Dynamic Difficulty Adjustment for Imperfect Information Games using Monte-Carlo Tree Search

CS407 Group Project Final Report

Sam James, Sam Medwell, Lukas Rutkauskas, Katie Yang

Supervisor: Professor Till Bretschneider

Year of Study: 4

April 30, 2024

WARWICK
THE UNIVERSITY OF WARWICK

Abstract

In games, player enjoyment is typically linked to the balance between challenge and skill. Dynamic Difficulty Adjustment (DDA) addresses this by artificially matching the challenge of the game to the skill of the player. The Monte-Carlo tree search (MCTS) algorithm has seen much research and application to perfect information strategy games, particularly board games. In this report, we design and implement DDA agents based upon MCTS designed to play Ticket to Ride, an imperfect information board game, as well as develop a game application for users to play against them. We use the Upper Confidence Bound for Trees algorithm to implement our MCTS agent, and implement two variations: Cheating MCTS (CMCTS) and Information Set MCTS (ISMCTS). Furthermore, we investigate two modifications to each agent to implement DDA: Reactive Outcome-Sensitive Action Selection and Proactive Outcome-Sensitive Action Selection. We propose a further modification known as momentum that aims to smooth out difficulty change. We test the agents and game application using both heuristic agents and user acceptance surveys, and found that the DDA agent was able to adjust to heuristic agents of varying difficulty. Testers found playing against the agents engaging and enjoyable, as well as finding the game environment pleasant to use. Our work demonstrates the possibility of using the MCTS algorithm as a basis for DDA agents, evaluates approaches for determinization of imperfect information games, and provides insights into the modifications that can be made to increase its performance.

Keywords Monte-Carlo Tree Search, Dynamic Difficulty Adjustment, Ticket to Ride, Imperfect Information Games, Upper Confidence Bound for Trees

Acknowledgements

We wish to thank Prof. Till Bretschneider, for his guidance and support throughout the project, including his insights into both technical and general improvements to the project, as well as efforts spent communicating with clients during project formation.

In addition, we wish to thank our user testers for their valuable time and feedback.

We would like to thank the contributions of Kristijonas Kojelis whose significant effort in leading and bringing together the team during its formation are appreciated despite not being able to continue with the project, and the contributions of Varun Chodanker.

Finally, we would like to thank the colleagues, family, and friends of the team members who provided invaluable support throughout the project.

Contents

Acronyms

C4T	Coupon for Travel.
CMCTS	Cheating Monte-Carlo Tree Search.
DDA	Dynamic Difficulty Adjustment.
GUI	Graphical User Interface.
ISMCTS	Information-Set Monte-Carlo Tree Search.
MCTS	Monte-Carlo Tree Search.
POSAS	Proactive Outcome-Sensitive Action Selection.
PRNG	Pseudo-Random Number Generator.
ROSAS	Reactive Outcome-Sensitive Action Selection.
SFML	Simple and Fast Multimedia Library.
TRNG	True Random Number Generator.
TTR	Ticket to Ride.
UCB	Upper Confidence Bound.
UCT	Upper Confidence Bound for Trees.

Chapter 1

Introduction

From the invention of chess more than 1500 years ago to the rising in popularity of VR gaming in the modern day, games have been a prominent part of human culture and history. Today, video and board games alike provide entertainment to millions of players across the globe. There are a myriad of reasons why games have captivated us throughout history, such as their social dimension, competitiveness and sense of progression. But a significant factor would be attributed to Csíkszentmihályi's Flow theory, which describes the flow state, a mental state in which a person is able to feel fully immersed in an activity with energised focus [1]. This can be induced by finding a balance between skill and challenge, whereby the player feels neither bored with the activity nor anxious from its difficulty. While the design of a game is the crucial factor for striking this balance, Dynamic Difficulty Adjustment (DDA) tackles this from a programmatic angle by changing parameters in a game in real-time to match the player's skill level.

The difficulty in competitive games is directly linked to the strength of the opponent's play. Board games, which typically require skill in strategising, lend themselves to the application of algorithms without the use of machine learning much more easily than games of hand-eye coordination or social deduction. The Monte-Carlo tree search algorithm is a heuristic search algorithm which has demonstrated promising results for such games, previously applied in chess, Go, as well as turn-based strategy video games. MCTS allows very strong agents to be produced with little to no domain knowledge, and a number of modifications can be made to adapt it to the task of DDA.

Significant work has been already done investigating DDA in *perfect information* games, that is games whereby players have access to all game state information, for example chess. However, there is less work focusing on the *imperfect information* setting. We would like

to investigate how well DDA techniques work in board games without access to perfect information. The modern, popular and critically acclaimed game Ticket to Ride combines strategic planning, adversarial interactions, imperfect information, resource management as well as an element of luck, which combines to create an interesting environment for developing DDA AI agents.

1.1 Problem Statement

Our project's goal is to create a Dynamic Difficulty Adjustment AI agent that can play in a game of imperfect information. The agent is based on Monte-Carlo tree search, a popular heuristic search algorithm that has been shown to perform well on board games, which we modify to achieve difficulty adjustment to the opponent. We use the board game Ticket to Ride as a test bed for the model, owing to its balance in complexity and ubiquity, and limit the agent to two-player games to keep player interaction simple.

1.2 Report Structure

We will begin the report by detailing required background information regarding central topics of the project. Then, we propose the objectives and deliverables we aim to fulfil, and discuss relevant existing literature. We address the legal, social, ethical considerations of the project, our approach to project management, as well as detail the tools used and the rationale behind our technology stack. In the *AI Agent* chapter we will detail the exact outline for the agents we aim to implement and their behaviour. In the *System Design* chapter we will discuss the design considerations and describe the architecture of the user and testing applications. Following on from these section, the Implementation section contains a detailed overview of the technical implementation of each project component, taking into consideration performance and reproducibility. At this point, thorough testing is conducted, and we describe both our testing methods and summarise results. Finally, an evaluation of the project against its original objectives is carried out based upon outcomes achieved by our finished products, and we conclude the project, summarise its outcomes, and make recommendations for any future work.

Chapter 2

Background

This chapter introduces the reader to the background required to understand the problem setting (imperfect information games and Ticket to Ride), the algorithm we base our solution on (Monte-Carlo Tree Search), and our goal (Dynamic Difficulty Adjustment).

2.1 Dynamic Difficulty Adjustment

Dynamic Difficulty Adjustment (DDA) is a mechanism employed in various games that adjusts the level of challenge in a game in real-time to match the player's skill level [2]. The majority of the existing work relating to DDA is focused on video games, where the game environment itself can be manipulated by subtly altering rules and mechanics behind the scenes. For example, in the racing game Mario Kart, players who are losing are often provided with power-ups not available to players that are leading the race, in order to level the playing field and provide the players with a chance to catch up. However, this approach to DDA may be unsatisfying to human players, as these artificial changes can be perceived as unfair advantages [2].

Dynamic Difficulty Adjustment in board games presents a different set of challenges. In board games, the AI operates as if it is another player, constrained by the same rules and structures as human players. Therefore, implementing DDA in board games requires adapting the AI's decision-making process rather than modifying the game environment.

2.2 Imperfect Information Games

In perfect information games, the full state of the game is visible to the players and there is no uncertainty on the current state of the game. On the other hand, imperfect information games are games with hidden information to one or many players, that are not revealed at the beginning of the game (or possibly at all). Most board games have imperfect information - Scrabble and Battleship to name a few.

Crucial to the understanding of uncertainty faced by players in imperfect information games are information sets [3]. An information set comprises all possible game states that are indistinguishable to a player based on the information available to them at a particular point in the game. In perfect information games, the information set consists of a single member, representing the current state of the game, since each player has identical information. Conversely, imperfect information games may have a multitude of game states within their information sets, reflecting the player's incomplete knowledge regarding the state of the game. This adds complexity to players' strategies, as it is necessary to make decisions while in a game state with varying degrees of ambiguity.

2.3 Ticket to Ride

Ticket to Ride (TTR) is a popular modern board game in which players battle it out to build the most successful railway network spanning a specific region of the world, from the USA to Europe to India. The game was designed by Alan R. Moon with its first edition released in 2004 to great success, winning the highly regarded Spiel des Jahres prize [4].

Ticket to Ride combines strategic planning, adversarial interactions, hidden and visible information, resource management, as well as an element of luck. This creates an interesting environment for developing an AI agent, and though many have attempted to implement high-performing agents for playing TTR and similar modern board games, few have explored the possibility of a dynamic difficulty agent.

2.3.1 Rules

Ticket to Ride can be played by between 2 and up to 5 players, depending on the edition.

As displayed in Figure 2.1, the game board represents a section of the world covering regions such as Europe, North America or Asia, and comprises a set of cities and possible

connections between these cities. Players interact with train pieces, train cards and ticket cards. Train pieces are used to claim a connection for the given player, train cards are resources that must be collected in order to claim a connection and ticket cards specify a route between two cities (which do not have to be neighbouring cities).

At the beginning of the game, players are dealt destination ticket cards, which they aim to complete by the end of the game (and should be kept secret from other players). Completing a destination ticket grants points, but failure to complete it deducts points.

In each turn, a player can either (i) draw train cards, (ii) claim a route, or (iii) draw additional destination tickets.

i Train cards are drawn from two piles, either a blind deck or from a set of 5 visible cards that everyone can view.

ii In order to claim a route, a player must play a set of train cards that are of the same colour and quantity as the spaces marked by the connection on the board. For example, a connection that requires four pink train cards must be claimed by

Figure 2.1: Ticket to Ride Europe Board. [CREDIT: Days of Wonder]

playing four pink cards from the player's hand. Once claimed, the player places their train pieces onto the corresponding route, that shows the player's ownership of that route.

iii A player can draw up to 3 additional ticket cards, but must keep at least one of these routes (they may discard the remaining routes if they should like).

The game continues until at least one player has 2 or fewer train pieces remaining, from which every player has one final turn. The player with the most points wins the game.[1]

The game is scored as follows:

- Players earn points for each connection they claim (longer connections yield proportionally higher score)
- Players earn points for every destination ticket they complete (and lose the same amount of points for those they do not).
- In non-small maps, additional points are provided to the player with the longest continuous path on the board.

2.3.2 Game Variants

Ticket to Ride has several versions, which cover different regions of the world. The game's second designed map, TTR Europe, is more balanced [5] when compared with the first edition, which covers North America. However, there also exist smaller maps such as New York or London, as well as medium size maps such as TTR Switzerland, in addition to the larger size original maps.

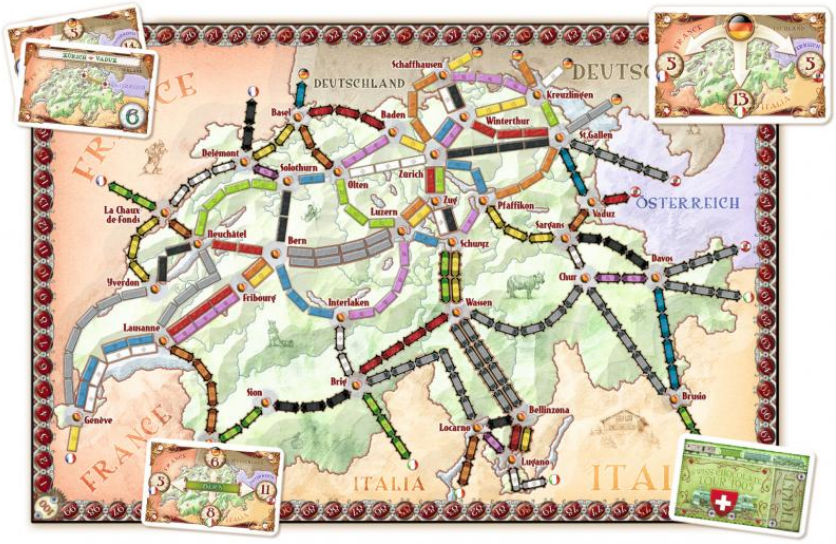
Different game editions may have additional rules. For instance, TTR Europe has different connection types — tunnels and ferries — which require different train card conditions, as well as train stations which can be used to unblock routes. Furthermore, TTR New York has a feature whereby players may score one additional point for every tourist attraction that is connected to it[2].

[1]The full rules of Ticket to Ride can be read at the ticket to ride website. The official rule book for the Ticket to Ride Europe edition can be found at https://ncdn0.daysofwonder.com/tickettoride/en/img/t2re15-rules-EN.pdf.

[2]Read the rules at https://cdn.1j1ju.com/medias/90/98/2f-ticket-to-ride-new-york-rulebook.pdf.

(a) Ticket to Ride: New York

(b) Ticket to Ride: London



(c) Ticket to Ride: Switzerland

Figure 2.2: A selection of Ticket to Ride game variants. [CREDIT: Days of Wonder]

2.3.3 Strategies

This section describes some strategies commonly used by players in Ticket to Ride, and explains the main rationale behind them.

2.3.3.1 Greedy Route Claim

Perhaps the most straightforward greedy strategy for TTR is greedy route claim. This strategy entails always prioritising claiming routes, drawing necessary train cards to do so. As only one player can claim a route and routes always give points, claiming more routes never results in a loss, and considering the sabotage factor for the other player,

this is a simple but often effective strategy.

Whilst a player may simply choose any route they are able to claim given their hand, a better strategy would be to take into account their tickets and the currently claimed routes. More advanced players can factor in the cards visible on the table, as well as their perceived strategy of each opponent(s).

2.3.3.2 Hoarding Train Cards

Another strategy is to draw as many train cards as possible before claiming routes. As train cards are the resources required to claim routes, a player who has a large amount of cards will have a many routes available for them to claim at any point. This strategy sometimes has an edge, because in contrast to needing to take turns to draw cards before claiming a route, the hoarder player is able to quickly react to changes in board state in the mid-to-late game. The main downside to this strategy is the fact that it lets other players have first choice over claiming routes, and possibly helping other players by constantly "refreshing" the train cards on the table.

2.3.3.3 Sabotage

The last distinctive strategy is that of sabotage. A player playing a sabotage based strategy might prioritise claiming routes that are beneficial for other players. For this to be effective, the player has to be able to make an accurate assessment on which route is most wanted by their opponents. This is often incorporated by players in their overall strategies, but not usually as a standalone tactic, as the main way to score in TTR is by building your own routes according to the tickets in hand.

2.3.4 Complexity

The state space of TTR is expansive, with the original edition (American Map) having more than 10^{54} possible states — this even surpasses the state space of chess [6]. This large state space stems from all the possible configurations of resources the game board can have throughout gameplay, such as ownership of routes and availability of train cards. The vast state space itself does not inherently imply high complexity. However, the multitude of possible configurations contribute to a high branching factor in the game tree for TTR. This high branching factor significantly increases computational costs for Monte-Carlo methods and other search algorithms.

The Ticket to Ride board can be considered as a coloured weighted graph $G = (V, E)$, where cities are represented as vertices V, railroad routes between the cities are rep-

resented as edges E, the costs associated with routes represented in a weight function $w : E \longrightarrow \mathbb{N}$ and colouring as a function of edges $c : E \longrightarrow 2^{\mathcal{C}}$. Solving problems such as Multiple Pairs Shortest Path, which is needed to calculate the optimal set of routes to complete all destination tickets, is hard [7], [8] and this complexity is further increased when colouring the different valid combinations of train cards that can claim each route are taken into consideration.

Moreover, Ticket to Ride is an imperfect information game, as discussed in Section 2.2. Players have to formulate their strategies with hidden information, regarding opponents' destination tickets and train card draws. Navigating this imperfect information successfully requires nuanced decision-making and adaptive strategies, rendering the development of agents to play this game more intricate than in perfect information games.

2.4 Monte Carlo Tree Search

The Monte-Carlo Tree Search (MCTS) method was first described by Coulom as a combination of tree search and Monte Carlo evaluation in his paper *Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search* [9], where he coined the term and applied the resulting algorithm to a Go program, which won the 10th KGS computer-Go tournament. The MCTS method has since been a rather popular topic of research in the AI field, as it shows a number of promising features as a strategising agent, particularly in board games: it requires little to no domain knowledge to implement, and its performance generally increases with the computational time given.

The core concept behind MCTS is quite straightforward. The algorithm starts from a root node, typically representing the current game state, and incrementally builds a game tree by recursively applying a selection strategy to choose a possible unexplored action, which it expands by simulating the rest of the game from that action, and returning the result. This process is repeated until a pre-defined termination point, and the best next action is selected. More concretely, one iteration of the MCTS algorithm is done in four steps: selection, expansion, simulation, and backpropagation [10].

1. Selection. Starting from the root node, apply the chosen selection policy to child nodes recursively until the first expandable node is reached. A node is expandable if it has unexplored children, and is not a terminating state (i.e. game end).

2. Expansion. From the expandable node, one or more child nodes are added, which represent the new game states upon taking one or more action from the parent node.

3. Simulation. Also known as playout, the game is simulated from the new node(s) according to the default policy until a terminating state is reached, usually by selecting the remainder of moves at random. This will produce an outcome, such as the loss of one player and/or the victory of another.

4. Backpropagation. The outcome of the simulated game is backpropagated up the tree, and each node in its path is updated in the process. The statistics for each node will contribute to their scoring for the selection policy.

This process is shown in Figure 2.3 below. Here, each node represents a state in the game, linked to its children by possible actions.

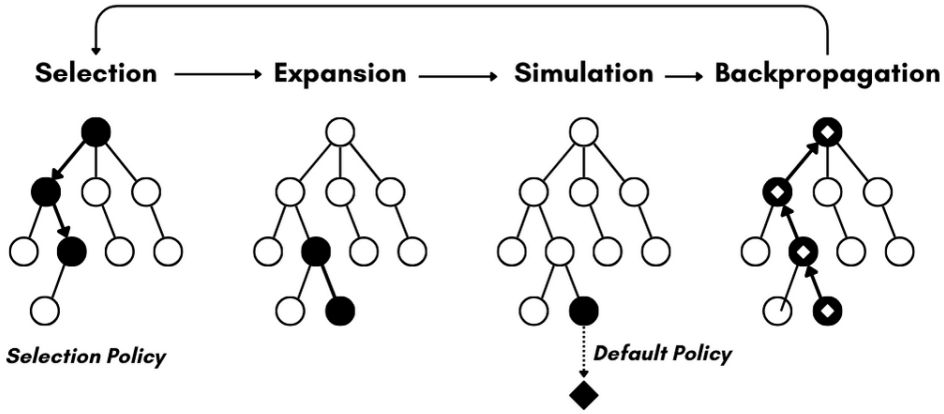


Figure 2.3: One iteration of the MCTS algorithm.

The MCTS method makes the assumption that a best-first strategy works well for the problem [10]. In the context of Ticket to Ride, this seems a reasonable assumption, as it is almost always beneficial to claim the most useful route as early as possible, or at least get closer to claiming it.

2.4.1 Bandit Problems

In a bandit problem, one is presented with K actions and has the goal of maximising one's cumulative reward from sequential selection of the actions [10]. The challenge arises from the fact that the underlying reward distribution of each action is unknown and can only be estimated from past observations. An agent must balance between exploring actions to gain greater certainty of their reward distribution, versus exploiting the action that currently has the best. The regret of a policy for selecting actions is the expected loss of not selecting the best action each time, and it has been shown that no policy exists where the regret grows slower than $\mathcal{O}(\ln n)$ where n is the number of plays. Each node in the MCTS search tree can be viewed as an individual bandit problem, whereby the

exploitation vs exploration of game actions is balanced in order to find the more optimal of the two.

2.4.2 Upper Confidence Bounds for Trees

Upper Confidence Bound for Trees (UCT) builds on the basic MCTS algorithm by providing a way to balance exploration and exploitation of child nodes. When performing an iteration of MCTS, the choice of which child nodes to expand is important: we could either try to cover a large amount of possible state spaces by prioritising nodes with more unexplored children, or try to achieve a better outcome by exploiting more promising nodes with higher scores. UCT proposes a solution to this using the Upper Confidence Bound (UCB) policy for bandit problems [10]. In the UCT algorithm, we select the child node j that maximises

$$UCT = \bar{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

where $\bar{X}_j$ is the average payoff at j, n is the number of times the parent of j has been visited across the selection phases, n_j is the number of times j has been visited, and C_p is a predetermined exploration constant.

In the UCT equation, the first term represents the score of child j, and the second term represents the exploration factor - the more j is explored, the greater the value of n_j, and thus the term decreases; the higher the number of plays without exploring j, the greater the value of n while the value of n_j remains unchanged, and hence the term's overall contribution increases. This is exactly the aforementioned balance between exploration and exploitation. It is understood that when $n_j = 0$, $UCT = \infty$, which means the algorithm will prioritise any child nodes that are completely unexplored. The exploration constant, C_p, has been observed to vary between domains and sometimes provide optimal solutions when equal to zero, and efforts have been put in to tune this constant both experimentally and automatically [10].

In a paper by Kocsis and Szepesvári, they were able to show that as the number of simulation grows to infinity, the result produced by the UCT algorithm converges to the best action, hence allowing MCTS to converge to a minimax tree. This means that given enough computation time, the UCT algorithm is optimal [11].

2.4.3 Cheating MCTS

The MCTS algorithm described can only deal with perfect information games. Games of imperfect information create further challenge, since an agent may be in one of many possible game states given a history of actions. The simplest approach to tackle these games is to allow MCTS access to the full game state and play as if the game was perfect information. Cowling et al. [12] in their work on Imperfect Information Monte Carlo Tree Search describes this approach as invalid but useful for benchmarking. In another work by Kelly et al. [13] which compares imperfect information MCTS algorithms in the game of Cribbage, they also use cheating MCTS as a benchmark to test the other algorithms. Unsurprisingly, they both found that Cheating Monte-Carlo Tree Search (CMCTS) outperforms other MCTS algorithms that do not have access to hidden information. We believe that CMCTS is a valid approach to the problem of creating a DDA agent since the aim is to create an interesting experience for the player as opposed to creating an agent for the purposes of competition or for applications where the hidden information is simply not available.

2.4.4 Determinization

A simple method to apply MCTS to games of imperfect information without cheating is an approach known as **Determinization or Perfect MCTS**. It works by sampling some game states from the current information set and creating separate search trees for each state, applying the algorithm separately. At the end the results are combined, for example by summing the total number of visits of each action over all the trees and selecting the one that was visited the most [12].

There are a few issues with this approach highlighted by Cowling et al. regarding certain assumptions that are made [12]. The first of these is known as strategy fusion in which the AI assumes it can make different decisions in game states contained in the same information set, which occurs because the different determinizations are considered in separate trees.

The second issue that is pointed out is known as non-locality is where game states in an information set are assumed to be equally likely, which is not necessarily the case. For example, if a player has claimed a route on the board, they are more likely to have tickets in their hand which require that route for successful completion. Modifications to address this issue aim to infer the hidden information based on opponent play, but they are not explored here.

2.4.5 Single-Observer Information Set MCTS

Single-Observer ISMCTS is a proposed modification to MCTS which aims to reduce the effects of strategy fusion [12]. It does this by constraining the search to a single tree, in which nodes represent information sets instead of game states; each node has children corresponding to actions legal in any state contained within the information set. On each iteration of the algorithm, a game state is selected from the information set uniformly at random, known as a determinization, and the algorithm proceeds as usual using legal actions consistent with the selected determinization. Therefore, only a subset of possible actions are considered during each iteration.

2.4.6 Other Modifications to Information Set MCTS

Further modifications can be made to further eliminate assumptions made by the agent. **Single-Observer ISMCTS With Partially Observable Moves** combines actions indistinguishable from the perspective of the agent into single actions in the search tree. For example, each combination of keeping two tickets when the opponent draws new tickets are treated as the same action [12]. In the case of Ticket to Ride, it is unlikely that this approach would reduce the effects of strategy fusion because even if the agent knows the combination of tickets kept, like it could in a real life game, this does not reveal any information about the tickets themselves. However, it would combine the computation for these actions, which could make the evaluation of them more accurate.

The previous algorithm solves an aspect of strategy fusion at the expense of assuming the opponent selects actions indistinguishable to the agent uniformly at random. **Multiple-Observer ISMCTS** removes that assumption by creating a separate tree which is descended in parallel, used to select actions for the opponent [12].

Chapter 3

Objectives

Based on our problem statement, we aim to develop a product that achieves the following objectives. Though our methodology and approaches have changed throughout the course of the project, the overall objectives remained the same.

1. Build upon existing research on Dynamic Difficulty Adjustment (DDA) to implement and analyse AI models for two-player Ticket to Ride, an imperfect information board game.

    (a) The AI should offer a balanced and appropriate level of challenge to players of any skill level, facilitating the player's improvement in the game.

    (b) The AI should consistently achieve a provided target win rate against opponent players.

    (c) Players should find the AI enjoyable to play against, based upon their assessment of key product indicators such as the AI's turn-time, "human-likeness" and challenge.

2. Develop an application that allows a player to play Ticket to Ride against the developed AI.

    (a) The user interface should be accessible, intuitive, and enjoyable to interact with.

    (b) The application should be able to be easy to install across several platforms.

Chapter 4

Deliverables

In this section, we outline the deliverables for the project. This provides a clear definition of the components that will be implemented, and bounds the project scope by specifying concrete, measurable products that fulfil the project objectives.

1. Develop the *game environment*: the core game system which encapsulates all the game mechanics of Ticket to Ride and maintains the game state.

   (a) Define an interface that allows communication between the game environment, game application front end and AI/user agents.

   (b) Build the core game system that implements Ticket to Ride game rules.

   (c) Build a console user interface that can affect the game environment, for internal testing and verification purposes.

2. Develop an AI agent to play two-player Ticket to Ride which dynamically alters its skill level to match the opponent.

   (a) Produce a report assessing the efficacy of various approaches to developing a DDA AI based upon the results of implementations for each approach.

   (b) Complete the back-end system of the game application by integrating the agent with the game environment.

3. Develop the front-end of the game application.

(a) Design and implement a graphical user interface, which dynamically displays the game state, as well as providing the player with ways to interact with the game environment.

(b) Implement additional game elements including a game menu, settings, and media such as art and sound.

4.1 Evolution of Deliverables

As the project evolved, we were able to keep to the original deliverables, as they turned out to be well-informed and suitable for purpose, as well as producing an appropriate workload for the team. Two extensions were initially planned for the project, as can be found in the Specification included in the Appendix of this report, but we found these drew focus away from our core product. Preferring increasing project depth over breadth, the team decided to focus on polishing the core deliverables as we felt they already combined into a substantial product with good results. To this end, we were able to produce a number of features that built and improved on the original deliverables, such as multiple DDA agents as well as heuristic agents for testing purposes. Furthermore, we were able to explore modifications to our DDA algorithm to improve its efficacy. These changes resulted in three additional sub-deliverables for the project, as listed below.

1. (d) Build a game simulation tool that can run AI agents against each other across multiple games in our game environment and record data for analytical purposes.

2. (c) Design and implement heuristic agents of varying strategy and difficulty for DDA efficacy testing and play.

   (d) Produce variations of the base MCTS DDA algorithm and carry out an assessment on their effects on agent strength and player enjoyment.

Chapter 5

Existing Literature

In the following chapter we conduct an in-depth review of relevant existing work. We look at AI methods for playing games of imperfect information, as well as those specifically used for Ticket to Ride. Then we explore approaches that have been used for introducing DDA into board games and video games. We focus on methods to integrate DDA into MCTS which forms the primary focus of the project.

5.1 AI for Imperfect Information Games

In the work *Reinforcement Learning Agents Playing Ticket to Ride–A Complex Imperfect Information Board Game With Delayed Rewards* [14] by Yang et al., the authors begin with a survey of methods for playing complex imperfect information games, which provided a great starting point for further research. They explain that state-of-the-art algorithms have achieved super-human performance in imperfect information games with smaller game tree complexities such as Poker using Counterfactual Regret Minimisation [15], but for more complex games like Ticket to Ride it remains an open problem in AI. To achieve this performance in more complex games, human data, game specific heuristics and or parameter tuning have been required in previous works [14].

