Game Tree Drawing

Candidate Number: *68123*

Supervisor: *Prof Bernhard von Stengel*

A Dissertation submitted to the Department of Mathematics of the London School of Economics and Political Science for the degree of Master of Science

Date: 01/09/2017

Abstract

The goal of this dissertation is to describe the process of creating a web-based software tool for the visual representation of game trees. A Flash version of the application currently exists and we aim to modernise it by utilising the latest HTML5 technologies and by further improving the usability and user interface. From a programming perspective, a lot of consideration will be given to the software architecture and structure in order to preserve maintainability, reusability, extensibility and thus, the overall quality of the code. These features of the project will be demonstrated with class diagrams and code snippets to better illustrate the advantages of the chosen architecture. We will also examine some specific challenges in the development of the software and their solutions, for example, input handling, tree serialisation, information sets representation, and strategic form implementation.

Contents

List of Figures and Code Snippets

Chapter 1

Introduction

The purpose of this dissertation is to document our implementation of a modernised version of *Game Theory Explorer* (GTE) – a software tool, which creates and analyses non-cooperative games in extensive and strategic form. The previous stable version of GTE has been written in Flash, which is a software for creating vector-based animations that can be incorporated in a webpage. As Flash support amongst browsers and operating systems continues to decline, an HTML5 version of the software needs to be developed. The goal of this new software is to conform to the latest web trends and technologies in order to keep it updated. The original GTE tool will eventually die out when the Adobe Flash support ends and at this point the HTML5 version created pursuant to this dissertation is intended to survive and supersede the Flash GTE.

In terms of implementation, we utilise a game engine called *Phaser* [2] for the visual aspects of the software. For the most part, the new GTE is written in *TypeScript* [9] which is a superset of JavaScript. Certain specifics relating to the usability and benefits of both Phaser and TypeScript are explored in detail in Chapter 3.

The structure of this dissertation is summarised as follows. Chapter 2 introduces some of the basic concepts in game theory, namely, game trees and games in strategic form, while Chapter 3 examines the HTML5 version of GTE that we have developed. Chapter 3 commences with a brief overview of the history of the software, then presents the core features of the new HTML5 version (including a new approach to creating game trees by operating on multiple nodes at once) with a detailed example. We continue by introducing the tools adopted in order to develop the new GTE and finally, we provide detailed instructions on running the software on a desktop computer. The main body of the dissertation deals with the architectural (Chapter 4) and algorithmic (Chapter 5) challenges we encountered during the development process. Chapter 4 analyses the chosen standard architecture called *Model-View-Controller* (MVC). Structuring the project in this manner allows for a clear separation of the logic and algorithms from the way the trees are drawn, thus enhancing the code readability and extensibility. At the end of the chapter, we illustrate the advantages of the chosen tools and architecture with two examples, namely, the implementation of input handling and undo/redo functionality. Chapter 5 presents certain algorithmic problems arising from the game tree data structure and their solutions in the HTML5 version.

To conclude, we present several options for further development of the new GTE. We also give a number of suggestions as to the way these features could be implemented within the project in the future.

Chapter 2

Game Trees and Games in Strategic Form

This chapter introduces the main concepts of non-cooperative game theory, namely, game trees with perfect and imperfect information and games in strategic form. The following is taken almost verbatim from the LSE course Game Theory I, given in [13], where all figures are replaced with the corresponding ones from the current HTML5 version of GTE.

Figure 2.1 shows an example of a game tree. In this dissertation we will use the convention that game trees are drawn downwards, with the *root* at the top. GTE currently draws trees this way and in the future it will have a feature which allows to draw trees from the bottom upwards, from left to right, or right to left. The *nodes* of the tree denote game states. Nodes are connected by lines, called *edges*. An edge from a node u to a successor node v (where v is drawn below u) indicates a possible *move* in the game. This may be a move of a "personal" player, for example move X in Figure 2.1 of player I. Then u is called a *decision node*. Alternatively, u can be a *chance node*. A chance node is drawn as a *square*, for example, the node u that follows move b of player II in Figure 2.1. The node v is then determined by a random choice according to the probability associated with the edge leading from u to v. In the game tree below, these probabilities are $\frac{1}{3}$ for the left move and $\frac{2}{3}$ for the right move.

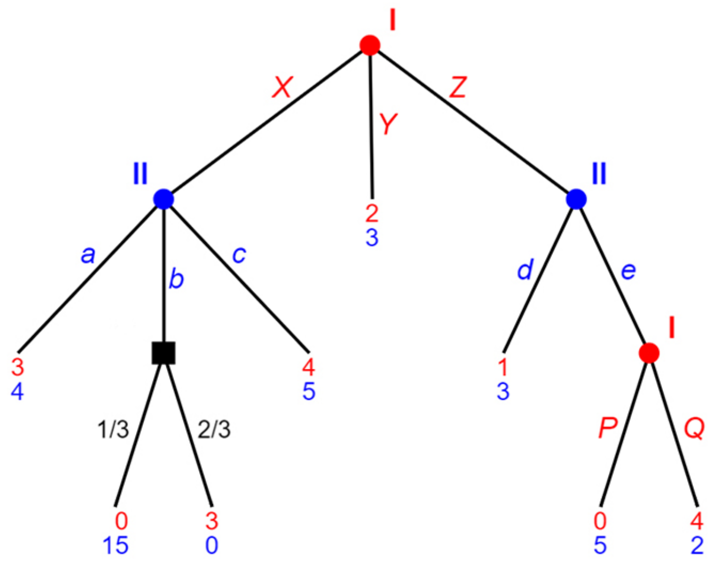


Figure 2.1: An example of a game tree with perfect information.

Nodes without successors in the tree are called terminal nodes or *leaves*. At such a node, every player gets a *payoff*, which is a real number. In GTE, leaves are not explicitly drawn, but the payoffs are given instead, with the top and bottom payoffs corresponding to players I and II, respectively. The game tree, with its decision nodes, moves, chance probabilities, and payoffs, is known to the players and defines the game completely. The game is played by starting at the root. At a decision node, the respective player chooses a move which determines the next node. At a chance node, the move is made randomly according to the given probabilities. Play ends when a leaf is reached, where all players receive their payoffs.

Game trees, as just described, are also called *extensive games* with *perfect information*. Perfect information means that a player always knows the game state and therefore the complete history of the game up to then. Game trees can be enhanced with additional structure that represents "imperfect information". In a game tree with *imperfect information* the non-terminal nodes are partitioned into *information sets*, i.e. every node belongs to exactly one information set, and no information set is empty. Information sets are typically shown as ovals around the nodes that they contain. Another convention is to draw the information sets by joining the nodes in the set with a single dashed line (which may be included in a future version of the software).

The interpretation of an information set is that in the course of a play of the game, a player is only told that he is at some node in an information set, but not at which particular node. Consequently, the information sets have to fulfil certain conditions: in an information set, all decision nodes belong to the same player; all decision nodes have the same number of outgoing edges; and the set of moves is the same for each node in the information set. Figure 2.2 is an example of a game with imperfect information for player II. The information set for player II has two moves, *a* and *b*. When the player has reached an information set, he makes a choice by picking a move, which is by definition the same move, no matter where he is in the information set.

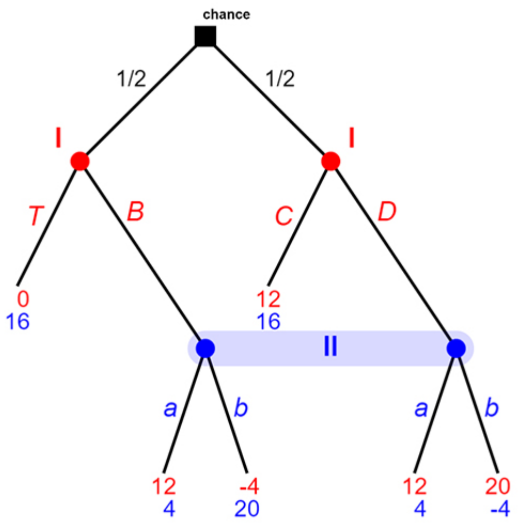


Figure 2.2: An example of a game tree with imperfect information for player II.

These conditions are trivially fulfilled if every information set contains only one node (a singleton). If all information sets are singletons, then the game has perfect information and the information sets will be omitted. This results in a game tree with perfect information. Only for information sets containing two or more nodes do these conditions matter.

In addition to the above conditions, additional constraints are imposed on the structure of the information sets. The first of these is that no two nodes in an information set share a path, as in the example in Figure 2.3. The single-player game in that figure has two decision nodes of player I, which belong to a single information set that has the two moves L and R. The only other non-terminal node is a chance move. Every condition about extensive games is met. However, the two decision nodes of player I share a path, because the first node is the root, from which there is a path to the second node. Information sets with this feature have a very problematic interpretation, because we require that a *move*, for example L, at such an information set is the same no matter where the player is . However, in Figure 2.3, if player I makes move L, he goes left from the root, but afterwards if the chance move goes to the right, player I has to move again. It is not clear whether he should make move L again. This is a problem, since by the interpretation of information sets, player I must have forgotten if he made a move at all when called to choose between L and R, and it is not even clear whether he is allowed to "choose again" or not. An information set could represent that a player forgets what he knew or did earlier. In that case, the player is said to have *imperfect recall*. However, players with imperfect recall, do not conform to the idea of "rational" players. Moreover, their forgetfulness is not adequately represented by information sets. We will therefore assume that all players have perfect recall.

Definition 1. Player i, in an extensive game, has *perfect recall* if for every information set h of player i, all nodes in h are preceded by the same sequence of moves of player i. The extensive game is said to have perfect recall if all its players have perfect recall.

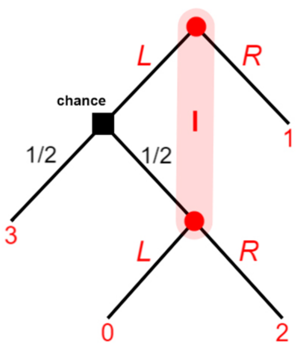


Figure 2.3: An example of a game with an information set where two nodes share a path.

In the rest of the chapter we briefly introduce games in strategic form. A more detailed discussion of this concept will be given in Section 5.4, where we will consider algorithms to transform a game tree to its strategic form.

Definition 2. In a game tree, a *strategy* of a player specifies a move for every decision node of that player. A *strategy profile* is a tuple of strategies, with one strategy for each player of the game.

If the game has only two players, a strategy profile is therefore a pair of strategies, with one strategy for player I and one strategy for player II.

Assume that a game tree is given. Consider a strategy profile, and assume that players move according to their strategies, If there are no chance moves, their play leads to a unique leaf. If there are chance moves, a strategy profile may lead to a probability distribution on the leaves of the game tree, with resulting *expected* payoffs. In general, any strategy profile defines an expected payoff to each player.

Definition 3. The *strategic form* of a game is defined by specifying for each player: the set of strategies, and the payoff to each player for each strategy profile.

For two players, the strategic form is represented by a table. The rows of the table represent the strategies of player I, and the columns represent the strategies of player II. A strategy profile is a strategy pair, that is, a row and a column, with a corresponding cell of the table that contains two payoffs, one for player I and the other for player II.

Figure 2.4 shows an extensive game in (a), and its strategic form in (b). In the strategic form, T and B are the strategies of player I, given by the top and bottom row of the table, and l and r are the strategies of player II, corresponding to the left and right column of the table. The strategic form for the game in Figure 2.1 is show in Figure 2.5.

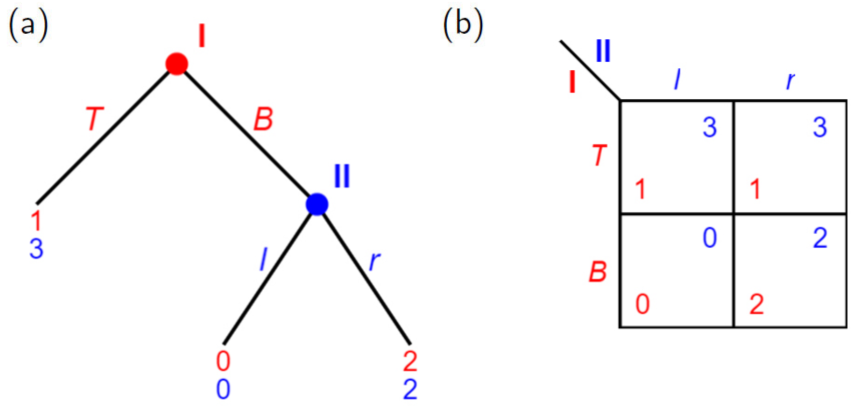


Figure 2.4: (a) An extensive game. (b) Strategic form of the extensive game in (a).

The strategic form can be simplified. In Figure 2.5 the two rows of the strategies XP and XQ have identical payoffs for both players, as do the rows YP and YQ. This is not surprising, because after move X or Y of player I at the root of the game tree, the decision node where player I can decide between P or Q cannot be reached, because that node is preceded by move Z, which is excluded by choosing X or Y. Because player I makes that move himself, it makes sense to replace both strategies XP and XQ by a less specific "plan of action" $X*$ that prescribes only move X. The star stands for an unspecified move at the respective unreachable decision node, to identify the node with its unspecified move in case the game tree has many decision nodes.

Leaving moves at unreachable nodes unspecified in this manner defines a *reduced strategy* according the following definition. Because the resulting expected payoffs remain

I \ II	a d	a e	b d	b e	c d	c e
X P	4 / 3	4 / 3	5 / 2	5 / 2	5 / 4	5 / 4
X Q	4 / 3	4 / 3	5 / 2	5 / 2	5 / 4	5 / 4
Y P	3 / 2	3 / 2	3 / 2	3 / 2	3 / 2	3 / 2
Y Q	3 / 2	3 / 2	3 / 2	3 / 2	3 / 2	3 / 2
Z P	3 / 1	5 / 0	3 / 1	5 / 0	3 / 1	5 / 0
Z Q	3 / 1	2 / 4	3 / 1	2 / 4	3 / 1	2 / 4

Figure 2.5: Strategic form of the extensive game in Figure 2.1.

uniquely defined, tabulating these reduced strategies and the payoff for the resulting reduced strategy profiles gives the *reduced strategic form* of the game. The reduced strategic form of the game tree on Figure 2.1 is shown in Figure 2.6.

