via simple desktop and web application UIs.

The key difference between Ludii and this project is this application uses a commonly known language called GDL whereas the Ludeme research team has come up with their own language which is a crucial distinction between this project and Ludii. Although Ludii has more games defined in the system, GDL is more commonly known and appeals to a larger audience.



# Chapter 3

## Design

This chapter will cover the design of the software: what it should look like, what it is capable of, and how the entire software is designed. After giving an overview, it will give an in-depth analysis of both the user interface was designed, how the system (the codebase) was developed, and how the user interacts with it.

### 3.1 Overview

The initial task of this project was to generate trees for two-player combinatorial games. There are two main tasks to achieve to deliver this project: a good tree-generating engine and a user interface. In figure 3.1 this idea has been visualized.

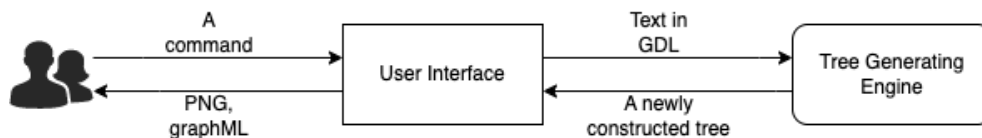


Figure 3.1: UI Design

The user puts in a "command" in order to generate a new tree, make a move, exit the game, etc. all of these will be handled by a user interface then the user interface will go to the engine to run the computations. The response from the engine will be outputted on the interface and then the program will again for a command from the user to run another computation. The program will run until the user manually terminates the app.

## 3.2 User Interface

In order for the user to interact with the system, it requires an interface to take input and show output. The app will utilize a Graphical User Interface so when the app starts there will be a screen pop-up that can be adjusted in size, simple to use, and allows the user all functionality of the system.

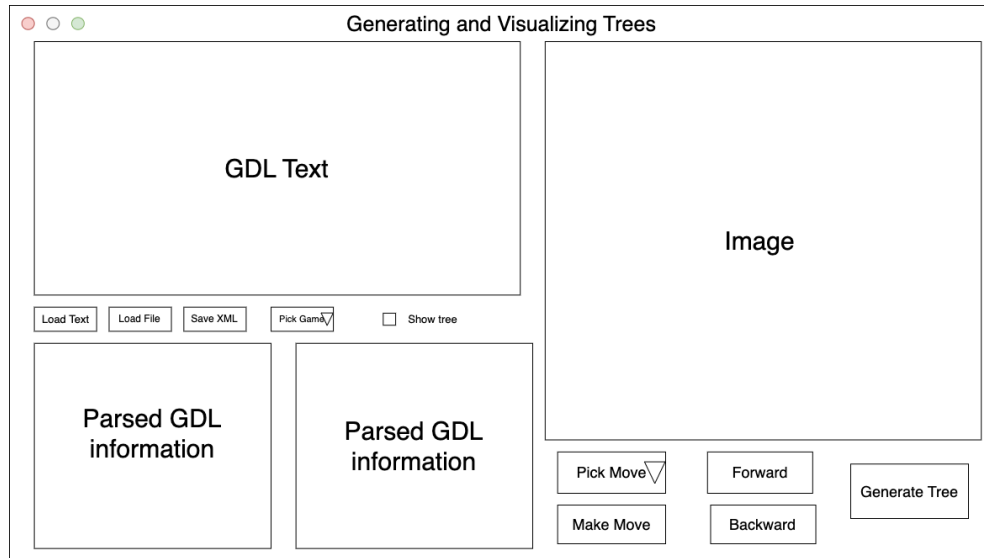


Figure 3.2: UI Design

Figure 5.1 demonstrates the outline of the user interface needed for the application. Essentially, it app needs 10 buttons for the user to fully interact with the system, this UI will be different from the product but only with additional buttons. This figure is only to represent the core buttons and screens needed.

- **3 buttons to load the game:** The app will allow its user to load from: a text file, a text box, or choose from the predefined games.
- **5 buttons for moves:** One button should be reserved for choosing the move, another one to execute the move, and 2 buttons for going back and forth on previously made commands.
- **3 buttons for tree generation:** A generate tree button is needed to create the graphML and the image of the tree from the current stage of the game. The app will first hide the image from the user so it needs a checkbox button to show/hide the tree. In order for the depth to be specified, another button is needed.

These are only the essential functionalities, later in the implantation, some extra features will be described and appropriate buttons will be added.

### 3.3 System Architecture

The system is very back-end intensive so it should be divided into multiple classes and layers to follow good coding practices.

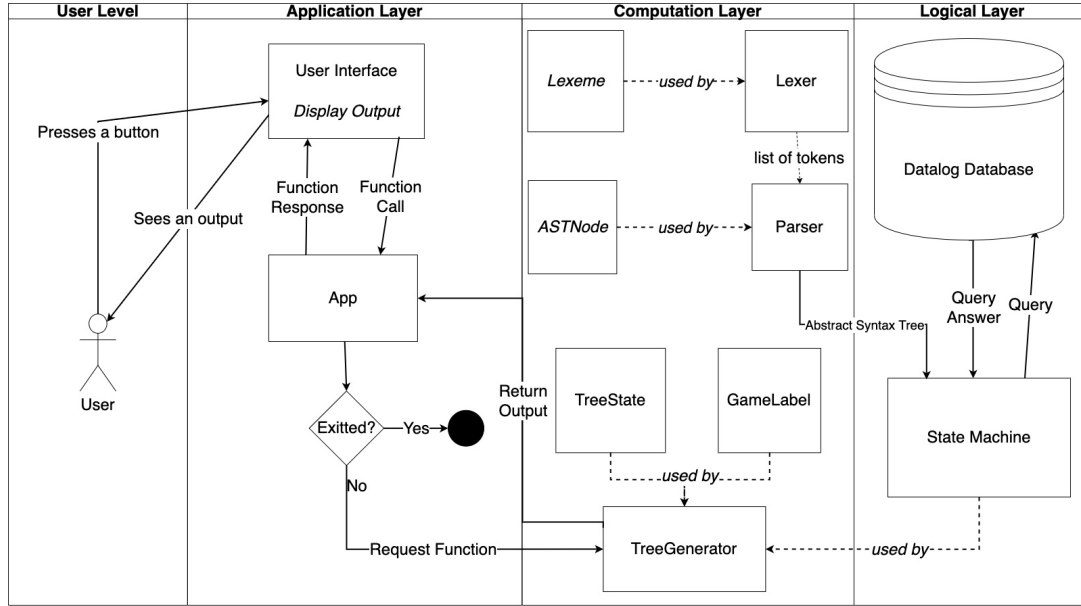


Figure 3.3: The System Architecture

As seen from figure 3.3 the application design consists of 4 layers. Each layer creates another layer of abstraction so it is important to examine them individually and understand how they are connected to each other.

#### 3.3.1 User Layer

This layer only has a user interacting with the system. Input is given to the system and output is received. As seen on 3.4 user provides the application layer with the necessary parameters for the tree to be generated.

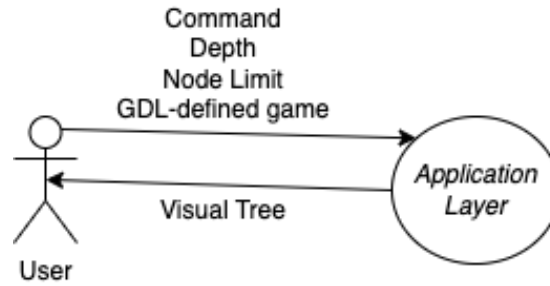


Figure 3.4: User Layer

### 3.3.2 Application Layer

This layer has the front-end-related parts of the application and acts as a middleware between the Computation and User Layer. Once the application has been started, the app class controls the current state of the application. It creates a simple UI that has buttons that the users can interact with. Every button triggers a function call within the application class. These functions can vary by saving XML files, loading a game, etc.

If a function that is related to generating a new tree is called, it will need a computational layer to execute the calculations for generating trees.

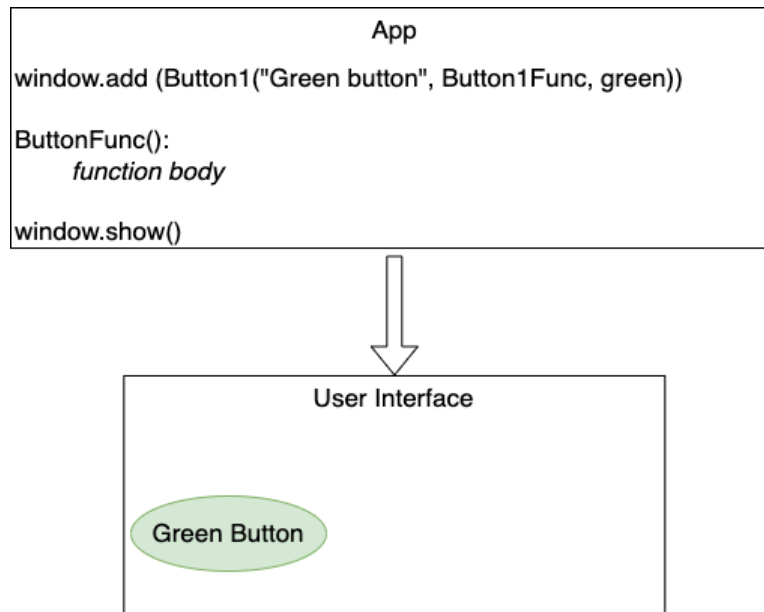


Figure 3.5: Application Layer

As seen from the simple pseudo-code in figure 3.5 demonstrates how this code will be designed and how a single button will be added. Essentially, the code will build and add all

these buttons to an object one by one defining its attributes and each one will have a function that it will be calling. That function will call the computation layer to run the calculations necessary, then the result will appear on the UI. At the end of the code, a window will be displayed and the UI will show.

### 3.3.3 Computation Layer

The crucial element in this layer is the tree generator. When a function is requested from the application class, this class runs numerous calculations to generate a tree.