The first approach discussed by Yang et al. is the use of heuristic rules and evaluation functions in order to create a rule-based agent. These are agents that implement predefined strategies developed using human knowledge, and some examples are given, including agents developed for Ticket to Ride. While this approach has the potential to be less computationally expensive compared to algorithms such as MCTS depending on the rules used, they do suffer from some drawbacks. This approach is inflexible

since its behaviour is hard-coded and does not to apply to other games with different rules. Moreover, the resulting gameplay is usually predictable for this reason, making the agent exploitable and potentially providing an uninteresting experience for a player. However, these rules can be integrated into other algorithms to boost their performance, and heuristic agents can be used for bench marking [14], which we will discuss in more detail in later sections.

The second class of algorithms discussed are tree-search algorithms, which include MCTS. These methods work by searching through possible game trajectories from a given starting state in order to find optimal moves. For small games it is possible to exhaustively search the whole game tree with an algorithm like minimax, but for games with large game trees, including Chess and Ticket to Ride, this is not feasible due to the large state-space complexity [14].

Finally, they go on to discuss reinforcement learning methods such as Temporal Difference Learning used to play Backgammon [16]. The general approach is to model the problem as a Markov Decision Process, which is composed of a set of states and transitions between them. Through self play, a reinforcement learning agent attempts to reach an optimal value function or policy that allows them to select the best actions in each state. Alpha Zero, which achieves super-human performance in Go and Chess, is an example where both a value function and policy are learnt using neural networks through the use of MCTS which guides the training process [14].

5.2 AI for Ticket to Ride

In this section we discuss the previous work on creating AI agents to play the game Ticket to Ride. The first of these works is that by Strömberg et al. [17] in which the Double Deep Q-learning reinforcement learning algorithm (DQN) is implemented. The game is represented as a Markov Decision Process, and neural networks are trained through self-play to learn a function that maps values to actions in different game states. However, they decided to modify the rules of the game in order to reduce game-tree complexity, making the game easier to approach. Instead of being able to draw from both visible table cards and the draw deck, a player can only draw two blindly from the deck. The player also gets no choice in which ticket they draw. They collect results comparing the performance of four DQN bots which use different reward functions, but they do not give any results involving human games, making it hard to gauge the strength of the produced agents.

The work by Huchler compared the performance of various MCTS based algorithms including Single Observer ISMCTS, Multi Observer ISMCTS and Determinization [6]. They even look into the performance of Flat Monte-Carlo which is a simpler algorithm that carries out simulations without building a search tree. A selection of modifications is investigated in order to improve performance, including progressive bias and unpruining, which uses heuristic knowledge to bias the selection of nodes during the selection phase. Our work focuses on the performance of DDA modifications, but this could be readily applied to increase the maximum strength of the MCTS agents.

The work by Yang et al. [14] discusses different approaches for creating agents for imperfect information games. They start by describing the challenge in creating AI to play Ticket to Ride stating that previous work, including the last two, either do not achieve human-level gameplay or use many heuristic rules. The approach they focus on is Proximal Policy Optimization (PPO), which is a general-purpose reinforcement learning algorithm developed by OpenAI, with the aim of producing superhuman performance without the use of human knowledge.

In the work by Smith et al., new Ticket to Ride maps are generated and MCTS is used to optimise the various aspects of the board, such as the colour of each route, in order to produce a fair win ratio between heuristic agents [18]. It improves on the work by Silva et al. which uses genetic algorithms with the aim of creating fair and interesting Ticket to Ride maps [19]. The wealth of previous work on creating new maps reinforced our decision of using TTR as our test bed, since we would be able to create maps of our desired size if required for testing.

5.3 Dynamic Difficulty Adjustment

A simple example of DDA demonstrated by Silva et al. in the video game *Defense of the Ancients* [20] defines three static difficulty levels which dictate the behaviour of the AI opponents. For example, the enemies can only cast spells when in "hard mode." A heuristic function which accounts for attributes such as the player's level and death count is used as a measure of the player's current strength. At certain time intervals, the relative increase or decrease of player strength is calculated and if these differences fall outside a defined range, the difficulty level will be adjusted up or down. This approach is simple but demonstrates how DDA could be implemented using heuristic, rules-based agents. However, it is likely that it would produce unnatural gameplay with large shifts in difficulty, since the adjustments are not continuous in either time or difficulty level.

In the work by Kennedy, a DDA agent for the game of chess is implemented [21]. The difficulty of the game is modified by adjusting the search depth used in the minimax algorithm, and the blunder probability that dictates the chance a suboptimal move is selected. The author uses the Stockfish chess engine to calculate the evaluation of the current position to determine which player is winning and at non-terminal nodes when the depth limit is reached during search. A handcrafted function was created that maps a state evaluation to a corresponding search depth and blunder probability, such that a higher depth and lower blunder probability is selected if the AI is currently in a losing position. This approach would create a much smoother transition between difficulty levels compared to the previous approach, since predefined skill levels are not used.

AlphaZero is a reinforcement learning algorithm combining deep neural networks and MCTS to achieve super human performance in board games including chess and go which makes it a natural target for modification to produce DDA. In the work by Fujita [22], three different methods of implementing DDA are proposed. One approach modifies the time allocation given to MCTS and another purposely damages the neural network using dropout to inhibit its performance. The final approach modifies the UCT equation to select moves based on its perceived chance of winning the game.

Another reinforcement learning approach known as modifies the Q-learning algorithm to implement DDA. The algorithm learns an action-value function, which is then used to create a ranked order of actions in a given state. The agent starts by selecting actions at a level of 0.5, which represents the 50th percentile, but the level is adjusted as the game progresses in order to match the strength of the opponent. Demediuk et al., whose work is discussed in the next section [23], points out that this may not be an effective approach since game states may have a disproportionate number of strong or weak actions. Therefore, a high level can still result in weak actions and vice versa.

5.4 DDA For MCTS

Our project focuses on implementing a DDA agent using MCTS, partly because the research in this specific area is limited. One of these works by Hao et al. [2] proposes the adjustment of computation time for MCTS in order to modify the game difficulty based on previous work, which shows that increasing the simulation time improves the intelligence of the agent. They use the video game Pacman as a test bed, where MCTS is used to determine the actions of the non-player ghost opponents. They suggest that DDA based on MCTS is better than previous methods that modify difficulty by adjusting game parameters such as the number of enemies, because it allows difficulty to be adjusted in

a continuous and dynamic way. Moreover, MCTS does not require domain knowledge in order to play the game. To estimate a function that maps a desired win rate to computation time used by MCTS, the researchers carried out groups of 250 simulations for different time allocations. However, the simulations were carried out against a Pacman with a fixed strategy in which it could only make forward or right moves, so it is unlikely that the derived function could generalise to other opponents. Another problem with this approach is that it is hardware dependent, since a more performant computer could carry out more simulations in the same amount of time [2]. In order to address this, they propose a method in which neural networks are trained on data generated by MCTS in order to approximate its performance without requiring online computation.

Further limitations of the previous work are identified by Demediuk et al. [23] who propose modifications to the algorithm itself in order to produce DDA. They point out that an equation would need to be developed for each new game, which is time-consuming compared to modifying the algorithm itself. Furthermore, it is possible that the algorithm will find strong actions in a restricted time frame in certain states and, on the other hand, weak actions when given significant computational time. Given that the derived equations which map desired win rate to computation time do not take into account the starting state, unbalanced behaviours could be produced [23]. Instead, they propose two action selection policies and two play out evaluation strategies, with the aim of selection actions which result in evenly matched game states.

The first action selection strategy they discuss is called **Reactive Outcome-Sensitive Action Selection (ROSAS)**, which is used in the final step of the algorithm to select the best action based on search tree statistics. Instead of selecting the child with the highest average payoff, the child with a score closest to zero is selected instead. A positive score suggests a resulting game state in which the agent is winning and one where the opponent is winning with a negative score. Therefore, actions with an average payoff close to zero keep the game closely matched. The following equation transforms this problem into a convex optimisation problem:

$$action = \arg\max_{a} -|r[a].score|$$

where r is the root of the search tree (the current game state), a is an action available in that state and $r[a]$ is the resulting state after applying action a to r.

However, a limitation was identified with this approach: the produced behaviour is only

reactive, and it can not make any proactive plays because it is always aiming for a score of zero, mirroring the opponent. **Proactive Outcome-Sensitive Action Selection (POSAS)** is put forward to address this issue. All actions which have a score within a defined interval around zero are equally weighted and are selected uniformly at random. Therefore, the agent can select actions which are slightly advantageous or disadvantageous, which aims to make behaviour more realistic and less exploitable. The following equation is used to select the action:

$$action = \arg\max_{a} -(|r[a].score| - I_h)^+$$

where $I_h \in \mathbb{R}_+$ defines the interval $[-I_h, I_h]$ about zero in which actions are selected uniformly at random. The + denotes a function which returns 0 for negative numbers and identity for positive numbers.

Finally, they propose two more modifications which exploit the asymmetric nature of MCTS. In the standard MCTS algorithm, the most promising actions which have the highest payoffs during simulations are explored in more detail. However, this is undesirable in the context of DDA where well-balanced game states would be left under-scrutinised in comparison. To fix this, **True ROSAS/POSAS** utilises the previous selection strategies during the selection step of the algorithm in which expandable nodes are selected for exploration. For example, True ROSAS will select the child action that has a score closest to zero in each selection step as opposed to just the final step at the end of the algorithm. At the conclusion of the algorithm, the child with the most visits can be selected, since the ROSAS selection strategy has already been baked into the algorithm.

To gather results on the performance of these developed modifications, the DDA agents were implemented to play a real-time fighting video game called FightingICE and tested against agents previously entered into a 2016 competition. The health difference between the two players was used as payoff, such that the DDA agent using ROSAS would select actions that produced a health difference closest to zero. They used a POSAS threshold of $\pm 10\%$ health points. They found that the ROSAS agents achieved average health differences much closer to zero compared to POSAS with lower standard deviations. However, they find that POSAS produces more natural game play in human games, since the ROSAS agent would play very passively when the health difference is small and perhaps unfairly if the player is ahead by a large margin. In automated games, the final win rates they recorded were: **ROSAS - 40%, POSAS - 37%, True ROSAS**

- 24% and True POSAS - 39%. Based on previous work, they suggest that a 50% win rate will provide a suitable level of challenge, but none of the algorithms achieve this target. This is because the algorithms are inherently responding to the actions of the opponent which POSAS aims to address. They suggest that a target value above zero could be used to achieve win rates closer to 50%. On the other hand, they propose that maintaining a score difference close to zero regardless of the resulting win rate keeps the game unpredictable and exciting.

5.5 Heuristic Agents

Yang et al. [14] test their PPO agent's performance against various heuristic bots. These bots make decisions based on a cost function. The most relevant bot to our project is CFBaseAI.

CFBaseAI uses the cost function to perform several different tasks. Firstly, it considers all possible paths that complete its destination tickets; these paths are evaluated according to the cost function and the best one is chosen. Once the optimal path is selected, the agent employs another cost function (with specified values assigned to train resource cards) to choose a route on the path to claim. If the agent does not have sufficient resources to claim the selected route, it will draw the missing train resource cards required to claim the route. Finally, if the agent cannot find a best path completing the destination tickets, it will either choose to claim a route with the best score or draw additional destination tickets.

Chapter 6

Legal, Social, and Ethical Considerations

Developing a DDA AI for an existing board game raises a number of legal and social issues, and it is important that these are considered prior to the commencement of the project, rather than retrospectively, as they can inform key decisions made during development. This section details any such considerations, the restrictions they might pose, and how the team takes them into consideration in relation to the project.

6.1 Legal Considerations

The major legal issues we have taken into consideration are to do with copyright and licensing, as detailed below.

6.1.1 Copyright

Ticket to Ride is an existing and copyrighted product owned by Days of Wonder [24]. Caution should be taken when implementing and presenting our adaptation of the game, and this is a particularly important consideration when it comes to the distribution of the final product - as we have replicated its mechanics and design, making our work publicly accessible could be unlawful and could lead to copyright infringement claims.

As such, the project will not be distributed freely or commercially without proper permission and licensing agreements. As it stands, our project falls under non-commercial research and private study, which is categorised under Fair Use [25], causing negligible financial impact to the copyright owner, and compliant with intellectual property rights.

To ensure this is the case, game copies provided to testers are instructed to be used for strictly academic purpose, and redistribution is prohibited.

6.1.2 Tool Licensing

The technologies and tools used in this project (see Chapter 8) are all free to the public, or free to use through our institution. Some of these software are also open source. The final software will not be made available to the public, and hence should not be a cause of concern for the potential breach of licences.

6.2 Social Considerations

Ticket to Ride features several locations and cultures, which should be represented with sensitivity and accuracy by avoiding stereotypes and misrepresentations to maintain respect and avoid potential backlash from users. Within our adaptation of the game, we eliminate this issue by featuring a fictional world, in which the cities do not reference real-life countries or locations.

Dynamic Difficulty Adjustment in games has social implications. The enjoyment of video games can be used to exploit its players, which is particularly seen in microtransactions in mobile games [26]. Developers often aim to maximise engagement in a product, due to the association with revenue. The strategies employed and excessive time spent using the product can lead to unintended consequences such as a reduction in focus, addiction and social withdrawal [27]. The success criteria of our product is based upon user enjoyment not explicitly engagement, however it is important not to use psychological tricks to keep players engaged beyond what is appropriate for a board game based application.

DDA algorithms, if effective, could be used to exploit vulnerable players. Dark Pattern Games [28] describes "dark patterns" in video games as "something that is deliberately added to a game to cause an unwanted negative experience for the player with a positive outcome for the game developer." An example of a dark pattern facilitated by DDA is giving players the illusion of control, which keeps the player interested by manipulating the difficulty, potentially giving players the illusion of improvement in their skill. This artificial shift in difficulty can be utilised at points to make a game easier or more difficult, abusing the player's emotional response to encourage more payments into the game. As a proof of concept our work is unlikely to cause such issues, however DDA in future products could.

When used in conjunction with other tactics, dark patterns can be very effective in

retaining players, even if it is to their detriment. However, it is difficult to see how this can be avoided. DDA, much like all technology, can be used in a malicious way for monetary gain, and lawmakers are increasingly struggling to keep up with effective regulations [29]. Ultimately, it is up to those who have access to these technologies to not abuse them for personal gain, however optimistic this outlook may be.

6.3 Ethical Consent

Research involving data obtained from human participants raises ethical considerations that must be carefully addressed.

It is important that we respect and not abuse user testers' time and commitment. This includes being transparent about the expected time commitment and keeping it to a minimum by taking care to make matters of installation and use as simple as possible. In addition, it is important to treat testers with dignity and recognize their autonomy and intelligence. This includes allowing users the flexibility to provide answers according to their understanding and avoid pressuring them to complete the entire test.

Also, it is important that a testing sample does not affect the outcome in a discriminatory way. Whilst answering the project's objectives do not involve generalisations about sensitive populations, we have tried to include a range of ages and video game ability. However, it is important to acknowledge our results are unlikely to take into account individuals who are not already interested in board games or technically proficient.

We aim to preserve a tester's privacy to the maximum degree. We do this by not requiring any uniquely identifiable information but age, for which we will provide large bounds. Thus, we are able to report their performance on the user test in a way that preserves anonymity. As we only collect user feedback for software evaluation and iteration in an anonymous approach, rather than utilising user data to train models, application for university approval and ethical consent is not required.

Chapter 7

Project Management

An appropriate management methodology is essential for any successful project. In this chapter, we will detail the management methods chosen for this project and justify their suitability, as well as discuss the various tools used and artefacts we have produced throughout the project's development.

7.1 Stakeholders

The stakeholders are an integral part of the project and inform the methodology, as their expectations and availability determines the feedback cycle and project flexibility. There are five groups of people we consider key stakeholders in the project: the project supervisor, the module organiser, the project team, the client, and the potential end users. The project supervisor, Till Bretschneider, is a subject expert who oversees the project and is able to advise the team on both technical and administrative aspects. The module organiser, Sara Kalvala, does not work directly with the team but is primarily interested in ensuring the project aligns with the module's goals; nevertheless, it is important that they are informed of any major changes in the project.

The project team designs, structures, and carries out work needed to ensure the project's successful completion. An important aspect of this is carrying out communications with the client, who is a student at the University of Warwick, as well as a member of the Tabletop Games society. They are curious to explore the application of DDA to a board game, as well as to provide an end-user perspective as a representative from the potential user base of the system. As the client is not available to meet on a regular basis, we opted to instead keep the client up to date with major milestones in the project, with which we

hope to maintain good transparency of the process and ensure the final product meets the client's expectations. The end users are the target demographic for the final product. Whilst they are primarily represented by the client, the team should keep the broader user base in mind when designing the product. In practice, we ensured the end users' interests were taken into account by obtaining user feedback for intermediate designs, such as for the GUI sketch.

An additional stakeholder is the Department of Computer Science, which enforces deadlines we must adhere to for both the project and other pieces of academic work. For the purposes of this project, the relevant restrictions posed by the department are communicated by the module organiser, hence it is not considered a direct key stakeholder.

We use the power-interest grid shown in Figure 7.1 to segment the aforementioned stakeholders into four groups, each with a respective management expectation. The general expectations are that stakeholders who hold more power should be kept satisfied with its direction and progress, and those who are interested should be kept up-to-date with the project's status. It is possible for a stakeholder to hold a lot of power as well as having a high interest, or to have a high interest but no direct power over the project.

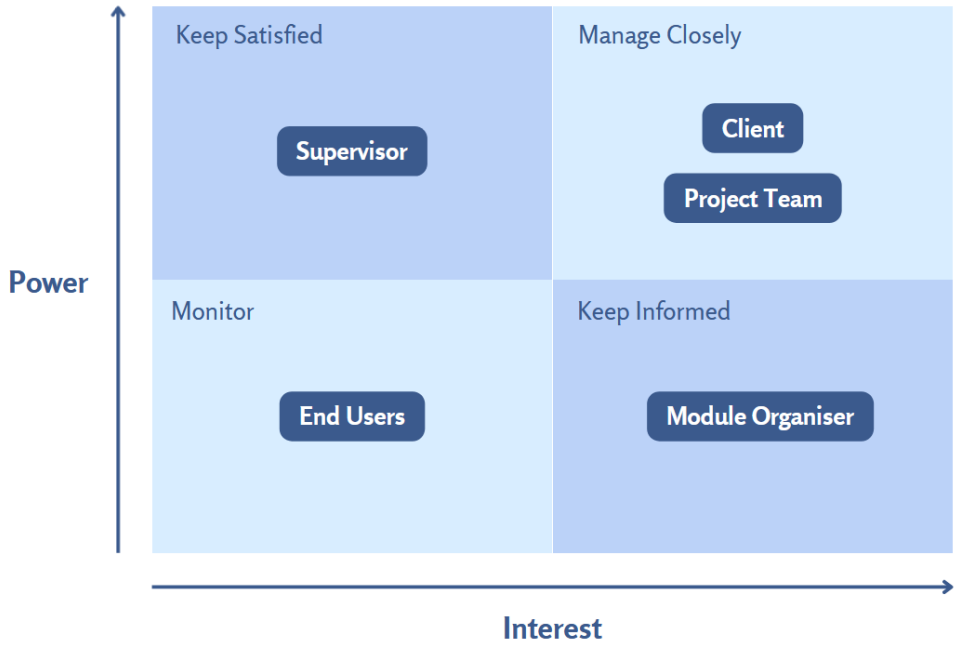


Figure 7.1: Power-interest grid for stakeholder analysis.

Based on the power-interest grid, we have designed a number of engagement strategies for each key stakeholder, taking into account their availability and preferred forms of communication. This can be found in Table 7.1. The strategies chosen remained largely

unchanged since the commencement of the project, however, during development it became apparent that the client was not available for frequent meetings as we had initially planned. Hence, we adapted our communication plan to be less frequent, only updating the client when important milestones are achieved in the project.

Table 7.1: Table of stakeholders and engagement strategies.

Name & Role	**Description**	**Engagement Strategy**
Client (Robert Crawley)	Has high interest and influence, informs the general direction of the project.	Keep informed of project updates, and should review key design documents.
End Users	An extension of the client, represents the general users of the system.	Feedback should be taken into account when considering the system's designs.
Supervisor (Till Bretschneider)	Has high influence in the project, is a subject expert and an assessor of the project.	Keep up to date with project status. Regular fortnightly meetings with the project team are scheduled, during which discussion of the project will take place.
Module Organiser (Sara Kalvala)	Sets the framework of the project itself.	Keep informed of substantial changes to the project.
Project Team	The developers working on the project.	Team leads to manage members closely, team keeps each other informed of updates and hold weekly sync-up meetings for discussion of ideas and project direction.

7.2 Methodology

We have identified a number of important project conditions that have influenced our methodology:

I We are a small team of four developers.

II We have had highly variable schedules dependent on external workload.

III The project has hybrid characteristics, involving both research and software engineering components.

IV The project had a number of immutable deadlines, including the production of a

specification, two presentations, a full-length project report, as well as the software product itself.

V The team had to respond to both the feedback from the supervisor and the needs of the customer.

A project with changing customer requirements and ambiguity did not suit a traditional planning based approach. Additionally, the project's requirements were contingent on ongoing research developments, introducing further unpredictability. The varying work capacity and schedule made it difficult to plan long term goals in granular detail. Considering these constraints, a predominantly agile approach was deemed most suitable for our project conditions. However, we had long-term immovable deadlines that have enforced constraints on our flexibility in order to meet them. Thus, we have had to be pragmatic and take these considerations by loosening the formalities of our chosen methodology, Scrum.

7.2.1 Scrum

Scrum is an agile project management framework that known for its adaptability, transparency, and iterative nature [30]. This flexibility was particularly important for our project for two reasons. Firstly, the concurrency of research and development meant implementation details had to be changed in order to respond to new findings. This in turn lead to changes in the overall design of the system. Secondly, we needed to respond to potentially changing client and stakeholder requirements in a timely fashion. The continuous feedback and re-prioritisation of tasks in scrum allowed the project team to take these factors into consideration.

7.2.2 Artefacts

As part of our methodology, we had chosen to implement three main artefacts: the product and sprint backlog, as well as a Gantt chart. The team entertained the idea of an increment, however given the small team size, variable schedule and infrequent client communication, it was decided that a product increment would generate an excessive amount of unnecessary work to produce a stable build each week. For similar reasons, especially given the variability in the week-to-week workload, a burndown chart for each sprint was not produced [30].

Product and Sprint Backlog. We have used GitHub Scrum Boards to combine the product backlog and sprint backlog into one artefact. The product backlog contains all

planned features for the upcoming sprints, and can be added to as necessary. As shown in Figure 7.2, the sprint backlog is divided into four columns: the actual backlog for the current sprint, the ongoing sprint tasks, tasks awaiting review by other team members, and tasks done. This visualisation of tasks ensures transparency and allows each team member to understand sprint progress at a glance.

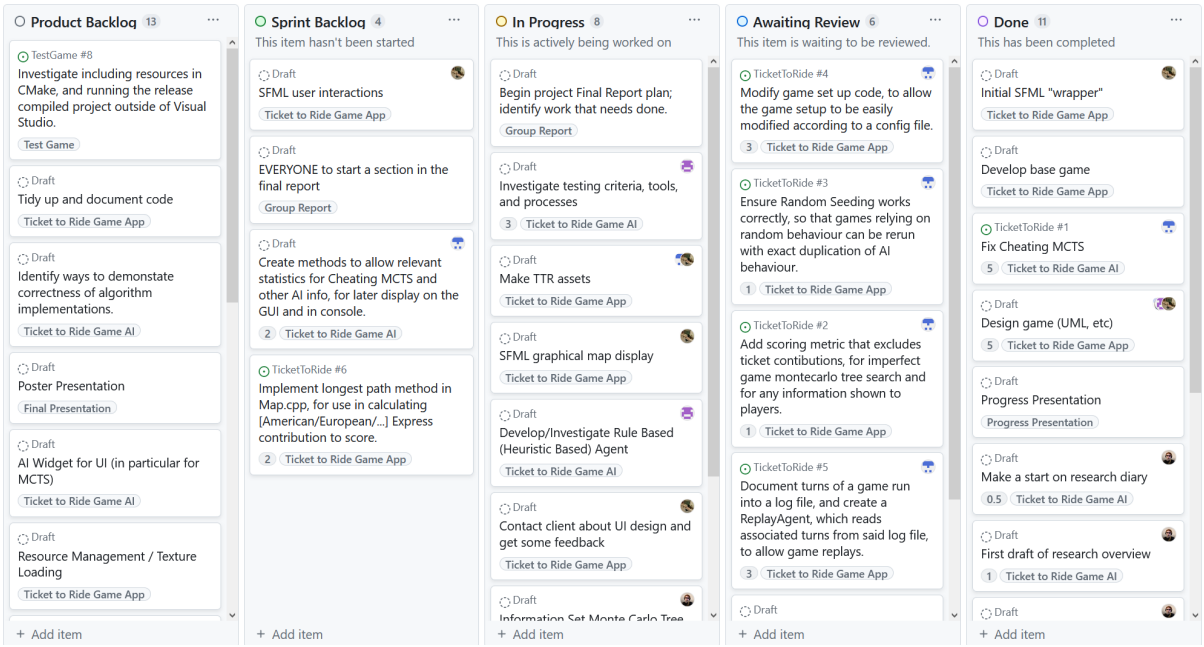


Figure 7.2: Screenshot of GitHub Scrum Board, with both product and sprint backlogs.

Gantt Chart. As expanded upon in Section 7.4, the Gantt chart acts as a high-level structure to ensure key externally set project deadlines are finished on time. Hence, the Gantt chart itself is not a detailed breakdown of every project task, and does not need to be strictly adhered to should changes be made. The Gantt chart was referred to during meetings as well as by members in their own time to ensure their current prioritisation of tasks aligned with project needs.

7.2.3 Ceremonies

Two-week sprints have been carried out, starting on Friday and ending on Wednesday. This sprint length provides a good balance between intensity and frequency, ensuring that significant progress is made within each sprint and that members are able to stay up-to-date with project status, always keeping the project fresh in mind. Three scrum ceremonies are carried out to ensure each sprint is clearly defined and productive: sprint planning, sprint review, and retrospective meetings. Daily stand-ups are not suitable here, as the team is not working on the project full-time.

Sprint Planning At the start of each sprint, sprint planning took place to outline the overall goal for the sprint, as well as the specific tasks that will contribute to it. This involved deciding which tasks are the most important at the current stage, as well as determining the availability of each member in the team so that work can be assigned. Team members were encouraged to be transparent with their schedule to avoid overburden. Everyone plays a part in prioritising tasks: though the overall next steps are determined by the product owner, each member contributes to assessing a task's scope and importance. When a suitable selection of tasks are chosen, they are added to the sprint backlog and assigned to individuals.

Sprint Review At the end of each sprint, the team demonstrated their progress to the supervisor, as well as to each other during a sprint review. A small presentation was put together to illustrate the work done in the sprint. The primary goal of this sprint review is to keep the supervisor in the loop and discuss any feedback, but it also helps to update the whole team with individual progress and serves as a wrap-up for the sprint. We also carried out a number of internal reviews which acted as knowledge sharing sessions, where team members could explain technical aspects of their work in detail, allowing for other members of the group to gain a more well-rounded understanding of the project and work cross-functionally if needed.