I \ II	*a d*	*a e*	*b d*	*b e*	*c d*	*c e*
X *	4 / 3	4 / 3	5 / 2	5 / 2	5 / 4	5 / 4
Y *	3 / 2	3 / 2	3 / 2	3 / 2	3 / 2	3 / 2
Z P	3 / 1	5 / 0	3 / 1	5 / 0	3 / 1	5 / 0
Z Q	3 / 1	2 / 4	3 / 1	2 / 4	3 / 1	2 / 4

Figure 2.6: Reduced strategic form of the extensive game in Figure 2.1.

Chapter 3

Game Theory Explorer

3.1 History

In this section we briefly discuss the history and the idea behind the Game Theory Explorer software, as presented in [12]. GTE allows to create games in extensive or strategic form, and to compute their Nash equilibria. It is intended for the applied game theorist, but can also be used for theoretical research or as an educational tool in game theory. The main purpose of the software is to allow the user to draw game trees, by incorporating a reliable and intuitive *graphical user interface* (GUI), where a game tree can easily be stored, retrieved, and changed.

A suite for game-theoretic analysis that currently exists is called *Gambit*. It has been developed for over 25 years and incorporates a library of solution algorithms, formats for storing games, ways to program the creation of games, and a GUI for creating game trees. The main difference between GTE and Gambit is the way of accessing the software, more specifically, Gambit needs to be downloaded and installed, which may require some patience and technical experience, while GTE can be run directly from a web browser.

Running the software in a browser hos some disadvantages. From a visual perspective, web applications still have limited access to the graphics card of a computer. Some advancements have been made in this direction which include a JavaScript *API* (Application Program Interface) for rendering 2D and 3D graphics called WebGL. The API, however, is not currently supported by many devices and often requires specific hardware. In GTE we do not use WebGL as a *rendering engine* (the software that creates the graphic environment). This in turn means that the software relies on the computing power of the processor. From a computational perspective, this disadvantage of accessing the software from a browser is further worsened by the fact that web applications can only use a single core of the CPU. These limitations are problematic because equilibrium computations for larger games are computationally intensive. For this reason all computations of Nash equilibria take place on a server, which presents its own challenges. One of the ideas for future development is to integrate GTE with Gambit's game solvers.

The previous stable version of GTE can be accessed here: `http://gte.csc.liv.ac.uk/gte/builder/` [1]. Flash, which has been the most popular way of creating interactive web applications, games, video players, and websites for over twenty years, is used as the rendering engine. Due to performance and security issues, the support for Flash-based applications is quickly declining in favour of modern HTML5 technologies (HTML5 is a standard for representing webpages and graphics). Currently, Flash is not running on Apple devices and can be accessed only via a plugin on specific browsers. Very recently [3], Adobe announced that it will stop distribution and updates for the plugin, at which point all Flash-based applications will become inaccessible.

A screenshot of the Flash GTE is given in Figure 3.1. In the rest of the chapter we introduce the functionality of the new HTML5-based GTE, some improvements of the user interface, and the technologies we have incorporated. We also provide installation instructions for developers who wish to contribute to the project.

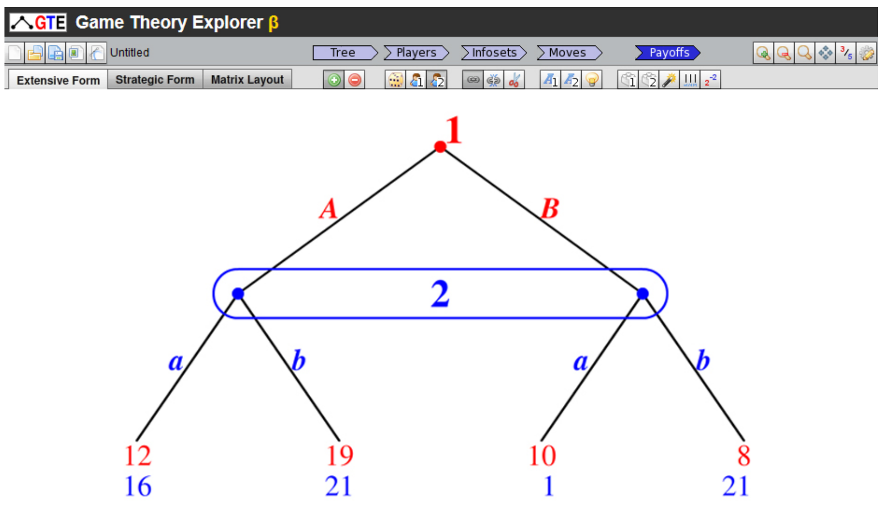


Figure 3.1: Flash version of GTE.

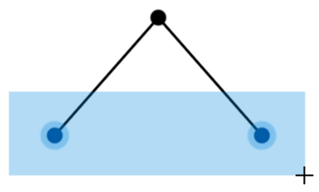
3.2 Features

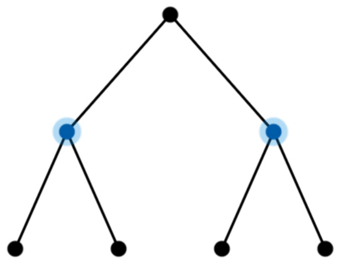
The new version of GTE supports almost all of the basic features of the Flash version such as adding nodes, players, information sets, editing labels, strategic forms, saving, and loading trees from a file, saving trees as images, undo/redo actions [1], zero-sum games, and random payoffs. One of the main objectives of the current version is to improve the way game trees are created. The step-by-step approach in the Flash version has been redesigned and made more automated for the user. One key feature that facilitates this is the ability to select and operate on multiple nodes. Similarly to the way Windows works with icons, by clicking and dragging the mouse, one can select a set of nodes and operate on them in a single step. Any such operation can be achieved in two ways: either with the keyboard, or with a menu which appears when clicking on or hovering over a set of nodes. For desktop users, we recommend using the keyboard shortcuts, because they allow a user to create game trees in seconds. The pop-up menu is more appropriate for mobile devices, when the software is ported to Android, iOS, and Windows Phone.

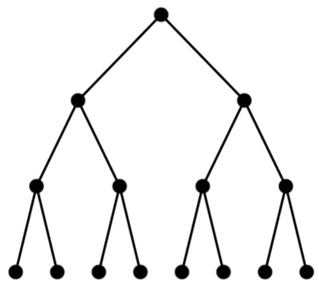
For a step-by-step guide for creating a simple game tree, please see Figure 3.2.

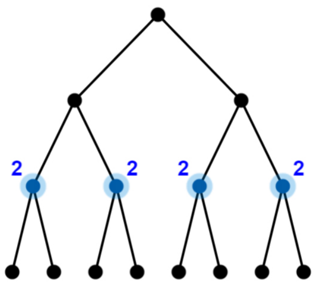
On image (a) one can select the two nodes by clicking and dragging the mouse. Pressing "*n*" on the keyboard creates children for both of them. When the nodes are further apart

[1]Note that undo and redo actions did not work in the Flash version and had to be disabled

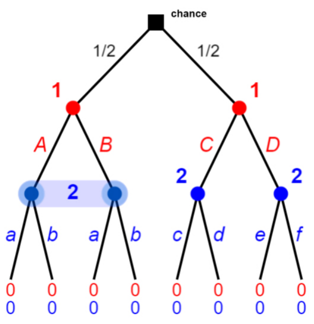

(a) Select the nodes with the cursor.


(b) Press *n* to add children to the selected nodes.

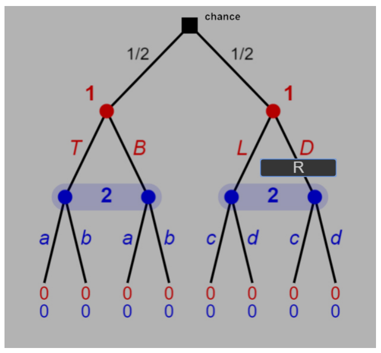

(c) Proceed by creating the tree in a similar way.



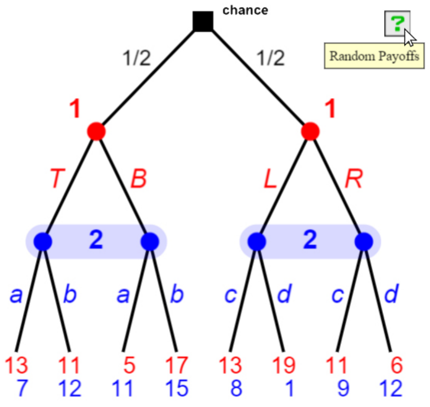
(d) Assign players by pressing *1,2* or *0* (for chance).



(e) Add information sets by pressing *i* on a selection of nodes.



(f) Assign move labels by clicking on them.



(g) Assign random payoffs from the top menu.

Figure 3.2: Creating a simple game tree with GTE.

and cannot be selected with a rectangle all at once, the user can hold the *"shift"* button on the keyboard and add nodes to the selection in parts. On image (d) we can press the number buttons on the keyboard to assign a player to a group of nodes. Currently, GTE supports up to four players. If the user wants to assign a chance player to a node, the corresponding keys are *"0"* or *"c"*. Adding information sets on image (e) is done in a similar manner: one selects a set of nodes and presses the *"i"* button on the keyboard. If an information set cannot be created, according to the rules mentioned in Chapter 2, then an error message appears. One can also dissolve information sets by pressing the *"u"* button on the keyboard.

Editing player, payoff, move, and probability labels is done by clicking on the respective label. After inserting the new value, pressing the *"Enter"* button on the keyboard will assign it to the item and will proceed to the next label in breadth-first order for move and player labels, and depth-first order for payoffs. We can also skip labels or go back to a previous label by pressing *"tab"* or *"shift+tab"* buttons, respectively. There is an algorithm for assigning chance moves probabilities to keep the sum of all probabilities (coming from a chance node) equal to one. This algorithm will be discussed in Section 5.3.

On image (g), we assign random payoffs by clicking on a button in the top menu. Currently, the top menu supports the following features: starting a new tree, saving and loading trees from a file, saving the current tree as an image, undo and redo, zero-sum toggle, random payoffs, and decimal/fractional representation of chance probabilities. Simple settings for adjusting the tree width and height have also been implemented and in the future we would like to extend the settings menu to contain more adjustable features, similar to the ones in the Flash version of GTE.

In the example above, we have created the tree by first adding the nodes first, assigning players second, then creating information sets, and finally, editing the labels. The user does not have to follow this order – he/she has complete freedom over the creation and editing of the trees. This includes removing information sets, then adding more nodes, following which, payoffs can be changed, and so on. If a mistake occurs, one can undo (*"ctrl+z"*) or redo(*"ctrl+shift+z"*). We should note that move and payoff labels appear only if every node is either assigned a player or is a chance node. This feature prevents the possibility of bugs appearing when editing the tree.

An interesting feature of the software is the user's ability to cut information sets By selecting an information set and pressing the keyboard shortcut *"c"* (or clicking on the respective button from the menu) a small scissors icon appears over the information set which follows the cursor. If the user clicks again, the information set will be split into two, depending on the position of the scissors icon (please see Figure 3.3 below).

The table in Figure 3.4 lists all keyboard shortcuts that are currently incorporated in the software.

One challenge when drawing game trees is to correctly place the nodes when there are information sets, so that no information set and move intersect. This is not always possible. In the Flash version of GTE, there is an algorithm which automatically repositions the nodes and in most cases it works well. For more complex trees, the algorithm does not always produce satisfactory results as shown in Figure 3.5 (the tree is taken from [14]). Improving the algorithm may require many more computations, which would slow down the software and may not work as expected in all cases. In the HTML5 version we have

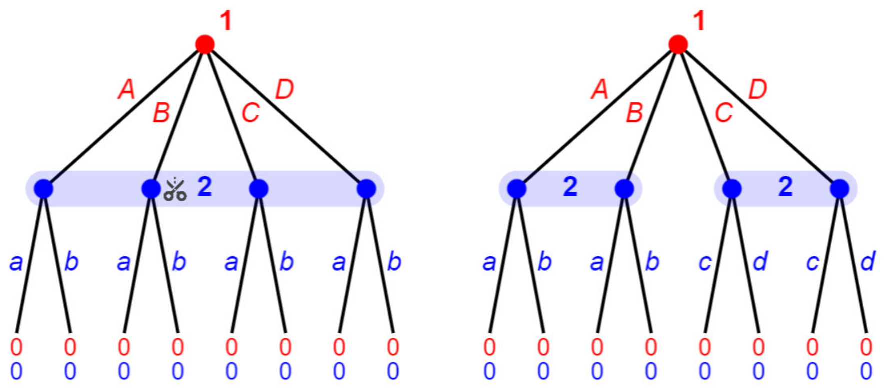


Figure 3.3: Cutting information sets.

Shortcut	**Action**
n	Adds children to selected nodes
d or *delete*	Deletes all children from the selected nodes If a node has no children, then it is removed
1–4	Assigns the corresponding player to the selected nodes
0 or *c*	Converts the selected nodes to chance nodes
i	Creates an information set with the selected nodes, if possible
u	Unlinks/removes information sets from the selected nodes
c	Cuts information sets
enter	Saves the label text and moves to the next label
tab	Moves to the next label in editing mode
shift+tab	Moves to the previous label in editing mode
esc	Exits the label editing mode
alt+n	Starts a new tree
alt+s	Saves the tree into a .txt file
alt+i	Saves the tree into a .jpg image
ctrl+z	Undo
ctrl+shift+z	Redo
(ctrl+)arrow keys	Manually reposition selected nodes

Figure 3.4: Keyboard shortcuts in GTE.

chosen a different approach – namely, we allow the user to manually reposition the nodes when a tree is completed. The user selects a set of nodes and by pressing the arrow keys on the keyboard, the selection can moved to the desired place. By holding the *ctrl* button on the keyboard, the user can fine-tune the position of the selected nodes, as they move with one pixel per frame in the desired direction. More information on nodes repositioning and information sets is provided in Section 5.2.