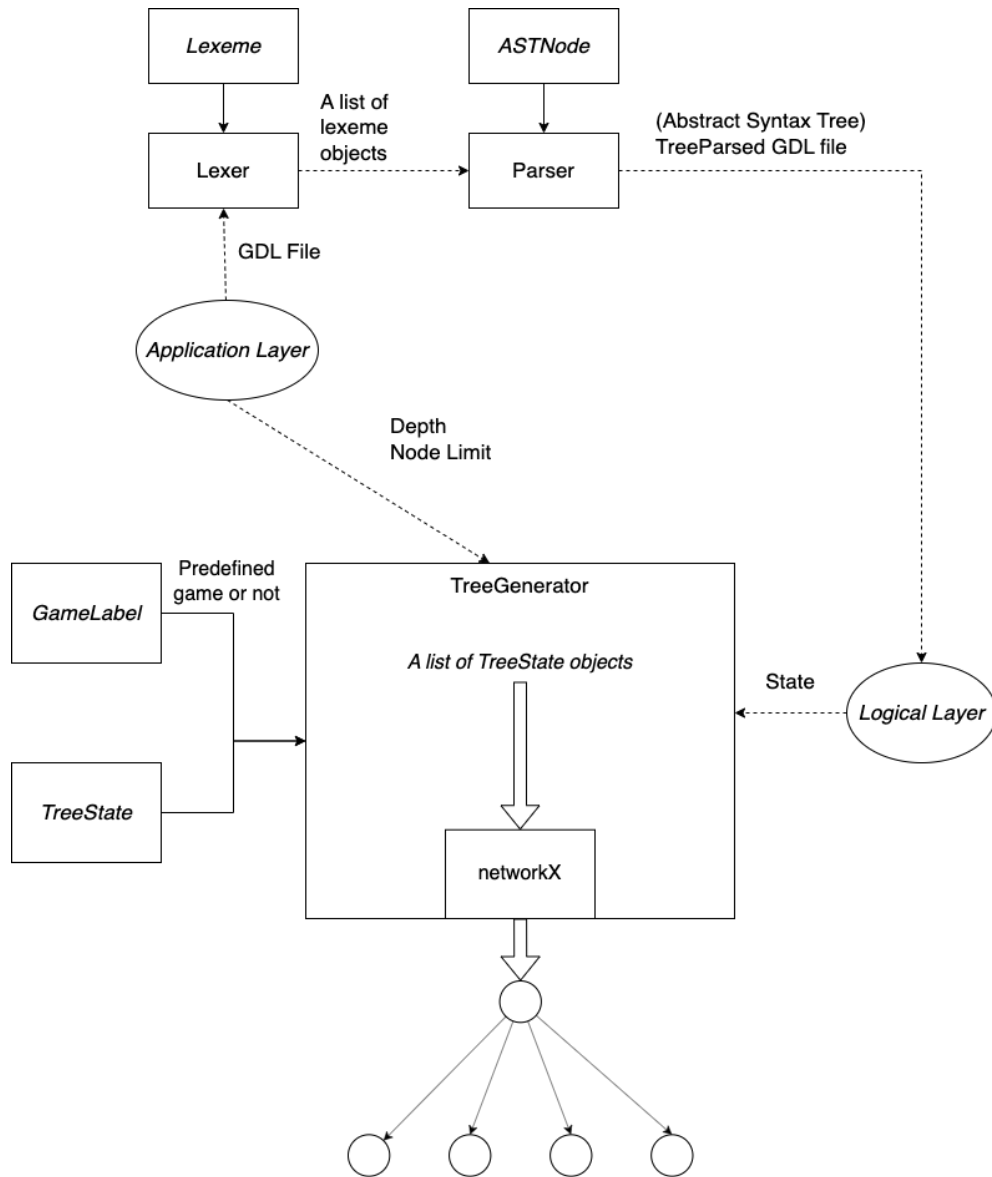


Figure 3.6: Computation Layer

This figure describes the process of treeifying a game. As seen from the figure above, certain parameters are required. The application layer passes on the user-inputted parameters like the depth of the tree, node limit (if these are set, these are not required and they have default values), and a GDL file.

The GDL file is passed onto the upper side of the figure to be processed and ready to be turned into a tree. First of all, it needs to be lexed and turned into lexeme tokens, which are then passed to a parser where these tokens are turned into an abstract syntax tree. By compiler design explained in early chapters, in order for the logical layer to understand to game it needs to be in an abstract syntax tree format because the logical layer is the stage where the "compiling" takes place. The logical layer will solve the most crucial questions in the game which are: "What are the rules?", "What state the game is at?" and "Which moves are possible from this state" and pack these into an object called state(or state machine). From the left side of the figure, it will get whether the game was one of the games already defined in the game, if so it will generate ASCII art while generating the trees for better visual aid.

After knowing where the game is at, their possible next moves, and the constraints given by the user, a list of trees will be generated which will then be passed onto a networkX library to turn that list into a tree. The arrow coming out of the TreeGenerator entity represents the generation of the tree.

### 3.3.4 Logical Layer

Besides computation related to the tree, some logical inferences are also required. This layer technically "understands" the game and how to play it. Computational Layer passes on the game in the form of a tree which is stored in the database and is divided into its rules and facts. Just like a normal DBMS(Database Management System) another class constantly queries the Datalog Database and gets the necessary information. A single-state machine object is returned to the computational layer which has been attributed to getting all its children, applying already played moves to the database, and more. The essential function of the state machine is to make the game playable and have an easier way of manipulating the database.

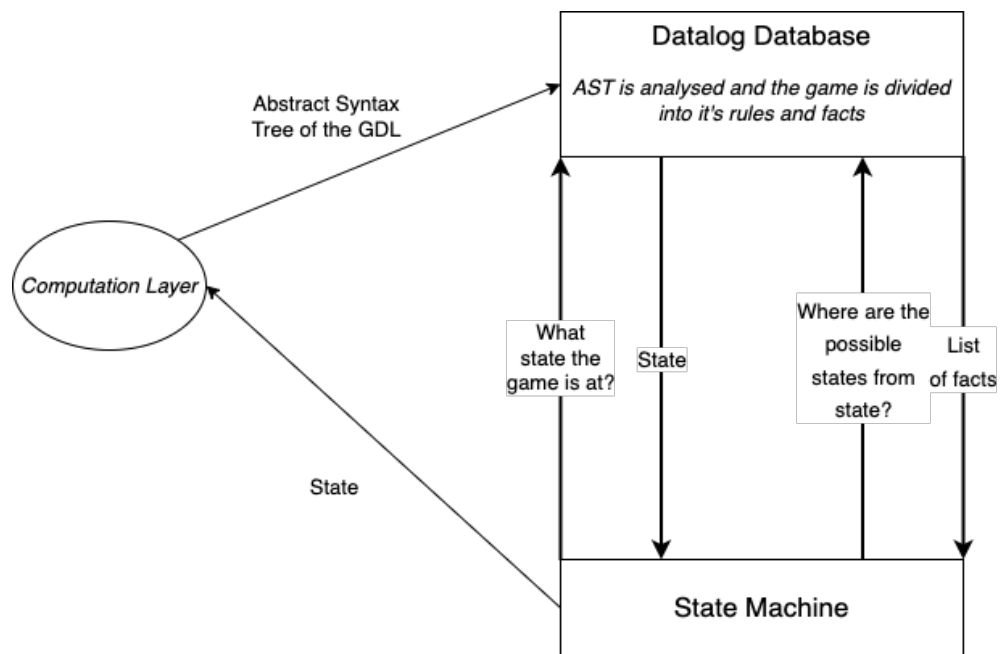


Figure 3.7: Logical Layer

## Chapter 4

# Requirements & Specification

Essentially, the system has two main functionalities: Read a GDL game from the user and generate a tree from any given state of the game. In order to achieve this, the user will need to input an initial state and a depth then the user can press run to simulate the tree.

The requirements can be divided into two groups one of them being User Requirements and the other being System Requirements. They will cover the expectations of users and the technical requirements respectively.

### 4.1 User Requirements

With specialized functions like viewing created trees and downloading graphML files, the system's main goal is to facilitate user involvement and interactivity. A simple user interface with a few buttons would be sufficient to accomplish this goal. This strategy would make it possible for users to successfully access and interact with the system's outputs, such as created trees and downloadable graphML files. Such an interface would be user-friendly and offer the best environment for users to communicate with the system, achieving the system's main objective.

1. The user can input a GDL game into the system.
2. The user can change the state of the game and generate a tree from that given state at the desired depth.
3. The user should be able to see the tree in some visual form.
4. The user can access the graphML file after interacting with the interface.
5. Every button should be intractable and functional.



6. The user should be able to understand the tree, its states, and the moves when looking at the GraphML file. In other words, the graphML file should enable the user to easily identify and interpret different states and moves.

## 4.2 System Requirements

The primary objective of the system is to provide a visual representation of the transition between different game states. To accomplish this, the system necessitates an engine capable of parsing GDL files. Once the file is parsed, the system must convert it into a graph and display it in one form: a graphML file.

### 4.2.1 GDL Parsing

1. The GDL parser should be able to parse GDL keywords such as roles, init, base, etc. correctly.
2. The parser should be able to distinguish between problematic games, such as games with more than two players or non-terminating games, that cannot be used in the system.
3. The parser should be able to parse complex games with multiple rules and conditions.

### 4.2.2 Tree Generator

1. The game generator must be able to collect information from the GDL parser and store it in the correct data structures.
2. The game generator should allow the user to input a desired depth and initial game state, and generate the game tree accordingly.
3. The software should produce correct trees for all games.

## 4.3 Specifications

There are other things to check in software other than what it is capable of nor capable to do. Specifications will highlight and try to challenge how the software is built rather than what it can do. This will ensure that the code is was built in the right way.

1. The software follows the Code of Conduct and Code of Good Practice by the British Computer Society.

2. The system shouldn't store any personal data of any of its users.
3. The software should produce trees of simple games until they terminate.
4. Users can interact with the tree and generate trees as they wish.

## Chapter 5

# Implementation & Testing

This chapter will demonstrate how the design in chapter 3 was implemented and it covers how testing was done on the software.

It will start off with the first describing the process of parsing and how a GDL file is read, interpreted and turn into an object that allows objectifying the game described in GDL. Then it will cover how this object is treeified and converted into another type of object that represents a tree. One of the most important elements of the application is the user interface so it will cover how the GUI was implemented and explain how trees are visualized so that they become visually appealing. This section will finish off by explaining external resources and explain the testing conducted against the requirements explained in chapter 4

While developing the implementation an incremental approach was taken. Every bit of the software has been cut into distinct parts and those were developed individually. This was particularly helpful because the software was more back-end intensive than front-ends. Also, this ensured that all classes were well encapsulated, enabling loose coupling.

### 5.1 Parsing GDL Files

The first thing the system does is interpret the GDL files when a GDL file is input. The system uses a mix of conventional and system-specific methods to interpret such files. This section will cover the entire process of parsing through each class used in the implantation.