Sprint Retrospective The sprint retrospective is a scrum ceremony which helps the team to reflect on what went well and what can be improved. Carried out as a quick meeting at the end of each sprint, members are able to relax and look back on the previous sprint and reflect on whether the methodology is serving its purpose, as well as raise any team or workload issues. This short reflection served to integrate continuous improvement to our workflow, and ensured the team was functioning smoothly without friction. As the team members were not familiar with each other initially, this meeting was helpful for everyone to address their concerns. By the end of the first term, members felt comfortable enough to address issues directly and in their own time, and the sprint retrospective became less formalised and was generally forwent.

7.2.4 Roles and Responsibilities

Our project required the team to not only partake in software development and research, but also project management, design, testing, as well as communicating with important stakeholders. To prevent stagnation and promote a division of responsibility, each team member has taken on one or many roles in addition to those of general development and research. These roles ensured that team members are able to individually push the

project forward and that decisions could be reached quickly.

Scrum Master Sam James

A core role in the scrum framework, the scrum master is the people's leader of the team. They are responsible for scheduling resources for each sprint, planning and leading meetings, and ensuring the overall effectiveness of the team is maintained. For this project, the scrum master also managed documentation, both developer and user-facing.

Product Owner Katie Yang

Another scrum role, the product owner ensures the client and other important stakeholders' needs are met. Tasked for defining and communicating a high level vision of the project, they are responsible for prioritising features and ensuring the product meets customers' expectations. They maintain the product backlog, a prioritised list of features for the development team to pull from. For this project, they were also the primary customer and admin contact.

Technical Lead Sam Medwell

This member of the team is the most experienced with the technical prerequisites of the project. They are research-focused and spearhead the direction of technical development, making important decisions in this area, for example, algorithms to be implemented for the agent. They also manage the research resources such as papers reviewed, and maintains the research diary.

Testing Lead Lukas Rutkauskas

Responsible for the overall testing and evaluation of the project, the testing lead ensures each project component functions as expected and are integrated correctly. They are accountable for producing a comprehensive and efficient testing framework to check for both correctness and effectiveness of the final software product.

7.2.5 Technical Areas of Responsibility

We divided the development in to two tracks, AI development and game development, under what we call *Dual Track Development*. These tracks would combine at key milestone points, as outlined in Figure 7.3.

Each member was assigned to one development track, with Sam Medwell and Lukas

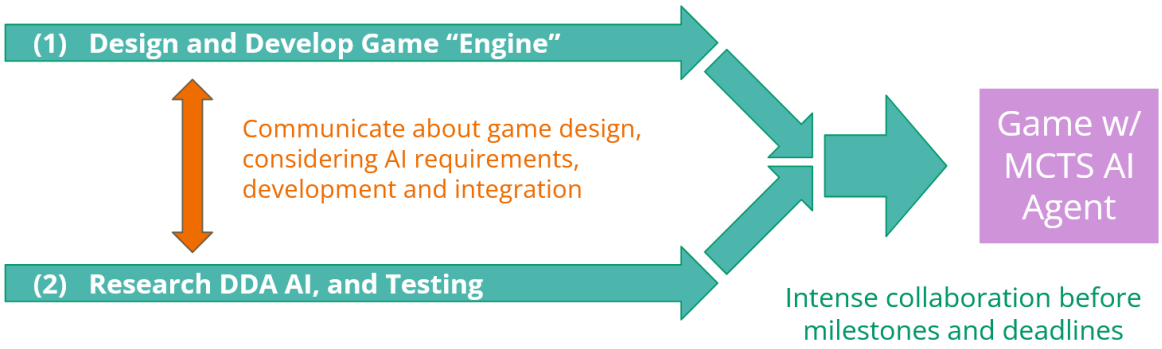


Figure 7.3: Dual Track Development during the first half of our project. The diagram was drawn for internal review. This allowed simultaneous work making the game environment as well as researching what AI models may be suitable for the project.

Rutkauskas starting on the AI track and Sam James and Katie Yang on the game application track. Members were not bound to these roles; as these were regularly assessed in retrospective meetings throughout the terms. However, in practice there was little divergence from these tracks as people were well suited to their initial roles.

The dual tracks further split into more defined areas of responsibility during the second half of the project. Sam James was responsible for the back end of the code including areas of performance, reproducibility and maintainability, providing tools for the AI team to use to test and develop their AI. Katie Yang was responsible for developing the GUI and creating game assets. Lukas Rutkauskas was responsible for developing a set of heuristic agents and testing their relevant strengths. Sam Medwell was responsible for developing the Dynamic Difficulty Adjustment agents and running tests to evaluate their performance.

In practice, whilst knowledge sharing occurred regularly to ensure the various components integrate well, the two tracks were able to remain fairly separate, with Sam James acting as an intermediary responsible for integrating game environment and AI code. This separation has allowed developers to maintain focus about their respective areas without needing to understand the whole system, reducing the individual burden of knowledge. This division of responsibility has been very beneficial, allowing each team member to hone their respective areas of expertise. The development of such specialisms utilise a depth rather than breadth of knowledge.

With integration responsibilities falling to a single person, each other developer has not had to concern themselves with the details of integration and merge conflicts, whilst the person responsible for merging has been able to improve their skills in managing

the different areas of code, resulting in a more streamlined integration process overall. Having a dedicated individual responsible for the codebase meant the team was able to improve its cohesion.

However, the division of responsibility and development of specialisms has raised the issue of single points of failure. For instance, if a team member was singly responsible for GUI and had became unavailable, the other members would have struggled to continue the work without the prior time spent developing that specialism. Limited knowledge sharing was used to mitigate this issue.

7.3 Tools and Technologies

This section details the tools used for general project management. Development tools can be found in the Technologies Chapter.

GitHub Scrum Board. As shown in Figure 7.2, this board is used for the creation, recording and assignment of sprint tasks. Managed by the product owner and updated every sprint, it is a useful tool for team members to keep track of their own ongoing tasks, as well as upcoming items for the team.

Overleaf. An online Latex editor used to create milestone project documentation such as the specification and the final report. The use of Latex is almost essential for a report of this scale, including the use of figures and references, and as Overleaf is browser-based it allows the team to collaborate effectively without being physically together.

Zotero. A research and reference tool used by the team to keep track of literature related to the project. With its browser extension and Overleaf integration, Zotero organises and streamlines the research process and bypasses the need to manually import references into reports.

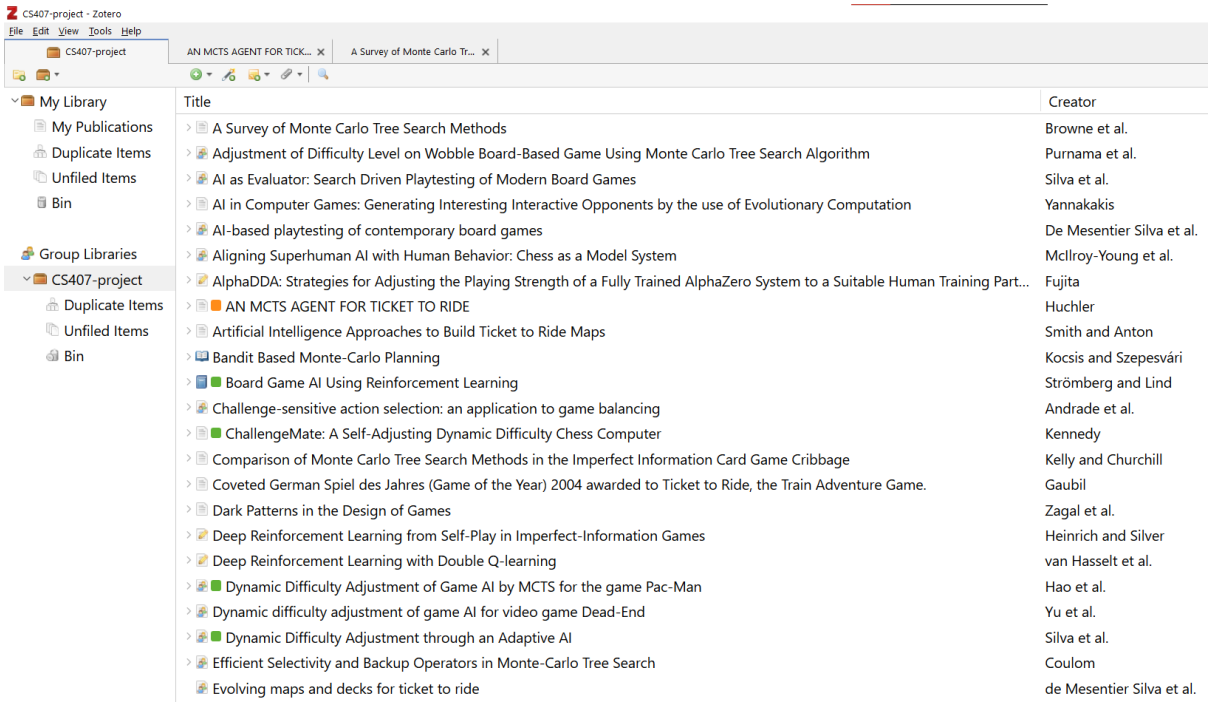


Figure 7.4: Screenshot of Zotero containing a number of papers reviewed.

Discord. The main communication channel used by the team. As an instant messaging platform, it is simple and casual, reducing communication barrier and thus increasing team engagement. Discord closely resembles platforms used in industry, such as Slack. Furthermore, its ability to create channels for chats dedicated to different aspects of the project facilitates conversations to take place simultaneously without confusion.

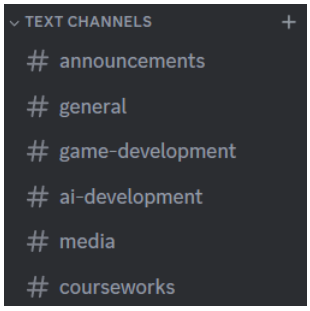


Figure 7.5: Discord channels. Note separate channels for different development tracks.

Google Drive. A file storage and synchronisation service which hosts much of the team's documentation, including meeting minutes, sprint review presentations, scheduling spreadsheets, and various internal documentation such as methodologies and testing plans.

7.4 Project Schedule

To guide project work at a high level, we produced and maintained a Gantt chart. The main motivation behind this is so that we were able to conform to immutable assessment

deadlines, and schedule our workload to allow each stage of the project to be carried out in time. To this end, the Gantt chart is composed of major work packages, their estimated lengths, and any dependencies. The aim was not to bind developers to an immutable schedule, but for the chart to be used as a guideline to track progress made, and ensure delays can be spotted early and be accounted for.

We created an initial schedule before starting development, as shown in Figure 7.6, where we had divided the workload into two tracks: the development of the game environment, and the development of AI agents. Significant time was allocated to research tasks at the start of the project, in order to give the team enough time to formulate well-informed AI agents and robust technology stack. We also allocated time for integration, which was recognised by team members to be typically underestimated, according to past project experience. This allows the team time to fix potential issues that could arise when merging various components in the codebase.

Throughout development, the team was split into two subteams, allowing simultaneous work building the game environment and user application and researching and developing the AI agents. Initially, the team planned to integrate both subsystems together once they are individually tested and sufficiently mature, but as the development process unfolded, it made more sense to continuously integrate the subsystems at milestone points. In practice, integration involved merging AI models and their responsible code to the latest stable game branch as and when they were fully developed. Regular integration avoided conflicting code that would have required significant refactor. In addition, it allowed both subteams to stay up-to-date with the work of the other, and use newly developed features more readily.

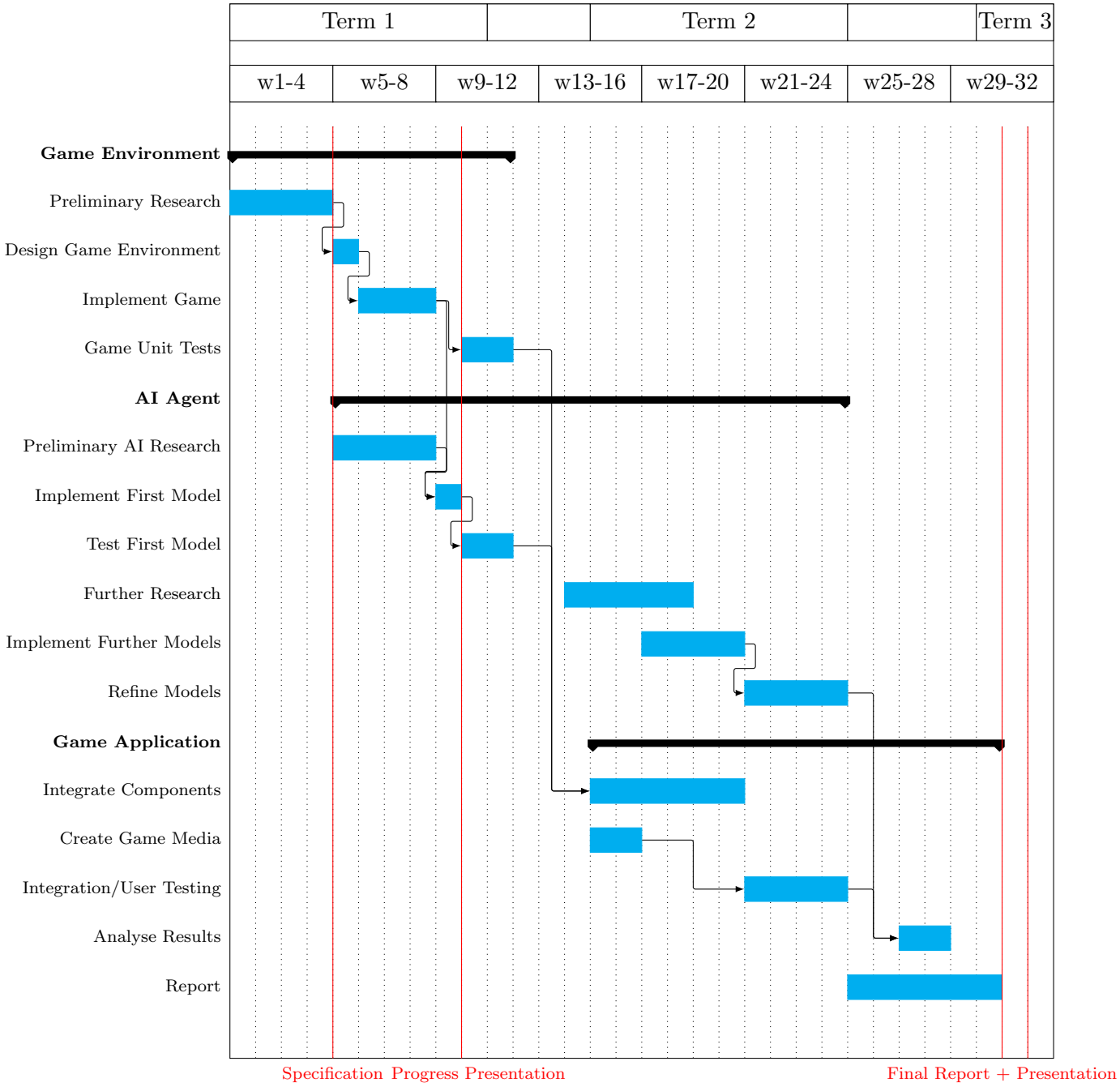


Figure 7.6: Initial Gantt chart with weekly breakdown of project progression plan.

The initial schedule proved to be an effective tool for mapping out long term goals during the first term. We kept this chart regularly updated so that it best reflects the current knowledge about the project and its progress. Figure 7.7 shows the state of the Gantt chart by the beginning of the second half of the project (from term two onwards). At this stage, the team had a better understanding of the remaining scope of the project, and was able to detail the specific types of agents. In particular, we extended the AI agent deadline to allow ourselves time to implement the modifications planned.

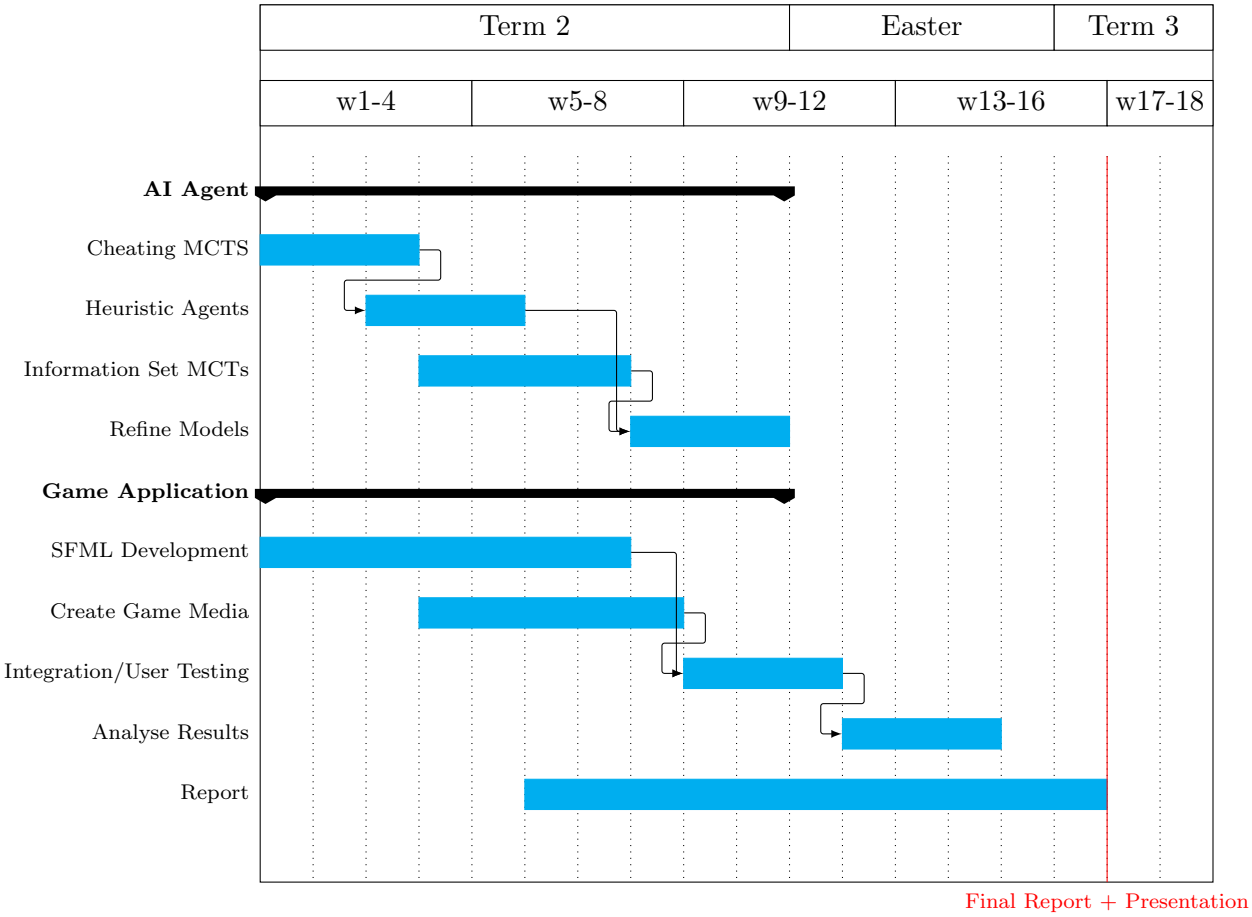


Figure 7.7: Updated Gantt chart at the start of term two.

In the final month of the project, the team experienced time crunch due to increased external workload, as well as internal delays. We felt that a more detailed schedule was required to ensure all aspects of the project have been taken into account and wrapped up. Hence, we created a planning spreadsheet (Figure 7.8), which contained member availability on a daily scale. Based on this, we planned a number of internal deadlines, such as for the testing plan or the GUI, as well as scheduling a number of formal meetings to checkup on project status and discuss next steps.

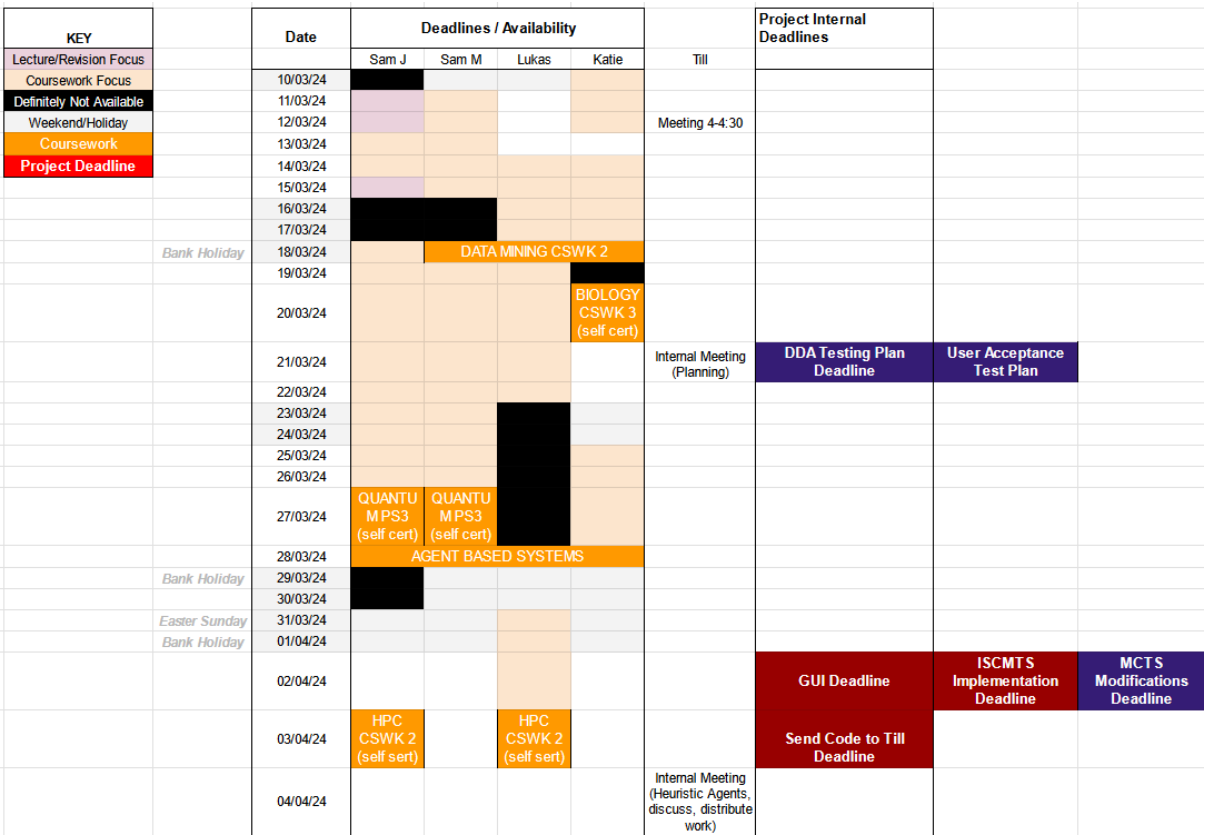


Figure 7.8: A portion of the spreadsheet used by the team to plan project schedule in more granular detail.

7.5 Risk Register

The creation of a risk register allows a team to proactively address and plan mitigation for issues that may arise during a project. At the time of specification - not long after the inception of the project itself - we created a risk register to address currently foreseen project risks, specifically detailing prevention plans to implement right from the start, as well as developing response plans in case the risk develops into an issue.

Table 7.2: Initial risk register.

N^o	Risk Overview	Likelihood	Impact	Risk Prevention	Risk Response Plan	Residual Risk
1	Ticket to Ride is too complex	Mid	High	Allow different maps to be loaded into the game so one of a suitable size can be used.	Switch to a game with lower complexity.	Mid

2	Development of AI does not finish in time for user testing	Mid	Mid	Start developing the AI after having researched the topic thoroughly and hence acquiring a good understanding of the technical details so that development time can be utilised efficiently.	Dedicate an extra team member to work on AI. Remove or amend lower importance requirements so that more time can be spent on AI development.	Low
3	DDA is effective but does not increase player enjoyment	Mid	Mid	Start user testing early, ideally as soon as the AI can be integrated into the game environment. Define quantifiable metrics for "enjoyment", incorporate them into the user survey.	Focus on specific target metrics and identify how each metric could be improved.	Low
4	Loss of a team member	Low	High	Pair programming / research to ensure knowledge is not lost.	Spend more time on developing the AI at the expense of UI.	Mid
5	Legal issues with board game	Low	High	Good understanding of proprietorship of the board game, maintain conditions of Fair Use.	Resolve by contacting board game developers/owners.	Low

It is important to keep a risk register regularly up to date as more information is learnt and circumstances change, so that it is adapted to the current project conditions. At the start of the second term, roughly halfway through the project, the team felt that some of the risks have changed either in its likelihood and impact, or have been non-issues altogether, such as risk 1 in the initial risk register, as much of the development of the TTR game has been finished at this point. The development and direction of the project also informed the team of new risks, calling for an updated risk register. This can be found in Table 7.3 below with the addition of risks 5, 6, and 7 and the removal of 1.

Table 7.3: Updated risk register.

N^o	Risk Overview	Likelihood	Impact	Risk Prevention	Risk Response Plan	Residual Risk

1	Development of AI does not finish in time for user testing	Mid	Mid	Start developing the AI only after having researched the topic thoroughly and hence acquiring a good understanding of the technical details so that development time would be utilised efficiently.	Dedicate an extra team member to work on AI. Remove or amend lower importance requirements so that more time can be spent on AI development.	Low
2	DDA is effective but does not increase player enjoyment	Mid	Mid	Start user testing early, ideally as soon as the AI can be integrated into the game environment. Define quantifiable metrics for "enjoyment", incorporate them into the user survey.	Focus on specific target metrics and identify how each metric could be improved.	Mid
3	Loss of a team member (single point of responsibility)	Low	High	Pair programming / research to ensure knowledge is not lost.	Spend more time on developing the AI at the expense of UI.	Low
4	Legal issues with board game	Very low	Mid	Good understanding of proprietorship of the board game, maintain conditions of Fair Use.	Resolve by contacting board game developers/owners.	Low
5	Time crunch due to external workload	High	Mid	Team members to set out and refer to coursework schedule when planning sprints and keep each other informed	Re-prioritise tasks to ensure the core product can be finished	Mid
6	Tech debt - significant refactor is possible due to different work streams	Mid	Mid	Semi-regular merge with main branch, and team members to inform others of major changes	Team members to discuss and pair program if necessary	Low
7	MCTS is not powerful enough to beat human player	Mid	Mid	Ensure that MCTS is correctly implemented, and the environment is set up suitably for it to perform maximally	Investigate and implement performance-enhancing modifications to MCTS	Mid

The risk register was effective in helping the team keep in mind potential risks before they developed into issues. One specific case for this was tech debt - risk number 6 on the updated register. As the development approached its halfway point, the team

was aware of the possible workload from integration of various project components. To prepare for this, we increased merges of working branches to main branch, typically after each iteration or added feature. Throughout the project, there were many potential points where massive refactoring could have been required for integration of branches, but the team managed to largely avoid this by these regular merges, as well as frequent communication and alerting others of potential changes to the system.

7.6 Effectiveness of Project Management

In general, we felt that the project management practices put into place were successful, with the exception of client communication. Initially, the client confirmed their availability for the schedule laid out by the team, which includes fortnightly progress updates and opportunities for feedback, as well as reviewing key documents (progress presentation etc.). The client seemed enthusiastic for the product, and the team decided to go ahead with the arrangement. However, as development commenced, communication from the client became sparse. We ran multiple checkups, providing them with project updates as discussed, but were not able to receive any useful feedback. Fortunately, the project did not have any direct dependence on the client, and the team was able to gather feedback from potential end users through both formal and informal user testing throughout the project. However, as a note for the future, regularly scheduled in person or video meetings could be implemented, as it may help keep up client engagement better through the project.