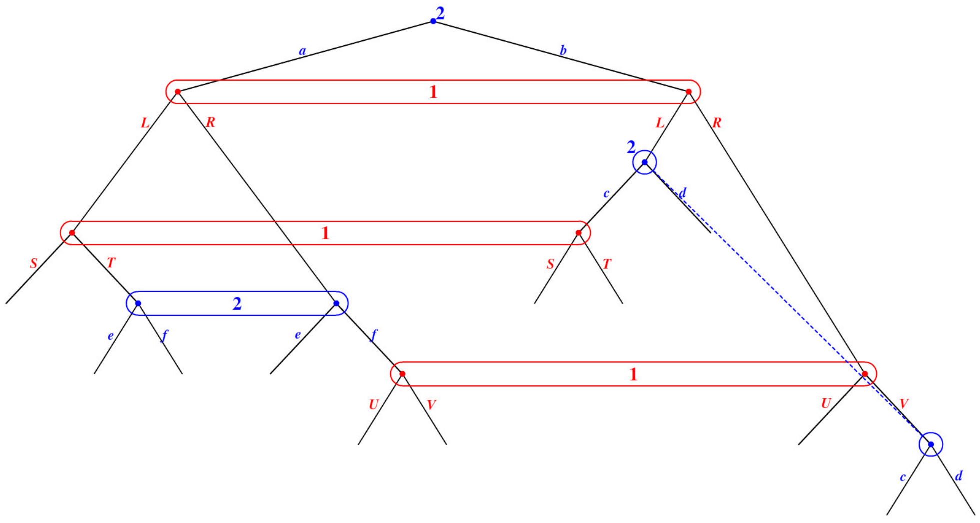
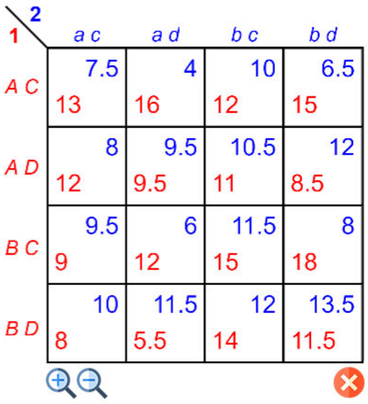


Figure 3.5: An example of a complex tree in Flash GTE.

Once a tree is created and the user starts changing the payoffs, the strategic form is calculated and appears on the same screen as the tree. The strategic form is updated every time the user modifies the payoffs and disappears when there is a change to the structure of the tree. We have also added controls for modifying the size of the strategic form and for moving it manually in case it overlaps with the tree. Figure 3.6 shows the strategic form of the game tree in Figure 3.2. We will explain the algorithms for calculating the strategic form in Section 5.4.



1 \ 2	a c	a d	b c	b d
A C	7.5 / 13	4 / 16	10 / 12	6.5 / 15
A D	8 / 12	9.5 / 9.5	10.5 / 11	12 / 8.5
B C	9.5 / 9	6 / 12	11.5 / 15	8 / 18
B D	10 / 8	11.5 / 5.5	12 / 14	13.5 / 11.5

Figure 3.6: Strategic form of the game tree in Figure 3.2.

3.3 Technologies

Front-end web development usually uses the three core technologies: HTML, CSS, and JavaScript. HTML is a markup language which provides the structure of a webpage. A single html file consists of a set of tags, denoting different building blocks of the page, e.g. text, images, links, lists, titles, headings, paragraphs, etc. CSS, the styling language, makes the page more user-friendly by adjusting the font sizes, colours, backgrounds, positions, and other features of the HTML tags. The functionality and behaviour of web content is most often written in JavaScript – a *"prototype-based"* scripting language which is directly interpreted by the browser. Prototype-based programming is a variant of object-oriented programming, where objects inherit functionality directly from other objects, as opposed to from predefined classes. In JavaScript each object has a property called *prototype*. The prototype is responsible for providing functions and fields, which are either inherited from other objects, or are directly attached. A book on JavaScript, which provides detailed review of the above, is given in [5].

Modern web applications are rarely limited to these three technologies. Instead, various libraries, packages, and engines are used, depending on the purpose of the relevant application. In the case of GTE, we substitute JavaScript for TypeScript [9] as the latter is easier to read and write, due to a variety of *syntactic sugars* (visually or logically appealing shortcut commands, provided by the language, in order to reduce the amount of written code). Most of the graphic representation in the browser will be implemented with a powerful game engine called Phaser [2]. Some additional features, such as menus and input fields, will be written in pure HTML and CSS. We will use a very popular library for an easy access to HTML tags and attachment of corresponding functionality, called jQuery [11].

Below, we examine the differences between TypeScript and JavaScript and present small overview of the Phaser engine to show its strengths and the reasons for using it in the project. We will leave jQuery out of our discussion as it used in very small parts of the code, which are mostly self-contained and relatively easy to follow.

JavaScript vs. TypeScript

Object-oriented programming is a methodology to modularise large programs to make them more abstract and easier to maintain. Its central tenet is the use of classes of objects with well-defined interfaces. Although object-oriented programming can be simulated in JavaScript, classes look very unnatural and are sometimes difficult to understand and read, especially in mid-to-large scale projects. We will be using TypeScript, which is a language developed by Anders Hejlsberg, a prominent Danish software engineer at Microsoft. Hejlsberg previously developed the popular programming languages Turbo Pascal and C# (which is a language similar to Java). TypeScript has many useful features, common in modern object-oriented languages, such as static typing, access modifiers, method overloading, properties, classes, generics and inheritance, amongst others. For the application to work, TypeScript files are compiled to JavaScript files, which are then interpreted by the browser. In Snippet 3.7 we can see a sample class, written in TypeScript, and its corresponding compiled version in JavaScript.

```
module GTE{
  export class Node{
    parent:Node;
    children:Array<Node>;

    constructor(parent:Node){
      this.parent=parent;
      this.children = [];
    }

    addChild(){}
  }
}
```

```
var GTE;
(function (GTE) {
    var Node = (function () {
        function Node(parent) {
            this.parent = parent;
            this.children = [];
        }
        Node.prototype.addChild =
            function () { };
        return Node;
    }());
    GTE.Node = Node;
})(GTE || (GTE = {}));
```

Snippet 3.7: A class in TypeScript (left) and the compiled JavaScript (right).

Looking at the TypeScript code, the keyword *module* is written to declare a scope for related classes, equivalently to a *package* in Java or a *namespace* in C#. In our application, all TypeScript files are within the *"GTE"* module. Next, the keyword *export* denotes that a class is visible to other classes within the module, in the same way *public* classes are visible in Java. We can also give a *private* access modifier to classes and variables, which hides them from public view. An important difference is that class fields and properties in TypeScript are normally given types, whereas in JavaScript all variables are declared with the keyword *var* and have no fixed type (such as string or integer).

In the JavaScript code the class is constructed solely by means of functions and variables, which often make it more difficult to read and follow. For example, when we want to add the method *addChild()*, it is attached as a function to the "prototype" of the class, instead of being written within the class. When implementing inheritance, the differences are more significant. In TypeScript we only need to add the keyword *extends* after the class name (e.g. *class Node extends INode*), whereas in JavaScript this is achieved by many additional function calls between the "base" object and the "derived" object.

Another advantage of TypeScript is achieved by the compilation process. When compiling files, meaningful warnings and error messages are shown if something is not written correctly. This is not the case in JavaScript as the code is directly interpreted by the browser, which makes finding errors and debugging the code more burdensome and time-consuming.

For the above-mentioned reasons, TypeScript has gained much popularity over the past few years and an ever increasing number of libraries are substitued for TypeScript-compatible versions. Examples include Angular 4 [7], a modern and very popular framework for creating web applications, developed by Google, and the upcoming Phaser 3 which will be entirely written in TypeScript.

Phaser

Due to security and performance concerns, the support for Flash-based applications, animations, and video games progressively declines among browsers. With the advancement of web technologies, there are now modern HTML5 alternatives which can be used instead. In GTE we incorporate Phaser, which is currently one of the most popular engines for developing web games. It was initially released in 2013 and as of August 2017 the latest stable version is 2.6.2. Phaser is the engine of choice for many game developers, due to its great community, detailed documentation, and the availability of numerous examples and resources. The engine is very powerful, providing useful features such as animations, events, physics, sounds, and particles, amongst others. It is also easy to set up and develop applications on Phaser. For our purposes, we will mostly be using the many options for *sprite* (an image object with additional properties, such as anchor, tint, opacity, scale, width, height, etc.) and text objects, the built-in events system and the *bitmapdata* object, which allows drawing simple shapes in the browser.

The example in Snippet 3.8 illustrates the basic functionality of Phaser. All classes which use the engine have a reference to the main object called *game*. We will manipulate the object to create a circle, using a bitmapdata, and will attach it to a sprite object.

```
module GTE{
  export class NodeView{
    game:Phaser.Game;
    bmd:Phaser.BitmapData;
    circle:Phaser.Sprite;

    constructor(game:Phaser.Game){
      this.game = game;
      this.createBitmapCircle();
      // parameters: (x, y, key)
      this.circle = this.game.add.sprite(0, 0, this.game.cache.getBitmapData("
          point"));
    }

    createBitmapCircle(){
      // parameters: (width, height, key, saveToCache);
      this.bmd = this.game.make.bitmapData(30, 30, "point", true);
      this.bmd.ctx.fillStyle = "#000";
      // parameters: (x ,y, radius, fromAngle, toAngle);
      this.bmd.ctx.arc(15, 15, 10, 0, Math.PI*2);
      this.bmd.ctx.fill();
    }
  }
}
```

Snippet 3.8: Creating a circle in Phaser.

When executed, the above code will create a white circle, with radius 10, situated in the top left corner of the screen. Shapes are drawn with bitmapdata in a similar way to drawing with Canvas in Java. There may be easier ways to create a circle in Phaser. However, the advantage of creating a bitmapdata as per Snippet 3.8 is that there will be many nodes on the screen and they will all use the shape which was saved to the cache. In other words, we only need to draw once and then reference the already created shape in the code.

Having created the sprite, we can set some parameters, as illustrated in Snippet 3.9.

```
setCircleProperties(){
  this.circle.anchor.set(0.5, 0.5);
  this.circle.position.set(this.game.width/2, this.game.height/2);
  this.circle.scale.set(2, 2);
  this.circle.tint = 0xff0000;
}
```

Snippet 3.9: Changing sprite properties.

It is often useful to set the reference point of a sprite to be its centre, because this makes the positioning easier. In Snippet 3.9, we place the sprite in the centre of the screen. With the last two lines of code we make the circle twice as large and change its colour to red.

One of the great features of Phaser is the Tween Manager, which allows the easy creation of animations. In Snippet 3.10, we animate the opacity of the circle until it becomes transparent for a duration of one second.

```
  this.game.add.tween(this.circle).
  to({alpha:0},1000);
```

Snippet 3.10: Animation with the Tween Manager.

Further details on the workflow and overall structure of a Phaser application are covered in Chapter 4.

3.4 Installing and Running GTE

GTE is available and can be tested live here: `https://lse68123.github.io` (named after the candidate number for this dissertation). We recommend accessing the page from Google Chrome or Mozilla Firefox since, at the time of writing, the software has not been optimised for all browsers. Furthermore, the software should be run on a Full HD resolution (1920x1080) or on a similarly scaled lower resolution (1600x900, 1366x768 or 1280x720), as scaling the application on different screens requires significantly more work. Due to reasons yet to be established and resolved, the software does not perform as expected on Linux (with low framerates and various bugs). GTE runs well on Windows and macOS, usually with the highest framerate of 60. The official repository of the code is stored on the Gambit Project's github page, which can be accessed here: `https://github.com/gambitproject/tsGTE`.

This section provides two ways of running the software locally. The first is almost entirely comprised of typing commands in the *Command Prompt* (or *Terminal* for Linux). For development purposes, we strongly recommend using a TypeScript supported *Integrated Development Environment* (IDE) such as Visual Studio Code [10] or Webstorm [8]. We provide instructions for the latter, as the new GTE has been implemented on it. Please note that instructions are only provided for Windows, but the equivalent steps are very similar on Linux. Let us first show how to download the project.

1. Download and install *Git* from `https://git-scm.com/downloads`. We will use it to get the project from github.
2. Download and install *Node.js* from `https://nodejs.org/en/download/`. Node will be used to download several packages needed to run the project.
3. Open the *Command Prompt* on Windows, e.g. press *"WinKey+R"*, type *"cmd"* and press *"Enter"*.
4. Navigate to your desired directory, e.g. type *"cd c:\"*.
5. Type *"git clone https://github.com/gambitproject/tsGTE"*.

By following these steps, the project will be downloaded to the desired folder, for example *c:\tsGTE*. In order to run the software without an IDE we need to download a TypeScript compiler, compile the project, and run it from a local server as follows:

6. Navigate to the project folder, e.g. type *"cd tsGTE"*.
7. Type *"npm install -g typescript"*. This command installs TypeScript globally on the computer.
8. Type *"tsc"* to compile the files into JavaScript. In the project folder for every *.ts* file, the corresponding *.js* file should be created.
9. Type *"npm install connect serve-static"*. We will use these libraries to create a local server and run the software.
10. Open the project folder in File Explorer and create a new file called *server.js*. Open the file with Notepad and type the following commands:

```
var connect = require('connect');
var serveStatic = require('serve-static');
connect().use(serveStatic(__dirname)).listen(8000, function(){
  console.log('Server running on 8000...');
});
```

11. Save the file and go back to Command Prompt.
12. Type *"node server.js"*. You should see the message *"Server running on 8000..."* written in Command Prompt.
13. Inside your browser go to *localhost:8000* (or *localhost:8000/index.html*). After a short loading period, GTE should now be running in the browser.