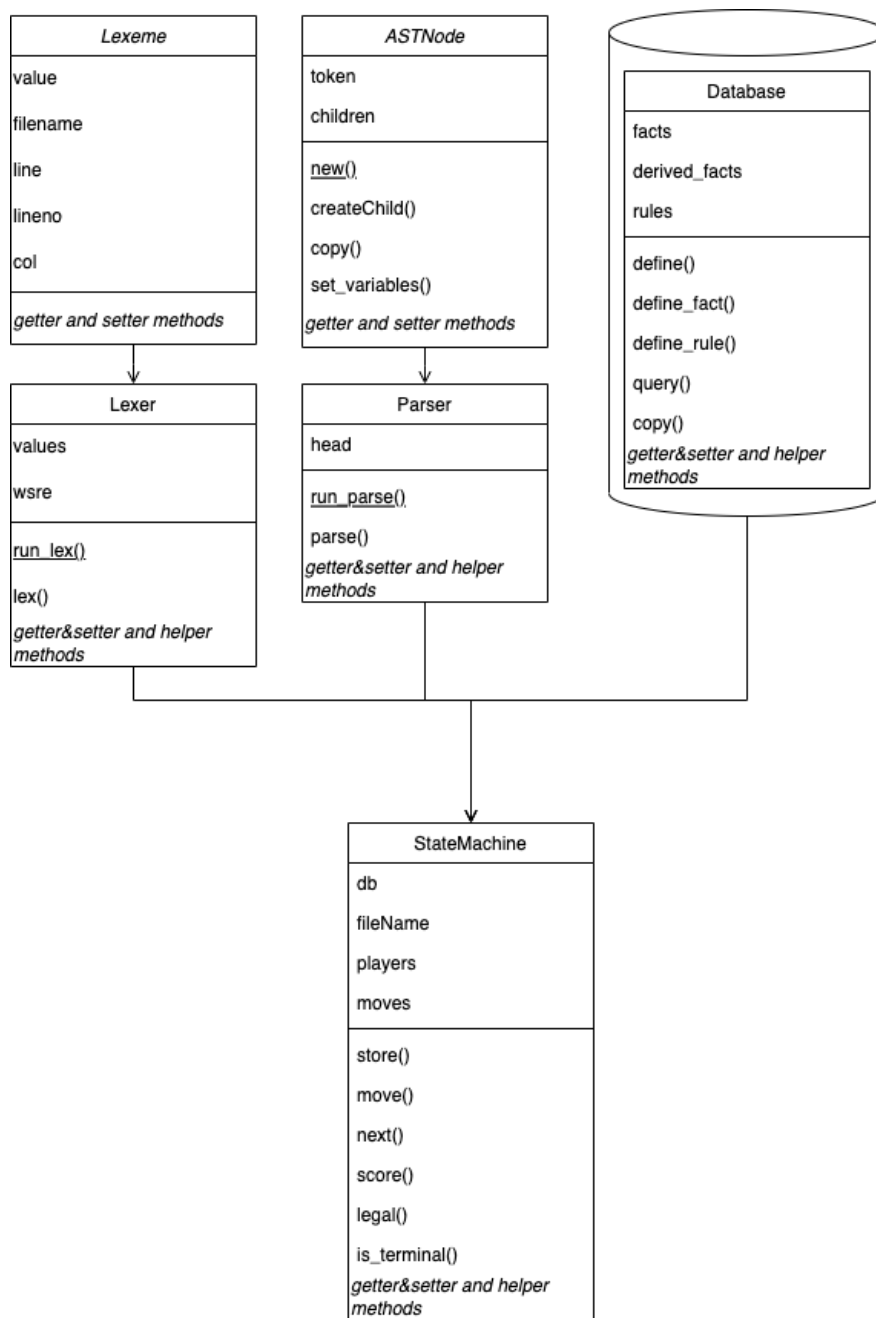


Figure 5.1: Class Diagram

The class diagram above demonstrates how the parser operates as a whole. Each class has a dedicated section it which will cover the depth of what they do.

### 5.1.1 Lexeme

The word "lexeme" is formally defined as: "a unit of meaning in a language, consisting of a word or group of words" by Cambridge Dictionary [8]. In the context of this project, lexeme

refers to the most basic unit of lexical analysis. In literature, lexical analysis is also referred to as tokenization, a lexeme object representing a single token.

The python class is initialized with 5 parameters:

- **filename:** the name of the file that the token belongs to.
- **line:** the actual line the token is taken from
- **lineno:** the line number of the generated token
- **column:** the position of the first character of the token
- **value:** the value of the token, what is actually needed.

These objects are generated in the lexer class during lexing. This class has a method to generate a new token when a new token is needed upon the new one. Also, it checks to see whether a token value is a special case like opening and closing parenthesis. To put this into perspective here is an example of a sample lexing and what are the lexeme tokens that are generated:

```
(role player1)
(init (heap a 1))
```

Figure 5.2: The Sample GDL Snippet

For the sample GDL snippet, the following objects will be generated:

```
Value: (, Filename: nim1.gdl, Line: (role player1) , Line no. : 1, Column: 1
Value: role, Filename: nim1.gdl, Line: (role player1) , Line no. : 1, Column: 2
Value: player1, Filename: nim1.gdl, Line: (role player1) , Line no. : 1, Column: 7
Value: ), Filename: nim1.gdl, Line: (role player1), Line no. : 1, Column: 14
Value: (, Filename: nim1.gdl, Line: (init (heap a 1)), Line no. : 2, Column: 1
Value: init, Filename: nim1.gdl, Line: (init (heap a 1)), Line no. : 2, Column: 2
Value: (, Filename: nim1.gdl, Line: (init (heap a 1)), Line no. : 2, Column: 7
Value: heap, Filename: nim1.gdl, Line: (init (heap a 1)), Line no. : 2, Column: 8
Value: a, Filename: nim1.gdl, Line: (init (heap a 1)), Line no. : 2, Column: 13
Value: 1, Filename: nim1.gdl, Line: (init (heap a 1)), Line no. : 2, Column: 15
Value: ), Filename: nim1.gdl, Line: (init (heap a 1)), Line no. : 2, Column: 16
Value: ), Filename: nim1.gdl, Line: (init (heap a 1)), Line no. : 2, Column: 17
```

Figure 5.3: Lexeme tokens produced

### 5.1.2 Lexer

Lexer class is the one that turns programs like the one shown in figure 5.2 into lexemes. It is an essential part of the parsing process as lexemes are the foundation of abstract syntax trees.

The lexer that is used in this implementation has taken a slightly less common approach to lexers as it is written in an object-oriented manner. Generally, compilers (therefore lexers and parsers) are developed functionally using functional programming languages like OCaml and Haskell. This is discussed in a StackOverflow thread on <https://stackoverflow.com/questions/2906064/why-is-writing-a-compiler-in-a-functional-language-easier>.

This class has been designed to generate a list of lexemes (like the one in figure 5.3 ) from a series of characters, the program. The object is initialized with two parameters:

- **values:** a list that will store the lexemes
- **wsre:** the name "wsre" is an acronym for whitespace regular expression. This compiles a regular expression that ignores the whitespaces [3]

To use this class, a static method named `run_lex()` method has been created so that it can be used without needing a lexer object. When this method is called it creates a `Lexer` object and calls the non-static method `lex()`. `Lex()` method checks whether there is any data, if there is it cuts it into lines and passes it onto the method `_lex_input()`. From this, every element on the list of lines is then passed onto a method called `_lex_line()`. This method will process every line, in the meantime, it will add the necessary values to the values list thus returning a list of values at the end of this method. Then the `_lex_line()` iterates over every character in the line and calls the method `_process_char()`. This method checks for opening and closing parenthesis, whitespaces and whether it reached to end of the line to lex the lines into words and parenthesizes. Once it finds a sequence of the suitable lexeme, it creates an object and adds it to the values list.

### 5.1.3 ASTNode

Before getting to parsing, where the tokens are going to be put into an abstract syntax tree, `ASTNode` class needs to be explained. `ASTNode` class defines a single node in the Abstract Syntax Tree. The parser will use this structure to form the AST. Here are the `ASTNode`'s initializing parameters:

- **token:** This is a `Lexeme` object of the token it is going to represent
- **children:** This is a list that holds children of this node in `ASTNode` type.

This class allows the tree-like structure to be formed through its methods. It allows to create children to form a tree, it has a few getter and setter methods that are used in the class

which will be crucial in the logical inference of the tree. Here is the abstract presentation of the program in figure 5.2 and its tokens 5.3.

When the program is tokenized, parsed and when the AST node is printed. There are two distinct AST nodes generated:

```
Term: role , Arity: 1
Token: 'role '
Children: [
    Term: player1 , Arity: 0
    Token: 'player1 '
]

Term: init , Arity: 1
Token: 'init '
Children: [
    Term: cell , Arity: 3
    Token: 'cell '
    Children: [
        Term: 1, Arity: 0
        Token: '1 '
        Term: 1, Arity: 0
        Token: '1 '
        Term: b, Arity: 0
        Token: 'b '
    ]
]
```

Figure 5.4: The AST fo the figure 5.3

If this was visualized in a tree form, here is how it would look:

#### 5.1.4 Parser

This is the class that turns the list of lexemes in figure 5.3 into a tree-like structure in figure 5.6. It is designed in a really similar way to Lexer where it has a static method called `run_parse()`

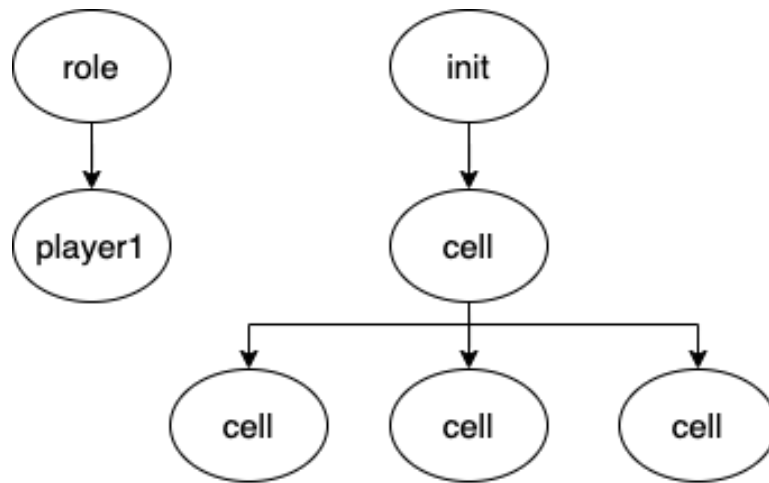


Figure 5.5: syntax tree

that calls the `parse()`. During the initialization of the Parser class, it creates `ASTNode` as a head. This is because all programs actually originate from a single head and every single becomes a new child of the head. So in figure 5.6 the parse class does this under the hood:

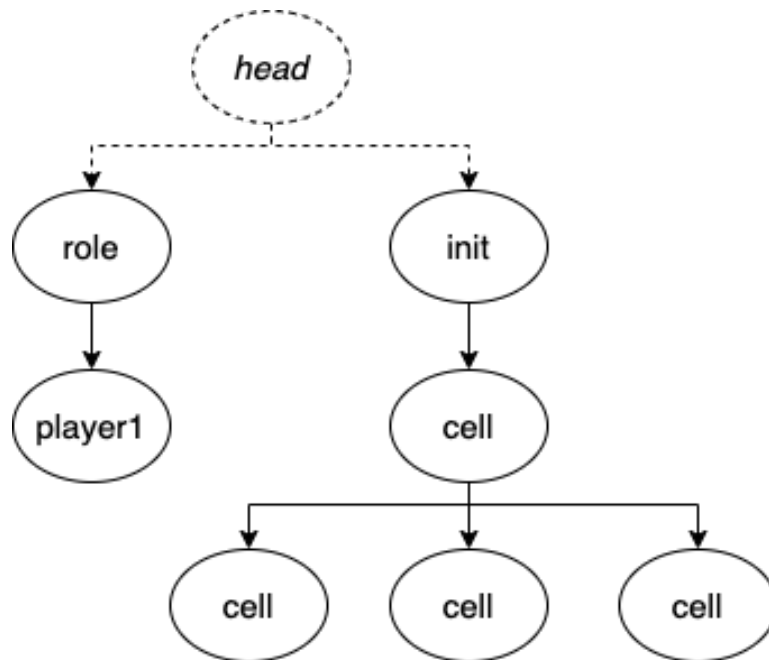


Figure 5.6: The tree version of figure 5.6

The essential function of the parser is to give the tokens a hierarchical structure and follow the syntax and semantics of GDL. When it encounters a parenthesis or a new keyword it creates a child of the current node. If it encounters an error parsing it raises a parser error. Here are the errors it can raise:



- **EXPECTED\_CONSTANT:** When the parser expected a constant (anything other than '?', '(' and ')') but doesn't get one
- **DOUBLE\_NOT:** When there are two consecutive "not" operator
- **UNEXPECTED\_CLOSE:** When there is a closing parenthesis without an opening parenthesis
- **MISSING\_CLOSE:** When there isn't a closing parenthesis for an opening parenthesis
- **BAD\_PREDICATE:** The arity of the node doesn't match what it is supposed to according to the reserved list. The reserved list holds the keywords of GDL and their arities. Refer to the table A.2 in the appendix to see the full table of keywords.