The team was able to progress at a pace in line with the project schedule for much of the first term. In the latter half of the project, development slowed down owing in part to the unexpected workload brought on by other academic commitments and in part to the ease of underestimating time taken to achieve a given software engineering tasks. This resulted in some adjustments to the later schedule, in particular extending the timescale of SFML development and the collection of user feedback, which informed further improvements to the DDA agents and user application. Overall, the software development aspect of the project continued all the way until the report deadline. This required the team to prioritise certain features and leave out others to only be discussed in the final report as future improvements.

To make sure everyone was up-to-date with key developments in the project, knowledge sharing sessions were held occasionally so that members can share exactly how their part of the system works. This allowed members to work cross-functionally in the second term of the project, including refactoring the game environment and debugging the im-

plementation of the MCTS algorithm. There was also a good balance of technical and non-technical focused roles, as the scrum master and the product owner were able to manage the high level direction of the project and keep the team on track, and those with technical-oriented roles were able to focus on research and development.

Chapter 8

Technology

This section gives rationale to the format of our game and how these decisions have affected our choice of technology. In particular, we discuss the platforms and approach chosen for the user application, as well as giving a comprehensive list of further development tools. Finally, we describe how we have validated our technology stack through a feasibility study of a smaller game.

8.1 Application Format

We chose to develop our system as a desktop application. Initially, we considered a web based application - in theory, this lends itself to easy distribution across multiple platforms. It also would more readily allow for hypothetical future online multiplayer extension. However, web hosting an application would have created several implementation complications, such as the need to send messages to and from a local server and potential security vulnerabilities when compared to desktop applications, which in our opinion outweighed the ease of cross platforming. The litany of issues brought about by such complexity would significantly increase total programming hours, which would dominate other features given our small team size. Desktop applications can also be compiled to single executable, which lie closer to hardware and give us increased options of graphics frameworks.

A Graphical User Interface (GUI) is preferred to a command line interface. A graphical application allows users and testers to easily interact with the game. We considered two approaches to develop the GUI: use a game engine, or implement the game directly from straight code. We considered two game engine candidates, Unity [31] and Godot

[32], both of which come with a wealth of built-in tools and features and visual editors which can speed up development. However, these game engines have steep initial learning curves, which the team may never fully overcome given the short timeframe of the project. Furthermore, Ticket to Ride is a game that could be implemented with simple 2D graphics, and the framework of game engines induce unnecessary features for our project which can lead to performance and storage overhead, which is not ideal given the criticality of particularly of performance for AI techniques such as MCTS. In addition, the restriction of a game engine for development limits the portability of our development environment.

For these reasons, the second approach explored is directly implementing the game in C++ using media APIs, such as those provided by the Simple and Fast Multimedia Library (SFML). SFML is a cross-platform software development library, which is designed to provide simple APIs to multimedia components, such as window, audio, and most importantly graphics [33]. Compared to game engines, using a media API like SFML is much more lightweight while still providing sufficient utility for everything planned for the game, supported by hardware-accelerated 2D graphics using OpenGL. For these reasons, SFML was chosen as the tool for game development. Keeping the game development stack as simple as possible allows more focus to be placed on the AI aspects of the project. Furthermore, this minimises the chance of complications in the integration between MCTS and the game environment. The library was initially favoured for its bindings for other languages like Python and Java, however the project evolved to be fully based in C++, rendering this feature unused.

8.2 Development Tools

Visual Studio [34]. Visual Studio offers a comprehensive Integrated Development Environment that is specifically designed for C++ development. Particularly useful are its integration with Microsoft's C++ compiler, as most of the development is done on Windows machines, and its advanced features such as IntelliSense and debugging tool set. Furthermore, Visual Studio provides an excellent live share feature that allows developers to modify the same code simultaneously, which proved very helpful for remote pair-programming sessions.

C++. Initially, Python was planned to be used for AI development. The rationale was that it is an extremely popular language for machine and deep learning, able to provide the team with pre-developed tools, as well as the team already having familiarity with the language. However, we chose not to explore reinforcement learning agents in favour

of focusing on MCTS development, which meant that Python was no longer a necessary choice. Given this, the team opted to implement the whole project in C++, as we felt that the project could be made less complex if the AI agent was able to directly communicate with the game environment without the need for interprocess communication. Furthermore, as computational resources is a limiting factor in the strength of MCTS, the use of C++ will likely result in better performance, as C++ provides increased execution speed as well as more efficient memory management due to finer control over memory allocation.

CMake [35]. CMake is the de facto standard for building C++ projects, and whilst complex to design and understand as a developer, is easy to use as a user and affords portability. With a single CMakeLists file, we can run the CMake tool to generate a Visual Studio project on Windows, whilst on Linux we can use the same CMake tool in order to generate a Makefile. This simplifies the installation and development process across differing platforms.

SFML. SFML was chosen for GUI development, as it was lightweight and easy to integrate with the rest of the project.

Git & GitHub. The use of a version control system is essential for any software engineering project, and Git is undoubtedly the most popular choice. Having multiple versions of the code facilitates separate tracks of development. GitHub builds on this by hosting Git repositories on the cloud, allowing team members to share their work instantly and remotely.

Procreate [36]. The chosen tool for game sprite creation is Procreate, a digital illustration tool for the iPad. With full-fledged features for artists and easy export of work in various formats, it is a good candidate to produce the media required for the GUI of the user application.

Valgrind, Callgrind and QCacheGrind. Valgrind [37] is a framework for building dynamic analysis tools, and most usefully for us has the tool Memcheck which checks leaking memory. Callgrind [38] is a profiling tool that records the call history among functions in a program's run as a call-graph, and is run from within Valgrind with command `valgrind --tool=callgrind`. QCacheGrind [39] is a Windows version of KCacheGrind: a KDE/Qt based GUI used to easily navigate the large amount of data that produced by Callgrind.

8.3 Feasibility Study

To verify our game stack, members of the team spent around a week implementing a test game of Noughts and Crosses. In particular, this was to trial the use of SFML, the Visual Studio development environment and ZeroMQ. ZeroMQ is an asynchronous messaging library designed to facilitate inter-process communication, an aspect that was initially planned to be a part of the system design. Later on in the project, the criteria of cross-communicating between a Python agent and a C++ game was no longer needed as we no longer planned to use Python reinforcement learning libraries, which also meant that ZeroMQ was no longer required.

Figure 8.1: Screenshot from noughts and crosses test game.

The test game accomplished a number of feats. Firstly, it validated that our technology stack was functional with no incompatibilities. In addition, developers were able to familiarise themselves with the Visual Studio integrated development environment workflow, making use of tools such has the debugger. Furthermore, the project served to validate cross-platform development and game portability using CMake. Secondly, by taking a (very simple) game from start to finish, developers were better informed of the relative complexity of developing Ticket to Ride, so were better calibrated when budgeting time. Thirdly, whilst developing the test game, developers less familiar with C++ were able to brush up upon their skills. Finally, we were able to learn lessons from the process, which helped when designing the architecture for TTR user application. One example is the action system in the test game, in which actions are encoded as messages to be sent between the agent and the game environment. This in turn inspired the action system in

the final game design, whereby actions are encoded as player events to be generated by agents to be decoded by the game environment. In summary, the process of developing the test game provided the team with the confidence to design and build the real game environment using the previously described technology stack.

Chapter 9

AI Agents

In this chapter, we detail the design decisions and direction taken on the developed AI agents, justifying our choices of certain methods over others. The first and main objective of the project is to create an AI agent that can modify its skill dynamically to provide an interesting experience for players. To make the scope of the project manageable, we decided to focus our attention on creating DDA agents using MCTS methods, with the two main categories being Cheating Monte Carlo Tree Search (CMTCS) and Information Set Monte Carlo Tree Search (ISMCTS). We are interested in comparing the performance of these two algorithms in terms of their maximum playing strength and their behaviour as perceived by human opponents.

We opted to use Single-Observer ISMCTS as the representative for the class of ISMCTS algorithms. It has been shown that the performance of each variant depends on the specific game, with no single approach being the best in all cases [12]. Huchler, in their work on MCTS for Ticket to Ride [6], find that the base Single-Observer ISMCTS algorithm performed as well as or better than the base Multiple-Observer ISMCTS algorithm. We focus on the effects of DDA modifications so we restrict our investigation to a single class of ISMCTS algorithm, but we believe general trends will likely apply across all of them.

We will be exploring the four DDA methods for MCTS introduced by Demediuk et al. [23]: ROSAS, POSAS, True ROSAS and True POSAS. We will compare which of these modifications can produce the most balanced games and achieve win rates closest to 50%. Furthermore, we introduce our own novel modification, which introduces momentum by penalising actions which produce large swings in game score. We theorize that this modification will have a greater effect in making the gameplay of CMCTS more natural

since it can more effectively detect opponent blunders compared to ISMCTS.

Earlier in the project, we also intended to explore the use of reinforcement learning algorithms but after conducting our feasibility study, we later focused project scope since it was not feasible to implement them and explore several DDA techniques in detail. We prefer to investigate a topic with depth rather than breadth. However, ideas that we explore for MCTS can be applied to reinforcement learning methods. For example, we mentioned previously how AlphaZero uses MCTS methods [22].

To evaluate the MCTS agents, we implement a number of heuristic agents based on previously discussed strategies which have different difficulty levels. These heuristic agents are used to mimic players of differing strategies in order to synthesise additional player data to evaluate our models' effectiveness, given we have a restricted pool of available human user testers. The design and selection of heuristic agents will also be discussed in this chapter.

9.1 Cheating MCTS

The first MCTS model we approached implements the standard MCTS algorithm, but with access to information normally hidden from players. We propose that Cheating Monte-Carlo Tree Search (CMCTS) will work well because the algorithm will focus its search on the true game state, as opposed to others in the same information set, allowing it to more accurately select moves that will keep the game well-balanced. However, there are some potential drawbacks with this approach which we will be investigating. It is likely that the agent will make frustrating moves from the prospective of the opponent, given it knows the full game state. For example, the agent may claim a critical route required by the opponent to complete a ticket, even if information leaked by the opponent would not suggest that route was desirable to them. This approach may generate unnatural behaviour from the agent, since it can plan ahead with full knowledge of the opponent's goals and the state of the draw piles. Moreover, its access to perfect information means it can detect blunders by the opponent and quickly adjust its own strength to return to a balanced game state. This should help the agent achieve a win rate closer to 50% but at the expense of producing unnatural behaviour.

Algorithm 1 shows the *uctSearch* method, which executes each iteration of the MCTS algorithm, constructing a search tree with payoff and visit statistics for each node. The algorithm can be limited by number of iterations or by time. Our MCTS design is based on that provided in the survey of MCTS methods by Browne et al. [10].

We decided to separate the logic of the algorithm into an agent class and a tree node class. The agent is responsible for setting up and controlling the stages of each iteration of the MCTS algorithm as shown in Algorithm 1, whereas the specific methods to generate, traverse and manipulate the search tree are contained in the node.

Algorithm 1 uctSearch method

```
procedure UCTSEARCH
    create root node v_0 with current state s_0
    while computational resources not exhausted do
        v_1 = v_0.treePolicy()
        Δ = v_1.defaultPolicy()
        v_1.backup(Δ)
    end while
    return partial game tree with root v_0
end procedure
```

Firstly, a root tree node is created, storing the current game state, which will get returned after the allocated amount of computational resources have been exceeded such as time or maximum iteration count. The best action in the given game state can be queried from this root node using the *bestChild* method shown in Algorithm 3.

At the start of each iteration, the *treePolicy* method, shown in Algorithm 2, is called on the root node which carries out both the selection and expansion steps described previously, returning the node which was expanded. From the game state corresponding to this new node, a simulation is carried out by selecting random moves for each player until the conclusion of the game. The result of the game is then propagated through the nodes visited in the tree policy to update their statistics.

Algorithm 2 treePolicy method of the MCTS node

```
procedure TREEPOLICY
    if current game state is terminal then
        return this
    end if
    if there are expandable actions then
        return this.expand()
    else
        return this.bestChild(C_p).treePolicy()
    end if
end procedure
```

The *treePolicy* method in Algorithm 2 recursively selects the best child node until one is found with unexpanded actions, i.e. actions which have not yet been explored or the

end of the game is reached. The *bestChild* method in Algorithm 3 shows how the UCB1 formula is applied to select the most promising child node, balancing between nodes with a high average payoff and those that are least explored. If two children have the same value, then one should be selected uniformly at random.

Algorithm 3 bestChild method of the MCTS node

procedure BESTCHILD(C_p)
  if current player in state is MCTS agent **then**
    return $\underset{c \in \text{ children}}{\text{argmax}} \frac{c.totalReward}{c.visitCount} + 2C_p\sqrt{\frac{2\ln(this.visitCount)}{c.visitCount}}$
  else
    return $\underset{c \in \text{ children}}{\text{argmin}} \frac{c.totalReward}{c.visitCount} - 2C_p\sqrt{\frac{2\ln(this.visitCount)}{c.visitCount}}$
  end if
end procedure

If the player to take their turn in the corresponding game state is the MCTS agent, then the formula is maximised, whereas in opponent states it is minimised. However, the exploration term should always be maximised so it is negated in opponent states [40]. The approach originally proposed by Browne et al. [10] for two player zero-sum games is to modify the backup method, such that the negated payoff is accumulated in opponent states. These approaches both achieve the same goal, but we opted with the former to contain all the logic in the *bestChild* method, which will also contain all the DDA logic later.

Algorithm 4 expand method of the MCTS node

procedure EXPAND
  select unexplored action a uniformly at random
  remove action a from unexplored actions
  return new child node with resulting game state after applying a
end procedure

The expand method in Algorithm 4 simply selects an unexplored action at random and creates a new child tree node which contains the game state where that action has been applied. It is possible that the expand method could create multiple child nodes on each iteration, but there does not seem to be too much work analysing the performance difference [10]. We opted with the more simple approach of expanding one child each time to reduce memory usage and maintain our research focus on analysing the performance of different DDA modifications.

Algorithm 5 defaultPolicy method of the MCTS node

```
procedure DEFAULTPOLICY
    while current state is non-terminal do
        select legal action uniformly at random and apply it to current state
    end while
    return own score - opponent score
end procedure
```

When a new node is created in the search tree, a simulation is carried out from that game state. The most basic form of the algorithm randomly selects moves for each player until the end of the game, but many extensions to this have been explored in previous work [10]. One example is implemented by Huchler in their work on MCTS for Ticket to Ride [6] which uses an ϵ-greedy strategy. On each turn of the simulation, there is a $1 - \epsilon$ chance that a move will be selected using a heuristic strategy instead of one at random. This helps to guide the simulation along game trajectories that are more likely to occur in a real game. Again, we have implemented the default algorithm which selects moves at random.

Finally, when the simulation is complete, the difference between the agent's score and its opponent score is returned to be used as a payoff. The agent therefore is rewarded with a larger payoff if it defeats its opponent with a greater score difference. Different functions could be used that produce different behaviours, for example the agent could be rewarded a flat payoff for winning and corresponding negative payoff for losing. The payoff is returned and then given to the backup method shown in Algorithm 6.

Algorithm 6 backup method of the MCTS node

```
procedure BACKUP(Δ)
    this.visitCount += 1
    this.totalReward += Δ
    if parent node is not null then
        parent.backup(Δ)
    end if
end procedure
```

Starting from the newly expanded tree node from which the simulation was carried out, the payoff from that simulation is propagated up the nodes visited during the selection phase of that iteration. Each node stores the number of times it was visited during the selection phase and the total payoff it receives from simulations from that node, which allows us to calculate its average payoff. The root node of the tree has no parent at which the recursive backup method terminates and the iteration of the algorithm is complete.

When the computational budget has been exceeded, the *bestChild* method is applied to the root node to select the action that the agent should play. However, it is called with an exploration constant $C_p = 0$ since balancing exploration and exploitation is no longer required. Therefore, the action that results in the largest average payoff is selected. There are alternative ways to select the best action such as selecting the one that has been visited the most. This generally matches the previous approach but not always [10]. The performance difference is likely very small but using the highest average reward allows the reuse of the *bestChild* method by passing in an argument of 0 for the exploration constant, making the code cleaner. All the mentioned modifications in this section are just a few of the many possible modifications that can be applied to attempt to improve performance, which can be explored in future work.

9.2 Information Set MCTS

The second approach, Single-Observer ISMCTS, is more complicated to implement and will likely have a lower skill ceiling when compared to CMCTS. However, we hypothesise that this approach will produce more natural gameplay which is more enjoyable for the opponent since it will not be using information unavailable to the player. As described in Section 2.2, the algorithm works by sampling potential game states within an information set on each iteration, and this represents the major difference between the two algorithms. After a determinization has been selected at the start of each iteration, the course of the algorithm is practically identical to the standard algorithm apart from action selection.

The game contains two different types of hidden information: the trains cards that the opponent draws from the face-down draw pile and the tickets they secretly choose. All other information is public including the state of the board, the face-up train cards on the table and the number of trains each player has. An agent can also keep track of any face-up train tickets drawn by the opponent; it will increment its counters when it sees the opponent draw cards and decrement the counters when routes are claimed with a certain train resource. By subtracting the number of tracked cards from the opponent's total number of cards we get the number of hidden cards. When the current information set is determinized, the opponent discards their hidden cards and draws the same number at random to produce a possible hand state from the perspective of the agent. The opponent also returns their tickets to the draw pile and take the same number back at random. The result is a possible game state in the current information set [6]. If the agent is currently drawing cards in the game state, those cards must remain on the top of the deck so that it can "see" them during the simulations because drawn tickets won't be stored in some intermediate area when a player is choosing them. Algorithm 7 shows the method of the

game that selects a possible determinzation which is used on the root node game state before every iteration of the algorithm.

Algorithm 7 determinize method of the game state

```
procedure DETERMINIZE(perspectivePlayer)
    return train cards from opponent hand that are not tracked by perspectivePlayer to the draw pile
    shuffle draw pile
    draw opponent the number of cards they discarded
    return opponent tickets to draw pile
    if agent is currently drawing cards then
        pop top three tickets
    end if
    shuffle ticket deck
    return same number of tickets opponent discarded
    return any popped tickets to top of the draw pile
end procedure
```

After determinization, the algorithm continues as normal, only using legal actions consistent with the current determinization. The expandable actions need to be updated in each call to the tree policy with the actions currently legal in the state, minus the actions which have already been expanded in previous iterations [12]. A list of expanded actions is maintained, which is used to calculate which legal actions are expandable. The ISMCTS tree node inherits the CMCTS node since most of the methods remain the same.

Algorithm 8 treePolicy method of the ISMCTS node

```
procedure TREEPOLICY
    expandable actions = legal actions \ expanded actions
    call standard tree policy in the CMCTS node
end procedure
```

9.3 DDA for MCTS

We implement the four modifications proposed by Demediuk et al. [23] which require modifications to the *bestChild* method of the MCTS tree node. We create a method known as *getValue* shown in Algorithm 9 to store the logic for calculating node values separately which can be reused. It takes in a DDA variant and returns the corresponding node evaluation depending on the selected node. When the given variant is disabled, i.e. none of the four DDA variants, the standard UCT value is used, and the agent plays at its full strength.

Algorithm 9 getValue method of the MCTS node

procedure GETVALUE(C_p, ddaVariant, POSASThreshold)
  calculate averageReward
  calculate explorationTerm
  if ddaVariant is ROSAS or True ROSAS **then**
    return $abs(averageReward) - explorationTerm$
  else if ddaVariant is POSAS or True POSAS **then**
    return $max(0, abs(averageReward) - POSASThreshold) - explorationTerm$
  else
    return standard UCT value with exploration term negated for opponent states
  end if
end procedure

If a ROSAS variant is used, then the absolute value of the reward is taken such that a reward closer to zero yields a smaller node score. For all DDA variants, the updated *bestChild* method shown in Algorithm 10 minimises the value returned by *getValue*, so the exploration constant is negated. Note that the exploration term is outside the *abs* function so that it is maximised separately. For POSAS variants, the POSAS threshold is subtracted from the absolute value of the average reward and then set to zero if that is negative. The result is that all nodes that have a value within the POSAS threshold either side of zero are treated as having a score of zero, such that they have an equal chance of being selected.

Algorithm 10 modified bestChild method of the MCTS node

procedure BESTCHILD(C_p, ddaVariant, POSASThreshold)
  if ddaVariant is disabled **then**
    if current player in state is MCTS agent **then**
      return $\underset{c \in \text{ children}}{\text{argmax}}\ c.getValue(C_p, ddaVariant, POSASThreshold)$
    else
      return $\underset{c \in \text{ children}}{\text{argmin}}\ c.getValue(C_p, ddaVariant, POSASThreshold)$
    end if
  else
    return $\underset{c \in \text{ children}}{\text{argmin}}\ c.getValue(C_p, ddaVariant, POSASThreshold)$
  end if
end procedure

For the non-true variants of ROSAS and POSAS, the *ddaVariant* is set to disabled during the tree policy so that the DDA selection only occurs at the end of the algorithm to select the best action at the root node after the search tree is constructed.

9.3.1 Momentum

In the work by Demediuk et al. which originally proposed the methods for DDA for MCTS [23], they found that ROSAS achieved a winrate closest to 50%. However, during human games the agent would produce a noticeable rubber-banding effect where the changes in difficulty were noticeable. We propose a novel modification called momentum, which adds a new term to the UCT equation which penalises actions that cause a large change in state score. The aim of this modification is to prevent the agent from mirroring the opponent, on both blunders and brilliant moves. For example, if the opponent makes a large mistake, the momentum term would bias action selection to those that produce smaller changes in state score, promoting a more gradual change in difficulty. It is likely that this modification could reduce the effectiveness of the DDA in terms of closeness to the target 50% win rate, since the agent will adapt more slowly to the changes in opponent strength. However, it could produce a more enjoyable experience for the player. There may be scenarios where the agent becomes more exploitable if the opponent purposely plays suboptimally during the start of the game and then quickly increases the strength of the actions. The equation of the momentum term is as follows:

$$\text{momentumTerm} = m \times (\text{parent.averageReward} - \text{this.averageReward})$$

where m is the momentum constant, which controls the weighting of the term in the UCT equation. The constant can be set to zero in order to disable momentum. The momentum term is added to the UCT equation which is minimised. For true variants of DDA, the momentum term is also applied during the selection policy. We will be exploring the performance of this modification when applied to the four DDA algorithms in automated and user testing.

9.4 Heuristic Agents

Along with MCTS agents, we developed a number of agents that play the game according to various heuristics. The purpose of these agents is for testing against our MCTS algorithms. This testing will help evaluate whether MCTS, with no domain knowledge, is able to perform equally well or better than agents specifically designed to win Ticket to Ride games and evaluate the effectiveness of DDA.

9.4.1 Base Agent

The base agent outlines the base functionality that most other heuristic agents extend, adding their own unique strategy. This heuristic agent shares a lot of its ideas and design with the CFBaseAI agent we have described in Section 5.5.

The base agent's strategy is shown in Algorithm 11: at the start of every turn, it seeks out a minimum cost spanning forest that connects source-destination city pairs for each destination ticket owned by the player. We call this spanning tree the *optimal path* for brevity. Then, it aims to claim a route that has a good balance between cost and the score it gives. If the agent lacks sufficient resource cards to claim any route on its chosen path, it will draw cards that are most needed to complete the path. In the case where the agent cannot find a viable path that connects the destination tickets (either due to having already completed the path, or having all paths blocked by the opposing players), the agent will consider two options: if it still has a substantial amount of train cars left, it will opt to draw more destination tickets; otherwise, it will choose an unclaimed route and claim it, ultimately resorting to drawing train resource cards if unable to claim any routes.

Algorithm 11 Base Agent Take Turn

```
procedure BASE AGENT STRATEGY
    identify an optimal path connecting all the agent's tickets
    if no path found then
        if sufficient amount of train cars left then
            draw destination ticket
        else
            if any route is claimable then
                claim the route
            else
                draw cards from the pile
            end if
        end if
    end if
    if any route on the optimal path is claimable then
        select the best route to claim and claim it
    end if
    identify the best cards to draw and draw them
end procedure
```

The base agent determines the best path, route, or card draw using a heuristic cost function.

In order to identify the optimal path, the agent first evaluates all routes individually, favouring lower costs. Subsequently, it considers all paths that connect each of the destination tickets; finally, it combines the paths of separate tickets into larger paths, for each combination. Ultimately, it iterates through all of these paths by calculating the total cost of each route and chooses the cheapest one.

As the agent reevaluates the optimal path at the beginning of each turn, it will adjust the path if any routes from the previously calculated path have been claimed by the opponent. The updated path will include only unclaimed routes and those claimed by the agent, allowing the agent to have a concrete objective throughout most of the game. Upon completing its tickets or having all paths to complete its tickets blocked, the agent will evaluate all available routes. It will try to claim the route offering the highest score; if there are no claimable routes, it will draw resource cards from the pile.

To decide which cards to draw, the agent calculates all the cards required to claim all the routes on the path. It then prioritises choosing the most needed cards from the face-up options, giving preference to locomotive cards whenever they are available. If no face-up cards are needed for the agent's path, the agent draws cards from the pile. This straightforward heuristic ensures that the agent consistently draws cards essential for claiming all routes in its selected path, enhancing its efficiency.

9.4.2 Random Smart Agent

This agent has a parameter that dictates the probability of taking a random move; at the start of each turn a random number is generated and based upon that value, the agent either performs exactly as the base agent would, or takes a random, yet smart action. This agent shares some similarities, primarily calculating the optimal path but incorporating an element of randomness, with CFRandomAI from [14].

Random smart agent always identifies the optimal path according to the behaviour of the base agent. However, it incorporates an element of randomness into its decision-making process, while still somewhat adhering to the base agent's strategy. Instead of selecting the best route to claim, according to a cost function, it may opt to randomly choose a route from the chosen path to claim. If no route on the path can be claimed, it resorts to drawing random cards, possibly a mix of drawing from the face up cards and from the pile.

9.4.3 Random Base Agent

Similar to random smart agent, this agent has a parameter that determines the probability of randomness; however instead of choosing a random move that still follows the base agent's overall strategy, it will pick a completely random move. Analogously to random smart agent, this agent shares the same similarities with CFRandomAI.

9.4.4 Hoarder Agent

This agent's strategy revolves around only drawing train resource cards before making any route claims, similar to the strategy Section 2.3.3.2. It determines when to transition from drawing cards to claiming routes by checking the percentage of cards needed to complete the entire chosen path the agent has accumulated. Once the threshold has been reached, it will revert to the base agent's strategy. Since the base agent always claims routes whenever possible, this agent will claim every route it can from once this threshold has been reached until it can no longer, before returning to hoarding resources again.

9.4.5 Hoarder Ticket Agent

This agent is the same as the hoarder agent, but it draws an additional ticket in its first turn. This results in the agent hoarding cards for several additional turns, since the threshold will be higher.

9.4.6 Pile Draw Agent

The heuristic of this agent is the same as base, but it will only draw cards from the face down pile. This strategy is interesting, as the agent relies only on random chance when drawing train cards. This allows the agent the possibility of drawing a locomotive card along with a coloured train card, or even 2 locomotive cards. Additionally, it does not give any information to the opponents which is beneficial to the agent. However, the drawback is that the agent is not guaranteed cards it needs to claim the routes on its chosen path.

9.4.7 Pile Ticket Hater Agent

The Pile Ticket Hater agent will not draw additional destination tickets. It will perform the same as base until it can no longer find a path connecting all destination tickets. Subsequently, it will never choose to draw more destination tickets, it will always simply try to score points by picking the longest route it can claim and will draw cards if it

cannot claim any route. Additionally, this agent only draws cards from the pile, making use of the previous agent's heuristic.

9.4.8 Villain Agent

This agent primarily focuses on impeding its opponent's progress by blocking any routes the opponent may attempt to claim, closely following the strategy outlined in Section 2.3.3.3. The Villain agent prioritizes claiming routes that are adjacent to those claimed by the opponent. If such routes are unavailable, the agent adheres to the standard behaviour of the base agent.