For development purposes, it is quicker and easier to use an IDE, in our case, WebStorm. We substitute steps 6–13 to run the project with WebStorm as follows:

6. Download and install WebStorm from `https://www.jetbrains.com/webstorm/`.
7. Open the project with WebStorm.
8. Go to: File $\rightarrow$ Settings $\rightarrow$ Languages and Frameworks $\rightarrow$ TypeScript
9. Fill in the path to the *node.exe* file.
10. Under *Compiler* tick the *Enable TypeScript Compiler* and the *use tsconfig.json* options.

11. Exit the Settings menu. Press the *"shift"* keyboard button twice and type *"TypeScript Compile all"*.
12. Open the *index.html* file and run it from the browser menu in the top right-hand corner.

WebStorm is free for students and open-source projects and is available on Windows, macOS, and Linux. The advantages of using this IDE include the built-in TypeScript support, code auto-complete, and the automated process of running websites on a local server, amongst others.

Chapter 4

Software Architecture and Design Patterns

A general focus of the new GTE is the overall improved quality of the code and its structure. By following certain rules and choosing appropriate design patterns, we aim to create a software which can be easily understood, maintained, and extended. This way GTE would not have to be rewritten soon, as the HTML5 framework, technologies and Phaser (our engine of choice) are unlikely to become obsolete in the near future. With the upcoming major release of Phaser 3, and once it becomes stable, it should be considered whether the current version of the framework should be updated in the event of any performance issues. The developers of Phaser have indicated that upgrading will not be complicated. The fact that we do not use many of Phaser's features (such as physics, particles, audio, camera manipulations, filters, etc.) ensures that updating the engine is unlikely to be problematic.

There are several principles that developers should follow when writing code. This chapter explains some of them and our aim is to choose an architecture and design patterns that conform to these rules. Firstly, every project should adhere to the *DRY* (Don't Repeat Yourself) principle. This general and simple rule may sometimes be difficult to follow, especially in large projects. Some repetition of the code may be necessary but should be avoided to the largest extent possible. Secondly, classes should have *high cohesion*, meaning that each class should be focused and do only what it is supposed to do, nothing more. *Loose coupling*, on the other hand, states that classes should be independent of each other as much as possible. If a set of classes is tightly coupled, this means that modifications to one class will have to correspond to modifications in all other related classes. Projects developed with tightly coupled classes are very difficult to extend and maintain.

Another popular set of rules are the *SOLID* principles, listed in Figure 4.1 below. They relate to larger projects which contain a great number of abstractions and interfaces. In the new GTE we rarely use inheritance and abstract classes, therefore, the SOLID principles will not be covered in much detail in this dissertation, however we have tried to conform to them as much as possible, where they apply. One can read further on the SOLID principles Wiki page [15].

Complying with all of the above-mentioned principles is not always easy, so in order to overcome some of the related difficulties we use *design patterns*. A design pattern is an abstract and reusable solution to a problem occurring in software design. These problems mostly appear when one wants to add new functionality to a project or to structure it properly, in order to conform to all of the principles mentioned above. For more information on design patterns please see [6].

Initial	Principle
S	*Single responsibility principle* – a class should have only one reason to change, meaning that a class should only serve a single purpose.
O	*Open-closed principle* – software entities should be open for extension, but closed for modification.
L	*Liskov substitution principle* – subclasses should be substitutable for their base class, without altering the correctness of the application.
I	*Interface segregation principle* – many client-specific interfaces are better than one general-purpose interface.
D	*Dependency inversion principle* – software entities should depend upon abstractions and not upon concretions.

Figure 4.1: SOLID principles.

Although, we can pinpoint specific locations in the code which incorporate design patterns, in this chapter we will turn our attention to the chosen classical architectural pattern Model-View-Controller. By structuring the code in this manner we achieve a clear separation of responsibilities within the project, which in turn preserves extensibility, maintainability, and clarity. In Section 4.2 we present the overall workflow of the project and how the MVC pattern fits in the structure. Sections 4.3 and 4.4 proceed with examples of some problems and their solutions when applying the chosen architecture.

4.1 The Model-View-Controller Framework

The MVC design pattern is one of the most widely used architectural software solutions. It is integrated in many programming frameworks, including Sprint (for Java), Django (for Python) and ASP.Net MVC (for C#).

MVC fits in perfectly with GTE, as it separates the tree, related data structures and algorithms (the Model) from the way they are presented on the screen (the View). We use Controllers for changes in the Model to be reflected in the View, or user actions to modify the Model. The purpose of these Controllers is to transfer signals from the Model to the View and vice versa in order to preserve the separation between data and algorithms from the user interface. Figure 4.2 shows the relationship between the three structures.

Decoupling the Model and the View has a significant advantage – if at some point we decide to change the graphics engine, this can be achieved without modifying the Model (i.e. the data structures and algorithms will remain intact). Such a modification should easy, since we only need to “translate” the corresponding View classes in the language of the new engine (for example, in the *NodeView.ts* class we have to reimplement the drawing of a circle).

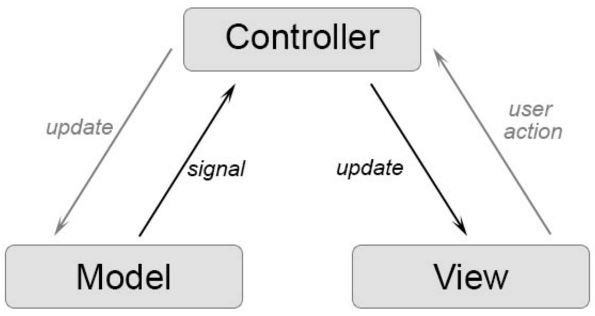


Figure 4.2: The MVC pattern.

Furthermore, the MVC architecture allows a relatively easy extension of the project. If we wish to add the strategic form of a game tree at some point in our development, we can do so by creating a new *StrategicForm.ts* class in the Model, which will contain the logic and algorithms. Next, we would implement its corresponding View class, which will be concerned with drawing a table.

Another advantage of the MVC architecture is the ability to quickly pinpoint and fix bugs in the application. If the data and the interface were to be put in the same class, this would not be true, as a given class (e.g. of type Node) would get significantly larger. Then, it would not be clear which part of the object is not working correctly.

We proceed by examining the three parts of the MVC architecture, in turn.

Model

In GTE, the Model consists of the building blocks of a game tree. The main classes in the Model are *Node.ts*, *Move.ts*, *Node.ts*, *ISet.ts*, *Player.ts*, and *Payoffs.ts*. The complete representation of a game tree is written in *Tree.ts* and it is composed of instances of these building blocks. Another class in the Model is the *LabelSetter*, which calculates the correct labels for the moves with respect to information sets and players. Finally, the strategic form is in a separate class and is decoupled from the tree structure, i.e. it is not part of *Tree.ts*. Figure 4.3 represents the class diagram for the Model which includes the main data structures, how they relate to each other, and some of their properties and methods.

These classes are linked in various ways, indicated by the lines between rectangles (which represent the classes in the Model). For example, a node has references to its parent, children, parent move, children moves, information set, and to a player. We also keep node references in the corresponding information sets and moves. For these reasons, the *Tree* object becomes a circular data structure. It may seem tightly coupled, but every link is created to serve a purpose – namely, to save time in accessing specific elements. This is done in constant time, instead of searching for an item in linear time.

As an example, suppose we omit the array of nodes in the class *ISet*. In order to find all elements with a specific information set we would have to traverse the nodes of the tree and compare their information set reference against the one we are given.

In general, working with such data structures is slightly more challenging and one must be careful to keep track of all the references, especially when removing elements from the

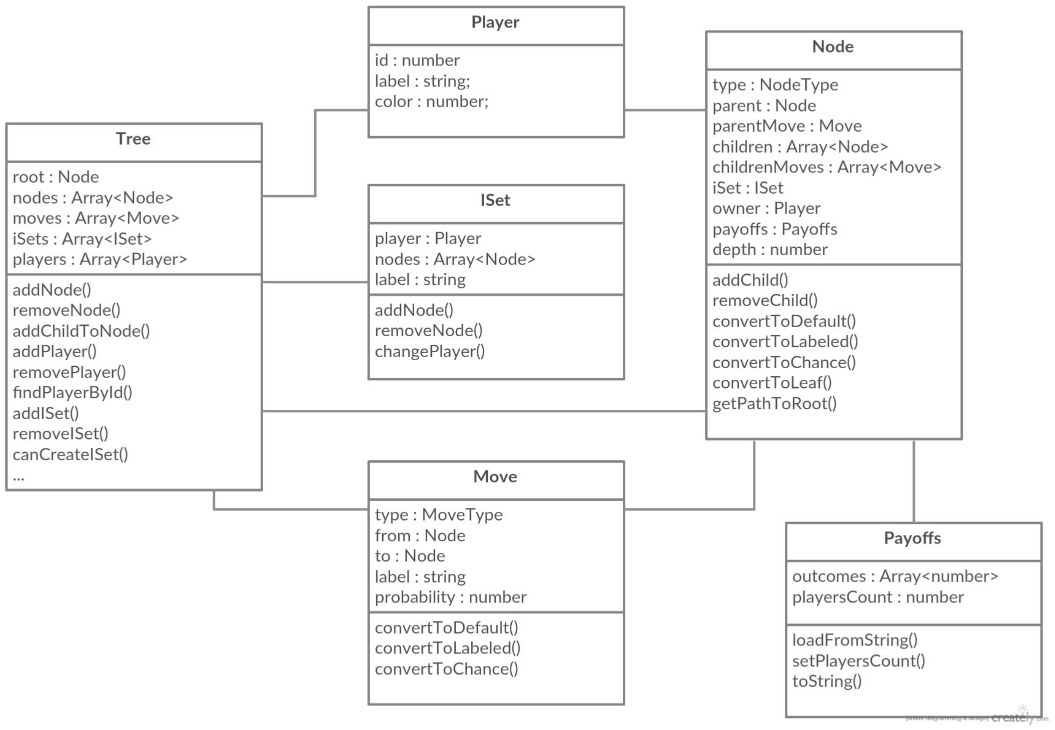


Figure 4.3: The Model structure.

tree. On the other hand, these references are convenient, as they allow constant access to the elements needed, in order to draw tree and implement various related algorithms.

Another problem which arises with circular structures is serialisation, i.e. transforming the data structure into an object which can be stored or transmitted and reconstructed later. A solution to this problem in the context of game trees and the functionality that serialisation provides is examined in more detail in Section 4.4.

We proceed with an example of creating a simple game tree, by using only the data structures presented in the Model. In Figures 4.4–4.6 below, we show the code on the left and the corresponding tree which would be produced by the View, on the right.

We start by creating the nodes of the tree.

```
let tree = new Tree();
tree.addNode();

tree.addChildToNode(tree.nodes[0]);
tree.addChildToNode(tree.nodes[0]);

tree.addChildToNode(tree.nodes[1]);
tree.addChildToNode(tree.nodes[1]);

tree.addChildToNode(tree.nodes[2]);
tree.addChildToNode(tree.nodes[2]);
```

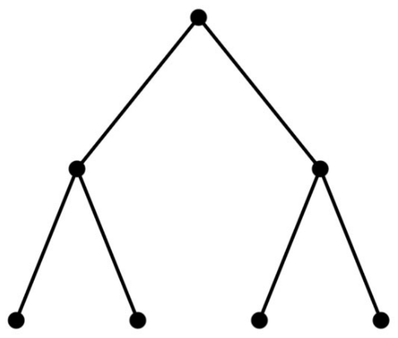


Figure 4.4: Creating a simple tree in the Model.

Next, we can add the players and an information set. Note that on the first two lines of the code the colours are given as strings in hex format and correspond to red ("#f00") and blue ("#00f").

```
// parameters: (id, label, colour)
tree.addPlayer(new Player(1,"1","#f00");
tree.addPlayer(new Player(2,"2","#00f");
let n1=tree.nodes[0],n2=tree,nodes[1],
  n3=tree.nodes[2];
n1.convertToLabeled(tree.players[0]);
n2.convertToLabeled(tree.players[1]);
n3.convertToLabeled(tree.players[1]);
// parameters: (player, nodes)
tree.createISet(tree.players[1],
  [n2,n3]);
```

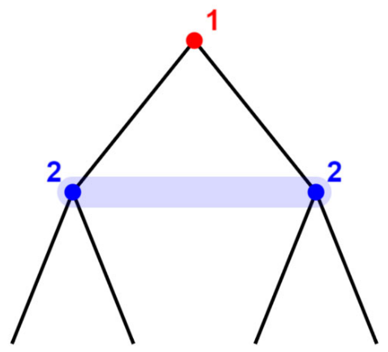


Figure 4.5: Adding players and information sets.