Some of the games from section 2.4.3 weren't be able to be parsed so the parser class was extended and adapted to fit a larger set of games.

### 5.1.5 Database

The database is a datalog database as mentioned in section 2.5.3. This class acts like a logical engine in the system. Just like a normal database management system, it provides some functionality for outside classes to use as an API. Since it is based on Prolog its functionality is also very similar. It stores all the facts and rules in the database object. Another class can query a statement to check whether it is correct or not and also define new axioms in the database. A sample database snapshot can be found in the Appendix A.3

This class store 4 main lists:

- **facts:** This is a list of current facts that are true. These facts are the roles (the two players), facts that are initialized using init, and the actual facts that are true in the game (like in a board game a cell being occupied)
- **derived facts:** This list is used when the game proceeds and new facts are coming in so facts need to be derived. For instance, when a player makes a move the legal moves and the current state of the game
- **rules:** Rules store keywords that are standard to every game like next, legal, terminal, and goal. It also stores the rules of the game-specific keywords and how they become true.

- **requirements:** Requirements list stores game-specific keywords and how they are structured.

This class is very complex and has a lot of private methods that help this function. To understand how this is implemented, an analysis of every function defined (documented as the API of the class) would suffice:

- **define():** This is a general definition function to write something into the database. It takes an AST as input, if the node is a rule it calls the define *define\_rule()* function, and if it is a fact it calls *define\_fact()*. In the Lexeme class, there is a method named *is\_rule()* to decide this.
- **define\_fact():** To define fact, three parameters are needed: the name of the fact, the arity of the fact, and a list of AST node arguments of that fact. First, these arguments are checked and ensured that they do not contain any variables or the keywords "or", "distinct", or "not". This check is referred to as a sanity check within the document.
- **define\_rule():** This function has an extra parameter which is the body of the argument. It has numerous sanity checks. First, it checks that the head of the rule does not contain any of the keywords "or", "distinct", or "not". Then, it checks that the rule does not create a recursive cycle that contains a "not" edge. Finally, it checks that the rule has a variable in the head or a negative sentence ('or' or 'distinct') in the body that also appears in a positive sentence in the body.
- **query():** This is the method to retrieve whether something is right or wrong. It checks all the facts and derived facts to see whether the following query is true.
- **copy():** Makes a copy of the database.

For instance, the program given in figure 5.2 would be stored as ('role', 1): [[xplayer], [oplayer]] in the database since they are facts. As defined in *define\_fact()* method, the first parameter is the name of the fact which is 'role', the second parameter is 1 which is the arity and the third parameter is the values stored in a dictionary format.

Essentially, the state machine uses this as a data warehouse which separates the data in a certain way and makes logical inferences. State machine class will utilize this database by querying and defining new facts.

### 5.1.6 State Machine

State Machine's sole purpose is to use make translate the act of game playing to logical premises to the database. This helps with creating a flow in the game, it constantly updates the database with recent moves and changes in the game state. It has a function to store the data in its database. Then it can update the database by inputting a player move, updating the current state of the game, updating their scores, and getting possible legal moves (which is essential for generating trees).

Just like the Database class this class also has an API with only a few functions:

- **store():** This is a function that takes the GDL file, lexes it then parses it, and then passes it onto the instance variable database. It iterates over the trees generated and checks whether they pass certain criteria, if they all pass the trees are added to the database
- **move():** Takes in two parameters: player and move. Adds the fact: ('does', 2, [player, move]) to the database.
- **next():** Updates the database by applying the does and turning them into true facts. It returns a fresh copy of the database.
- **score():** It returns the score of the asked player by querying the database about the score of the player.
- **legal():** Checks whether a certain player can do a certain move or not. If there is no move input then all of the possible legal moves are returned. If neither is provided returns the legal moves for all players. This is essential when generating a tree because all legal moves from that state will become the children of that state
- **is\_terminal():** Check whether the game has come to an end.

### 5.1.7 Summary

The multi-layer architecture of the parsing process makes the GDL come as a playable state machine at the end. Firstly, when a GDL is inputted a file is lexed into lexeme tokens, then these tokens are parsed into ASTNode which are abstract syntax trees of the sentences. These nodes are passed onto a database which passes them onto a state machine that makes constant queries and alterations to the database so that it changes the game state. Each state machine

object represents a single state with its facts in the database which can change to update the state of the game.

## 5.2 Tree Generation

This is the part where every piece comes together and fits like a jigsaw puzzle. As previously described in figure 3.6 TreeGenerator is where the state machine is used to retrieve the states to put in the tree.

By design, the TreeGenerator class takes a state of a game and turns it into a tree. Once it has a starting point of where to start the tree, the state machine gets all legal moves from class methods and generates a list of them. The *generate()* method is the main method that generates the trees.

Generate method uses a method called "level order traversal using queue" method which is a way of traversing trees. This method was chosen because it fit perfectly with the tree generation process. It has an iterative approach and a really simple implementation explained in 5.2.1. Here is the level order traversal using the queue algorithm.

---

**Algorithm 1** Level Order Traversal Algorithm

---

```

Create an empty queue  $q$ 

Push  $root$  in  $q$ 

while  $q$  is not empty do
     $currentNode \leftarrow$  first element from  $q$ 
    process( $q$ )
    Push  $currentNode$ 's all children to  $q$ 
    Remove  $currentNode$  from  $q$ 
end while

```

---

First of all, a new queue is created to store these states. The initial root  $q$ , which is the initial state, is pushed on the queue. the root  $q$  is processed and all of its children are added to the queue. This means that after each iteration a level is present in the queue as the root is always removed and all there are left are the children of the root.

This algorithm helps a lot in generating trees in the implementation as the aim of the tree is to find the playable moves from a state. The queues elements are states however states cannot be directly put into a queue which is why there is another class called TreeState that holds all the necessary information to generate the tree which has state, id, depth, graphLabel,

isTerminal, and winner information in it. While depth is related to how big the tree is, the other attributes are related to the visual representation of the state. Similar LVT algorithm, the same method is used in creating a graph from a single state.

---

**Algorithm 2** Creating a Graph from Tree State

---

```

Create an empty graph  $g$ 

Create an empty queue  $treeStateQueue$ 

Generate  $firstTreeState$  with  $firstState$ 

Push  $firstTreeState$  in  $treeStateQueue$ 

while  $treeStateQueue$  is not empty do
     $currentTreeState \leftarrow$  first element from  $treeStateQueue$ 
    add  $currentTreeState$  to  $g$ 
    Push  $currentTreeState$ 's all children to  $treeStateQueue$ 
    for all  $childNode$  in  $currentTreeState$ 's children do
        add an edge from  $currentTreeState$  to  $childNode$  in  $g$ 
    end for
    remove  $currentTreeState$  from  $treeStateQueue$ 
end while

```

---

The algorithm starts with creating an empty graph object,  $g$ , and an empty queue. From the initial state, an initial tree state is generated, then that tree state is pushed into the queue. To keep track of the current processing state the first element of the queue is stored in a separate variable. The current element is put into the  $g$  object and all of its children are pushed into the queue. In order to find all the children of a state, a function in the util class called *getAllLegalMoves()* method is called. All of its children make an edge from the parent state. The initial current state is removed from the queue and this process is repeated until no state is left in the queue. In the end, the graph object will have nodes and edges that make up a tree.

When generating trees there are two different limit controls. The first one is the depth of the tree, or levels, which determines how deep the tree will go. For instance, if the depth is given as 1, the tree will only generate the children of the root node but if the limit was given as 2 the children of children will be generated. In the 0 case, the tree will generate all the children of all the nodes until the game terminates (the image might get unclear if generated from the very beginning of the game). The other limit is the node limit where the user can limit the number of total nodes generated in the tree. These two variables can be set in the

user interface however if not changed, both of their default values are 1 for depth and 100 for nodes.



Figure 5.7: Limit Buttons on the Interface

Alongside the *generate()* method, there is also a method named *generateWithMoves()* which will generate the tree below with only a single move tree in a single line. It takes a parameter called *movesList* and it essentially applies all the moves present on that tree. The below figure demonstrates how this looks on a simple tictactoe game

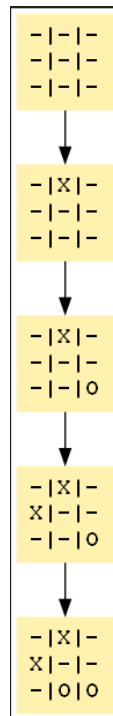


Figure 5.8: Single Move Tree

Alongside the methods already described, there are a few more methods that in the Tree Generator class, here is a list of what each of them do:

- **reset():** Before the generate method starts again, some of the variables have to be resettled back to empty values. These variables are: *gameTreeDot*, *gameTreeML* and *id\_counter*. The first two are tree objects and the last one the counter for the state id (when each state is created the counter goes by one so each state has a different id).

- **getStateFromFile():** This method is used when a state is not passed on the tree generator object, and it helps to find the first state of the game.
- **newTree():** Using the networkX library, an empty directed graph is generated.
- **generateTreeState():** Using state and other variables it generates a TreeState object
- **addStateToTrees():** Adds the tree states to gameTreeDot and gameTreeML using id, labels, and some parameters related visual of the tree.
- **createFiles():** The gameTreeDot and gameTreeML objects are turned into actual files of dot and graphML respectively. These files are stored within the results directory and the image version of the dot file is generated and saved into the results directory.

### 5.2.1 Level Order Traversal

Level Order Traversal (LVT) of trees is a method used for traversing trees in a left-to-right manner [10]. This method allows the tree to be generated on a level basis meaning it aims to output all the nodes of a certain move before moving on to the next one.

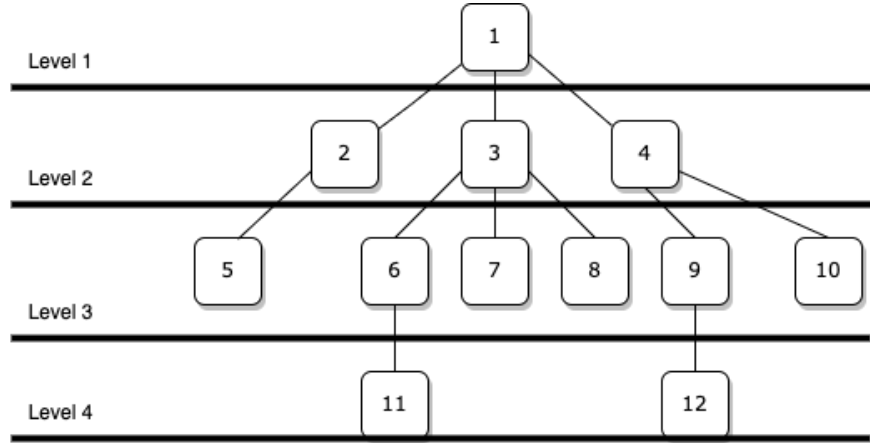


Figure 5.9: A sample tree

In figure 5.9, a sample tree is given. If the tree was generated using LVT, the nodes would have been generated in arithmetic order. In other words, First, level 1 (1), then level 2 (2-4), then level 3 (5-10), and finally level 4 (11-12). This allows the system to explore the options of a game state before going on to another state which is the desired outcome.