9.4.9 Greedy Random Agent

This agent does not use any of the heuristics from the base agent. It will simply claim a random route from among the list of routes it can claim, matching the strategy outlined in Section 2.3.3.1. If it cannot claim any routes, it will draw cards randomly. This extremely simple greedy heuristic still proves to be relatively strong, especially in smaller maps, as it will always claim routes if it can and so it will always gain some points. However, it claims routes without taking into account its tickets, so will likely always have negative ticket score contributions.

9.4.10 Fully Random Agent

The fully random agent picks an action from a list of all available legal actions uniformly at random every turn. It will be interesting to see how MCTS is able to adjust difficulty against a bot that has no overarching strategy.

9.4.11 Mixed Agents

A mixed agent is one that is capable of swapping its strategy in the middle of a game, potentially resulting in significant shifts that strongly influence opponents' perceptions of the agent's skill level. Furthermore, swapping of behaviour can imitate more irrational human play behaviour. The primary purpose behind creating these mixed agents is to evaluate how the dynamic difficulty adjustment AI would respond and adapt to changes in gameplay style.

Chapter 10

System Design

Our second objective concerns the development of an application that allows a human player to play a version of Ticket to Ride against our AI. In addition to being a vehicle for users to experience our DDA agents in an accessible, intuitive and enjoyable manner, elements of this game application are also used for AI development and testing.

To avoid confusion when referring to our adaptation of Ticket to Ride and to give the project its own flair, we have named the game application we have developed *Coupon for Travel*[1].

This chapter is focused on the system architecture and design of the game application, detailing the design of core components and how they integrate, as well as how the design aligns with the project's objectives. Our scrum-based development approach means an interplay between design and implementation. The design is validated by implementation, which in turn encourages discussion and adjustments to system design. This to-and-fro ultimately should settle to a good outcome that takes into account constraints, akin to any other dialectic method. As a result, however, it is hard to separate these two stages of development in a chronological manner, since design has evolved during development.

[1]Whilst the name Coupon for Travel sounds as if it is some legally dubious knock-off brand, we have not used this naming as a way to bypass copyright restriction. This name is mainly used to differentiate our own twist on the application, whilst not shying away from being tongue-in-cheek and self-aware. As we outline in Section 6.1, we can use the rules and name of Ticket to Ride so long as it remains under fair use.

10.1 Coupon for Travel

Our game is restricted to two player games. Whilst we have set up the infrastructure for up to five players, allowing the game to be extended if required, many of our game AI work in a two player adversarial setting. Most notably our DDA agents, which are much more complex in a multiplayer setting Chapter 9, are designed to play against a single opponent.

For the user application, two maps have been chosen and implemented to be playable. The smaller of the two is the Yang et al. map, which allows for very quick games and is largely used for testing purposes; the larger is the map from Ticket to Ride New York, which is a small map overall but an appropriate size for 2-player games. The 1st edition game rules are implemented, which means additional game mechanics such as tunnel connections or hotels are not used. In addition, following the two player set up rules, we have not implemented double connections between cities. Lastly, in accordance with map size constraints, longest paths do not grant additional score at the end of the game as it may skew the total score drastically.

Additional maps have been used in testing, but don't have a GUI developed for it. See 12.6 for a formal description of each map used for testing.

10.2 Architecture

The Coupon for Travel system is composed of four components: a *game environment* which encapsulates all game mechanics of Ticket to Ride and maintains game state, *agents* whether human or AI which affect the game environment, a *user application* which consists of a human-game graphical interface that we present to users and finally a *testing application* which is used for internal validation and insight to agent behaviour. The high-level interaction of these components can be seen in Figure 10.1 below.

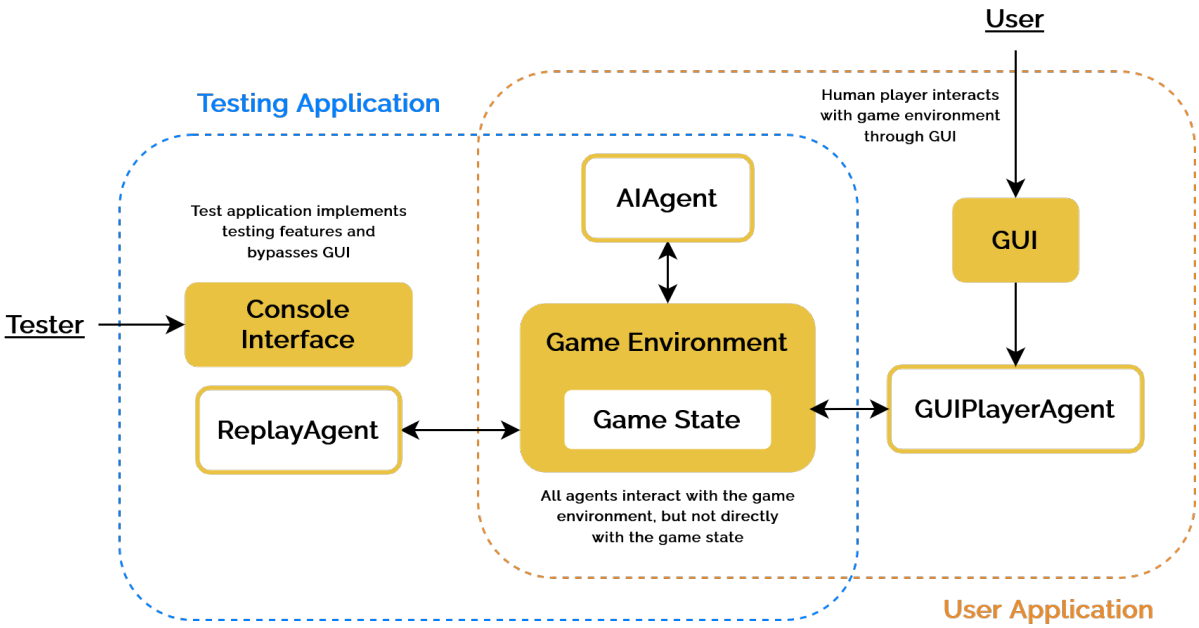


Figure 10.1: Interaction between main components of the system.

In summary, the game environment can be seen as the "core" system. Human and AI agents alike can interact with the game environment, but not directly with the game state. Human players interact with the game using the user application - the SFML implementation of the game's front end - and the user application interacts with the game environment itself using the `GUIPlayerAgent`. The testing application largely intersects with the user application, but is used instead for developers to test the game, which is helped by the testing application's record and replay functionalities.

10.3 Design Principles

Our design decisions were motivated by several software design and software engineering principles.

Firstly, we aimed to produce a generalisable system. The primary motivation was to allow different combinations of agents and maps. However, there are additional benefits to generalisation, such as improved code reusability, maintainability, and scalability. Whilst code designed for special cases is simpler to write and therefore takes less design and development time in the short run, it balloons.

We aim to follow object-oriented design principles for much of the game, such as encapsulation and inheritance, polymorphism, and abstraction. In addition, we followed some object-oriented design patterns, such as the builder, decorator, servant and command

patterns.

10.4 Game Environment

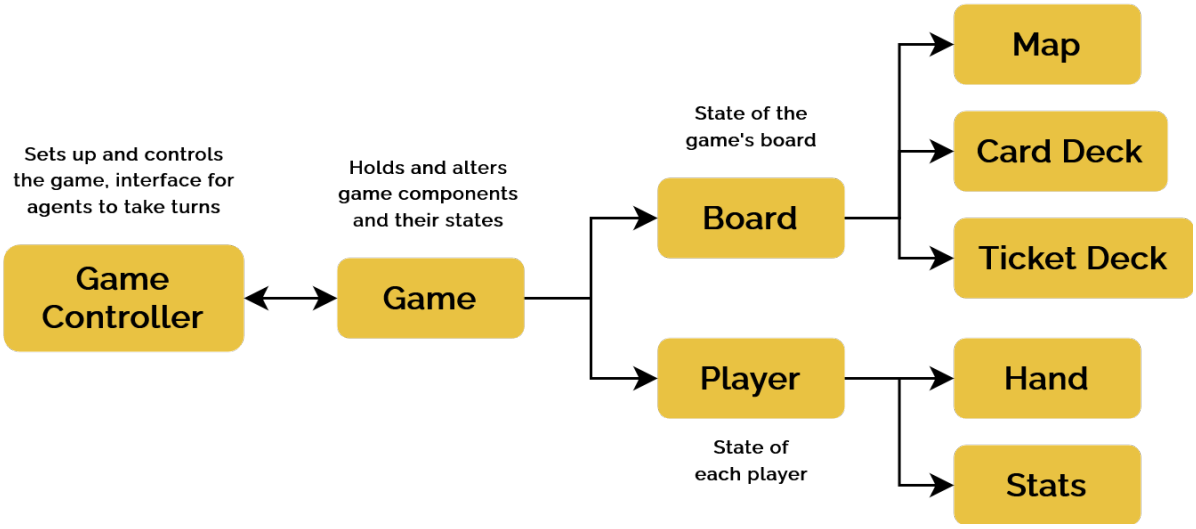


Figure 10.2: Architecture of the game environment.

The game environment is the foundation which all other components in our system build upon. The game environment was identified early on as a core deliverable — in fact, it is our first deliverable. It implements the rules and mechanics for the chosen game, stores game states, and provides operations players can perform to alter the game state. It is designed to provide a common interface for the human players and artificial intelligence agents alike, and is responsible for receiving, executing and passing on relevant information of the game state.

10.4.1 Game State

The game state refers to all the relevant information about the current turn of a Ticket to Ride game. More specifically, the game state contains the map layout as well as current route ownership, the train card and ticket resource decks, resources held by each player (*player state*) and the current turn and player. It also is responsible for certain statistical functions, such as calculating and tracking the scores of each player. The game state is controlled exclusively by the game environment (although its copies can be modified by agents).

A key design consideration on the game state was that game states must be able to be efficiently duplicated. Monte-Carlo tree search based agents have to iterate through game

simulations, where often thousands if not millions of game states have to be searched in order to make a decision about what action to take.

Individual game resources, such as tickets, have been designed in an object-oriented line of thought. Each resource should be treated as an individual object in its own right, and have been designed to be passed around from board to player as and when needed. This object-oriented line of thought helps to ensure no unintended behaviour, as well as mimicking the actual game behaviour. As an example, tickets must be digitally handed from the board to players, and collected by the deck at the end of the game.

10.4.2 Actions

The game environment is also responsible for performing game moves, or *actions*. This means that the game environment has to be able to handle every possible action a human player might do playing the board game. Such actions include setting up game resources, drawing and discarding train cards, claiming routes and scoring the game.

Drawing from object-oriented design principles, and following on from lessons learn in our feasibility study, we decided to use an "action system", whereby action objects which encapsulate all the relevant information for a given action in one place. For example, a claiming route action should include the name of the two cities and the resources used by a player to claim the route. The beauty of this system is that the same action system can be for several functions, such as for querying the validity of an action or to perform this action to alter the game state itself. Furthermore, actions can be sent by agents to the board, which helps to detach the game environment and agents themselves.

In the original game, picking cards from the table and drawing tickets are two-step processes. We decided to simplify the ticket drawing by not replacing the first train card the player selects from the table. This reduces the complexity of the game logic and helps reduce the size of the game tree. However, it is not possible to reduce ticket drawing since a player must choose to draw the tickets and then select which ones they want to keep. To allow MCTS algorithms to reason about ticket drawing, the process had to split up into two separate actions. To draw tickets, MCTS first selects a draw tickets action as its best child. The algorithm is then repeated and the agent selects between all possible legal combinations of tickets it can keep.

10.4.3 Maps

We designed our game environment so that it can readily accept differing Ticket to Ride maps and resource allocations. This is so that we can test the game AI in different scenarios, and provide more value to the user by allowing more replay-ability. Different maps have differing colour selections, cities, routes and tickets. We store this information in the game state, and decided to use a board constructor object based on the builder design pattern [41] to set up a game based upon a given input game configuration. Whilst this adds more complexity to the design, the generalisation ensures that the code is modular and expandable.

10.5 Agents

Whilst we do not discuss specific AI strategies in this chapter (see Chapter 9 instead), it is important to see how these agents fit in with the wider application.

The primary design decision is to design agents to appear indistinguishable from each other in the game state, regardless of whether it is a human player using a graphical interface, a Monte-Carlo tree search AI, or an agent replaying a game. This indistinguishably allows the introduction of more types of agent easily. Furthermore, it simplified recording analytics for agents for test purposes. In practice, this generalisation allowed the easy integration and combination of agents of differing forms of input. The chosen agent data is presented alongside a game configuration to specify the precise nature of a given game run.

Information access is another important design decision for agents. Agents should not have the power to look at the information of other players that they are not allowed to. With the exception of our cheating Monte-Carlo tree search based agents, which are intentionally able to infer implicit knowledge about its opponent's resources, this should be forbidden. In turn, agents are designed to be able to query the game state whether a given move is valid, view its own resources as well as view shared information such as the table train cards and current route claims.

Agents have been designed to have no direct control over the game state: only the game environment should be responsible for affecting the game state, and has the final say. This prevents illegal modification and manipulation of data by agents, and improves safety and system robustness by allowing the game environment to check whether the performance of an action could lead to undefined behaviour. Agents request to perform an action, and the game environment can then refuse or accept this request and enact this action

appropriately. The game environment is also thus responsible for controlling and alerting agents to know when it is their turn.

In this way, agents are detached from their player states: they cannot directly control their own resources. This might sound strange, but in practice it enforces the detachment from the game state and in theory allows the ability for other agents to switch out mid-game to join or leave the game, much like how you can let a friend take over your turn without taking the resources with you.

10.6 User Application

The user application is the piece of software which we have produced that is designed to accomplish the objective of allowing a player to play Ticket to Ride against developed dynamic difficulty AI. In particular, this application has been designed to be accessible, intuitive, and enjoyable to interact with.

The user application consists of the game environment, a user interface as well as a variety of agents for users to play against.

The user interface is designed to be able to parse information into actions and send them to an appropriate human player agent, which is then able to send actions to the game environment, which in turn can affect the game state. Furthermore, the user interface is designed to receive information from the game environment and display it to the player.

10.6.1 Platforms

Our objective also specifies that the application should be able to be easy to install, across several platforms. To ensure this cross compatibility, we have developed the system across Windows (10/11), Mac (M1 based) and Linux (Ubuntu distributions) and verified compatibility for these systems. To ensure ease of installation, we took care to reduce the amount of software dependencies and choose installation software carefully.

10.6.2 User Interface

We decided early on (as our third deliverable) that the users would interact using a graphical interface as this is a more intuitive means of interaction than, for instance a command line interface. As Ticket to Ride is a board game, the various resources and game states are better represented as game sprites than as strings. One case in point is the game's map. To represent a map you have to represent the structure of the graph

of cities, represent the colour and length of each route as well as ownership. Not only is it harder to interpret written text than image data, it is almost impossible for players to strategise on larger maps due to the large amount of information processing the brain has to take to visualise and make sense of this stringified data.

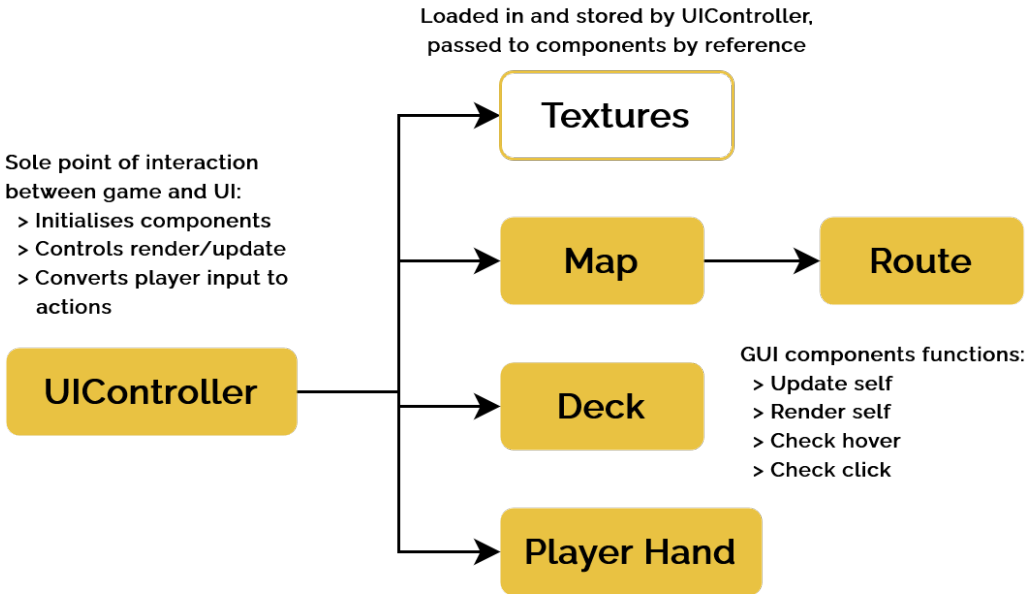


Figure 10.3: The interaction between GUI components.

Similar to the game environment, the user interface would have a single point of access, being the `UIController`. This component should handle texture loading and initialisation of all GUI components. Furthermore, it should be responsible for propagating render/update calls down to and actions back up from individual components.

10.7 Testing Application

It is important to validate our application and agents. When testing our game environment, we need to verify the system's robustness and hardware performance. In addition, to test our DDA agents, we need to validate their behaviour works as intended and note statistics such as win rate against other agents.

The user application is not best suited to these testing criteria. For starters, the GUI adds overhead to game runs. In addition, it limits the available maps we can play against to those that have had a graphics pass. On the other hand, the testing application tracks information and requires unnecessary features for an end-user that could bloat the installation of the user application.

Thus, we remove the interface layer, and add additional testing specific features that allow us to collect several pieces of analytical data that we describe in Chapter 12, to create a test application. The test application uses the same shared game environments and agents as the user application, but is designed strictly for non-human agents.

The test app is designed as a command-line application, which can be run with different flags to promote different behaviour. Flags we have considered include a help flag, game configuration flag and number of game runs. The flag based system allows for different test cases to be trialled from a script, which helps to automate testing.

10.7.1 Reproducibility

Early on, game run reproducibility was identified as a very important feature to validate game behaviour.

The board and several agents require random behaviour. For example, the train decks need to be shuffled, and a random agent may want to take a random choice of all available actions. But this random behaviour means that it is nearly impossible to reach a same game state and setup on any two runs.

We thus took great consideration in handling random devices, to ensure they could be seeded (and this seed be tracked) to ensure game repeatability.

10.7.2 Recording and Replaying Game Runs

We extend game run reproducibility, by recording each turn taken by various agents. This allows a static analysis of agent behaviour. Since we had decided to record games, being able to run games from this recording was a natural extension.

10.7.3 Multiple Runs

For ease of testing, we allowed simultaneous game runs on the same object, without having to reload the map and application. This requires the game state to be reset. This functionality provided additional value to the game app, as it allows end-users to play infinite games without reloading the application.

Chapter 11

Implementation

This chapter describes the practical implementation of the project. The focus will be on technical details of the software engineering aspects of the project, rather than research- and design-focused like the previous chapters.

11.1 Game Environment

As discussed in Section 10.4, the game environment consists of a system that encapsulates the game state and core game mechanics of Ticket to Ride. An overview of the architecture can be seen in Figure 11.1, and we will expand on each component below.

GameController The game environment is contained within a *game controller*: either the base controller, `GameController`, or extensions that act as wrappers on the game environment `RecordGameController`, `ReplayGameController` and `SaveGameController` which have been used for testing (see Section 11.4.3).

Each game controller is responsible for setting up, maintaining, manipulating and resetting each game state and agents. Crucially, it provides an interface to interact with agents, to request actions from an agent when it is its turn and query and perform these actions to manipulate the game state. As the name suggests, the `GameController` is built around the controller design pattern so that all requests will be handled by a single handler.

`GameController` comprises a game state (`Game` object), a collection of agents (`Agent` objects in a container), a player order, turn tracking information and finally, score data for analytical purposes.

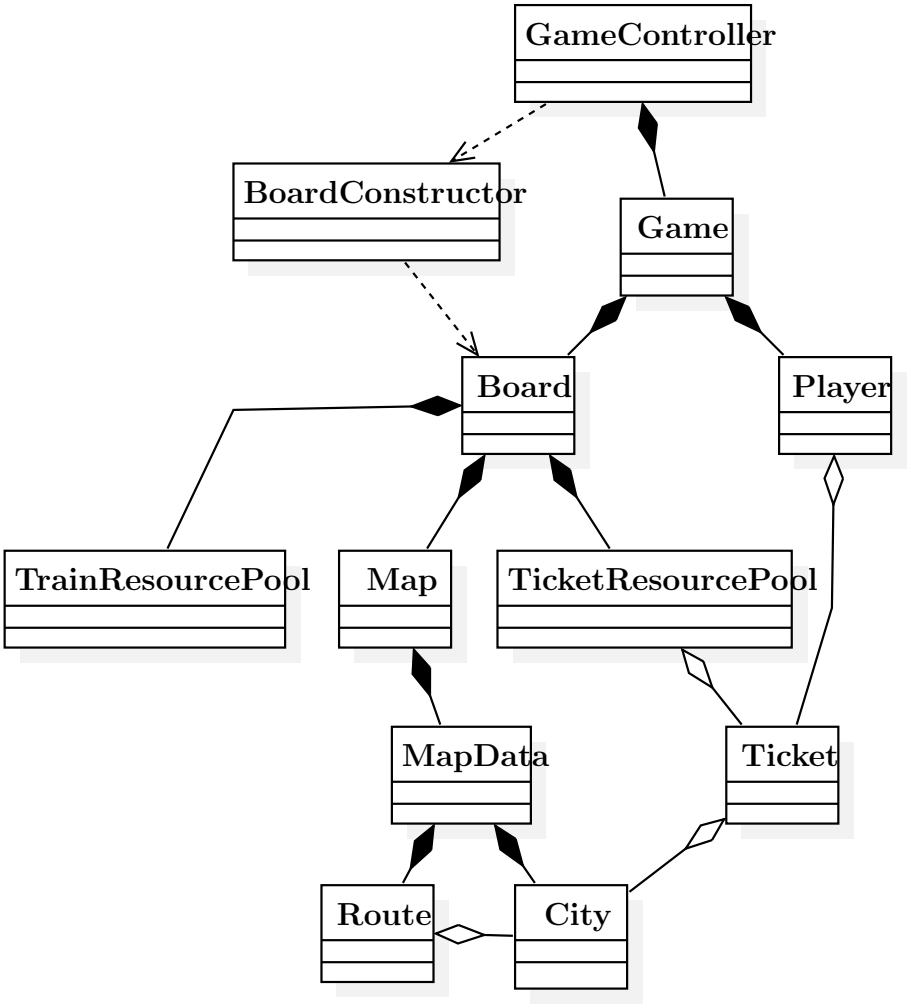


Figure 11.1: UML Class diagram for game environment.

The user application largely only calls `GameController::doNextTurn` to perform a next turn: the controller handles the rest. The test application has even less direct control and leaves everything to `GameController::playConsoleGames`, which then runs the number of games as specified.

Game The `Game` object contains the game's components, its board (`Board` object) containing Map and game resource, its player states (`Player` objects) and variables tracking the current player and turn, and whether the game is in its last round or is in a terminal state. It is responsible for performing the actions `GameController` sends to it through the function `Game::performAction`, and querying that these actions are valid with `Game::queryAction`. These functions propagate down to an appropriately responsible object; for example, `Map` handles the claiming route actions.

Also, `Game` objects provide useful information to agents. Most crucially, it provides a list of valid actions for a given player with the function `Game::getPossibleActionsForPlayer`,

which is used by all of our AI agents.

Board Maintaining well encapsulated code is an important object oriented design philosophy, and as such the `Board` object is separate from player states. It is comprised of a `Map` object which contains map structure (in a `MapData` object) and current route ownership, a train resource pool which handles train draw and discard piles and table cards as well as a ticket resource pool which is responsible for ticket draw and discard piles.

Since routes and tickets reference cities and are all interconnected, board construction is complicated. This prompted the creation of a `BoardConstructor` class, which loads city, route and ticket data from CSV files, generates their respective `City`, `Route` and `Ticket` objects that contain correct cross-references, and load these resources into their responsive container whether the `MapData` or `TicketResourcePool`. This use of the builder design pattern allowed us to create this complicated object.

Agent, Player and PLAYER The `Player` class should not be confused with the `Agent` class: `Agent` is the class through which AI and player agents interact with the game, whereas the `Player` class only holds the player state of each player. The `Agent` and `Player` are detached so that agents have no direct control over their resources, and have to go through the game environment in order to affect the game state as we reason in Section 10.5.

A `PLAYER` enumerated type stores a unique identifier (such as Alpha, Bravo or Charlie) for each distinct player. These link together `Agent` and `Player` and are a computationally cheap way to keep track of a given player and are used for record purposes.

11.1.1 Console Game

During implementation, we developed a developer-facing console interface to validate and interact with the game environment before starting work on the graphical interface. The console game allowed the AI to be integrated with the game environment more quickly and allowed for simultaneous development of GUI and AI agents. We will refer to this earlier state of console user interface and game environment as the *console game*.

Developers are able to manually debug and test the console game using its console. This includes printing the game state at each turn and the creation of a console line agent which a human can input their own actions. We do not show this to users, however, and mask this behaviour under `verbose` and `clear` flags. To run some console games

we use the following function: `GameController::playConsoleGames(n_games, true, true)`

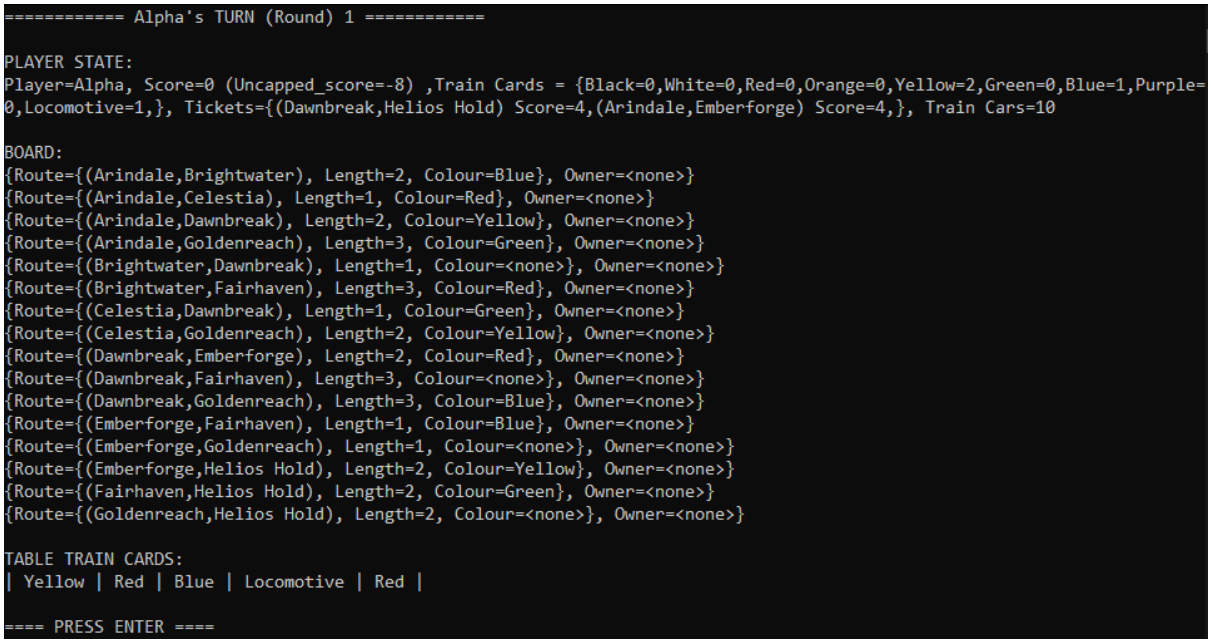


Figure 11.2: Interface of the console game, with player state, map, and board display.