Finally, one can add the corresponding payoffs and labels, as shown in Figure 4.6. Please note that the *LabelSetter* class calculates the move labels automatically with an algorithm, which requires the nodes in breadth-first order and the array of players. In this algorithm, the labels are the letters of the alphabet, where each move label for player one is assigned (in the breadth-first order) a capital letter and for player two a lower case letter.

```
let nodes = tree.BFSOnTree();
let players = tree.players;
tree.calculateLabels(BFSnodes, players);

let leaves = tree.getLeaves();
leaves[0].payoffs.loadFromString("1 3");
leaves[1].payoffs.loadFromString("5 2");
leaves[2].payoffs.loadFromString("4 4");
leaves[3].payoffs.loadFromString("3 0");
```

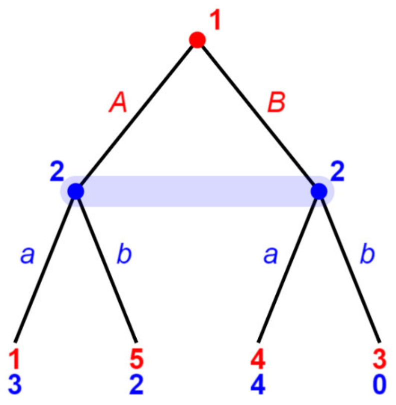


Figure 4.6: Adding labels and payoffs.

When working with the Model, one should use the built-in functions in the *Tree.ts* class, because they manage the references in the circular structure, as mentioned above. For example, adding players directly to the array *tree.players* may cause errors in the application. This is where access modifiers would be invaluable, e.g. setting a *private* access modifier to the players array. Unfortunately, in JavaScript every property and method of a class is public by nature and can be modified externally. In TypeScript access modifiers do not change this fact as the code is compiled to JavaScript. The modifiers are mostly added for clarity of the code and to show developers that a given property or method should not be accessed from outside of the class.

View

The purpose of the View part of the MVC pattern is to receive a game tree (from the Controller) and to draw it on the screen. This is, by definition, very much dependent on the graphics engine, in this case Phaser. We have briefly introduced the framework in Section 3.3 and this subsection concentrates on the way it is incorporated in GTE.

Most classes in the Model have corresponding ones in the View: these are *NodeView.ts*, *MoveView.ts*, *ISetView.ts* and *TreeView.ts*. There is an additional file *TreeViewProperties.ts* which can be thought of as a structure (a class consisting of only a list of properties, without methods) and which contains the basic settings of how a tree is drawn (e.g. height between two levels in the tree, width between two nodes, level of zooming, node radius, line width, and so on). For now, we will leave out the strategic form of our discussion and will concentrate on the tree.

Each building block of the *TreeView* extends the *Phaser.Sprite* class, which means that they are essentially sprites with additional properties and functionality. For the moves and information sets this is exactly the case – an image is attached (a line or an oval) directly to the object. In the *NodeView* the inheritance serves a different purpose. The associated sprite is invisible and is larger than the circle. Signals and events are attached to invisible sprite, in order to make hovering and clicking easier for the user. To draw the node, we use three properties which are also of type *Phaser.Sprite* – namely *circle*, *square*, and *previewSelected*. The *previewSelected* property is a slightly larger light-blue circle, displayed around the node and indicates if a node is currently selected.

We examine the *NodeView* class in more detail. The diagram on Figure 4.7 lists most of its properties and methods.

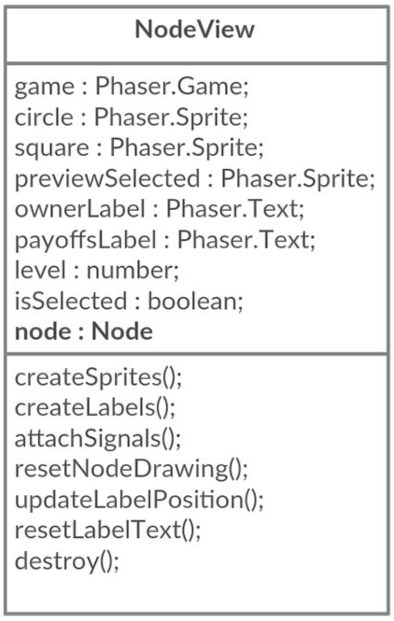


Figure 4.7: The NodeView class.

As noted previously, all classes related to the View and the engine contain the main *game* reference. There are three sprite objects for the circle, square, and preview, and two labels which are of type *Phaser.Text*. Text objects are very similar to sprites in Phaser with some additional properties for setting specific fonts, font weights, line spacing, etc. The *level* property is used to correctly position the node with regards to information sets. By default, it is equal to the depth of the node (distance from the root) in the Model. When creating an object of type *NodeView*, one first creates the sprites and labels by invoking the corresponding methods. Whenever there is a change in the model, the way the sprites are drawn is reset, which includes the circle, square, preview circle, and the labels.

In order to improve the readability of the code and make it easier to follow, we deviate slightly from the pure MVC framework by including Model references in the View. In the *NodeView* class there is a property *node* of type *Node* which is highlighted in Figure 4.7. To see how this makes the code clearer, let us consider a portion of the method *resetNodeDrawing()* in Snippet 4.8.

```
resetNodeDrawing(){
  // Selected and not Chance
  if (this.isSelected && this.node.type !== NodeType.CHANCE) {
    this.circle.alpha = 1;
    this.circle.tint = NODE_SELECTED_COLOR;
    this.square.alpha = 0;
    this.previewSelected.alpha = 0.3;
  }
  // Selected and Chance
  else if (this.isSelected && this.node.type === NodeType.CHANCE) {
    this.circle.alpha = 0;
    this.square.alpha = 1;
    this.square.tint = NODE_SELECTED_COLOR;
    this.previewSelected.alpha = 0.3
  }
  ... // All other cases
}
```

Snippet 4.8: A sample method from NodeView.ts.

The “proper” way to implement this method in terms of MVC would be to add a parameter *nodeType* to the method and send it from outside of the class (from the TreeView) as opposed to keeping a node reference in the View. In the *resetNodeDrawing()* method, there is no problem in doing so. However, an issue arises when we draw the whole tree as this requires a very large list of parameters which will have to be transmitted from the Controller to the View. Such methods (which require many parameters) are very difficult to read and this is the reason for our deviation (namely keeping Model references in the View). There is another reason for our deviation which will be discussed in the next subsection.

The above paragraph summarises what is, in most cases, the “correct” way to structure the links between the View and the Model when using Phaser. As Phaser is a game engine it is capable of redrawing the tree 60 times per second. Thus, whenever there is a change in the Model, the View can automatically detect it because of the above-mentioned referencing between the Model and the View. The corresponding drawing of the tree is seen immediately due to the high frame rate of Phaser (as a game engine). Such implementation may become problematic in the future if the project is extended (e.g. with more complicated node repositioning or label setting algorithms), because of the calculations overload per frame. Therefore, to keep the application optimised and scalable we only update the

View when there is a change in the Model.

The *TreeView* looks very similar to the *Tree* class - it has arrays of nodes, moves, and information sets, this time taken from the View. The View also includes a tree reference so that whenever we want to redraw it, the class will have all the necessary information, as discussed above. The main method here is *drawTree()*, which is called when there is a change in the Model. It repositions the nodes and moves to their respective correct places, activates player, move and payoff labels when the conditions are met, and creates information sets. Some of these algorithms will be explained further in Chapter 5.

Controller

The Controller serves as the main link between the Model and the View. This is where all objects come together and communicate with each other, normally via signals. There are five main classes – *TreeController.ts*, *UndoRedoController.ts*, *UserActionController.ts*, *Keyboard-Controller.ts*, and *HoverMenuController.ts*. The links between them are presented in Figure 4.9, where an arrow indicates that a reference of a class is kept in another class.

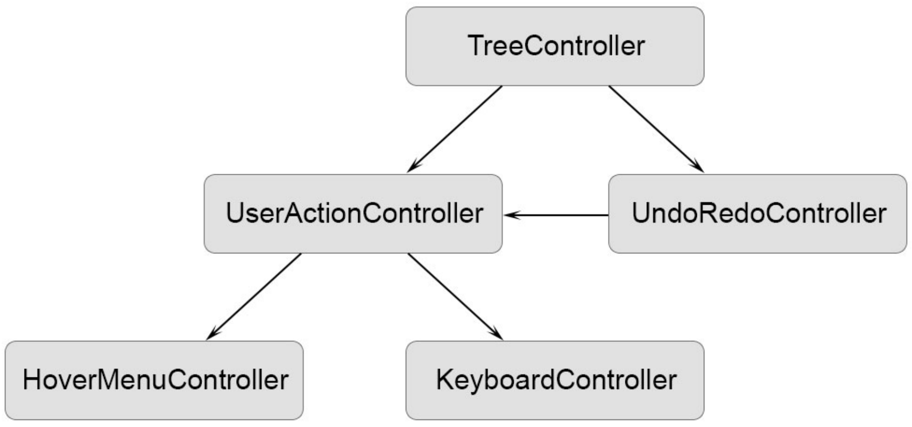


Figure 4.9: Links between Controllers.

TreeController.ts is the main class in the application. It creates, changes, and manages both the TreeView and Tree classes. Any signals and actions from the user are handled in *TreeController* and the corresponding modifications are made in the structure and drawing of the tree. The procedure works as follows – a user action is detected from the View (e.g. a node is hovered and children added with the button) ; the *TreeController* receives a signal and makes changes to the Model ; and finally the View is instructed to redraw the tree. The above method works almost as in Figure 4.2, the only difference being that there are no signals from the Model to the View, as explained in the previous subsection.

Another reason for the deviation from the MVC structure is a problem called "Fat Controller". Since the whole project is operated from a single class, it may get very large very quickly as the application is extended. This is why the *TreeController* is "relieved" of some of its duties to keep it more concise and self-contained, by including Model references in the View.

Input from the user has been implemented in a flexible way. Assuming that changes related to the tree are handled in the *TreeController*, the *UserActionController* keeps a list of all actions that can be performed by the user (such as adding or removing nodes, changing players, assigning labels, undo or redo, etc.). These actions can easily be called from different places in the user interface. Let us consider for example the action of assigning a player to a given node. The logic is implemented in the *TreeController*, and the *UserActionController* can invoke this method on a set of nodes. There are two ways such a call can occur: from the hover menu, or by pressing a keyboard shortcut key. Both corresponding classes (*HoverMenuController* and *KeyboardController*) contain a *UserActionController* reference and they each use the same method in the same way. By having this structure of Controllers one can quickly add new ways for the user to interact with the application. Although not given in the diagram above, the *Top Menu* operates in a way identical to the one described above. It keeps the same reference and adding an undo button, for example, can be created with a single line of code. Furthermore, implementing additional functionality in the tree can be relatively easy. The corresponding algorithm should be written in the *TreeController* – a method is added in the *UserActionController* which can invoke this algorithm. Then, depending on how the user should interact, a key in the *KeyboardController*, a button in the hover menu, or a button in the top menu, can be added.

4.2 Project Structure

In this section we complement the discussion on the architecture of GTE by showing the file structure and the workflow of the Phaser application. Figure 4.10 shows how classes and libraries are organised in folders.

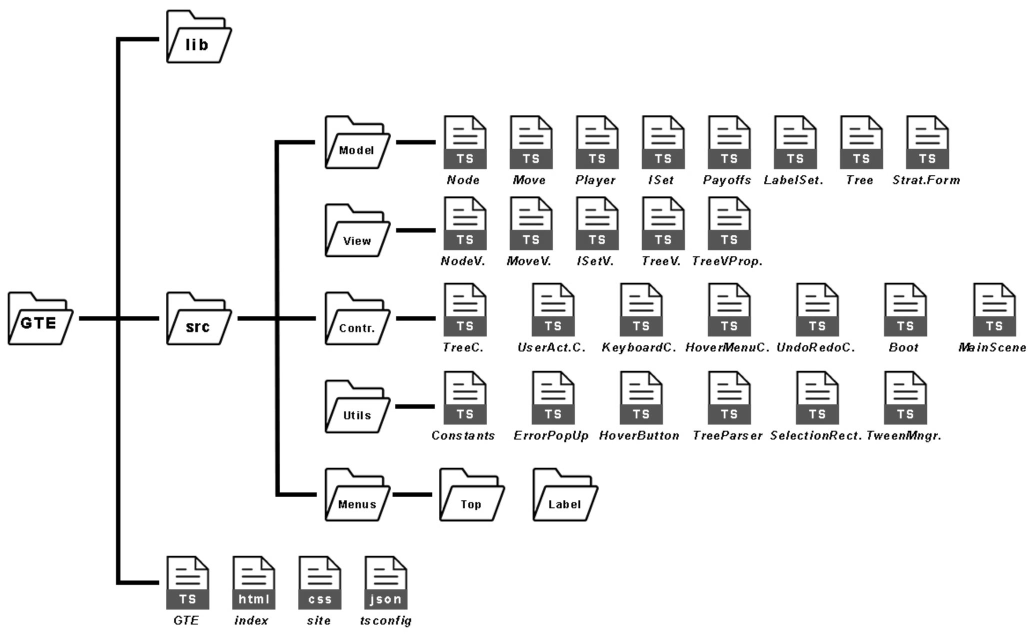


Figure 4.10: GTE file structure.

The *index.html* file has a *div* tag which is the container of the application and *site.css* positions and stretches it to cover the screen. The *tsconfig.ts* file is standard in TypeScript applications. It contains different options for the compilation process of JavaScript files from TypeScript files, external libraries, and dependencies. For simplicity of downloading and running the project, all additional libraries are pre-downloaded and stored in the *lib* folder. Currently it contains Phaser, jQuery, and a plugin called *FileSaver* for saving and loading files from a computer.

The application starts from *GTE.ts*. The file tells the engine how to find its container, i.e. the *div* tag, and loads the states of the application. For a video game, these would be for example *Preload*, *Menu*, *Settings*, *Game*, *Game Over*, and so on. In GTE, we only use two states – *Boot* and *MainScene*. In the Boot state, we load all assets, draw the circles and lines (which are saved to the cache for later use) and start an initial animation. The main body of the application is initiated from *MainScene.ts*. The file is very short as it only creates the Controllers, which take care of the tree, user input, and menus as explained above.