## 5.3 Graphical User Interface(GUI)

There are two main reasons a GUI was required for this application. Firstly, the most important part of the project is to generate and visualize these trees, therefore the application requires a graphical interface to deliver these outputs. The second reason is for the user to interact with the system. It wouldn't be practical and user-friendly to have a terminal-based simple scripting interface to use the entirety of the software. A GUI allows the users to interact with the system and have downloadable versions of the trees.

When parsing, generating trees or making moves there was only an abstract concept of GDL and GGP, all of the moves are being made as `move('xplayer', '(mark 1 1)')` and there hasn't been anything specific to a game. This distinction between games having nicer graphs has been done by creating specific labels for the predefined games. For instance, the game tictactoe actually displays the board rather than writing `(mark 1 1)`. Here are a few demonstrations of how they look:

```
PAST MOVES :
1 : xplayer - (mark 2 1)
2 : oplayer - (mark 3 3)
3 : xplayer - (mark 1 1)
4 : oplayer - (mark 3 1)

STATE INFO :
(cell 1 1 x)
(cell 1 2 b)
(cell 1 3 b)
(cell 2 1 x)
(cell 2 2 b)
(cell 2 3 b)
(cell 3 1 o)
(cell 3 2 b)
(cell 3 3 o)

TURN : xplayer

LEGAL MOVES :
(mark 1 2)
(mark 1 3)
(mark 2 2)
(mark 2 3)
(mark 3 2)
(mark 3 3)
```

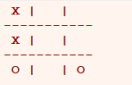
A 3x3 grid representing a TicTacToe board. The top row contains 'X', '|', and '|'. The middle row contains 'X', '|', and '|'. The bottom row contains 'O', '|', and 'O'. Horizontal and vertical dashed lines separate the rows and columns.

Figure 5.10: TicTacToe

```
PAST MOVES :
1 : player1 - (reduce c 2)
2 : player2 - (reduce c 0)

STATE INFO :
(heap a 1)
(heap b 5)
(heap c 0)
(heap d 2)

TURN : player1

LEGAL MOVES :
(reduce a 0)
(reduce b 0)
(reduce b 1)
(reduce b 2)
(reduce b 3)
(reduce b 4)
(reduce d 0)
(reduce d 1)
```

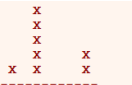
A visualization of the Nim game state. It shows four rows of 'X' characters. The first row has 1 'X', the second has 2 'X's, the third has 3 'X's, and the fourth has 4 'X's. Horizontal dashed lines separate the rows.

Figure 5.11: Nim

These labels are also nicely put into the tree image generated in the figure below:



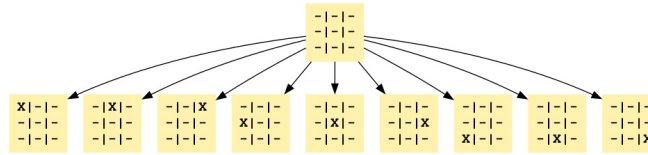


Figure 5.12: How the state labels look in the tree

To achieve this a simple UI with a few buttons was necessary just like the one described in the section 3.2. An external library called PyQt5 handled this task which had numerous functions to create an interface. This part of the project is done in the initialization method of the app class.

## 5.4 Tree Visualisations

Tree visualization is the last and most critical step of the application. State Machine itself cannot be visualized so it needs it can be broken down into a form that can be used in a tree. Therefore this state machine should be put into another class where its states are put into another structure of a TreeState. Then it will be ready to be visualized.

An external library called networkX [2] was used in this implementation. It is a library that uses a predefined data structure called DiGraph(short for directed graph). Once this object is created the library can turn this into a dot file and a graphML file which was the form of the graph. The dot file is the basis for the image created in the UI. The reason for this is networkX library has better visual presentation built-in functions that make it easier to have nice and colorful images.

Both of these formats share a common ground in terms of they are generated. Firstly, they are inertly created from the same type of object using the library but using different functions. They are generated two using two different objects because a dot tree can take more parameters like font and color but graphML tree object them slightly different parameters. These will be discussed in their respective sections. Since they are built-in functions this report will not go deep into how they are created under the hood. These files are generated via a function called *createFiles()* in the TreeGenerator class in the TreeGenerator.py file.

Here is how the dot file is written:

```
dot = nx.drawing.nx_pydot.to_pydot(self.gameTreeDot)
dot.write(self.fileNameDot)
```

This line converts the dot file into an image:

```
dot.write_png(self.fileNamePng)
```

Lastly, the code that converts to graphML:

```
nx.write_graphml_lxml(self.gameTreeML, self.fileNameMl)
```

Here is how the overall process works:

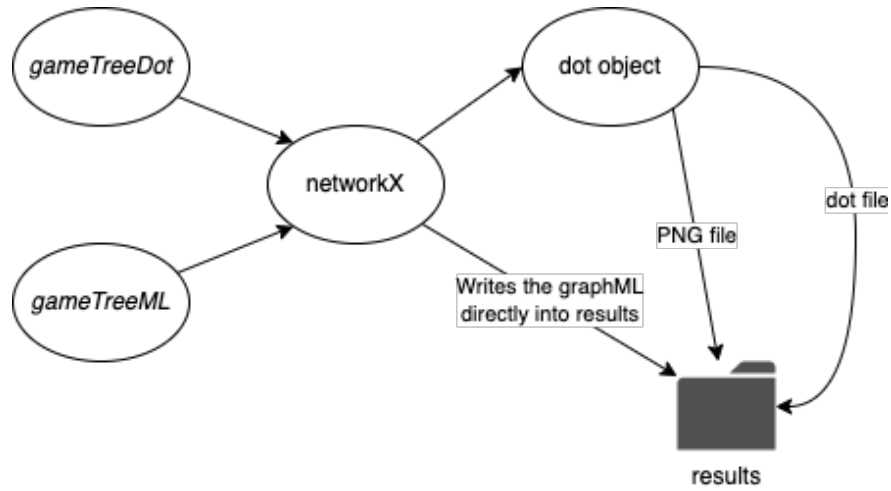


Figure 5.13: networkX Flow

Here is the difference between two graphs (figure 5.15 and figure 5.14) when they are compiled the same Tic-Tac-Toe tree:

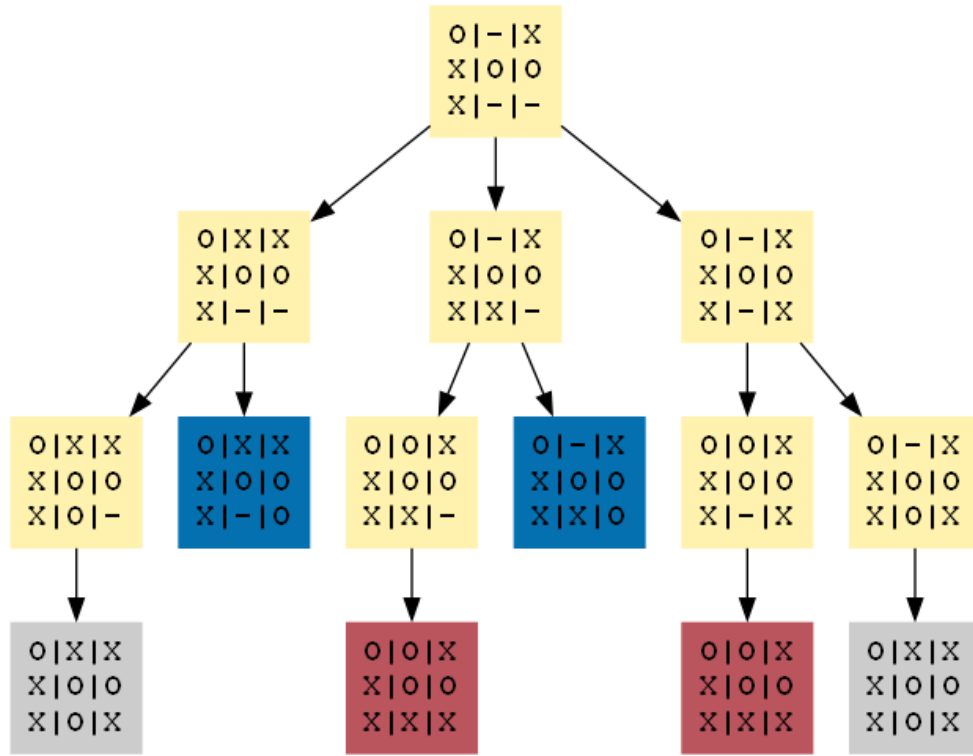


Figure 5.14: The dot graph

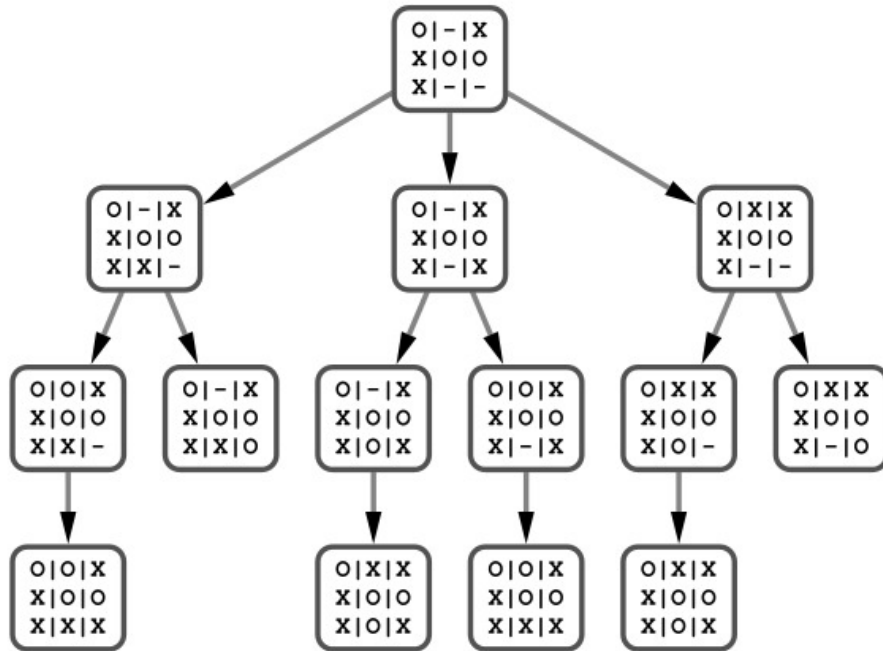


Figure 5.15: The GraphML graph

Note that there are many graphML compilers out there and each of them can compile it differently, this figure 5.15 has been formatted in cytoscape [7] (a network visualization platform). If the graphML was plotted using networkX, the figure 5.16 would be the outcome.

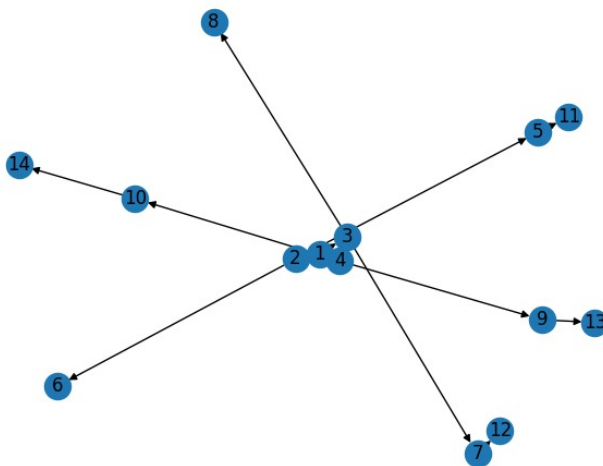


Figure 5.16: NetworkX Compilation

#### 5.4.1 Dot Content

As mentioned before, dot files can be edited using python code. Once styled they can then be translated into a PNG file however, in order to translate it into a PNG file, dot files are needed to be saved. These are the parameters of a dot file:

- **Fontname** : Courier-bold
- **Fontsize** : 15
- **Shape** : square
- **Label** : (State label)
- **Color** : (color)
- **Style** : filled

Label and color are different for every node. Label differs from game to game. Some games are already defined so their labels depict a real-world representation of the game while others are represented by how they are used in the GDL syntax.