11.1.2 Actions

The game environment implements the aforementioned action system using a number of 'PlayerEvent' objects. It is these events that contain information relating to a player's action, and are sent by player agents to 'GameController', which queries and propagates the event to board components. There are four general categories of events: ticket draw, card draw, route claim, and bad event. The inheritance hierarchy is illustrated in Figure 11.3. Our action system is based upon the command design pattern.

- Ticket draw: Consists of `DrawTicketPlayerEvent`, which signifies the player's choice to pick up tickets, and `DrawAndDiscardTicketPlayerEvent`, which performs the draw and discard actions. This is a product of the two-step ticket draw action.

- Card draw: Consists of four events that describe possible scenarios for train card draw - drawing two face-up cards from the table, two cards from the deck, one from each, or one locomotive from the table.

- Route claim: Consists of two versions of the same event - the player could either claim a route by the names of the cities it is adjacent to, or by the route's ID.

- Bad event: Used to identify invalid or erroneous events.

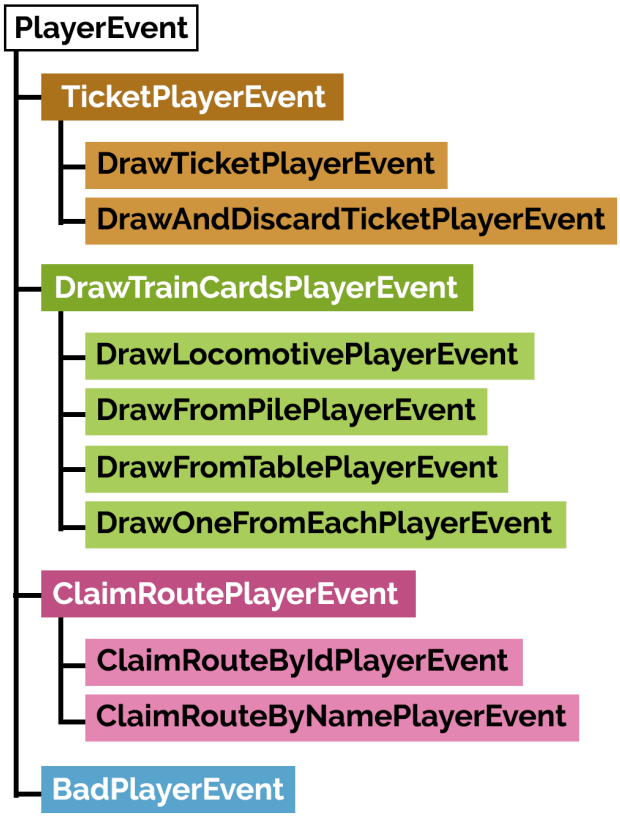


Figure 11.3: PlayerEvent inheritance hierarchy and list of all actions. Valid actions are in black text. `PlayerEvent` itself is an abstract class.

11.1.3 Setting Up and Resetting Games

We discuss the creation of Agents and game configuration in Section 11.4.2.

Game set up is handled by the function `GameController::setUpGame`. This resets the game state, sets up agents and links game objects to them, deals train card resources and initial ticket turns. The initial ticket draw is different to additional draws, as according to TTR official rules, you must keep two tickets on the first game rather than one for normal ticket draw.

We outlined the need to run several games without recreating a given GameController object in Section 10.7.3. To do this, the Game uses its reset functionality, which propagates down through the structure. Due to the object-oriented way we designed our game state, players need to return resource objects back to the board, and resource piles must be reset to their starting position. We applied special care to ensure that ticket objects were not lost or duplicated.

In `Map::reset()`, route ownership is reset, in `TicketResourcePool::reset()`, players hand over their `Ticket` objects to the discard pile which are then reshuffled into the draw pile and in `TrainResourcePool::reset()` the existing draw pile, all player cards and table cards are placed into the discard pile, which is shuffled into the draw pile and new train cards are laid on the table again.

11.2 Graphical User Interface

This section concerns the implementation of the Graphical User Interface (GUI), the front end of the user application. We will highlight the evolution of the GUI itself from illustration to SFML implementation, as well as discuss its accessibility features.

An initial sketch for the GUI, shown in Figure 11.4, was made by referencing the interface of the official TTR games as well as digital adaptations of other board games. The main portion of the screen is taken up by the map display, and residing in the left and bottom sidebars are the card decks and player hand.

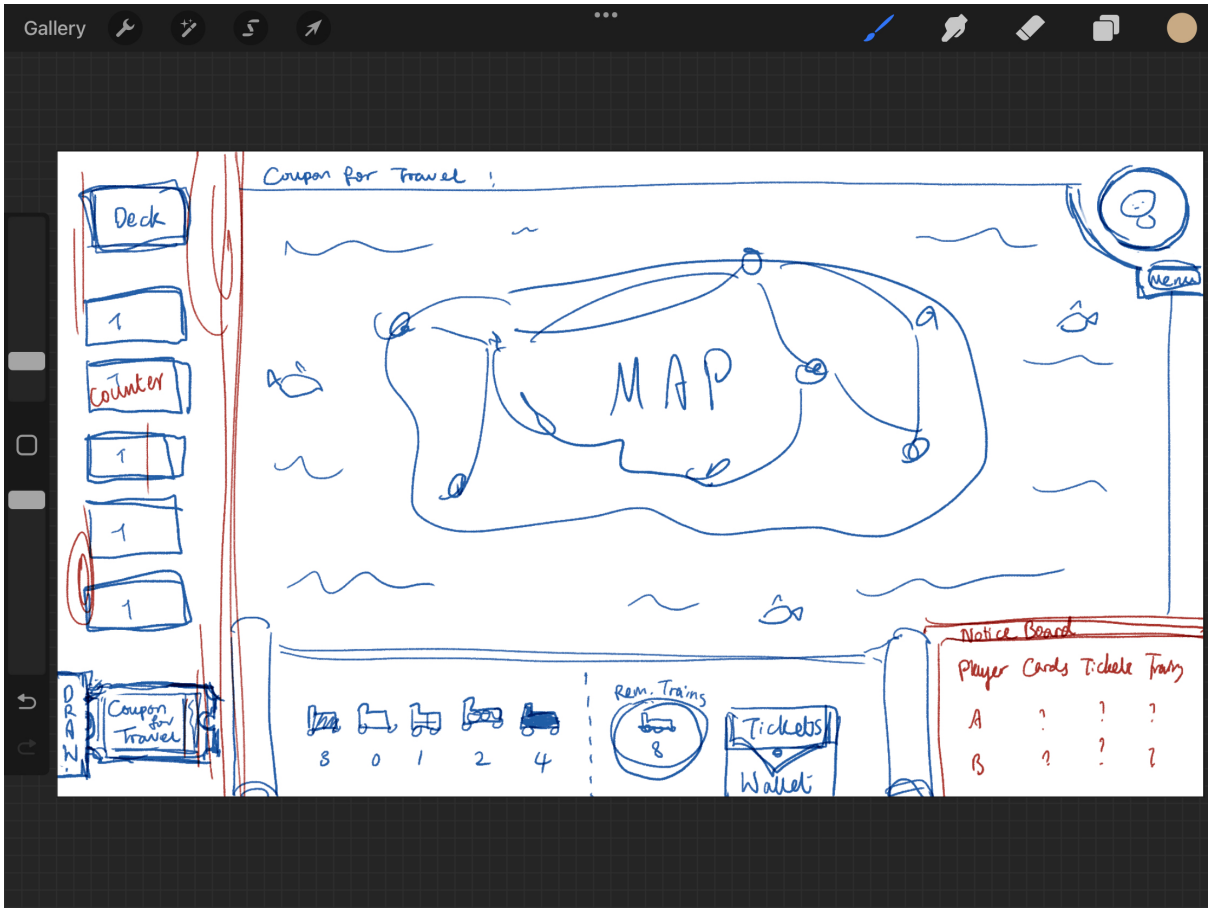


Figure 11.4: Initial sketch of the GUI.

As the SFML development commenced, the game was prototyped according to the design sketch, with a drawn background but mostly placeholder sprites for resources, and a text representation of the map much like in the console game. This prototype can be seen in Figure 11.5. At this intermediate stage of the GUI, we make sure that console game map print out can be translated to the SFML window, and that rendering works as expected.

The GUI rendering loop is illustrated in Figure 11.6. Rendering is done at the frame limit of 60 times per second, and works as follows. The main program holds a number of internal objects, such as `GameController` and `UIController`, and boolean variables relating to the current state of the game. At the start of a rendering loop, the state variables are checked to decide whether to render the menu, the game, or the end screen; then, the current mouse position is checked for hover animations; if a click event is registered, the mouse position is passed to the `UIController`, which reports back whether the player made a meaningful interaction via a `PlayerEvent` - if so, the game state is updated by `GameController`, the window is rendered, and the loop repeats.

Figure 11.5: Early prototype of the GUI.

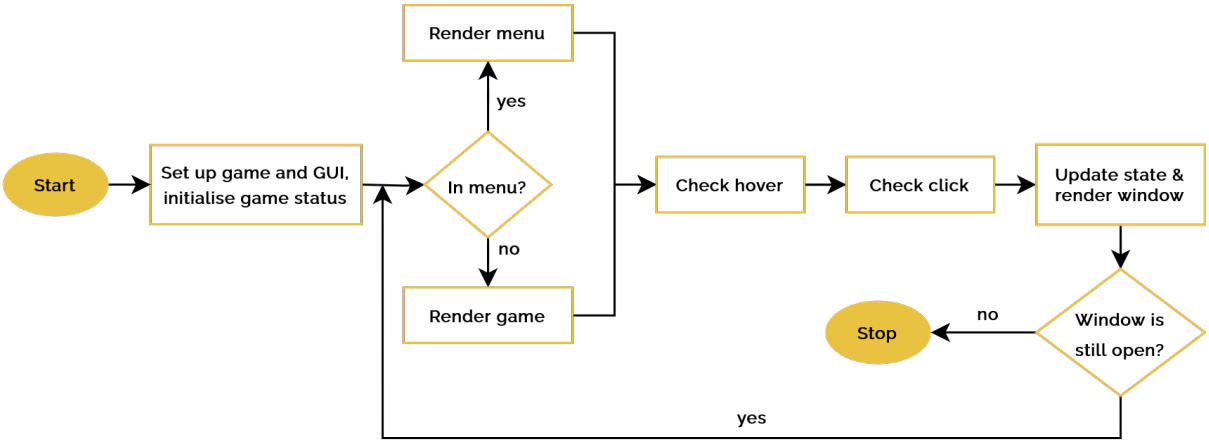


Figure 11.6: GUI rendering loop flow diagram.

Once it was verified that SFML components work as expected, a number of game sprites were added to create a more polished final game interface. The game sprites were drawn by hand for a simple but cohesive look. One downside for this approach is that it is not as flexible as vector graphics or 3D models, which are able to scale up to larger screens while maintaining the resolution. However, 2D sprites are by far the most cost-effective option when it comes to storage and performance. To further ensure rendering is as computationally efficient as possible, sprite textures are only loaded once by the `UIController` and passed to other components by reference, eliminating unnecessary duplication of resources.

Figure 11.7: Example media created for the GUI.

The final GUI is shown below in Figure 11.8. It aims to display all game resources and information on the screen and minimise the need for cross-referencing several screens. It is possible to fit all this information in without making the window feel too cluttered

because TTR is a relatively simple game without a huge amount of information, especially given that a smaller map is used.

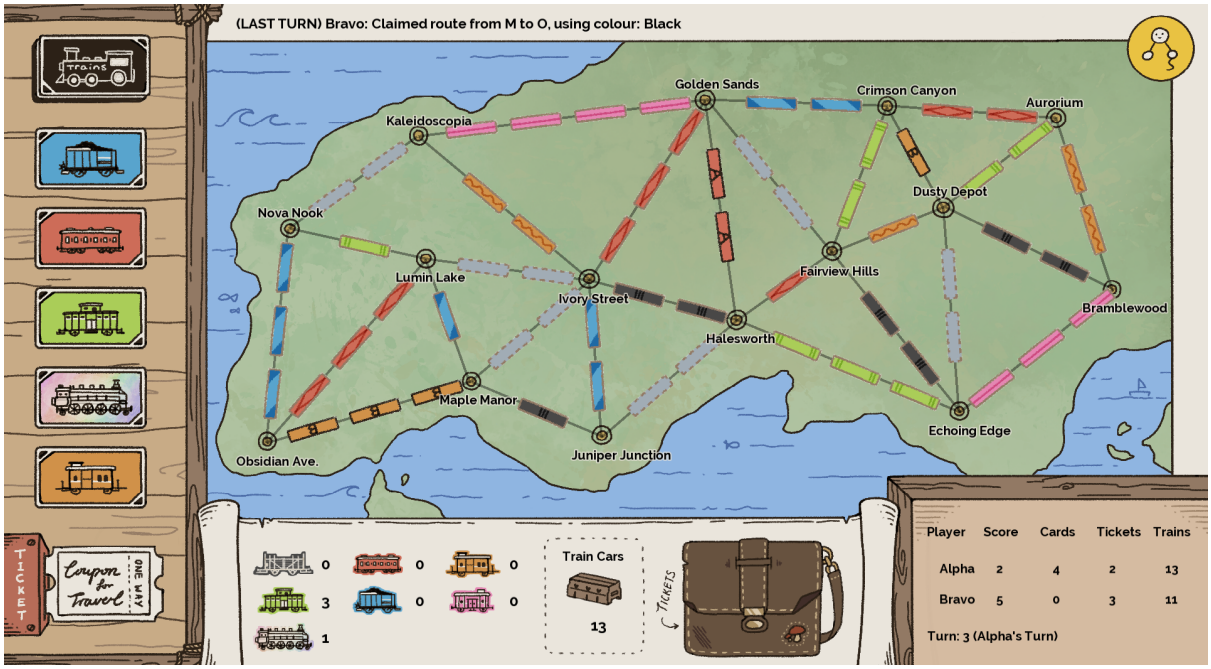


Figure 11.8: Final GUI.

The only information which the player is allowed to access that is not shown in the main screen are the MCTS statistics and the ticket wallet, which can be activated by interacting with their respective widgets. The MCTS statistics shown in Figure 11.9 is not really a part of the game, but entirely for the user's information; thus, there is no reason for it to take up screen space. Instead, it can be found via clicking on the top right icon, which represents the AI opponent. The player is intuitively informed of the correlation between the icon and the opponent, because the same icon also displays a status marker when it is the opponent's turn.

Ticket display is not immediately shown, but instead accessed by hovering over the ticket wallet. This provides players with a new interface, where they can scroll through the tickets they own. This interaction is demonstrated in Figure 11.10.

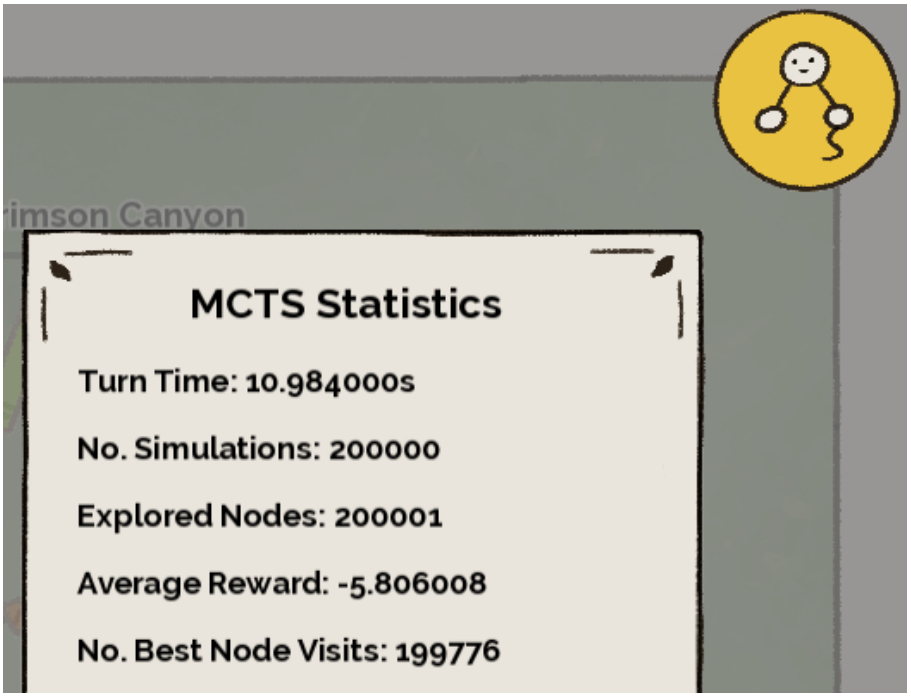
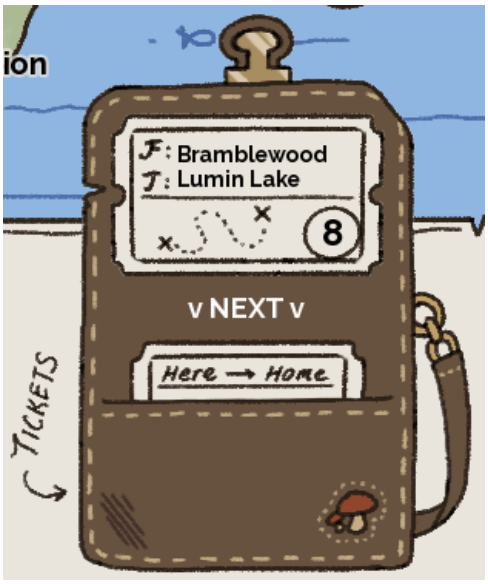


Figure 11.9: MCTS statistics expands on click.

(a) Ticket display when not hovered over.



(b) Tickets are displayed on hovering the wallet, and can be scrolled through using next button.

Figure 11.10: Ticket wallet interactions.

Before we implemented actions, we checked that the GUI was correctly updated according to game state, comparing GUI to console output for an example game. Once we were sure the display was updated accordingly, we then allowed users to modify state through the GUI.

On click, the `UIController` propagates the mouse location to each UI component in a prioritised order, and each component checks their own subcomponents' bounding boxes to see whether the player clicked on something meaningful. If so, they return the player's action to the `UIController`, encoded as a vector of integers, of which an example is shown in Figure 11.11. GUI interactions are sent encoded and interpreted by the game environment by using the same record format used by the `ReplayAgent` (see Listing 11.5). Finally, we present an overview of the GUI components and their hierarchy in Figure 10.3.

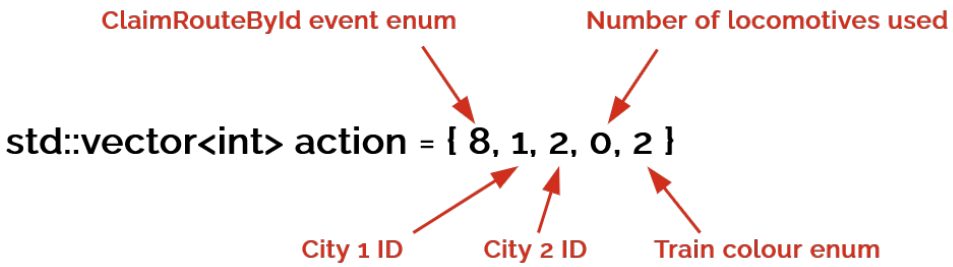


Figure 11.11: Example action vector returned to `UIController`, where the GUI player claimed a route between cities 1 and 2 using red cards.

11.2.1 Accessibility

In the design of the application, as described in Section 10.6, we have taken into account different disabilities and accessibility preferences. In particular, we aim to ensure the game interface's visual elements are suitably accessible to those with visual impairments, such as forms of colour blindness. One potential accessibility issue is the colour-based routes on the game map; for those who might struggle to differentiate train colours, patterns are used to distinguish routes of different colour. The game also uses a clear, bold font throughout to maximise readability. Visibility is further increased by making the text black with white outline to help the text show in front of a textured background with a mixture of light and dark colours.

We also provide a reference manual for playing the game, which can be found in the Appendix. For users who might not be familiar with the game itself, or those who have played the game but are not familiar with its digital adaptation, the manual aims to familiarise users with our interface so that they are able to interact with the game without issue, as well as summarising the general game rules in condensed form.

11.3 AI Agents

We implemented all the AI agents and their extensions as we described in Chapter 9. All of our developed agents fall into the inheritance structure visualised in Figure 11.12. To ensure indistinguishably as explained in Section 10.5, AI agents implement a base `Agent` interface, which is the lens through which the `GameController` interprets each of our agents. This includes the most important (pure virtual) function, `Agent::takeTurn`, which must be implemented by any concrete agent implementation, that `GameController` uses to receive the `PlayerEvent` it is meant to perform on the game state for the given player on a given turn.

Further sub-categorisation is used to keep code modular and allow for more specific game environment set up from configuration. The sub-categorisation of `AIAgent` distinguishes AI agents from human and replay agents and has the requirement of needing a reference to the game state, whilst the further categorisation of `TwoPlayerAdversarial_AIAgent` has a requirement of a reference to the opponent's ID.

11.3.1 DDA Agents

We implemented pseudocode as outlined in Section 9.1, adapted to our game implementation. The CMCTS agent constructs a `CMCTSNode` from which the search tree is built during the algorithm. This tree is created and explored in `Cheating_MCTS_Base::uctSearch` and comprises individual `CMCTSNode` objects each of which contain a pointer to all child nodes and its parent node. The inclusion of both parent and child pointers allow for both downward and upward tree traversal which is essential for the MCTS algorithm. Each node contains its incoming action and expandable actions, as well as a game state copy. Special care has been taken to ensure that the whole tree is properly deleted after the conclusion of the algorithm to avoid leaking memory.

Most of the functionality is shared between differing MCTS variants. Thus, ISMCTS and CMCTS variants both inherit from the base CMCTS agent. ISMCTS expands upon the base agent by storing additional information (`m_expandedActions`) as well as modifying behaviour to manage determinization.

11.3.2 Heuristic Agents

All agents have enumerated values associated with them, and these values will frequently be used to represent these agents. The list of enumerated types is found in Appendix A.

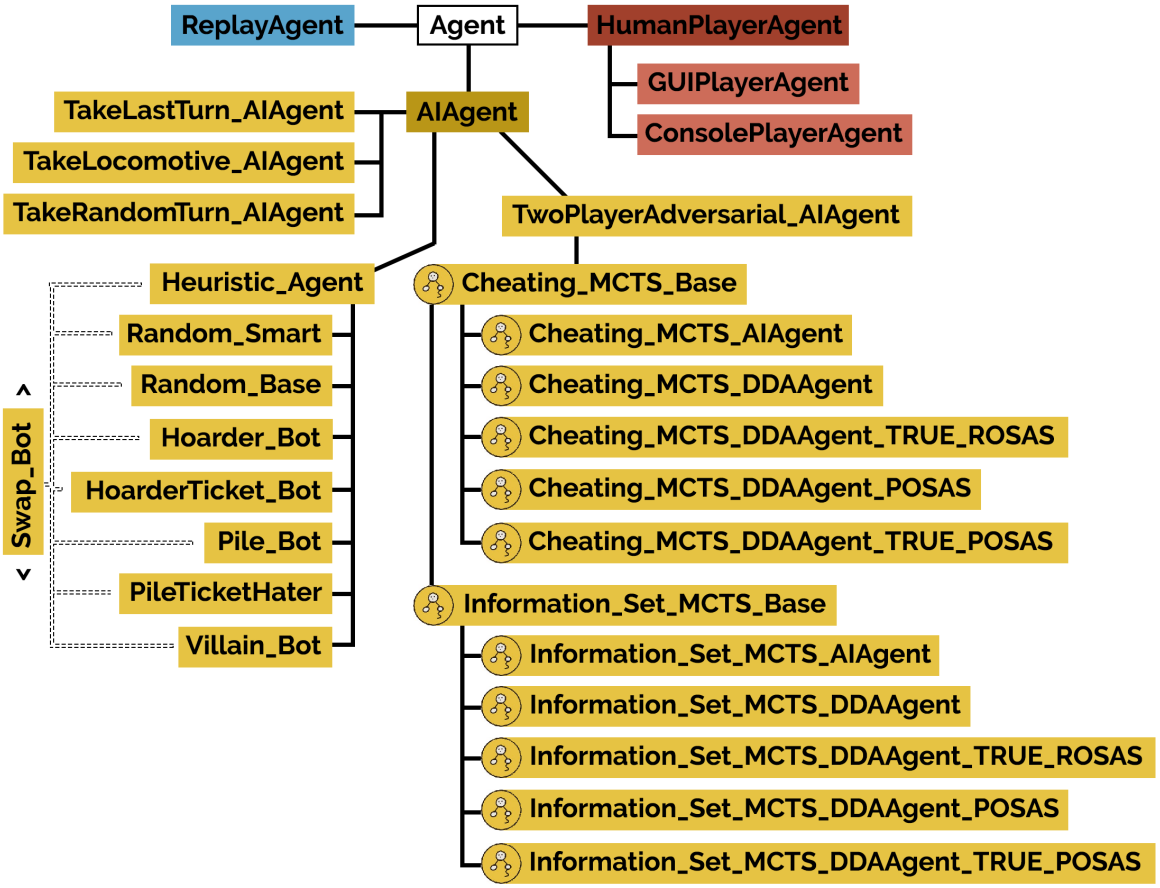


Figure 11.12: All implemented agents and their inheritance hierarchy. Solid lines are inheritance relationships, from the base `Agent` abstract class. Dashed lines are composition relations — `Swap_Bot` uses two of any agent.

The majority of heuristic agents are built upon the foundation of the base heuristic agent, `Heuristic_Agent`, whose implementation we describe in detail in this section.

11.3.2.1 Base Agent

The base agent (known as `Heuristic_Agent` in the codebase) inherits from the `AIAgent` class. It overrides the `takeTurn` function and possesses a range of class functions and variables. We will refer to key member variables, of `m_path` and `m_edgeCosts`. `m_path` is a vector of `Route` objects, representing the selected optimal source-destination path of the base agent. Meanwhile, `m_edgeCosts` is a vector of `double` values that stores the calculated costs of each route on the map.

Base agent's `takeTurn` function primarily consists of calls to two key functions: `updatePath`

and `makeDecision`, both of which are explained in detail.

updatePath To update `m_path` at the start of each turn, the `updatePath` function was developed. Initially, the function calculates the cost of all routes by calling `evalEdge` for each route, thereby updating the `m_edgeCosts` class variable. The cost of the route is evaluated based on its colour and length, as outlined in 12. Subsequently, each ticket the agent possesses is considered: all paths that connect the ticket destination cities are identified. This is facilitated by the `findAllPaths` function, from the `Map` class.

Algorithm 12 Evaluate a route

```
procedure EVALEDGE(route)
    multiplier = 8
    if route's colour is grey then
        multiplier = 1
    end if
    cost = multiplier * route.getLength()
    cost = cost - 0.1 * route.getRouteScore()
    return cost
end procedure
```

For efficiency, the base agent has the class variable `m_allPathsPerTicketTable` which caches all source-destination paths for each ticket the player owns. This eliminates the need to recalculate all paths using `Map::findAllPaths` each time `updatePath` is invoked, which is an expensive operation.

Finally, equipped with all updated necessary information, the most optimal path can be identified. To facilitate this process, the `findBestPath` function is created. Initially, it calculates the cost of each path completing the tickets, utilising `m_allPathsPerTicketTable`. These costs are stored in `m_costPerPathPerTicket`. Subsequently, `m_costPerPathPerTicket` is sorted and the cumulative costs of the cheapest paths from each ticket are summed up and stored in a local variable `threshold`.