In the additional folder, named *Utils*, there are some files which complement the MVC architecture. The *Constants.ts* file has a list of fixed values for the basic settings of the tree such as initial node width, player colours, error messages, and so on. The reason for extracting these settings is that when one wants to change some of them, one does not need to look through the code to find them. There is a file which draws a red error message rectangle, normally displayed when the user tries to create an information set which does not follow the rules explained in Chapter 2. *SelectionRectangle.ts* contains logic for the left mouse click and drag operation which selects multiple nodes. The *TreeTweenManager.ts* animates the tree and the *TreeParser.ts* serialises it. The latter will be discussed in detail at the end of this chapter.

Finally, it is worth pointing out that some parts of GTE are more easily written with pure HTML5 instead of Phaser. These are the *Top Menu* and the *Label Setter*. Each of these folders contains three files – *.html*, *.css*, and *.ts* which all are self-contained. The label is a simple input field which changes the text for moves, players, and payoffs. It has a simple functionality – the *UserActionController* tells it when and where to appear and takes the input text from the user.

For the remainder of this dissertation we move away from the abstractness of the architecture and the structure and we focus on specific problems in the development process and their respective solutions.

4.3 Input Handling and Signals

One way to handle input in an application is to detect the coordinates of the mouse when, for example, the user clicks on the screen. This is very inefficient, since for every click one must to scan through all objects on the screen to check which specific object has received input. It would be even more difficult and computationally intensive to determine various actions such as hovering over and outside of a sprite.

Phaser has its own input handling system – it is fairly simple to implement, quite powerful, and optimised, allowing complete freedom over user interaction. The idea is that every display object in the application (sprites, texts, buttons, etc.) has an event handler. We attach functionality to an event (for example *onClick*) and the display object *"knows"* when the corresponding event has been triggered. In GTE, we also sometimes require that, when triggered, such an event is handled from outside the class. For instance, if we would like to add children to a node when it is clicked, the corresponding functionality cannot be accessed from the *NodeView* but from the *TreeController* or *UserActionController* classes instead. In this case the *NodeView* fires a *signal* when it has been clicked and the *TreeController* contains a *listener* which waits for such signals to be triggered. Upon receiving a signal, the *TreeController* handles such signals by invoking the corresponding method (e.g. for adding children to a given node).

We consider a detailed example of integrating the input handling system in GTE below. When the user hovers over a node, we open a menu with different possible actions for the node (adding or removing children, assigning players, and so on). When the user clicks on a button from the menu, we invoke the corresponding method from the *UserActionController*. The UI for accessing and using the menu is portrayed in Figure 4.11.

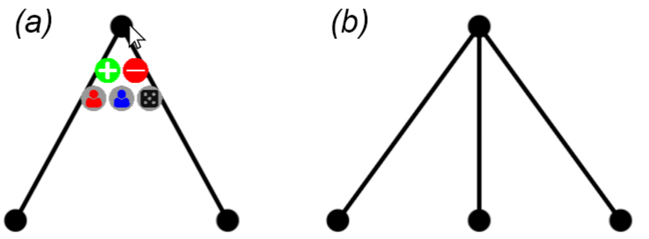


Figure 4.11: (a) Hovering over a node. (b) Clicking on the plus icon adds a child.

The functionality explained above, is achieved in the following way: the node dispatches a hover signal from the corresponding *NodeView* instance, which in this case represents root; the *TreeController* is subscribed and listens for signals from the *NodeView*; upon receiving the hover signal, the controller fires another signal, which in turn is picked up by the menu (and which then appears on the screen). Since the *HoverMenuController* class has a *UserActionController* reference, upon clicking on a button from the menu, the corresponding functionality is invoked directly without firing additional signals. In the figure above, the *plus* button has been clicked, which adds another child to the root. The transmission of signals is illustrated in Figure 4.12 below.

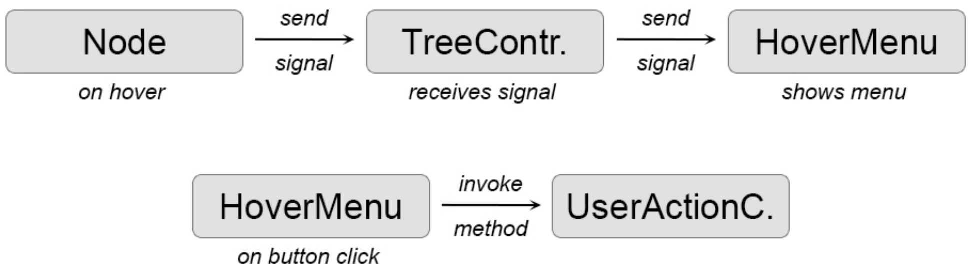


Figure 4.12: Transmission of signals for adding a child to the root.

We now turn to the implementation of this operation. As mentioned in Section 4.1, the *NodeView* class extends *Phaser.Sprite*. When the class is referenced with the keyword *"this"* we can use properties and methods which are inherited from the *Phaser.Sprite* class. We make each *NodeView* invisible, as it is only concerned with handling input – it is larger than the circle so that the user can interact with the node more easily.

Snippet 4.13 below sets some properties of the sprite, in particular, setting *inputEnabled* to true means that the sprite will dispatch signals when the node is hovered over or clicked. These signals are then handled externally.

```
this.alpha = 0;
this.anchor.set(0.5,0.5);
this.scale.set(3);
this.inputEnabled = true;
```

Snippet 4.13: Enabling the dispatch of signals in the NodeView class.

Next, in the *TreeController* class we create a custom signal called *hoverSignal* and dispatch it when the user hovers over a specific instance of the *NodeView* class. We also send a reference to the node that has been hovered, because, depending on its position in the tree, the menu shows different options (e.g. if the node is a leaf, then players cannot be assigned to it).

```
// "this" references the TreeController class
this.hoverSignal = new Phaser.Signal();

private attachHandlersToNode(n:NodeView){
  n.events.onInputOver.add(()=>{
    this.hoverSignal.dispatch(n);
  }
}
```

Snippet 4.14: Receiving a signal from the node and firing another signal to the menu.

Finally, the menu receives a signal which tells it to appear. Upon clicking a specific button, we invoke the corresponding method from the *UserActionController*.

```
// "this" references the HoverMenuController class
this.userActionController.treeController.hoverSignal.add(()=>{
  // arguments[0] is a reference to the hovered node
  this.triggerMenu(arguments[0]);
});

this.plusButton.events.onInputDown.add(()=>{
  this.userActionController.addNodesHandler(this.hoveredNode);
});
```

Snippet 4.15: Receiving the hover signal and adding a child when the user clicks on the "plus" icon.

The whole procedure, as explained in this section, may seem overly complicated. It could appear that, it would be much easier to keep a menu reference in the *NodeView* class and simply instruct the menu to appear when the user hovers over a node. However, keeping such references heavily couple the code, which is something we would like to avoid. By integrating the input handling system from Phaser, we transmit signals more abstractly and globally throughout the project, whilst keeping it decoupled, self-contained, and cohesive.

4.4 Undo/Redo Functionality

In general, there are two major techniques for implementing undo/redo functionality. The first is based on *states* and the second – on *actions*. Implementing the latter requires a list of actions. If the user chooses to undo a specific action, its corresponding reverse is applied. This technique is relatively lightweight (in terms of memory) and usually preferred when programming a text editor, for example. If the user highlights a piece of text in bold, the corresponding reverse action is to remove the highlighting. In other cases, reversing an operation may be extremely difficult, for instance, applying an *"oil-paint"* effect to an image.

Implementing an undo/redo feature based on states is in some cases easier, but less memory-efficient. For every user action, the current state of the object is saved in a list of states. When the user wants to access a specific state (not necessarily the one immediately preceding or following the current state), the object is restored from the list. GTE incorporates the state-based approach as the tree structure does not use up a lot of memory. This method fits in well with the chosen architecture. At each step, the current tree is saved from the Model to the *UndoRedoController*. When the user accesses a different state, such state is given to the *TreeView* which, in turn, draws the tree in the desired state.

Cloning the Tree Data Structure

Saving a tree in a list requires to clone the data structure for each user action. There are a number of ways in which this can be implemented, for example with a *clone()* method inside the *Tree.ts* class. We have chosen another approach which is to serialise the tree, in this case we convert it to a string. The string is then parsed which creates a deep clone of the object.

JavaScript has a built-in format for serialisation called JSON (JavaScript Object Notation). In theory, the tree can be converted to a string object by calling *JSON.stringify(tree)* and then deserialised via *JSON.parse(treeAsString)* which would return the cloned object. In practice, however, JSON does not support serialisation of circular data structures. Snippet 4.16 shows a simple circular object and the error we receive when we try to convert it to a string.

```
var a = {};
a.b = a;
JSON.stringify(a);

// Produces the following error:
Uncaught TypeError: Converting circular structure to JSON
```

Snippet 4.16: Stringifying a circular data structure with JSON.

The problem arises because *JSON.stringify* tries to recursively handle circular references.

In the case of a *Node* object, the list of children is stringified. Then, for every child, the parent reference has to be converted as well, which includes a list of the same children.

A simple solution to this problem is to incorporate another serialiser. In JavaScript a popular serialiser is YAML [4], which is a superset of JSON. The way YAML handles circular references is to keep an address of each stringified object and whenever an object appears twice in the serialisation, it is replaced with the corresponding address.

In GTE, we have chosen to limit our dependence on external libraries as much as possible, so the above procedure of decircularising the tree is performed manually by our code. This is implemented in the *TreeParser* class inside the *Utils* folder and works as follows. Given an instance of type *Tree* we create a simplified version of it called *strippedTree*, which contains arrays for the players, nodes, information sets, and moves. Firstly, we copy all players from the original tree to the simplified tree. Secondly, we copy all nodes from the nodes array, without any parent-child relationship. Finally, we copy all moves and information sets, where each node reference is replaced with its corresponding index in the array of nodes.

In order to save a valid tree (to be drawn by the *TreeView* class) it needs to be parsed from the simplified version. The only information that is missing is the parent-child relationship between the nodes, which is taken from the moves. Snippet 4.17 illustrates this operation, where *m* denotes the simplified move with indices instead of references.

```
// Create a Move and copy the primitive data from the simplified move "m"
let move = new Move();
move.label = m.label;
move.probability = m.probability;

// Save the node references from the tree to the move
move.from = tree.nodes[m.fromIndex];
move.to = tree.nodes[m.toIndex];

// Copy the parent-child relationship between nodes from the move
move.from.children.push(move.to);
move.to.parent = move.from;
move.from.childrenMoves.push(move);
move.to.parentMove = move;
```

Snippet 4.17: Parsing circular references of the nodes from a move.

Next, we show how cloning helps in the implementation of the undo/redo functionality in GTE.

Undo/Redo Controller and Storing Trees

Once the tree cloning functionality is completed, utilising it in the *UndoRedoController* and, additionally, for saving and loading trees from files, is relatively simple. In order to save trees, the class keeps an instance of the *TreeParser*, a list of trees and the index of the current tree which is shown on the screen. For every user action, the new tree is saved with the following lines of code:

```
// "this" references the UndoRedoController class
let clonedTree = this.treeParser.parse(this.treeParser.stringify(tree));
this.treesList.push(clonedTree);
```

Depending on the user action (undo or redo), the currently visible tree on the screen is destroyed (both the corresponding View and Model instances) and a new tree is created with the new index from the list.

The *UndoRedoController* is instantiated and exclusively used in the *UserActionController*. For example, the method of creating an information set and saving the tree looks as follows:

```
// "this" references the UserActionController class
createISetHandler() {
  this.treeController.createISet(this.treeController.selectedNodes);
  this.undoRedoController.saveNewTree();
}
```

By implementing the undo/redo functionality in this way, the developer has complete freedom over when to save a tree and how to load it. For example, as discussed in Section 3.2, the user can perform actions on multiple nodes or on a single node. In each case the new tree is saved as it appears on the screen. If the action-based approach of the undo/redo functionality was incorporated, reversing a user action for multiple nodes could be more difficult to implement and store. For further development, a new UI feature can be added relatively easily – namely, an additional menu with a list thumbnail images, where the user can quickly see the steps of creating a tree and can access any one of them from the list.

Saving to and loading a tree from a file has also been implemented by utilising the serialisation of a tree. When the user saves a tree, a *.txt* file is created which stores the stringified object. Loading a tree from a file is virtually the same as changing the current tree from the *UndoRedoController*. A desired feature of the software for the future is to have a user system where one can save and access previously created trees. This can be implemented by saving trees on a server which is both quick and efficient as only the string representation needs to be stored.

Chapter 5

Development Challenges and Solutions

In this chapter we analyse some of the difficulties in the development process of GTE. We will focus on problems that arise from implementing specific features and we show how these problems were handled in the current HTML5 version of the software.

5.1 Tree Drawing Algorithm

We begin our discussion with a basic algorithm for drawing a tree, which incorporates a bottom-up approach by first positioning the leaves and then the parents. The corresponding method, called *drawTree()*, is situated in the *TreeView.ts* class. It has an instance of an external class *TreeViewProperties* which contains two properties *levelHeight* and *treeWidth* (shown in Figure 5.1 below). The reason why these and other properties are abstracted in a separate class is because they can be modified by the settings menu.

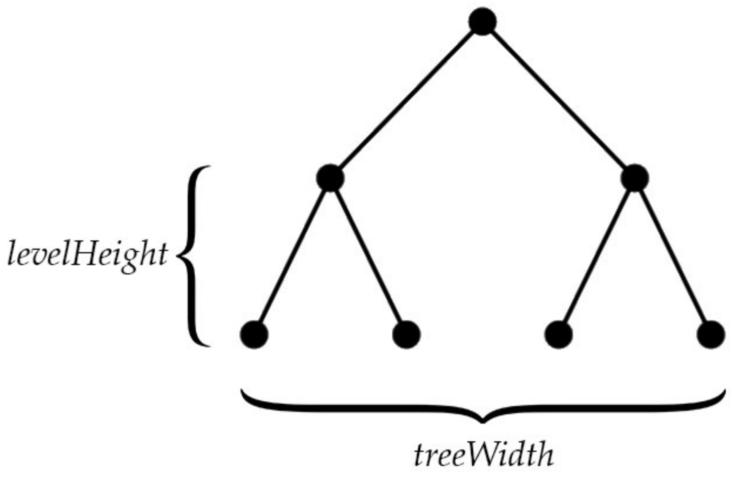


Figure 5.1: The *levelHeight* and *treeWidth* properties.