Here is an example node:

```

1 [color="#FFF2AE",
    fontname="Courier-bold",
    fontsize=15, id=1,
    label="O|-|X
        X|O|O
        X|-|-",
    shape=square,
    style=filled];

```

And if there were another state going from state 1 to state 2 it would be shown like this:

```
1 -> 2
```

Here is the color palette used in the implantation:

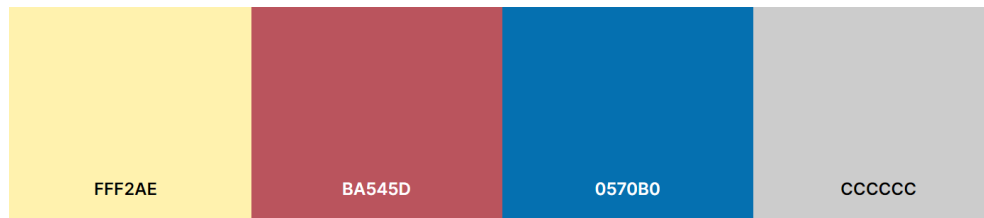


Figure 5.17: Colour Palatte

The yellow represents a non-terminating state, the red represents the first player winning the game, the blue represents the second player winning the game and lastly grey represents a draw in the game where the game ended without a winner. These values are determined here is how this looks in code:

```

if s.winner == 1:
    color = "#BA545D"
elif s.winner == 2:
    color = "#0570B0"
elif s.isTerminal:
    color = "#CCCCCC"
else:
    color = "#FFF2AE"

```

## 5.4.2 GraphML Content

The graph type doesn't contain any styling it only stores the necessary information:

- **Label:** (StateLabel)

- **StateType:** (StateType)

The label would be the same with the dot file for the same tree. Since there is no color in graphML, StateType represents who won the game. 0 means the game hasn't ended yet, 1 means player 1 has won it, 2 means player 2 has won it and 3 means the game ended in a draw.

For the same game state in figure 5.4.1, here is how the nodes and edges would look in graphML:

```
<node id="1">
  <data key="d1">0|-|X  X|0|0  X|-|-  </data>
  <data key="d2">0</data>
</node>

<edge source="1" target="2" />
```

The second line represents the state label and the third line represents the state type.

## 5.5 External Resources and Libraries

There were several outside code resources used to help the development process of the application. Note that there are other libraries used but these are the ones that help with the functionality directly.

### 5.5.1 Repositories

- **Parser:** The repository called pygdl [4] by Jason M Barnes adapted in this project. Pygdl was an old project that aimed to parse GDL games and store them in a Datalog database so that games can be playable in the future. It was implemented as a library in pip but it has been outdated as it was developed for Python 3.4, his library was adapted to work with 3.10 Python and altered to better suit the system. His work has been put under the parser directory in the implementation, also they are represented as Lexeme, ASTNode, Lexer, Parser, Database and State Machine entities in figure 3.3.
- **Games:** Part of the repository called ggp by Markus Partheymueller [16] was used in the implantation. The games tat were written were slightly changed and formatted to fit the parser that was used.

### 5.5.2 Libraries

- **networkX:** Essential for building the trees where a Directed Graph, a class of this library, is used to transform the graph into graphML, dot, and PNG.
- **PyQT5:** This library builds the graphical user interface of the application. It adds functionalities like interactive buttons and a display screen that enables users to input commands and see their outputs.
- **re:** Stands for regular expression, used in the lexer to match the words.

### 5.5.3 Software

- **PyCharm:** PyCharm is an integrated development environment that allows smooth python development. The product has been developed using this tool.
- **Cytoscape:** This is a network visualization tool and it is used to test the correctness of graphML files.
- **Drawio:** A flowchart generation app that is used in most figures in this paper.

## 5.6 Testing

The testing section will cover all the needed requirements that needed to be tested and evaluate the performance of how the tests went.

### 5.6.1 Testing Against User Requirements

1. The system enables the user to input from both files and enables direct writing on the UI to supply their own GDL file.
2. There are buttons that allow the user to go back and forth between moves that the user made, it also allows the user to input a depth of their choice for tree generation.
3. There is a checkbox that enables the user to choose to see or not to see the image version of the tree.
4. Once the user presses the GenerateTree button, the graphML file is available to be downloaded using a button.
5. Every button has been tested and they are all fully functional.

6. The graphML output is human-readable and it clearly states the edges from one state to another.

## 5.6.2 Testing Against System Requirements

### GDL Parsing

1. Parser class has a set of keywords list reserved which covers all the keywords in GDL
2. The parser is very well round created, here are the test cases it has been tried against:
  - (a) *Parser Correctness*: The parser raises an error when:
    - there is only 1 or no players defined
    - there is a game where three players have legal moves.
    - there is no terminal state
    - there is no goal state and the game reaches a where either of the players wins
    - there are no legal moves.
  - (b) *Game State*: For every game defined after every move the Game State changes accordingly.
  - (c) *Legal Moves*: For every game defined after every move the legal moves alter appropriately
  - (d) *Moving forward and backward* : After a mix of Move Tree and Generate Tr commands, the system reverts back to the correct order.
3. Although the parser is able to parse complex games, the system might sometimes slow down, or depending on the machine it is being executed and the command executed it may even crash sometimes. This is not about the correctness of the code but it is about the do with the performance.

### Tree Generator

1. With the help of the parser, the GDL file is parsed and stored as a state machine behind the curtains.
2. The system allows input for depth and if the GDL is edited the initial state can also be altered.
3. Transformation from the parser to TreeGenerator doesn't create any issues. All of the labels are colored and all the nodes and edges have been tested on the existing games.



### 5.6.3 Testing Against System Specifications

1. Discussed in section 6.5.
2. Discussed in section 6.1
3. System can go into the full depth of the trees of simple games like nim and tic-tac-toe.
4. The system is made so that allows a to have maximum interaction with the system.

### 5.6.4 Quality Testing

The software cannot only be tested on functional requirements on what it can or cannot do factors like performance and usability. This section will analyze the quality of this software through a few main topics:

- *Resource Management:* In terms of space, it occupies a very insignificant space in the RAM and uses a very small portion of the CPU while computing the hardest games so it uses its resources very efficiently.
- *Usability:* The GUI focuses on simplicity rather than complex looks, the app's silver lining is a strong engine. A basic Qt application with a few buttons is more than enough for this app to be fully functional thus it is designed to be usable.
- *Maintainability:* The software uses the latest technologies and good practices of the software make it easy to maintain.

## Chapter 6

# Legal, Social, Ethical and Professional Issues

This section is dedicated to discussing possible non-technical issues that may rise. They are categorized under four categories while being discussed: Legal, Social, Ethical, and Professional. Each is equally important and will be discussed in this chapter in that order respectively.

### 6.1 Legal Issues

In every project, there are a lot of legal issues may arise. One of the typical issues is using someone else's work. Usually, they are protected under either of the following: copyrights, patents, trademarks, and licenses. Additionally, privacy is a key component in legal matters as software is not meant to store any information obtained from the user.

In the context of this project, it is important to keep an eye on whether any of the pre-defined games are patented and if there is such an issue, permission should be requested from the patent holder. The parser of this project Parser directory and games (GDL files in games folder) has been adapted from GitHub's different repositories online, both of these have been disturbed and they have been made open to the public so it will not raise any legal issues in using it in non-commercial purposes. This means the application shouldn't be disturbed or published and it should only be used for personal use.

In terms of privacy, the software doesn't store any data from the user. These include the files that they input, the moves they make, and the images they generate. Therefore, it doesn't violate any laws.

## **6.2 Social Issues**

The social implications of the software are also noteworthy. Gaming in general is highly associated with addiction, this software also allows you to play the games while generating trees. This may raise some social issues however, every game ever created has a chance of causing addiction therefore this will be neglected.

Also, another issue might be accessed, the software is not designed in a way that inclusive is inclusive for every user. This is a weakness of the project an area that can be improved which will be discussed in future chapters.

## **6.3 Ethical Issues**

Ethical considerations include the platform doesn't promote or enabling unethical behavior. The system does not promote or contain any sort of discriminative or offensive content. It is designed in a way that minimizes addictive behavior.

## **6.4 Professional Issues**

As this is a software-centered project, is also very important to discuss professional issues. The code written follows certain principles like DRY and writing clean code. The software is not intended for distribution or commercial purposes.

## **6.5 Code of Conduct & Code of Good Practice issued by the British Computer Society**

### **6.5.1 Code of Conduct**

The Code of Conduct consists of core principles like integrity, respect, and taking responsibility. It enforces ethical practices which are a must in the tech industry. Every member or non-member should follow these guidelines to stay professional.

In the software implantation Code of Conduct was followed and everyone's work has been cited and given credit for. There is also an Originality Award available at the beginning confirming that all of the work belongs(unless said otherwise) to me.

### **6.5.2 Code of Good Practice**

The Code of Good Practice is a set of guidelines that ensure people in the Tech industry are developing products in maintainable and good-quality products. This applies to every area of the Tech industry that includes but not limited to software development, quality assurance, and risk management.

All of these guidelines were followed during the development and professionalism has always been considered. The code has been written in a way that is easily maintainable, expandable, and readable.

# Chapter 7

## Evaluation

This section will evaluate the end product in numerous aspects and at the end, it will make an overall evaluation of the entire process development.

### 7.1 Limitations

Every software ever created has flaws and small bugs that make them not perfect. This system also comes with its limitations on what it can do possible improvements have been discussed in chapter 8.2. Firstly, the app cannot process games that have computationally hard commands because of the design of the compiling process(this can be observed in games like tictactoe and pawn hopping where pawn hopping is a larger game; there is a significant difference between computations). The nature of the parser isn't meant to be fast or efficient; it is meant to be a parser for simple GDL games of small scales. Also, the parser isn't technically perfect as it still has some difficulties sorting out parenthesizes. As GDL has no official documentation it should follow the syntax of the games already in the game to make the games work. Moreover, the app has no accessibility features limiting the number of possible users. Lastly, it only offers 8 games to play; this may be expanded to more games for people who don't know the initial 8 games.

### 7.2 User Interface

The implementation has a very simple Graphical User Interface that is functional which was the aim. There wasn't any requirement that required a more complex UI than the one developed. Using a pre-existing library, QT5, was enough to develop everything needed, the simplicity of

the syntax also eased the development.

## 7.3 Imported Code

The code has benefited from outside resources in terms of parsing and GDL files for predefined games which became the bedrock of the entire back end of the system. As previously mentioned in the scope of the project, the project wasn't to write a parser for GDL or create GDL files from scratch, it was aimed at treeifying games. Therefore implementing these into the code was very sensible as it saved a lot of time during development.