The algorithm for finding the best path, Algorithm 13, operates as follows: initially, all paths completing the first ticket are stored in a separate variable called `allPaths`. For subsequent tickets, each combination of paths from the current ticket and `allPaths` is merged to connect the paths together, removing duplicate routes, and then added to `allPaths` for the next iteration of tickets.

We can represent the game map as an undirected graph $G = (V, E)$ where V is the set of vertices (cities) and E is the set of edges (routes). Let $m = |E|$. As each route can

be in either of the 2 states of being claimed or unclaimed, the state space of the map is 2^m. Due to the exhaustive nature of the algorithm used, the worst case scenario for computation time would be $(2^m)^k * 2r$ where m represents the number of edges (routes) in the graph, k represents the number of tickets being investigated and r represents the length of the longest edge (route).

The `threshold` variable serves a crucial purpose for managing the hard computational complexity associated with finding an optimal path by connecting multiple cities in an exhaustive manner [7], [8]. It achieves this by effectively trimming the search space, thereby improving the efficiency of the search algorithm. Before adding the merged path to `allPaths`, a check is made: the path is only included if its cost is less than `threshold` multiplied by 1.4; if it meets this criterion, the threshold is updated to the cost of the newly added path.

Ultimately, the function selects the path with the lowest cost as the optimal path and returns it.

Algorithm 13 Find the most optimal path connecting all the input tickets

```
procedure findBestPath(tickets)
    calculate the threshold
    for ticket in tickets do
        ticketPaths = m_allPathsPerTicketTable[ticket]
        for path1 in allPaths do
            for path2 in ticketPaths do
                joined = combine path1 and path2
                if joined.cost ≤ threshold * 1.4 then
                    add joined to allPaths
                    update threshold to joined.cost
                end if
            end for
        end for
    end for
    return min cost path from allPaths
end procedure
```

makeDecision After the optimal path, `m_path`, has been identified, or conversely, no path has been found, the `makeDecision` function 14 is invoked. Firstly, let's discuss the scenario where a path has been identified.

Algorithm 14 Make a decision on what action to take

```
procedure makeDecision
    if m_path is empty then
        return cantFindPath()
    end if
    action = selectEdge()
    if action not empty then
        return action
    else
        return drawBestCard()
    end if
end procedure
```

In this case, since the agent has a concrete path it's seeking, it chooses a route on the path to claim via the `selectEdge` function. This function works by iterating over every route on `m_path` and finding the most optimal claim action by employing a cost function to evaluate each potential claim action.

These claim actions are evaluated based on the number of locomotives used and the quantity of cards of the route's colour required for completing the entire path. Generally, claim actions that use locomotives tend to be more costly. However, if claiming all the routes on the path necessitates a significant quantity of a particular coloured resource, prioritising locomotives becomes advantageous.

To facilitate finding the quantity of cards needed for any given path, the `cardsNeeded` function was developed. It systematically iterates through each route and increments the corresponding count of resource colours by the length of the route.

`selectEdge` keeps track of the most optimal claim action so far, and its cost; ultimately returning the cheapest claim action.

However, it is possible that the agent does not possess sufficient resources to claim any route on the map. In that case it will draw resource cards; for this purpose, the function `drawBestCard` has been created.

Initially, `drawBestCard` makes a call to `cardsNeeded` with `m_path` as input. Subsequently, it iterates through the face up cards, prioritising the colours for which the quantity needed to complete the route is greatest. It then draws the 2 most needed face up cards. However, if one or more of the face up cards is a locomotive card, it prioritises drawing those instead. Potentially, only 0 or 1 of the face up cards are needed for path completion. In this case,

2 or 1 cards will be drawn from the pile respectively.

However, the other case still persists; what if no path is found, meaning `m_path` is empty? In this scenario, the `cantFindPath` function is executed.

Firstly, `cantFindPath` checks whether the agent still possesses a sufficient amount of train cars. If that is the case, the agent will draw an additional ticket. It will only draw a ticket if it has a path that is claimable by the agent.

Alternatively, if the agent does not have enough train cars, it will attempt to claim the longest route possible. The `getPossibleClaimActionsForPlayer` function from the `Game` class is called to obtain all legal claim actions for the agent. These actions are then iterated over, and the one providing the highest score, corresponding to the greatest length, is selected. If no legal claim actions are possible, the agent will draw cards from the pile.

11.3.2.2 Other Agents

Most other agents are derived from the base heuristic agent, thus have inherited all previously mentioned variables and functions from the base agent. They add their own unique strategy by overriding the `Heuristic_Agent::takeTurn` function.

11.3.2.3 Swap Bot

We've developed a template enabling the creation of a bot that combines any two heuristic agents. A swap bot agent itself contains other bots as member variables. This bot can dynamically switch its strategies mid-game by transitioning from one of these agents to another according to some condition.

11.4 Reproducibility

As we described in Section 10.7.1, we wanted to design the game such that each game run can be fully reproduced. This is both for good scientific practice so that others can reproduce our results, and for software debugging and data collection. It also sets up the back end for a future user application feature that will allow players to recapitulate their previous game runs.

11.4.1 Randomness

Since some of our agents have random behaviour, this in theory makes game-play unrepeatable. However, Pseudo-Random Number Generators (PRNGs)[1] allow repeatability by producing a deterministic, albeit erratic, sequence of numbers from an initial seed with some very large period, such that the numbers appear very closely match a random distribution of some kind [42]. PRNGs are usually used for statistical purposes. There are several different random devices to choose with a trade off between quality and performance, for reasons as we discuss in Section 11.6.3, we chose to use the Xorshift algorithm [43].

We made our own `Random` class in order to provide functions alongside a Xorshift PRNG, such as shuffling iterators or returning a random index of an iterator, to encourage code reuse.

The game state and every agent is provided its own unique seedable random device. We have given agents their own devices separate to the game state, ensuring an agent's own pseudo-random behaviour cannot impact the game state or its opponents. Thus, each of these components use exclusively their own random device. These random devices can be seeded by user defined values, or by the time and the seed is recorded before the game has begun. This seeding ensures reproducibility.

We chose to copy the internal state of the random device between game state copies. This is primarily so that MCTS agents are not able to affect the original game state used by other players. However, preserving random state between copies also ensures any information that CMCTS learns about train and ticket resources, which are shuffled using the game state's random device, is not destroyed after each simulation. This is crucial: if this were not the case, then the CMCTS agent would not be playing with perfect information. This necessity to copy random states comes with penalty, as it increases the memory resources allocated per tree node and increases game state copy time. Thus, we have carefully considered our random device to reduce these overheads, as explained in Section 11.6.3.

[1] Whilst True Random Number Generators (TRNGs) exist that rely on physical processes and are less predictable, these are not suitable either due to high latency or the need for expensive additional hardware [42]. More critically, sequences produced by such TRNGs cannot be reproduced without very detailed modelling of the physical source of entropy, which disqualifies their use.

11.4.2 Game Configuration

Our game environment is designed to play Ticket to Ride games across a variety of maps, with differing colours, using differing quantities of starting resources and across different selections of agents. We identified early on that it is essential to store and load this game run information readily for user and testing purposes. In order to keep track of this snapshot of information, we formalised a game configuration standard which encapsulates all this information.

In particular, the game configuration stores the map name, colours used, agent types, game seed and agent seeds. We decided to store and load game configuration from text files, for reasons of reproducibility and to allow tests to be pre-configured out of application and loaded in.

Being able to painlessly load these configurations was crucial for our test plan, since we wanted to test dozens of game configurations.

11.4.2.1 Configuration Format

We compressed included colour information into a single integer that we call a *colour code*, whereby the first 8 bits of the colour code correspond to the inclusion or exclusion of the following colours with values 1 or 0 respectively: black, white, red, orange, yellow, green, blue, pink. For instance, the map created by Yang et. Al [14], uses only the colours red, yellow, blue and green, so has colour code $2^2 + 2^4 + 2^5 + 2^6 = 116$.

Each agent has its own `AGENT_CODE` enumerated type, for example, the base AI CMCTS agent has agent code of 16384. We have attached a list of all agent codes in Appendix A. We also allow users to include the frequency of the given agent type as a hangover from legacy code to preserve internal backwards compatibility, although for the two player setting these have only valid values 1.

Game configuration format is written in Listing 11.1.

These configurations are loaded from a text file into an internal `GameConfig` C-struct which is then used by the game environment to set up the game. We outline some example configurations in Listing 11.2.

11.4.2.2 Agent Hyperparameters

Our MCTS agent variants have several hyperparameters such as the expansion coefficient and iteration bounds. These are handled separate to the configuration file, and are

Listing 11.1: *Game Configuration Format.* It is structed in the order of map configuration; agent 1 configuration; agent 2 configuration; starting resource configuration and random device configuration.

```
<MAP_NAME>,<COLOUR_CODE>,
    <FREQUENCY_AGENT_1>,<TYPE_AGENT_1>,
    <FREQUENCY_AGENT_2>,<TYPE_AGENT_2>,
    <NUM_STARTING_CARS>,<NUM_CARDS_PER_COLOUR>,<NUM_LOCOMOTIVES>,
    <GAME_SEED>,<AGENT_1_SEED>,<AGENT_2_SEED>
```

Listing 11.2: *Example Game Configurations.* (1) is a test map which uses all colours and plays one greedy claim route agent against another and is manually seeded. (2) is the Yang et. Al Map which uses a reduced colour set and plays the GUI player agent against a CMCTS DDA agent and is time seeded. (3) is an exaxmple of a BigMap configuration which also sees a GUI Player but pitched against a CMCTS DDA agent with POSAS.

```
(1) testGame,255,1,256,1,256,45,12,16,501,300,762
(2) YangEtAlMap,116,1,2,1,32768,10,6,6,-1,-1,-1
(3) BigMap,237,1,2,1,32770,15,6,8,-1,-1,-1
```

instead handled by our testing infrastructure. This is since for our users, the parameters will be fixed, so we do not need to add these, often unused, parameters in every game configurations. We wrap all of our parameters into a C-struct, `AIParametersConfig`, which can then be loaded by the game controller.

11.4.3 Game Record and Replay

We are able to reproduce any games with just AI agents, using just a configuration if we include the correct seed. However, how do we reproduce nondeterministic inputs: that is, how do we extend game run reproducibility to human players? In addition, how can we monitor and analyse game moves after a game's completion? To answer both of these questions, we record every turn taken by each play and save a recording. These records allow for static analysis of agent behaviour.

We designed recording behaviour to wrap over the existing game environment. Whenever an action is executed in the game environment, the recording wrapper momentarily steals the action, records it to a log file before returning it to the game environment to act as normal. This is the `RecordController`, which inherits most behaviour from the `GameController`.

We had to take special care when handling initial set up; in particular initial ticket selections are recorded in a zeroth turn.

Listing 11.3: *Record Turn Format.* Each turn contains turn and agent information (whose turn is it?). In addition each line records a timestamp at the end of this turn for analytical purposes (how long has this turn taken?). Finally the action performed by the player is captured with its action code and associated parameters.

```
<GAME_NUMBER>,<ROUND_NUMBER>,<AGENT_NUMBER>,<TIMESTAMP>,
    <ACTION_CODE> [ ,OPTIONAL_ARGS : <ACTION_PARAM_1>,
                                     <ACTION_PARAM_2>, ... ]
```

Listing 11.4: *Example Recorded Turns.* (1) On the third round of the first game player 1 (BRAVO) drew two cards from the pile. (2) On the first round of the second game player 0 (ALPHA) claimed a route from the city with ID 4 (Emberforge) to the city with ID 5 (Fairhaven) using 0 locomotive cards and using the colour with value 6 (blue).

```
(1)  1,3,1,1796,4
(2)  2,1,0,3020,8,4,5,0,6
```

11.4.3.1 Recording Format

Each turn requires various data, such as the agent ID, turn tracking and action performed. This action data is in turn parsed into an action object that performed by the appropriate player. The format is displayed in Listing 11.3, with example recorded turns in Listing 11.4.

To start a recording, we store game configuration data, as it is impossible for the game to determine the chosen map and resources from turn data alone. To inform the replay controller when the game starts, we reserve the character `~` and the character `%` for the end of a game.

Games are stored in a text file with an auto-generated timestamped filename, with format `ttr_game_replay_YYYYMMDD_hhmmss.txt`, stored in the local resources folder. In Listing 11.5 we demonstrate an example of such a recording.

11.4.3.2 Replay Agents

Since we had decided to record games, being able to run games from this recording was a natural extension. We run replays by creating a special replay agent which can read from a file, which is run as normal inside a game environment. We construct the `ReplayController` class which lightly wraps over the `GameController` and loads the replay file into memory, setting up the game according to the configuration line, setting up replay agents and distributing the file pointer to these agents. When it is a (replay) agent's turn, it reads the next line from the replay file, and parses the action data into a

Listing 11.5: *Example Game Recording.* The recording was captured using the test application with command `./bin/Release/TicketToRideTest.exe -C YangEtAlMap,116,1,1024,1,16384,10,6,6,-1,-1,-1 -N 2`. Note the Game starts with the configuration; followed by a `~` indicating the start of a game; 8 rounds and a `%` to indicate the end of the first game. The second game starts with a `~` and the same pattern continues.

```
    YangEtAlMap,116,1,1024,1,16384,10,6,6,-826872240,-818963940,
-818963840
    ~
    1,0,0,0,0
    1,0,0,0,1,2,1,1,0
    1,0,1,0,0
    1,0,1,561,1,2,1,0,1
    1,1,0,562,8,4,5,0,6
    1,1,1,1037,8,1,3,0,4
    1,2,0,1037,8,4,6,0,4
    1,2,1,1416,8,3,4,0,2
    1,3,0,1417,3
    1,3,1,1796,4
    1,4,0,1796,4
    1,4,1,2094,8,0,2,0,2
    1,5,0,2095,8,1,5,3,8
    1,5,1,2295,8,2,3,0,5
    1,6,0,2295,4
    1,6,1,2473,5,5,5
    1,7,0,2474,8,3,5,0,4
    1,7,1,2510,8,0,6,1,5
    1,8,0,2510,4
    %
    ~
    2,0,0,2409,0
    2,0,0,2409,1,2,1,1,0
    2,0,1,2409,0
    2,0,1,3019,1,2,1,0,1
    2,1,0,3020,8,4,5,0,6
      ... more turns ...
    %
    ~
      ... more games ...
```

`PlayerEvent`, which is executed as normal.

11.4.4 Reproducibility in Practice

Seeded game configurations have been useful for debugging as they allow us to re-run the exact same computational steps again. For example, when running MCTS parameter tuning tests, we found that a non-trivial proportion of the runs ended up terminating with a segmentation fault after a hundred or so games. We did not find this out until we ran this massive volume of games. However, because we recorded the seed, configuration and parameters, we were able to perfectly reproduce the game in debugger, and discover how we missed a null pointer check. Without this functionality it would have been very difficult to uncover this infrequent bug.

In addition, our game reproducibility ensures others can reproduce our work. For example, anyone can produce an identical recording to Listing 11.5, by using the game configuration at the top of the record.

11.5 Test Application

While test game lightly decorates the game environment, and its features have been developed into the game environment, we still needed to make a command line interface. The test application has its own main file.

The `TicketToRideTest` executable parses flags, which determine features including the game configuration, which type of controller is used and the number of games to be run. See Figure 11.13 for each flag and its explanation.

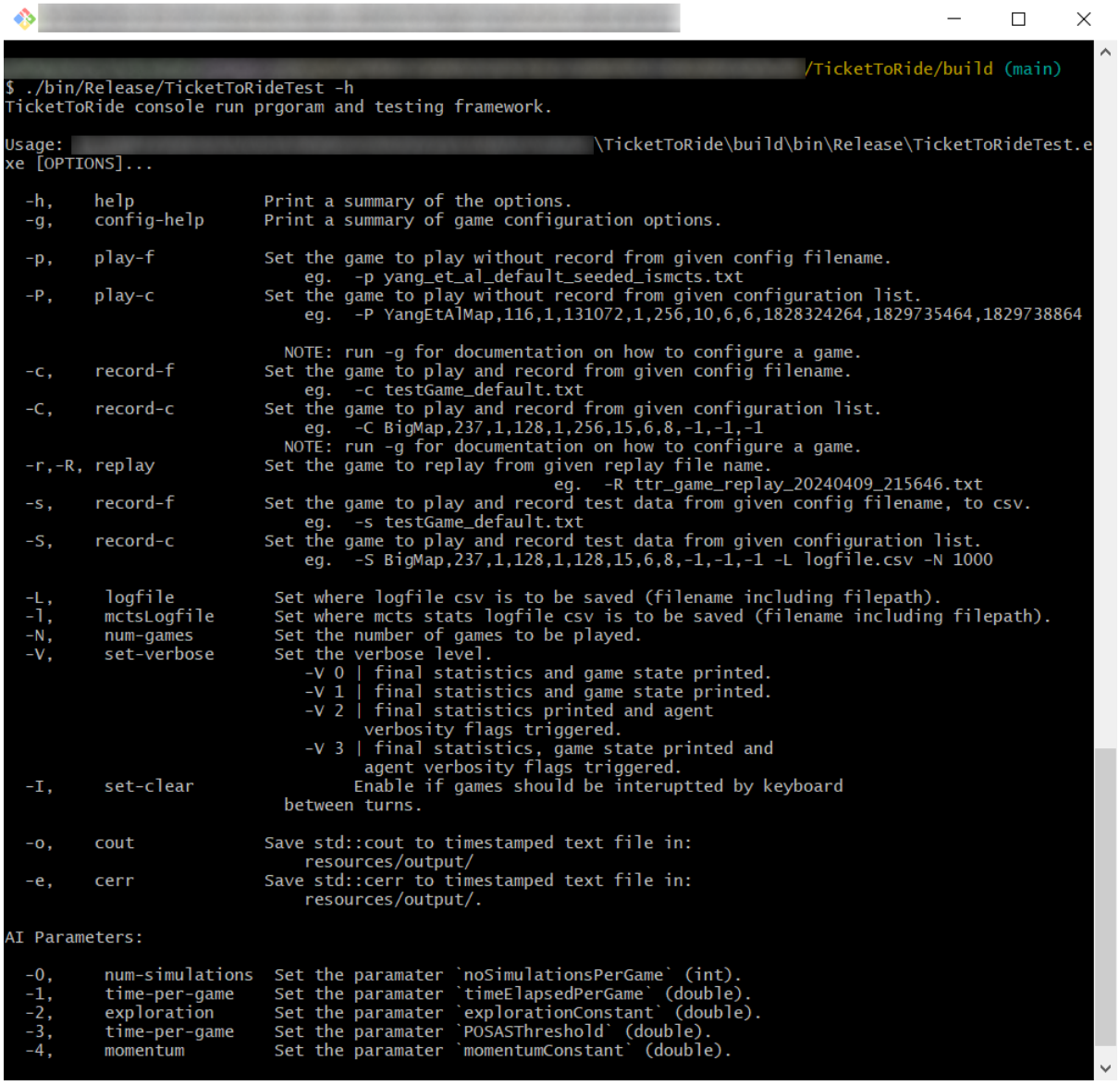


Figure 11.13: Test application help dialog. This shows all the various flags that can be used by a tester. To run the help dialog use the flag `-h`. On Windows run `.../TicketToRideTest.exe -h` or on Linux and Mac use `.../TicketToRideTest -h` .

SaveGameController is used only for gathering detailed test data on test runs. It decorates `GameController`, but unlike `RecordGameController` it does not record turns but records statistical data. Firstly it logs score data for every game and saves that to a CSV file (see Listing 11.6). Secondly, it logs MCTS stats into a CSV for every turn to analyse MCTS performance and behaviour in a separate CSV file (see Listing 11.7).

Listing 11.6: *Example Game Run Score Analytics CSV.* This recording was captured using the test application with command `./bin/Release/TicketToRideTest -S testGame,255,1,256,1,16384,45,12,16,212,212,212 -l ../test/runs/example.csv -L ../test/runs/example2.csv -0 77777 -N 5 -V 0 -o -e .` The format is as follows: GAME_NO; NO_TURNS_TAKEN; GAME_TIME_ELAPSED(ns); GAME_TIME_ELAPSED_FROM_START(ns); Alpha_SCORE; Alpha_SCORE_NO_CAP; Alpha_SCORE_NO_TICKETS; Alpha_SCORE_ROUTES_ONLY; Bravo_SCORE; Bravo_SCORE_NO_CAP; Bravo_SCORE_NO_TICKETS; Bravo_SCORE_ROUTES_ONLY.

```
0,45,34006747700,34006747900,0,-15,10,10,32,32,44,44
1,51,42387043700,76393791900,0,-20,6,6,49,49,48,48
2,43,35034661400,111428453400,0,-9,14,14,54,54,40,40
3,43,33967704900,145396158400,0,-9,12,12,58,58,42,42
4,45,37557626200,182953784800,0,-13,10,10,58,58,44,44
```

Listing 11.7: *Example MCTS Analytics CSV.* This recording was captured using the test application with command `./bin/Release/TicketToRideTest -S testGame,255,1,256,1,16384,45,12,16,212,212,212 -l ../test/runs/example.csv -L ../test/runs/example2.csv -0 77777 -N 5 -V 0 -o -e .` The format is as follows: GAME_NO; RUN_NO; TURN_NO; PLAYER_ID; TIME_ELAPSED(ns); NO_ITERATIONS; NO_EXPANDED_NODES; NO_VISITS_TO_BEST_CHILD; SCORE_OF_BEST_CHILD.

```
1,1,2,Bravo,2726829200,77777,77778,77565,-4.20727
1,2,4,Bravo,2516913500,77777,77778,77609,-5.53246
1,3,6,Bravo,2462292500,77777,77778,47831,-8.26562
1,4,8,Bravo,2309554700,77777,77778,77568,-0.000128919
1,5,10,Bravo,1951887100,77777,77778,57011,-0.130729
1,6,12,Bravo,2218888400,77777,77778,64591,-0.443421
1,7,14,Bravo,1859361300,77777,77778,72373,0.655479
1,8,16,Bravo,1618086900,77777,77778,75031,1.57917
1,9,18,Bravo,1860851800,77777,77778,77034,11.8129
1,10,20,Bravo,1235266800,77777,77778,76877,12.5467
1,11,22,Bravo,1535840400,77777,77649,77680,32.6866
1,12,24,Bravo,1260470100,77777,77757,77692,29.2606
1,13,26,Bravo,1297153200,77777,77687,77536,31.8316
1,14,28,Bravo,360121000,77777,15646,77733,39.3004
1,15,30,Bravo,1076176500,77777,77778,77515,42.0175
1,16,32,Bravo,540569400,77777,42848,77742,39.9016
```

```
1,17,34,Bravo,787684200,77777,63641,41801,40.6954
1,18,36,Bravo,1066462900,77777,77778,62885,42.6891
1,19,38,Bravo,588613800,77777,59943,76681,45.4359
1,20,40,Bravo,201864700,77777,9847,77304,46.7584
1,21,42,Bravo,101340200,77777,1325,76546,47
1,22,44,Bravo,52334100,77777,9,9722,47
2,1,2,Bravo,2724205200,77777,77778,77605,-1.21166
                    ... continued ...
```

Work on the test application has led to improvements in the game environment, which have thus passed on features to the user environment. For example, we have been able to visualise replays in the user application, and play infinite games without reloading the application.

11.6 Performance

Performance, in terms of memory and time, is a crucial consideration for all of our components. However, we are most concerned with AI performance, because improved performance allows for more computation to be performed within a reasonable time limit. The user will have a threshold before the time taken for an AI to complete its turn becomes disengaging and reduces enjoyment. Thus, we are limited to a maximum computation time of at most tens of seconds. So to get more simulations in this limit, we have to increase the performance of our game environment and AI calculations. In addition, a small application size in memory and reduced number of CPU cycles per operation ensures the portability of our program to less powerful devices. Furthermore, improved performance allows for more games to be run in order to get higher resolution quantitative data.

In this section we outline general practices taken as well as specific optimisations that took place after profiling code as outlined in Section 12.3.2. All our optimisations have been data driven, assessing whether to keep a given optimisation by comparing expectation time and space before and after each change.

11.6.1 General Practices

As we outline in Section 10.4.1, MCTS performance is deeply tied to the number of simulations it can perform. If turns are time bounded, game state copying time must be suppressed to ensure a maximal simulations volume. One way to decrease copy time is

to decrease the amount of memory copied in the first place.

We do this by not duplicating *static components* between game state copies, like we duplicate *dynamic components*. Static components include factors such as the map layout, city names, route counts and ticket information, whereas dynamic components include current route ownership, resource decks and player resources. Static components of the game state include larger data such as strings for the names of each city, compared to dynamic components, which usually are numeric values or references to objects. In addition to reducing copying time overhead, reducing the amount copied reduces the overall amount of memory, and thus increases the compatibility of our game for smaller memory systems. However, references themselves do have a memory overhead, and access time overhead, so it is a balancing act.

Since we have coded using C++, we have to make the most of the features and quirks of this high performance language. This includes making the most of the strict control over memory through pointers, references and other such features. Certain data structure implementations have greater overhead, such as std::map than others such as std::vector. Where possible we try to use simpler data structures.

Where possible, we try to avoid duplicate computation, preferring to cache results where necessary. A key example of this in practice is storing a lookup table of all possible routes for given player resources and route lengths. This is needed to get a list of valid claim route actions, and is queried nearly 10^8 times in a given MCTS Ticket to Ride game. Caching all possible combinations in a four dimensional array, which although occupies several megabytes in memory, is much preferable and drastically reduces time taken to get a list of valid actions.

11.6.2 Memory

We want to reduce memory usage for two reasons: space and time. However, the consideration of *how* memory is stored is just as important as *how much* memory is used. Fetching memory from different physical locations is non-trivial in cost, therefore it is best to try and keep related data close together.

We describe some memory location considerations through an example. We found calculating player ticket score contributions to be expensive. During analysis, we determined this was due to (i) the constant re-creation of `hasBeenVisited` which tracks which nodes have been visited in a depth first search to prevent cycles and (ii) de-referencing smart pointers for the map structure, further compounded by the fact (iii) the map stored cities

and routes as smart pointers, rather than contiguously in memory, which also have to be de-referenced. In response to these performance issues, we designed the `MapData` class. `City` and `Route` objects are now stored contiguously in memory, addressing issue iii. A member variable `m_hasBeenVisited` is only created once and instead of being created and destroyed in each iteration has its flags reset, addressing issue i. Instead of sending the map structure to the Map which stores the route claim data we reversed this by sending the route claim data to the MapData so that the computation was closer to the data. By fixing issues i, ii and iii we saw a dramatic speed up, deceasing the total share of computation used by `Game::updateScores` from 28% to 8%.

In addition, we avoid creating and deleting objects which are used for intermediary computations, since the overhead of object allocation and deletion is significant.

Overuse of `std::shared_ptr`. In C++, is best practice to use smart pointers to deal with dynamically memory allocation rather than with C-style `new` and `delete` [44]. This is since smart pointers handle their own deletion and thus minimise the chance of the developer missing a memory leak.

Since we have data shared across multiple game state copies, shared pointers are more appropriate than unique pointers. However, we overused shared pointers, due to not understanding their nature: all copies of cities, routes and tickets, whether in map structure or for use in intermediary calculation, were shared pointers. This was problematic. Firstly, a circular dependency between routes and cities meant that neither objects were destroyed, as both reference counts would never sink to zero, which lead to a memory leak. Secondly, when route objects were used in intermediary calculations, new shared pointers would be created: this is expensive as shared pointers have to perform an atomic operation to increase their reference count [45]. We solved the circular dependency by storing Cities and Routes contiguously in memory, not as references, and storing references to each other as member variables. And instead of using route shared pointers we used route references instead which significantly reduced execution times for using functions.

11.6.3 Efficient Random Devices for Monte-Carlo Tree Search

As we describe in Section 11.4.1, we duplicate the random state of our PRNGs both for the sake of reproducibility as well as to ensure proper cheating behaviour.