The first step of the algorithm is to position all nodes (of type *NodeView*) vertically, depending on their depth in the tree. This operation is shown in Snippet 5.2. Please note that, in Phaser and most graphical engines the $(0,0)$ coordinate is situated in the top-left corner.

```
private setYCoordinates() {
    // "this" is a reference to the TreeView class.
  this.nodes.forEach(nodeView => {
    nodeView.y = nodeView.level * this.properties.levelHeight;
  });
}
```

Snippet 5.2: Positioning all nodes vertically.

The second step is to get all leaves in a depth-first order and position them horizontally at equal distances from each other. To determine the distance between leaves we divide the *treeWidth* property by the number of leaves as shown on Snippet 5.3.

```
private updateLeavesPositions() {
    let leaves = this.tree.getLeaves();
    let widthPerNode = this.properties.treeWidth/leaves.length;
    for (let i = 0; i < leaves.length; i++) {
        // Find the corresponding nodeView from the leaf
        let nodeView = this.findNodeView(leaves[i]);
        nodeView.x = widthPerNode * i;
    }
}
```

Snippet 5.3: Positioning all leaves horizontally.

The third step places all parents horizontally in the middle between their left-most and right-most children. In order to properly position the parents, they need to be handled in a reverse breadth-first order as seen in Snippet 5.4.

```
private centerParents() {
    this.tree.BFSOnTree().reverse().forEach(node => {
        if (node.children.length !== 0) {
            let currentNV = this.findNodeView(node);
            let leftChildNV = this.findNodeView(node.children[0]);
            let rightChildNV = this.findNodeView(node.children[node.children.
                length - 1]);
            currentNV.x = (leftChildNV.x + rightChildNV.x) * 0.5;
        }
    });
}
```

Snippet 5.4: Placing parent nodes horizontally.

Once all nodes have been properly placed, the moves are drawn automatically by accessing the respective parent (*from*) and child (*to*) *nodeView* properties. Move and payoff labels are shown only if every node is either owned by a player or is a chance node. Finally, the whole tree is positioned in the centre of the screen.

We have further enhanced the GUI of the application by adding animations when the user modifies the structure of the tree. This functionality is implemented in a separate class called *TreeTweenManager*. It works by collecting a set of previously stored coordinates for every node. Once the new positions have been calculated, an animation is started which translates the nodes from their old positions to their new positions, instead of immediately repositioning them. In every frame of this animation, the positions of the corresponding moves and labels are also updated. The main reason for implementing this feature is because game trees can be drastically transformed from one state to another, especially when adding information sets. By having animations the transformation is gradually seen by the user.

One great feature of the Flash version of GTE, which is not currently present, is the automatic zooming when the tree gets larger than the screen. A workaround we have introduced is a simple check at the beginning of the drawing process, which reduces the height between levels of the tree. Possible ideas for implementing the zooming feature will be given in Chapter 6.

5.2 Information Sets

This section examines the challenges in the implementation of information sets both in the Model and the View.

In the *Model* folder the class which represents an information set is called *ISet.ts*. It contains properties for the player and a set of nodes. There are simple methods for adding a node, removing a node, and changing the player. The *ISetView* class extends *Phaser.Sprite*, similarly to other previously discussed classes in the View. *ISetView* contains an array with the corresponding nodes of type *NodeView*. A *bitmapdata* is used to draw the rounded shape of the information set by creating a solid, thick line with rounded ends, which passes through all nodes in the array (sorted from left to right).

When the user hovers over the shape of the information set, the actions menu appears. Input over shapes in Phaser is by default detected from the rectangular bounds of the corresponding sprite. In this case however, the shape may be quite large. For more accurate input detection we set two properties of the sprite, namely, *input.pixelPerfectClick* and *input.pixelPerfectOver* to true. Pixel-perfect detection in any engine is computationally intensive and should be avoided in general. However, there are no physics or complicated algorithms with regards to the shape, so currently we have not noticed any performance issues. Should there be any, multiple rectangular sprites could be created in order to detect the input.

Although the *ISet* and *ISetView* classes are relatively simple, handling information sets with regards to the tree structure poses a great number of challenges. Upon creation of an information set, we check whether all conditions (given in Chapter 2) are satisfied – all decision nodes in an information set must have the same number of children, must be owned by the same player, and no two nodes in an information set must share a path. If the user tries to create an information set in which some of these conditions are not satisfied, an error message appears. If the conditions are fulfilled and an information set is created, we reset the move labels for each node in the set.

Modifying the tree when information sets are involved requires a lot of cases to be checked and handled manually. If the move label of a node in an information set is changed by the user, the corresponding move labels of all nodes in the information set are given the same value. The addition and removal of children of nodes in an information set has to be done for all nodes in the set. If the user executes these operations with the hover menu, children are added to or removed from all nodes in the set. On the other hand, if the user selects only a single node from an information set and presses the keyboard shortcut *"n"* to add a child, it is not clear what should happen. We handle this, and the case where a child of a node from information is deleted, by dissolving the information set. All of these and more cases are handled carefully, both in the Model and the View. Since the user can operate on multiple nodes, at the end of each operation, we perform checks whether all information sets in the tree satisfy the conditions mentioned above. If not, we delete the information sets both from the *Tree* and *TreeView* classes and redraw the tree.

As mentioned in Chapter 3, there are is no algorithm in the HTML5 version for repositioning the nodes of a tree containing information sets. Instead we have implemented a feature that manually moves a selection of nodes. By pressing the arrow keys on the keyboard, the selected nodes can be moved at a distance. In the case of horizontal repositioning the distance is $\frac{1}{4}$ times the distance between leaves (as given in Snippet 5.3) and for vertical repositioning it is $\frac{1}{4}$ times the current *levelHeight* value. In most cases, these fixed distances allow for a quick alignment of nodes and information sets. For more precise repositioning, the user can hold *"ctrl"* and move the nodes at a distance of 1 pixel per frame.

Although manual repositioning takes some time and patience, the user can create and modify game trees visually as he/she sees fit. Figure 5.5 below shows the game tree in 3.5 (taken from [14]), where information sets are aligned and the number of intersections is minimised. We believe that this additional feature in the HTML5 version is a great improvement over the Flash version as the game tree below cannot be visually recreated in the latter.

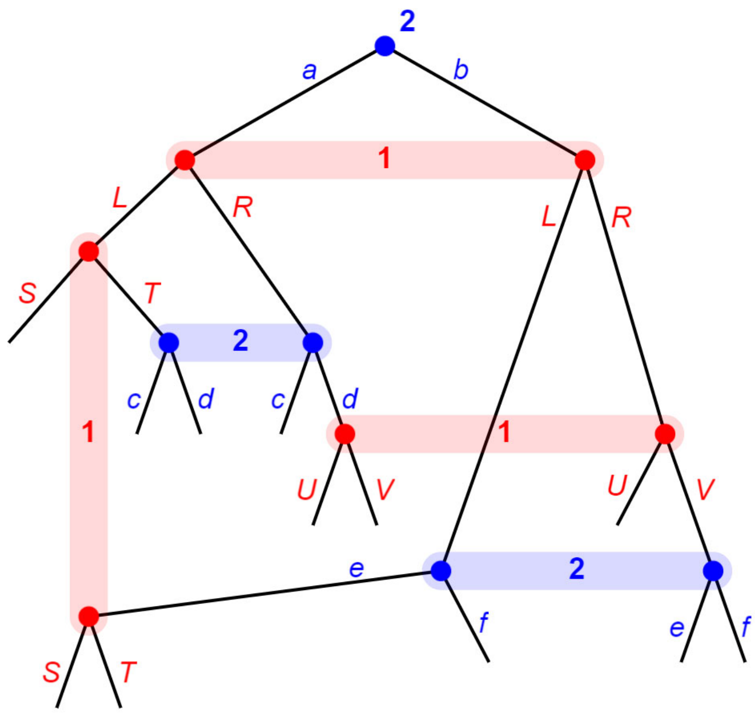


Figure 5.5: A game tree with manual repositioning of the nodes.

5.3 Chance Node Probabilities

An interesting problem occurs when the user changes the probabilities of a chance node, where the sum of all probabilities should always be equal to one. In the case where there are only two chance moves the solution is clear, but when there are more the problem of correctly assigning and resetting probabilities becomes more difficult.

Let us consider some examples. Suppose that there is a chance node with two moves. Initially, the probabilities are set to $[0.5, 0.5]$. When the user sets one of them to p the other should immediately change to $(1 - p)$. Suppose next that there are four chance moves and the corresponding probabilities are $[0.25, 0.25, 0.25, 0.25]$. It is not clear what should happen when the user decreases the first probability to 0.01. In this case the Flash version of GTE increases the last value and the new probabilities become $[0.01, 0.25, 0.25, 0.49]$. Next, if the user increases the first value to 0.75 the software starts decreasing probabilities from right to left until a total sum of one is obtained, so the resulting probabilities are $[0.75, 0.25, 0, 0]$. Finally, if the user sets a large probability for the last move the order for reducing is reversed and the new probabilities become $[0.1, 0, 0, 0.9]$.

In our project, a different approach is implemented, which we believe is slightly more intuitive. Instead of rotationally resetting probabilities, we take into consideration the exact position of the value that is modified. When the current probability is decreased all previous elements remain the same and all consecutive ones are given a value which is the average of the remainder. Starting with the same example with four equal values, suppose the second is changed to 0.15. The first value 0.25 remains unchanged and the last two become the average of the remainder 0.6, which gives the following set $[0.25, 0.15, 0.3, 0.3]$. Given the new probabilities, suppose we change the third to a large value, for example 0.8. In the Flash version, all probabilities but the first become 0. In our implementation the last becomes 0 and the remaining 0.2 is distributed equally between the previous elements, in this case to obtain the set $[0.1, 0.1, 0.8, 0]$.

The algorithm works in the following way. Denote the new probability as p. We calculate the sum s of all probabilities which are placed before p. We then check three different cases. If $p + s \leq 1$ we set all probabilities after p to be the average of the remainder $(1 - p - s)$. If $p + s > 1$, we set all probabilities after p to be zero and the ones before p become the average of the remainder $1 - p$. Finally, a border-line case needs to be handled when we set the last probability to be less than one, in which case we reset all previous probabilities as in the second case.

We should note that for two or three chance moves, or when the user correctly assigns probabilities from left to right, both algorithms work in the same way. Our modified algorithm may be useful in situations where the user would like to equalise probabilities without having to go through all of them. In the case of four values, changing the last to 0.25 makes all probabilities equal, independently of the current state of the set of values.

5.4 Strategic Form

The implementation of the strategic form of a game tree has been one of the most challenging tasks in the development of the software. In this section we examine the representation of the strategic form and the algorithms required to correctly calculate and visualise it.

The *StrategicForm* class in the *Model* contains a tree reference and properties for the strategies of player one, player two, and a payoffs matrix. For each player the strategies are represented as a two dimensional *jagged array* (the number of elements in each row can be different) of moves, where each row corresponds to a specific strategy.

In order to create a strategic form, we first check whether each player has perfect recall, namely, that each node in an information set has the same sequence of own earlier moves (as explained in Chapter 2). The algorithm for checking whether the game has perfect recall is fairly straightforward. For every node n in a given information set, we store the reachable nodes (that belong to the same player) on the path from the root to n. If any node in the reachability set has been reached via different moves, then the player has imperfect recall. We repeat this procedure for every information set in the game tree.

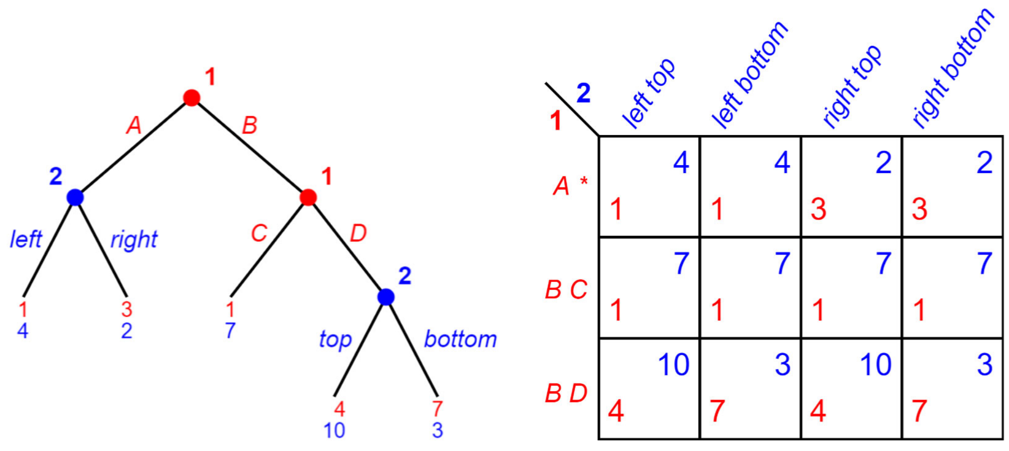
If the game has perfect recall, we proceed by calculating the reduced strategies for each player. The algorithm starts by collecting all information sets (including singletons) for each player from the tree in a breadth-first order. The order of information sets for a given player determines the order of moves, coming from an information set, in a strategy for that player. For example, in Figure 2.1 player I has two decision nodes, which determine two "slots" for moves at the respective decision nodes. The possible moves for the first slot are $\{X, Y, Z\}$ and $\{P, Q\}$ for the second slot. The algorithm then proceeds to generate all possible combinations of moves for each slot. If an information set is not reachable due to an own earlier move in the strategy, the corresponding slot is skipped and given a value of *null*, which is later replaced by a "$*$" in the string representation of each strategy.