Connecting two different libraries and repositories, translating the result from one, and turning them into a form that the other can understand, was a challenge as it required a deep understanding of compilers and the code that I have imported. At the end, everything was seamlessly integrated and the system was complete as a whole.

## 7.4 Engine

As this project was more back-end oriented, a big part of the project was the engine that creates the trees. As tested in section 5.6 there "rightness" of the engine is properly tested and there hasn't been an error. It does what it is intended to do and it can evaluate small games in a reasonable amount of time. Also, there were many parts tied together from the state machine to Tree Node to networkX which were well translated to one another. The intended graph format was graphML which was generated accurately during the process.

### 7.4.1 Visuals

One of the most important parts of the project was to create good visuals for the user when they generate a tree. The overall resulting tree has nice edges which clearly demonstrate the transition between states. Through the use of labels, the nodes were colored depending on who won the game. This image went above and beyond what was expected as no coloring and neat edges were necessary.

### 7.4.2 Additional Features

The project specification was limited and it achieved everything it set out for, alongside these requirements some extra features have been added. Making a random move, limiting the node

amount and the entire process of creating an image of the tree wasn't in the specifications but they are included to improve user experience.

## 7.5 Overall

All in all, the implementation was a success and it met all the requirements by passing an extensive set of tests. It actually exceeded expectations by adding additional features and having prettier visual aids. After discussing the non-technical issues as well, there were no significant issues found. One major issue is performance and the fact that it cannot process larger games but this can be easily overcome by implementing a new parser. There are possible areas of development and improvement left in the system which are discussed in future chapters.

## Chapter 8

# Conclusion and Future Work

This section will conclude the report by analyzing the outcomes of this product, how it can be used, and summarizing what has been done so far. It will also discuss the possible improvements to the product and how it can be studied in the future.

### 8.1 Conclusion

This project has demonstrated the power of conceptualization, or generalization, of games and how they can be utilized. In an attempt to visualize and generalize two-player combinatorial games, the end product turned up to be a tool that is extremely powerful and can generate a tree of such games.

Although there is a lot of research in this field, there isn't software that serves this purpose. There have been researches on how to play these games under the field of GGP [11] in order to put GGP into a structure there have been researches on GDL [13] and how to compile them [14]; researchers also found that representing GGP as directed graphs was a success [21]. Developers like Jason M Barnes have done an excellent job developing a Python library for GDL parsing [4] there hasn't been software that combines all of these concepts together and this project managed to do that.

In a way, this project is really similar to assembling a big jigsaw puzzle. Each piece of the system is a unique but distinct component but when they are carefully integrated and interlocked, they form a software far greater than the sum of their parts.

Using concepts applying concepts from GGP to combinatorial game theory allowed these games to be generalized in GDL. From generated GDL files the system was able to parse these



files and interpret the resulting tokens. There was already a library to generate trees as well so all there was to do was implement a way to pass all those tokens into the networkX [2] library to turn it into a directed graph.

Overall, this is a project that combines many concepts together into one big software. It acts as a bridge between all of the mentioned disciplines and finds a common ground between different Python libraries. The end product is a unique and niche implementation that could be useful for numerous fields.

## 8.2 Future Work

This project aimed to generate trees from simple games and the system was designed specifically for this purpose however, there is plenty of room for improvement and additional features and in-depth expansion opportunities:

1. **Performance optimization.** The code cannot work with large games like chess and checkers it becomes incredibly slow and therefore unplayable. If code is more optimized, designed for larger scales, or written in a different language it can run a greater amount of games. However, since there is no limit on how computationally exhausting the input can be, besides the computer specs are also important. So this is a very open-ended area of improvement.
2. **More Accessibility Features.** UI doesn't have any accessibility features so that limits the number of users that can use the software. A good addition to the code might be adding such features as text-to-speech features.
3. **Adding a GGP agent.** There is no AI involved in this project however there is a good foundation to build upon. The system has a functioning parser and an interpreter of the games. The only addition needed for a GGP agent is an algorithm(minimax, alpha-beta pruning) to find the best possible outcome in the quickest way possible.<sup>1</sup>
4. **User-against-AI feature.** If there were a GGP agent that could stimulate the game for a player, there could be a feature to play against AI instead of both players being controlled by the user. This feature would mean that a user can play against an AI in every combinatorial game imaginable.

---

<sup>1</sup>This is also mentioned briefly in the original implantation's README.md file

5. **Better UI/UX design.** UI/UX was never the primary focus of this project thus it only offers the bare minimum in this area so there is definitely room for improvement. The interface can be more interactive as the only way for users to attract with the software is by a few buttons. Some of these may include interacting with the graph directly instead of seeing a static image and possibly more multi-page design could have been more user-friendly.

# References

- [1] Ludii portal. Accessed: February 21, 2023.
- [2] Networkx graph library. Accessed: February 21, 2023.
- [3] Regular expression howto – python 3 documentation. <https://docs.python.org/3/howto/regex.html>. [Online; accessed 11-March-2023].
- [4] Jason M Barnes. pygdl: Gdl parser for python. <https://github.com/jazzyb/pygdl>, 2018. GitHub repository.
- [5] Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy. *Winning Ways for Your Mathematical Plays*. Academic Press, 1982.
- [6] Sebastian M. Cioabă and M. Ram Murty. *A First Course in Graph Theory and Combinatorics*. Chapman and Hall/CRC, 2009.
- [7] Cytoscape Consortium. Cytoscape. <https://cytoscape.org>, Accessed: 2023.
- [8] Cambridge Dictionary. Lexeme. <https://dictionary.cambridge.org/dictionary/english/lexeme>. Accessed on March 10, 2023.
- [9] The Getty Museum Education. The royal game of ur: A diy board game for the ancient classroom. [https://www.getty.edu/education/college/ancient\\_rome\\_at\\_home/pdf/ur\\_game.pdf](https://www.getty.edu/education/college/ancient_rome_at_home/pdf/ur_game.pdf), 2012. Accessed on 03 March 2023.
- [10] GeeksforGeeks. Level order tree traversal.
- [11] Michael Genesereth and Michael Thielscher. *General Game Playing*. AAAI Press, 2005.
- [12] JetBrains. Pycharm system requirements. [https://www.jetbrains.com/help/pycharm/prerequisites.html#min\\_requirements](https://www.jetbrains.com/help/pycharm/prerequisites.html#min_requirements), accessed 2023.

- [13] Jakub Kowalski and Andrzej Kisielewicz. *General Game Description Languages*. PhD thesis, University of Wrocław, 2016.
- [14] Jakub Kowalski and Michał Szykuła. Game description language compiler construction. In *Australasian Joint Conference on Artificial Intelligence*, pages 234–245. Springer, 2013.
- [15] Nick C. Love, Tim L. Hinrichs, and Michael R. Genesereth. General game playing: Game description language specification. Technical Report LG-2006-01, Stanford Logic Group, 2006.
- [16] Markus Partheymueller. GGP: games for the generalized game playing competition. <https://github.com/parthy/ggp/tree/master/games>, accessed 2023.
- [17] Eric Piette, Dennis J. Soemers, Mark Stephenson, Francesca Sironi, Mark H. Winands, and Cameron Browne. Ludii—the ludemic general game system. *arXiv preprint arXiv:1905.05013*, 2019.
- [18] Abdallah Saffidine and Tristan Cazenave. A forward chaining based game description language compiler. In *Proceedings of the IJCAI-11 Workshop on General Game Playing (GIGA’11)*, pages 69–75, 2011.
- [19] SourceForge. Datalog. <https://datalog.sourceforge.net/datalog.html>, 2023. Accessed: March 11, 2023.
- [20] Christian Urban. Compiler and formal languages 22/23 slides 1. Slides, 2017.
- [21] Guenter Wallner. Visual representation of game mechanics for analysis and comparison. In *Proceedings of the International Conference on the Foundations of Digital Games*, pages 234–241. Society for the Advancement of the Science of Digital Games, 2013.

# Appendix A

## Extra Information

### A.1 List of Games

This section explains the 8 games used in the implementation. Each game is describe with four subsections: Game Description, Initial State, Rules, Terminal State.

#### Tic Tac Toe

**Game Description:** A game is played on a 3x3 grid where players place Xs and Os. It starts with a blank grid. **Initial State:** All spaces on the grid are empty.

**Rules:** No player can play on top of an already-played space.

**Terminal State:** The player that makes a straight line either horizontally vertically or diagonally with their respective signs wins the game if not it ends in a draw.

#### Number Tic Tac Toe

**Game Description:** A variation of Tic-Tac-Toe played on a 3x3 grid where players place numbers instead of Xs and Os.

**Initial State:** All spaces on the grid are empty.

**Rules:** No player can on top of an already-played space.

**Terminal State:** The player that makes 15 in total either horizontally, vertically, or diagonally wins the game if not it ends in a draw.

## Nim

**Game Description:** A game where players remove objects from 4 piles.

**Initial State:** Piles have 1-5-4-2 objects respectively.

**Rules:** A player can remove any number of objects from a pile at once.

**Terminal State:** The game ends when there are no objects to remove, and the player that takes the last object wins the game.

## Sum 15

**Game Description:** There are numbers, 1 to 9, available to pick and each player takes a number in their turn.

**Initial State:** Both players have no numbers and all of the numbers are available to be picked.

**Rules:** Each player can only take one number in their turn. Each number can only be used once.

**Terminal State:** The player that reaches exactly three numbers that add up to 15 wins otherwise it is a draw.

## Connect Four

**Game Description:** Played on a 6x7 grid where players drop down different colored discs down at the columns.

**Rules:** Players can drop a disc down a column when it is not full and can only drop one disc at their turn.

**Initial State:** All of the columns are empty.

**Terminal State:** The game ends when four discs are connected horizontally, vertically, or diagonally. If no player managed to connect four the game ends in a draw.

## MiniChess

**Game Description:** Simplified version of chess played in a 4x4 board.

**Rules:** The rules are the same as the normal rules of chess.

**Initial State:** The white side has a king and a rook, while the black side only has a rook.

**Terminal State:** The end conditions are also the same as the normal conditions of chess.

## Sheep and Wolf

**Game Description:** The game board is 8x8. One side consists of a wolf while the other side consists of sheep. The sheep try to cross the board from one side to the other, while the wolf tries to catch the sheep. During the game, both sides make strategic moves and apply tactics to either help the sheep cross safely or to catch all the sheep as the wolf.

**Rules:** The wolf and sheep can only move one space at a time diagonally. The wolf always moves first, and then the sheep take turns moving. The wolf can capture a sheep by moving to the same space that the sheep occupies. The sheep cannot capture the wolf, but they can surround and trap it. If a sheep reaches the other side of the board, it is removed from play and considered safe.

**Terminal State:** If the wolf captures all the sheep, the wolf wins the game. If the sheep manage to get one sheep to the other side of the board, they win the game. If the game reaches a stalemate, with neither player able to make a move, the game is declared a draw.

## Pawn Whopping

**Game Description:** The Pawn Whopping game is a version of chess that only consists of pawns.

**Rules:** The movement and capturing conditions for pawns are the same as in normal chess.

**Initial State:** The initial positions of the pawns are the same as in normal chess. The positions where other pieces should be are empty.

**Terminal State:** The game is won by either side when one of their pawns reaches the other side of the board. Alternatively, if one player runs out of pawns, the other player wins. If at any point in the game, there is no possible move left but neither player has achieved a winning condition, the game is a draw.