Once the strategies for both players have been generated, the payoffs matrix needs to be calculated. The algorithm works in the following way. Initially all payoffs are initialised to be zero. For every leaf of the game tree we store for each player the sequence of moves which leads to that leaf. Each sequence determines a set of strategies for the corresponding player and both sequences together determine a rectangular subset of the payoffs matrix. If there are chance moves along the path to the leaf, we multiply the payoffs by the product of all probabilities on the path and the result is added to each cell in the rectangular subset.

As the implementation of these procedures is fairly long and complex we do not include them in this dissertation. The corresponding algorithms can be seen in the *StrategicForm.ts* file.

The *StrategicFormView* class is much simpler, as it only draws the information which is already calculated and provided by the *StrategicForm* class. The number of rows and columns are determined by the number of strategies for player one and player two respectively. For each cell in the table, a simple object of type *SCell* is created which contains a rectangle and two numbers taken from the payoffs matrix. Each strategy for player one is placed at the beginning of each row and for player two above each column. For player two if the string representations of a strategy is too long, we rotate it so that it fits above each column, as shown in Figure 5.6 below.

The strategic form is drawn on the same screen as the game tree. With this feature the user can interact with the tree and see the corresponding changes to the strategic form immediately. In case the strategic form becomes too large we have added options to scale it down (or up respectively) and to freely move it on the screen. An idea for future development is to implement a way of editing move labels from the strategic form and the corresponding changes to be made in the tree.



1 \ 2	left top	left bottom	right top	right bottom
A *	1, 4	1, 4	3, 2	3, 2
B C	1, 7	1, 7	1, 7	1, 7
B D	4, 10	7, 3	4, 10	7, 3

Figure 5.6: A game tree with long move labels for player 2 and the corresponding strategic form.

5.5 Label Editing

After a game tree is created and all nodes are either chance nodes or owned by a player, the labels for moves, probabilities and payoffs appear on the screen. Initially, chance move probabilities are equalised. Move labels are automatically set in a breadth first order, where labels for players 1 and 2 are upper and lower case letters, respectively, in alphabetical order. We also take into consideration information sets, as the move labels from nodes in an information set have to be the same. In this section we present the implementation of manually changing labels for moves and nodes.

Input Field

In the code all labels are instances of the *Phaser.Text* class and are similar to sprites with some additional properties for texts, such as font size, font weight, colour, alignment, and so on. As discussed in Section 4.3, when the user clicks on a label, a signal is sent to the *TreeController*. The controller has a property *labelInput* – a simple text field where the user can enter the new value of the label. Upon receiving the signal, the *TreeController* tells the *labelInput* to appear.

The development of this functionality seemed quite challenging at first due to the fact that Phaser does not have a built-in input field. In order to implement such a field in the engine, a separate class needs to written, which includes a rectangle for the field, a *Phaser.Text* instance for the text, another sprite for the blinking caret, and a great number of events for the selection, deletion, and modification of the text. An input field can be implemented in this manner, but it would be quite tedious and time-consuming. Instead, we make use of the fact that HTML tags and the corresponding CSS can be shown on top of Phaser. In this case, the *labelInput* property of the *TreeController* class is constructed with *vanilla* (pure) HTML, CSS, and the corresponding functionality is written in TypeScript, rather than implementing it in Phaser.

The *LabelInput* folder contains three files which comprise this functionality. In the *label-*

input.html file we create a tag which corresponds to the input field and an overlay background to indicate that the user is in input mode. Both are assigned IDs so that they can be styled to appear once the signal is sent from the corresponding label in the tree. The html file is given in Snippet 5.7.

```
<input type="text" id="input-label">
<div id="label-overlay"></div>
```

Snippet 5.7: The html file of the input field.

The *label-input.css* file modifies some properties of the input field and overlay by addressing their respective IDs. Initially we set the width, height, and size of both and give a black background colour for the overlay only. At first both elements are invisible. To make them visible we add classes *.show-label* and *.show-overlay*. We note that a *class* in CSS is different from a *class* in the context of object-oriented programming. CSS classes are primarily written in order to add and remove visual features of html tags. The css file is given in Snippet 5.8.

```
#input-label
{
    display:none;
    position:absolute;
    width:10%;
    height:3%;
    z-index:1001;
    left:30%;
    top:30%;
    font-size:1.5em;
}
```

```
#label-overlay{
    position:absolute;
    z-index:-1;
    background-color:
        black;
    opacity:0;
    width:100%;
    height:100%;
    transition: opacity
         200ms linear;
}
```

```
#input-label.show-label
    {
    display:block;
}

#label-overlay.show-
    overlay{
    opacity:0.3;
    z-index: 1000;
}
```

Snippet 5.8: The css file of the input field.

The *.ts* file contains the functionality of the input field and the overlay. When the class is instantiated, it appends both tags to the *index.html* file. A method *show()* is called (whenever the user clicks on a label), which appends the corresponding classes *.show-label* and *.show-overlay*. Whenever the input field becomes visible, it is repositioned close to the clicked label. A corresponding method *hide()* removes these classes to make the tags invisible.

We should note that both tags could have been added to the *index.html* directly, rather then appending them later. However, separating the logic into files, makes this functionality completely independent of the rest of the code and does not unnecessarily fill up the *index* file, which in turn increases the maintainability and extensibility of the code. Currently, a similar folder contains the logic for the top menu.

Changing Labels

The *LabelInput.ts* class contains a boolean property *active* which is set to true when the field appears on the screen and becomes false when the field is hidden. Another property in the class keeps a reference to the currently selected text. Both properties are accessed externally by the *Keyboard* and *UserAction* Controllers. When the user presses *"tab"* the field is moved to the next label (*"shift+tab"* moves to the previous, respectively). The next and previous labels depend on the currently selected label. If the label is a payoff, the order is depth-first, i.e. from left to right. In all other cases the order is breadth-first. By pressing *"Enter"* the current label is changed and the input field immediately moves to the next label, depending on the order. The user can exit the label editing mode by pressing *"esc"* or by clicking anywhere on the screen with the mouse.

The implementation of this functionality in the code works in the following way. The *UserActionController* has two methods, *changeLabel()* and *activateLabelField(next:boolean)*. The latter moves the field to the next or previous label, depending on the boolean parameter. When the user presses one of the keyboard shortcuts, the *KeyboardController* receives a signal and executes the corresponding methods from the *UserActionController* only if the input field is currently active. In Snippets 5.9 and 5.10 we present the algorithms for both methods only in pseudo-code, as the implementation in TypeScript is quite long. We would like to point out that changing the values of the labels involves computations which are performed in the Model, and the View is then reset with the new values of the labels. For example, if the current move is a chance move the algorithm presented in Section 5.3 is invoked. Furthermore, modifying a player label changes it globally for every node owned by the player.

```
activateLabelField(next:boolean)
1:  if (labelInput is currently active)
2:    reset BFS and DFS order
3:    if (currentlySelected is a Move label)
4:      find the next/previous move in the BFS order
5:      dispatch a click event on the next move label
6:    if (currentlySelected is a Node player label)
7:      find the next/previous node in the BFS order
8:      dispatch a click event on the next node label
9:    if (currentlySelected is a Node payoff label)
10:     find the next/previous node in the DFS order
11:     dispatch a click event on the next node label
```

Snippet 5.9: Activating the label fields preceding or following the current label field.

```
changeLabel()
1:  if (labelInput is currently active)
2:    if (currentlySelected is a Move label)
3:      update the move label with the new value
4:    if (currentlySelected is a Node payoff label)
5:      update the node label with the new value
6:    activateLabelField(true)
7:    save the tree in the UndoRedoController
```

Snippet 5.10: Changing the value of the current label.

The current implementation is an improvement over the Flash version. The process presented above is more automated, whereas in the Flash GTE the user has to click on a label, edit it, then click again on another label, and so on. By incorporating the *next* and *previous* functionality in the HTML5 version, we reduce the number of actions needed in order to modify the labels.

Chapter 6

Further Development and Concluding Remarks

In this final chapter we present ideas for future development (improvements new features) and we provide ideas for their implementation.

In Section 5.1 we mentioned the lack of a zooming functionality when the tree gets larger. In a vector-based engine, this feature could be implemented relatively quickly by making the objects larger or smaller without loss of quality. Phaser is a raster-based engine, as vectors are computationally expensive for game development where a high frame rate is always required. However, there are still ways to implement this feature. A relatively simple solution that can be attempted is to add all sprites of the tree into a group. A *Phaser.Group* object in Phaser is a container for visual objects (texts, sprites, bitmaps, etc.) and can be scaled down or up, depending on the width and height of the tree. Upon creation, the nodes and texts would have to be larger and scaled down. Scaling sprites (both up and down) in the engine usually makes them appear pixelated. There are options for fixing this problem which can be researched in several Phaser forums. We suggest another way of implementing the zooming functionality. In Section 3.3 we deliberately create the initial node circle as a bitmapdata and save it to the cache so that each *NodeView* instance can load it from a single place. A solution to the zooming problem could be to redraw the bitmapdata object and replace it in the cache, which in turn, would scale all node sprites on the screen. A method for re-caching the bitmapdata can be found in the *TreeController* class and is given in Snippet 6.1 below. We note that this operation is very similar to the creation of the circle as per Snippet 3.8.

```
setCircleBitmapData(scale: number) {
    this.bmd = this.game.make.bitmapData(this.game.height * NODE_RADIUS *
        scale, this.game.height * NODE_RADIUS * scale, "node-circle", true);
    this.bmd.ctx.fillStyle = "#ffffff";
    this.bmd.ctx.beginPath();
    this.bmd.ctx.arc(this.bmd.width / 2, this.bmd.width / 2,
    this.bmd.width * 0.45, 0, Math.PI * 2);
    this.bmd.ctx.fill();
}
```

Snippet 6.1: Re-caching the circle bitmapdata.

Implementing the zooming functionality with this method may require some work in the *TreeView.ts* class, but eventually could work.

Extending the settings menu is another important aspect of the software that should be considered. New modifiable features can be added to the *TreeViewProperties* class. When the user changes any of the settings, the tree can be redrawn in real time as it is implemented currently with the *levelHeight* and *treeWidth* properties (instead of pressing an *"apply"* button, for example). These settings can be saved to the local storage of the browser so that the user does not have to apply them every time when he/she visits the website. Alternatively, a backend service could be created where users can login and all settings and previously stored trees can be accessed. Storing trees on the server could be quickly implemented by utilising the JSON representation of the tree, as discussed in Section 4.4.

Another interesting new feature could be to draw the strategic form in a separate window and synchronise it in real time with the tree editing. This could be implemented again on a backend by creating a websocket to transfer data between the tree and strategic form.

One major idea behind GTE is to incorporate it with existing solvers. This may require the implementation of a new format for serialisation (e.g. XML), depending on the solver. As very large trees cannot be drawn in the browser, a possibility is to develop a console within the web version of GTE, where the user can create larger trees with simple commands. A simplified version of the tree can then presented as an image or a screenshot without any options for editing, for example. This way, the user can send larger trees to the solver, without worrying about the computing limitations of the browser.

In terms of completely new features, a separate website of GTE can be created, which would include the software, a user system, and a detailed documentation. By having such a website, GTE would become more accessible for developers and the end user.

The main objective of this dissertation was to introduce the ideas behind a modern implementation of the *Game Theory Explorer* software tool. During the development process we closely followed the overall framework and functionality provided by the Flash version of GTE, whilst also incorporating new features for better user experience, interface and usability. Due to the chosen tools and architecture, we strongly believe that the current HTML5 version of GTE provides a solid base for future development and growth of the project.

Bibliography

[1] Game Theory Explorer. `www.gametheoryexplorer.org`. Accessed on 17 August 2017.

[2] R. Davey. Phaser Game Engine. `https://phaser.io/`. Accessed on 17 August 2017.

[3] Engadget. "Adobe is ending development and support for Flash in 2020". `https://www.engadget.com/2017/07/25/adobe-ending-flash/`. Accessed on 3 August 2017.

[4] C. Evans, Ingy döt Net, and Oren Ben Kiki. YAML - Ain't Markup Language. `http://www.yaml.org/`. Accessed on 17 August 2017.

[5] D. Flanagan. *JavaScript : The Definitive Guide : Activate Your Web Pages*. O'Reilly, Beijing Sebastopol, CA, 2011.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[7] Google. Angular 4 – One framework. Mobile & desktop. `https://angular.io/`. Accessed on 17 August 2017.

[8] JetBrains. WebStorm: The Smartest JavaScript IDE. `https://www.jetbrains.com/webstorm/`. Accessed on 17 August 2017.

[9] Microsoft. TypeScript - JavaScript that scales. `https://www.typescriptlang.org/`. Accessed on 17 August 2017.

[10] Microsoft. Visual Studio Code – Code Editing. Redefined. `https://code.visualstudio.com/`. Accessed on 17 August 2017.

[11] J. Resig. jQuery. `https://jquery.com/`. Accessed on 17 August 2017.

[12] R. Savani and B. von Stengel. Game Theory Explorer: software for the applied game theorist. *Computational Management Science*, 12(1):5–13, 2015. `http://www.gametheoryexplorer.org`.

[13] B. von Stengel. Game Theory Basics - LSE Lecture Notes. pages 38–43,112–113, 2016. version from 8 September 2016.

[14] B. von Stengel and F. Forges. Extensive-form correlated equilibrium: Definition and computational complexity. *Mathematics of Operations Research*, 33(4):1012, 2008.

[15] Wikipedia. SOLID Principles. `https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)`. Accessed on 17 August 2017.