## A.2 List of Reserved Keyword

Below you find the table that summarizes the keywords used in a GDL game.

Keyword	Arity	Description
base	1	Defines the base state of the game
distinct	2	Specifies that the arguments are distinct
does	2	Specifies a move taken by a player
goal	2	Specifies the goal value for a player
init	1	Specifies the initial state of the game
input	2	Specifies input for a player
legal	2	Specifies a legal move for a player
next	1	Specifies the next state of the game
not	1	Negates the following proposition
or	2-9	Specifies disjunction of 2-9 propositions
role	1	Defines a player
terminal	0	Specifies the end of the game
true	1	Specifies a true proposition

### A.3 Snapshot of the Database

This section gives an example of how a database object looks for a tic-tac-toe state to demonstrate the structure for the figure A.1

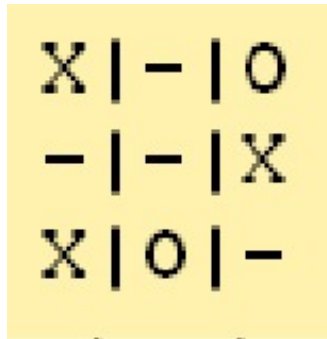


Figure A.1: The visual representation of the game state

As explained in the 5.1.5, the database has been implemented with 4 lists: facts, derived\_facts, rules, and requirements. Below you may find the contents of the list for figure A.1.



## FACTS

```
{('role', 1): [[xplayer], [oplayer]],
 ('init', 1): [[(cell 1 1 b)], [(cell 1 2 b)],
               [(cell 1 3 b)], [(cell 2 1 b)],
               [(cell 2 2 b)], [(cell 2 3 b)],
               [(cell 3 1 b)], [(cell 3 2 b)],
               [(cell 3 3 b)], [(control xplayer)]]],
 ('true', 1): [[(cell 2 3 x)], [(cell 3 2 o)],
               [(cell 3 1 x)], [(cell 1 3 o)],
               [(cell 1 1 x)], [(cell 1 2 b)],
               [(cell 3 3 b)], [(cell 2 1 b)],
               [(cell 2 2 b)], [(control oplayer)]]}]
```

## DERIVED\_FACTS

```
{('legal', 2): [[oplayer, (mark 1 2)],
                [oplayer, (mark 3 3)],
                [oplayer, (mark 2 1)],
                [oplayer, (mark 2 2)],
                [xplayer, noop]]}
```

## RULES

```
{('next', 1): [[(cell ?m ?n x)],
                [(does xplayer (mark ?m ?n)),
                 (true (cell ?m ?n b))]],
                [(cell ?m ?n o)],
                [(does oplayer (mark ?m ?n)),
                 (true (cell ?m ?n b))]],
                [(cell ?m ?n ?w)],
                [(true (cell ?m ?n ?w)), (distinct ?w b)]],
                [(cell ?m ?n b)],
                [(does ?w (mark ?j ?k)),
                 (true (cell ?m ?n b)),
                 (or (distinct ?m ?j)
                     (distinct ?n ?k))]],
                [(control xplayer)],
                [(true (control oplayer))]],
                [(control oplayer)],
                [(true (control xplayer))]]],
('row', 2): [[(?m, ?x),
               [(true (cell ?m 1 ?x)),
                (true (cell ?m 2 ?x)),
                (true (cell ?m 3 ?x))]]],
('column', 2): [[(?n, ?x),
                  [(true (cell 1 ?n ?x)),
                   (true (cell 2 ?n ?x)),
                   (true (cell 3 ?n ?x))]]],
('diagonal', 1): [[(?x),
                    [(true (cell 1 1 ?x)),
                     (true (cell 2 2 ?x)),
                     (true (cell 3 3 ?x))]],
                    (?x),
                    [(true (cell 1 3 ?x)),
                     (true (cell 2 2 ?x)),
                     (true (cell 3 1 ?x))]]],
```

## RULES

```

('line', 1): [[(?x), [(row ?m ?x)]],
              [(?x), [(column ?m ?x)]],
              [(?x), [(diagonal ?x)]]],
('open', 0): [[[]], [(true (cell ?m ?n b))]]],
('legal', 2): [[(?w, (mark ?x ?y)),
               [(true (cell ?x ?y b)),
                (true (control ?w))]],
               ([xplayer, noop],
                [(true (control oplayer))]),
               ([oplayer, noop],
                [(true (control xplayer))]]),
               [(true (control oplayer))]]],
('goal', 2): [[([xplayer, 100], [(line x)]),
               ([xplayer, 50],
                [(not (line x)), (not (line o)),
                 (not open)]),
               ([xplayer, 0], [(line o)]),
               ([oplayer, 100], [(line o)]),
               ([oplayer, 50],
                [(not (line x)), (not (line o)),
                 (not open)]),
               ([oplayer, 0], [(line x)])],
('terminal', 0): [[[]],
                  [(line x)],
                  ([],
                   [(line o)]),
                  ([], [(not open)])]]}

```

## REQUIREMENTS

```
{('does', 2): {('next', 1)},
 ('true', 1): {('next', 1), ('open', 0),
               ('legal', 2), ('diagonal', 1),
               ('column', 2), ('row', 2)},
 ('b', 0): {('next', 1)},
 ('row', 2): {('line', 1)},
 ('column', 2): {('line', 1)},
 ('diagonal', 1): {('line', 1)},
 ('line', 1): {('goal', 2), ('terminal', 0)},
 ('open', 0): {('goal', 2), ('terminal', 0)}}
```

# Appendix B

## User Guide

This chapter will cover how to use the software from installing, compiling, and using the application

### B.1 Setting up the environment

This section assumes that the computer doesn't have any tools that is used in this application. To get started here is a brief summary of what needs to be done in order to run this application. First of all, a free-to-download app called PyCharm needs to be downloaded, this is an IDE (integrated development environment) that will compile the app. Then python 3.10 and pip package manager installations are required. Once these are done the app will be able to be run through the use of a single run button. Before getting into installations system requirements are needed to specify. According to JetBrains official documentation [12] suggests these specs for a computer to run PyCharm:

- 4 vCPUs, either x86\_64 or arm64 architecture. Also, higher clock frequency is preferred to higher core count.
- 8 GB RAM
- At least 500MB of free disk space even if the IDE is already installed.

It is also important to note that in addition to PyCharm, some other installations are going to be run and some of the libraries used are highly dependent on other libraries therefore overall need at least 2 gigabytes of space for the entire project. This section will start off with the installation of PyCharm.

## PyCharm

In order to install PyCharm please refer to this website:

<https://www.jetbrains.com/pycharm/>

The Community Edition is free to download and use. The exact version of the ide that was used in the project was 2022.3.3, if there are any problems in the older versions please use this version of the software.

## Python and Pip

Before installation, it is common practice to check whether you already have python installed on your computer as most modern-day computers have python already installed. To check whether python is installed on your computer, run the command through your terminal/shell:

This command works in Unix-based operating systems and Windows. If you don't see a figure like this you will need to go ahead with the installation:

```
berkes-MacBook-Pro-2:~ berkemuftuoglu$ python3 --version  
Python 3.10.0
```

Please refer to the Python 3.10.0 installation page and choose a method of installation as you wish:

<https://www.python.org/downloads/release/python-3100/>

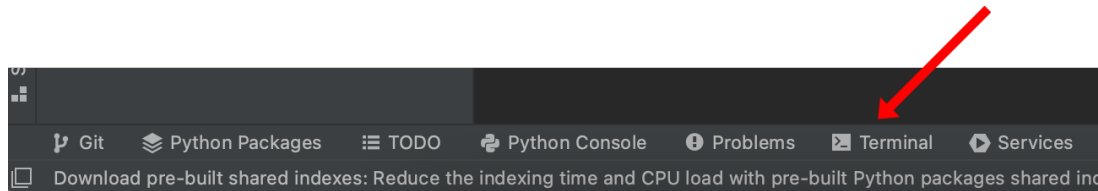
The downloaded file will automatically install the necessary binaries into your computer. Moving on to installing pip, you can run the same command, changing python to pip, to see whether pip is already installed. Python comes with pip already installed with it, if it doesn't work refer to this website for other forms of installation:

<https://pip.pypa.io/en/stable/installation/>

## B.2 Compiling

After these installations, it is time to open the app and start compiling the application. Click on open project and select the root directory of your application. This should open a gray screen and the directory hierarchy on the left. Wait for a minute for PyCharm to index the

files and add interest to the project structure. At first, it will generate lots of errors because none of the dependencies are libraries are installed yet. To download them all at once, go to the terminal button at the bottom of your screen:



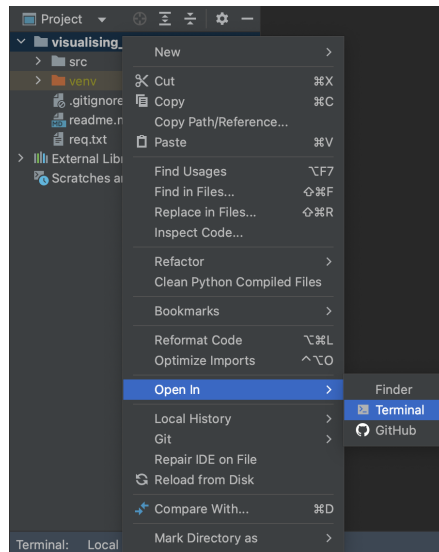
Once you open the terminal, this screen should pop from the bottom:

```
Terminal: Local x + v
The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
(venv) berkes-MacBook-Pro-2:visualising_trees berkemuftuoglu$
```

The *venv* at the beginning of the terminal prompt means that the terminal is in a virtual environment and the *visulasing-trees* is the name of the directory that you are in. In this case, *visulasing\_trees* is the root directory.

As default, PyCharm CE will create an virtual environment for every project. A virtual environment is a self-contained container that allows developers to install Python packages locally to their projects rather than to their machines. Therefore libraries that will be installed later will be localized to this project.

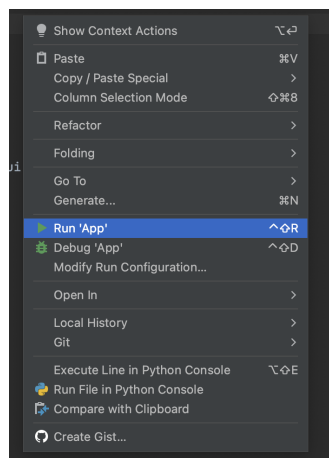
By default, PyCharm should have opened the terminal at the root of the project, however, if it didn't click on the root directory and press open in the terminal:



Once you are in this directory run this command in the terminal to install the dependencies:

```
pip install -r req.txt
```

This is a common technique in software development where through a single command pip can install the libraries written inside req.txt (short of requirements). "-r" stands for recursive as in it has iterates through the file and installs all the dependencies. Some output like "installed" should be printed on the terminal. Wait for a few minutes to let the machine finish the installation. Once it is done, the app is ready to run. Go to the src directory and there is a python script named "App.py" Right-click on the file and press "Run 'App' ". This should open a screen and now the app is ready for use.





# Appendix C

## Source Code

### Originality award

“I verify that I am the sole author of the programs contained in this folder, except where explicitly stated to the contrary”.

Berke Muftuoglu

April 6, 2023

This selection will cover the source code of the application. Below you may find the directory tree of the application and the code of each file below. Above the code there will be an indicator for which file the code belongs to.

visualising\_trees