The default C++ standard library random devices are large in size, have poor period or

are slow[2]. Some devices are designed to be cryptographically secure (CSPRNG), which comes with long time computational overhead. This consideration is one we need not to worry about, as there are no vulnerabilities present in MCTS that would warrant such security. However, poor disributions are unsuitable for MCTS as a large period is needed for the around 10^7 uses of the random device in a MCTS game. This disqualifies generators such as `std::ranlux24_base` that have very low period despite small space in memory and very efficient computational time [47].

For execution in software, Xorshift generators are among the fastest PRNGs and require very small code and state [43]. However, they do not pass every statistical test without further refinement, despite large sequence periods. the Xorshift128 algorithm has period $2^{128} - 1 \approx 10^{38}$ The 128-bit Xorshift algorithm passes the diehard tests [48]. Since XOR operations are invertible they are not cryptographically secure. A non-linear function can be introduced to further improve quality, but we decided away from doing it, preferring to avoid such expensive mathematical functions. We thus implemented a xor128 generator algorithm inside our `Random` class.

The change in random device to xor128 significantly decreased the random state copy time from 36.56ns to 7.62ns (a reduction of 4.8x)[3] when compared with previously used mt19937_64. Random number generation is cheaper too with generation time of 3.52ns vs 2.81ns (speed up of 1.3x)[4]. These speedups validate our decision to use anon-standardd library random device.

11.6.4 Unexpected Performance Impediments

Expensive Mathematical Operations: Logarithm and Square Root. Even the most efficient standard hardware implementations for logarithmic functions take hundreds of clock cycles at double precision if complying with the IEEE 754 floating point arithmetic standard [49]. This is a hundred-fold more expensive than floating point addition and subtraction for example. In MCTS, the UCT score, which uses a single use of both logarithm and square root, is regularly calculated. This led to these calculations making up for a surprising volume of computation. Indeed, whilst our MCTS was not working

[2]Several devices can be found in the C++ std::random library, such as the `std::mt19937` random engine which implements the Mersenne Twister by Matsumoto and Nishimura [46] and has size 5KB, or the `std::ranlux48` algorithm with smaller 120B size, but slower running time. Quicker linear congruential engines are both small in size and fast, but produce low quality distributions. See the standard library devices at https://en.cppreference.com/w/cpp/header/random.

[3]To reproduce random state copy test result on your own hardware, run the test `.../TicketToRideTest -T 0`.

[4]To reproduce the random generation result on your own hardware, run the test `.../TicketToRideTest -T 1`.

correctly, `std::log` and `std::sqrt` were together responsible for just under 51.2% of our computation (see Figure B.1).

These computation times can be reduced by three ways. Firstly, reducing precision from double (64-bit) precision to float (32-bit) precision. Secondly, further reducing precision by using the `-Ofast` compilation flag[5]. Thirdly, use alternate functions, such as fast square root [50] and fast log [51], which have reduced accuracy[6]. We found that the `-Ofast` flag yielded negligible performance and the fast log and square root functions were manipulating MCTS behaviour because they reduced the accuracy by too much. However, 32-bit precision standard library functions, `std::logf` and `std::sqrtf` provided a two-fold speed up for very little additional cost in precision.

Type Identifier Lookup . We relied upon using dynamic casting to type check the type of our events. This is particularly expensive due to the inheritance structure of these events. Inheritance and polymorphism introduce the need for dynamic dispatch, where the correct function or method to call is determined at runtime based on the actual type of an object. This requires additional type information to be stored and managed, adding to the overhead of type lookup [52]. We also used explicit type look up in an `operator==` overload which was used in ISMCTS, which originally took around half of the mass of computation.

Our solution was to add a function which returns the associated event enumerated type value, which was instead compared. This dramatically decreased the computational cost of querying actions and the whole ISMCTS algorithm.

11.7 Installation and Portability

We decided to use CMake to install our game files as justified in Section 8.2. We fetch SFML source code and add this to our list of build targets[7]. This means that we do not have to worry about lack of an SFML library already installed in a given user's system. However, we have found that additional dependencies may have to be installed, as outlined in Appendix D.

CMake also provides a straigtforward way to add several target executables. Since we

[5]The GNU C++ compiler has a list of optimisation flags which can be found in the documentation at https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[6]The fast square root function is able to achieve in very few cpu cycles with minimum maximal error of 3.5%. [50]

[7]To do this, we adopted the official SFML CMake Project Template, which can be found at https://www.sfml-dev.org/tutorials/2.6/start-cmake.php.

were developing the user application and test application, we added two executables `TicketToRide` and `TicketToRideTest` which share all the code but their main. However, since end users do not need to use the test application, we add a flag `TEST_BUILD` which must be explicity enabled in order to add, link libraries and compile `TicketToRideTest`.

Differing C++ compilers such as Clang++ [53], GCC [54] and Microsoft Visual C++ [55] that are default in Macintosh, Linux and Windows systems respectively, all have slightly different quirks that have to be dealt with. But since we were developing on these three platforms regularly, we were able to make the code compile on all three systems by fixing the varying errors and warnings raised by each. Building with multiple differing compilers makes our code safer and more robust, as some compilers raise code issues others ignore. In addition, we modify behaviour based on the platform with preprocessing directives such as `#if defined(__linux__)`.

Chapter 12

Testing and Results

This section details the testing plan for the final product and the results collected. Its objectives are to ensure the game prototype is functional and stable without significant bugs, as well as to assess the overall fun factor of the game, which is a key contributing factor to the success of the DDA agent.

Approach. First, we conduct small functional tests to ensure the game functions exactly as we intend. Then, we focus on the MCTS, verifying its correctness and carrying out performance and stress testing to verify the behaviour of the system under various conditions. Next, we use automated testing against heuristic agents in order to investigate the effect of different parameters and t determine the most effective algorithms for DDA. Finally, we perform user acceptance testing, which is a high-level test plan designed to primarily evaluate the system as a whole, ensuring that our product meets user expectations.

Tools. For heuristic, DDA efficacy, and performance testing, we use computational resources provided by the university's Department of Computer Science. On the user testing side, we use Google Survey as the main tool to collect feedback from testers.

Exit Criteria. The product meets the completion criteria, when we are able to ascertain that it achieves a minimum level of stability and robustness based on functional testing, as well as receiving satisfactory feedback from users indicating an overall positive user experience.

Stakeholder Involvement. Project stakeholders, including the supervisor and potential end users, are involved in providing feedback and game testing. The team will also

collaborate closely to ensure alignment between design and testing efforts.

12.1 Functional Testing

Before carrying out full user acceptance testing, it is important to ensure the game is fully functional and without major bugs. This can be carried out by the team, according to pre-specified test cases. The test cases are a simple but comprehensive set of possible actions when interacting with the game, for which we check that each input results in the expected outcome.

Table 12.1: Functional tests for user application.

Component	**Test ID**	**Test Case**	**Status**
Main Menu	1.1	Renders as expected	Passed
	1.2	Click on start button starts the game	Passed
Map	2.1	Renders as expected	Passed
	2.2	Unclaimed routes respond to hover	Passed
	2.3	Unclaimed routes can be selected on click	Passed
	2.4	Selected routes can be unselected on second click	Passed
	2.5	Selected routes are unselected and reset upon invalid train choice	Passed
	2.6	Selected routes are updated upon valid train choice	Passed
	2.7	Selected routes are claimed upon all valid train choices, and display the correct owner	Passed
	2.8	Claimed routes do not respond to hover and click	Passed
	2.9	Routes do not respond to click and hover on opponent turn	Passed
Train Resource Pools	3.1	Renders as expected	Passed
	3.2	Train draw deck and table cards respond to hover	Passed
	3.3	Click train deck twice draws two from deck	Passed
	3.4	Click train deck once, then one coloured table card draws both	Passed
	3.5	Click one coloured table card, then train deck draws both	Passed
	3.6	Click two different table cards draws both	Passed
	3.7	Click locomotive table card at any point only draws the locomotive card	Passed
	3.8	During card draw, selected state on cards persists	Passed
	3.9	Clicking outside while selecting cards resets selection	Passed
	3.10	Train draw deck and table cards do not respond to click and hover on opponent turn	Passed

Ticket Deck & Ticket Selection	4.1	Renders as expected	Passed
	4.2	Ticket draw deck responds to hover	Passed
	4.3	Ticket deck is not displayed when out of tickets	Passed
	4.4	Click ticket deck brings up ticket selection screen	Passed
	4.5	Cannot interact with normal game objects in ticket selection screen	Passed
	4.6	Ticket selection screen correctly displays top 1-3 tickets from draw deck	Passed
	4.7	Ticket choices respond to hover	Passed
	4.8	Tickets are selected by default	Passed
	4.9	Click on a selected ticket unselects it	Passed
	4.10	Click on an unselected ticket reselects it	Passed
	4.11	Keep one ticket	Passed
	4.12	Keep any two tickets	Passed
	4.13	Keep all three tickets	Passed
	4.14	Cannot discard all three tickets	Passed
	4.15	Ticket deck does not respond to click and hover on opponent turn	Passed
Player Hand	5.1	Renders as expected	Passed
	5.2	Train icons respond to hover and click when claiming route	Passed
	5.3	Train icons cannot be interacted with when not claiming route	Passed
	5.4	Train colour and car counts correctly updated after route claim	Passed
	5.5	Ticket wallet opens on hover	Passed
	5.6	Ticket wallet displays player tickets	Passed
	5.7	Click on next button in wallet cycles through tickets	Passed
	5.8	Opponent hand is not shown on opponent turn	Passed
Scoreboard	6.1	Renders as expected	Passed
	6.2	Updates all player information correctly	Passed
	6.3	Updates turn counter correctly	Passed
Action Banner	7.1	Renders as expected	Passed
	7.2	Correctly displays draw ticket action	Passed
	7.3	Correctly displays draw train cards from deck action	Passed
	7.4	Correctly displays draw train cards from table action	Passed
	7.5	Correctly displays draw train cards from deck and table action	Passed
	7.6	Correctly displays draw locomotive action	Passed
	7.7	Correctly displays claim route action	Passed
Game End Screen	8.1	Renders as expected	Passed
	8.2	Correctly renders winner and final scores	Passed
	8.3	Click anywhere returns to start menu	Passed

MCTS Statistics Display	9.1	Renders as expected	Passed
	9.2	MCTS icon responds to hover	Passed
	9.3	MCTS icon opens MCTS statistics display on click	Passed
	9.4	Statistics displayed are correct	Passed
	9.5	MCTS Statistics display can be closed by clicking outside	Passed

At the current scale, it is still feasible for functional testing to be carried out by hand, as the game environment and GUI are simple and there are relatively few actions to be chosen. However, should this be a product designed for commercial or public release, it might be necessary to carry out functional testing automatically using a testing framework. This will allow for rigorous testing for a more robust product.

12.1.1 Game Environment Testing

We used a host of tools that we have developed for the game, as well as external tools, to validate the correct behaviour of our game environment. We have made use of the Visual Studio just-in-time debugger in order to track the change of variables across turns, line by line, and compare values with our expectations. We have used the console game interface to view map and player states to observe more readily the manipulation of the game state. Furthermore, we made the most of game configuration, record and replay functionality to analyse the turns produced and reproduce games that produced faulty code.

12.2 MCTS Testing

To ensure our results are valid, it is critical to thoroughly test the functioning of the MCTS algorithms to verify they work as expected. To this end, a small test map [1] was created with four cities and 3 connections to form a path. Debugging statements were used to produce text files that detailed the flow of the algorithm, which was used to hand verify its operation.

[1] To use the map in our test app, use map name `MiniMap` and colour code `7`. For more details, consult Table 12.6.

12.3 Performance Testing

We have continually analysed performance during development, using the methods we outline in this section. Please consult Section 8.2 for an explanation of performance analysis tools we have used.

12.3.1 Memory Safety

In software, memory leaks are dangerous because they can gradually deplete available memory in a system, which can cause the system to slow down, become unresponsive, or in some cases even eventually crash due to running out of memory. We used valgrind to check for memory leaks in our game and MCTS agents. After several revisions, we have removed all memory leaks, and have a memory safe system, as we have demonstrated in Listing 12.1.

Listing 12.1: *Valgrind Output.*

```
==1512975== Memcheck, a memory error detector
==1512975== Copyright (C) 2002-2022, and GNU GPL'd, by
    Julian Seward et al.
==1512975== Using Valgrind-3.21.0 and LibVEX; rerun with -h
    for copyright info
==1512975== Command: ./bin/TicketToRideTest -V 0 -P
    YangEtAlMap,116,1,16384,1,256,10,6,6,500,500,500 -N 5
==1512975== Parent PID: 1495593
==1512975==
==1512975==
==1512975== HEAP SUMMARY:
==1512975==     in use at exit: 0 bytes in 0 blocks
==1512975==   total heap usage: 218,474,999 allocs,
    218,474,999 frees, 14,339,389,569 bytes allocated
==1512975==
==1512975== All heap blocks were freed -- no leaks are possible
==1512975==
==1512975== ERROR SUMMARY: 0 errors from 0 contexts
    (suppressed: 0 from 0)
```

12.3.2 Code Profiling

We profiled our game environment and agents in order to see where the mass of computation lies. Using Callgrind, we were able to determine the number of CPU cycles used per major function, as well as a total percentage utilisation. Taking repeated profiles throughout development was useful to analyse what functions should be targeted for optimisation. At the end of our optimisations, we found that CMCTS and ISMCTS have different profile, with ISMCTS spending more time (43.08% vs 16.49%) in tree policy when compared to CMCTS. However, the default policy computation was largely similar. See Appendix B for intermediary and final code profile graph visualisations.

12.3.3 Time Benchmarks

Firstly, we have analysed the time taken for copying game state across different maps, presented in Table 12.4[2]. This copy time is sufficiently small to only make up between 5.8% and 4.7% of total MCTS computation, depending on the variant used. The larger the map, the longer the game state copy computation takes. We think this is due to the increasing number of tickets, route claim information and train cards in memory.

Table 12.4: Game Copy Performance Across Maps. Game states include only two player states and we test using the xor128 random device. Results shown are averages over 100000000 game state copy operations.

Map Name	No. Train Cars	No. Train Cards per Colour	No. Locomotives	Game State Copy Time (ns)
MiniMap	10	6	6	134.49
YangEtAlMap	10	6	6	148.86
BigMap	15	6	8	243.25

Second, we have analysed the time taken to run a mcts iteration for each variant in the game, per turn and averaged across all turns. The average iteration and game run times for selected MCTS agents are listed in Table 12.5. We found ISMCTS variants to take between 11% and 48% longer than their corresponding CMCTS variants. However, the average game duration is between 9% and 47%. These results mean that any of our modifications do not significantly impact performance on the small map size.

We chose to run 30000 simulations, because the average turn time is sufficiently small to maintain engagement (around 1s), whilst being large enough such that CMCTS can readily beat experienced players on the small map when at full strength.

[2]To reproduce results on native hardware use command `.../TicketToRideTest -T` x where $x = 10$ for YangEtAlMap, $x = 11$ for MiniMap and $x = 12$ for BigMap.

Table 12.5: MCTS Iteration and Game Run Time Across Different MCTS Agents. The test was ran on YangEtAlMap, with random devices seeded with 500 with 30000 iterations per turn, across 500 games. w/m means "with momentum".

Agent	Time Per Iteration (ns)	Average Game Time (s)
16384	11371.576	2.6337
32768	14407.420	4.5409
32768 w/m	12357.513	3.8111
32770	10879.319	3.3872
131072	15421.237	3.8769
262144	16150.566	4.9651
262144 w/m	15881.453	4.8645
262146	16089.464	4.9099

12.4 DDA Efficacy Testing

The main purpose of the project is the creation of dynamic difficulty AI to play Ticket to Ride, so it is important to demonstrate that the implemented agents vary their skill level to match their opponent. As user testing can be highly subjective and variable, as well as time-consuming, we employ heuristic agents to carry out efficacy testing of the DDA. This allows us to run thousands of games automatically, and will inform us the win rate of our agent against heuristic agents of different skill level.

12.4.1 Testing Infrastructure

In order to test DDA efficacy, we made use of our test application and its in-built tools to run game simulations across multiple configurations.

Computing Resources We ran our tests using nodes from Department of Computer Science of Warwick University's super computers. In particular, we used the *cpu-batch* partition, which is a 27 node system with node specification as follows: Dual Intel Xeon E5-2660 v3 @ 2.6 GHz (20 cores/40 threads total) / 64GB RAM (1.5GB per thread / 3GB per core). Whilst the individual core clock speeds are slower than readily available consumer hardware, the ability to parallelise testing outweighs the increase in test computation time.

Test Scripting We produced different scripts for each type of test: round-robin; parameter tuning and final results. Each test sends a single job off to a node for each test configuration (game set up and MCTS-parameters). Each job stores out and error streams in text files and saves analytical data in CSVs in a well-organised folder structure.

More than a hundred jobs are produced for some test runs, so we gratefully benefit from the fact that jobs are computed in parallel.

Our script calls the TicketToRideTest application, with -S flag, constructing a configuration string within each script loop for the desired configuration. The -N flag determines the number of games we will simulate. In addition, for parameterised testing, we use the flags -0/1/2/3/4 to load hyperparameters.

Map Configurations We have used 3 maps for testing, with their characteristics outlined in detail below in Table 12.6.

Table 12.6: Map Details.

Map Name	No. Cities	No. Routes	No. Tickets	Colours Code	No. Cars	No. Train Cards per Colour	No. Locomotives
MiniMap	4	3	12	7	10	6	6
YangEtAlMap	8	16	8	116	10	6	6
BigMap	15	30	18	237	15	6	8

Statistics Recorded As described in, Section 11.5, test statistics are recorded and saved using `SaveGameController`, so that they can be reviewed and analysed.

12.4.2 Heuristic Agent Ranking

It is imperative to verify that the heuristic agents we have are indeed capable of exhibiting varying skill levels. To accomplish this, we conducted tests using the round-robin format, pitting all heuristic agents against each other. The map chosen for testing purposes was the Yang et al. map. Round-robin enables each heuristic agent to compete against every other agent, including a duplicate of itself. Moreover, we ensured that each agent starts the game an equal amount of times (100000) in each match-up, since the starting position, especially on smaller maps may yield a significant advantage. The configuration for the testing is detailed on 12.7.

The results are shown in 3 tables. Table 12.1 displays win rates ignoring draws, while Table 12.2 illustrates win rates scaled to include draws. In both tables, we show the win rate of the row agent against the column agent. Table 12.3 shows the average scores the heuristic agents achieved in their match-ups.

Table 12.7: Round-robin testing game configuration

Parameter	**Value**
Map	YangEtAlMap
Number of games played	100000
Seed	1964
Parameters	Random agents' probability for random moves = 50%
Platform	DCS cpu-batch

	128	256	1024	2048	3072	4096	5120	6144	7168	8192
128	48.29	35.44	9.8	11.98	20.71	26.48	7.69	29.14	13.53	13.53
256	60.72	47.71	20.52	24.49	32.5	32.66	19.77	45.43	22.83	22.83
1024	88.92	77.32	47.05	54.61	66.76	76.67	38.93	61.03	60.36	60.36
2048	86.6	72.84	39.8	47.27	60.9	72.38	33.47	54.14	56.7	56.7
3072	77.02	64.82	29.79	35.49	48.48	56.41	24.32	48.3	43.71	43.71
4096	70.0	63.95	20.71	24.84	40.4	47.59	13.27	38.94	23.16	23.16
5120	91.15	78.19	55.48	61.27	72.33	84.53	47.05	61.65	66.24	66.24
6144	68.54	51.07	36.81	43.4	49.14	58.36	35.84	48.56	48.31	48.31
7168	84.49	74.48	34.84	38.68	52.81	74.02	29.03	49.09	48.15	48.15
8192	84.49	74.48	34.84	38.68	52.81	74.02	29.03	49.09	48.15	48.15

Figure 12.1: Winrates of heuristic agent round-robin.

	128	256	1024	2048	3072	4096	5120	6144	7168	8192
128	50.0	36.86	9.92	12.15	21.2	27.45	7.78	29.83	13.8	13.8
256	63.14	50.0	20.98	25.16	33.4	33.81	20.19	47.08	23.46	23.46
1024	90.08	79.02	50.0	57.84	69.14	78.73	41.23	62.38	63.4	63.4
2048	87.85	74.84	42.16	50.0	63.18	74.45	35.33	55.51	59.45	59.45
3072	78.8	66.6	30.86	36.82	50.0	58.27	25.16	49.57	45.28	45.28
4096	72.55	66.19	21.27	25.55	41.73	50.0	13.56	40.02	23.83	23.83
5120	92.22	79.81	58.77	64.67	74.84	86.44	50.0	63.24	69.53	69.53
6144	70.17	52.92	37.62	44.49	50.43	59.98	36.76	50.0	49.6	49.6
7168	86.2	76.54	36.6	40.55	54.72	76.17	30.47	50.4	50.0	50.0
8192	86.2	76.54	36.6	40.55	54.72	76.17	30.47	50.4	50.0	50.0

Figure 12.2: Winrates (with draws removed) of heuristic agent round-robin.

	128	256	1024	2048	3072	4096	5120	6144	7168	8192
128	3.96	2.18	2.4	2.95	2.88	2.54	2.37	4.23	3.97	3.97
256	5.31	4.08	5.18	5.62	4.8	3.54	4.77	6.26	5.75	5.75
1024	13.73	11.61	12.16	12.92	12.76	12.67	11.36	13.33	12.99	12.99
2048	13.57	11.01	11.45	12.44	12.34	11.8	10.75	12.72	12.79	12.79
3072	10.03	7.49	7.89	8.8	8.63	8.02	7.49	9.77	9.62	9.62
4096	6.05	4.75	5.55	5.73	5.57	5.23	4.79	6.14	5.93	5.93
5120	14.28	11.09	13.19	13.67	13.7	13.82	12.75	13.62	13.73	13.73
6144	11.11	6.24	9.1	10.25	9.92	9.46	9.43	12.02	11.3	11.3
7168	12.86	11.09	11.13	11.85	11.36	11.78	10.47	12.33	12.19	12.19
8192	12.86	11.09	11.13	11.85	11.36	11.78	10.47	12.33	12.19	12.19

Figure 12.3: Average scores of heuristic agent round-robin.

Round-robin testing revealed the differences in the skill levels of the agents. Hoarder Bot (5120) was the most well performant by far, boasting a win rate of over 60% against all other bots, with the exception of the base agent (1024), against which it achieved approximately a 55% win rate. Surprisingly, base agent is the second strongest, suggesting that heuristics such as exclusively drawing from the pile or drawing extra tickets are not well-suited to the base agent's core strategy. Generally, the worst performing agents

were the ones incorporating varying degrees of randomness, with performance worsening as randomness increased. As expected, the Fully Random Agent exhibited the poorest performance among all agents.

Villain agent, which uses exclusively sabotage strategies, exhibited poor performance against most bots that either do not involve randomness or incorporate it partly. While it was somewhat successful in lowering the average scores of a few other agents, it was not able to complete its own tickets consistently, resulting in significant point losses.

Another noteworthy observation is that Pile Bot (7168) and Pile Ticket Hater (8192) achieved equal win rates against each other. This is attributed to their shared heuristic of exclusively drawing cards from the pile. However, in the small map used for round-robin testing, neither agent draws additional tickets, since Pile Bot does not ever have enough train cars after completing its path. Consequently, their behaviour is deterministically homogeneous, resulting in these outcomes.

The resulting ranking of heuristic agents is depicted in the following graph 12.4.

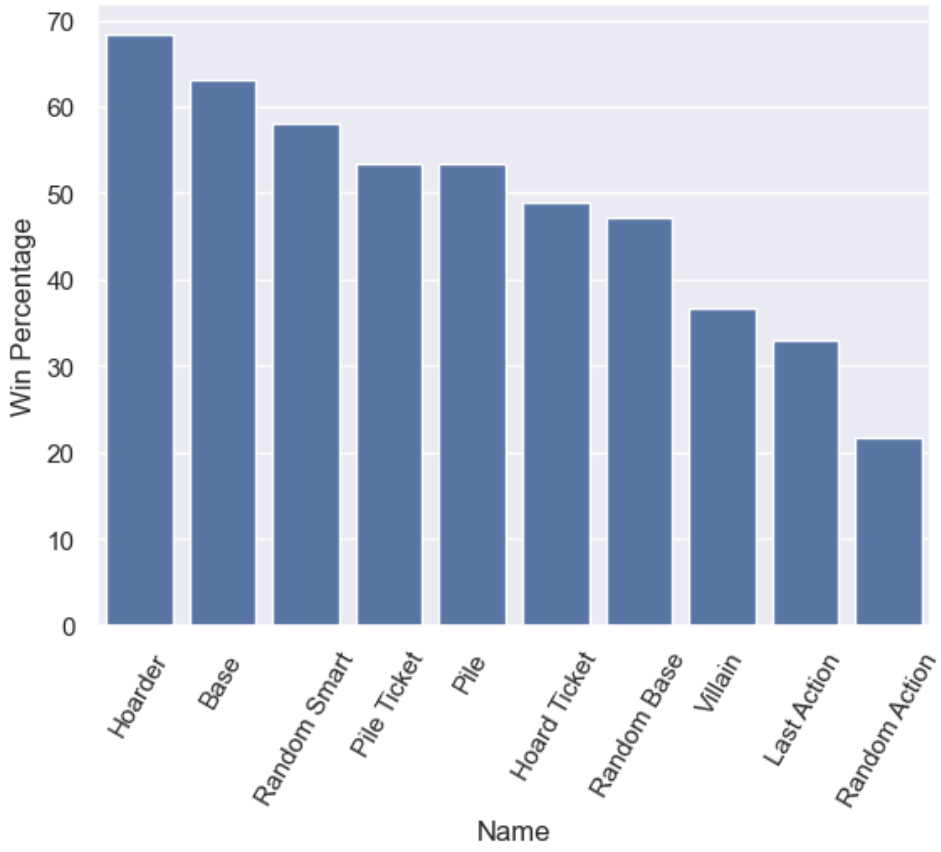


Figure 12.4: Ranking of heuristic agents based on average win rate resulting from round-robin testing

12.4.3 Hyperparameter Tuning

For the purposes of tuning parameters, we selected half of the heuristic agents in order to decrease the computational burden of running tests. Moreover, it helps prevent over fitting and allows us to see if the results generalise to all the heuristic agents in the 12.4.4 section. Table 12.8 shows the set-up for all remaining automated tests.

Table 12.8: DDA testing game configuration

Parameter	Value
Map	YangEtAlMap
Number of games played	500
Seed	500
Parameters	MCTS iterations = 30,000
Platform	DCS cpu-batch

12.4.3.1 Exploration Constant

Our agents require a number of hyperparameters that need to be tuned. For all agents, we have the exploration constant C_p in the UCT equation

$$UCT = \bar{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

which dictates the importance of the exploration term. As a reminder, the greater the value of C_p, the more the MCTS algorithm prioritises exploring nodes that have been visited less before. The purpose of tuning C_p is to find a value which maximises the strength of both CMCTS and ISMCTS, hence providing a good basis for their respective DDA variations.

As shown in Figure 12.5, we trialled C_p values between 0.5 and 7, which seemed a reasonable cap to not dominate the entire UCT equation with the exploration term, effectively giving all nodes equal importance irrespective of their score. The exploration constant has a small but non-negligible effect on the MCTS algorithm, producing a 2% and 4% difference in the win rates of CMCTS and ISMCTS against heuristic agents respectively. Through tuning, we find that the optimal value are $C_p = 3$ and $C_p = 4$ for CMCTS and ISMCTS respectively. Another important observation is that CMCTS performs significantly better against heuristic agents compared to ISMCTS, which is an expected result

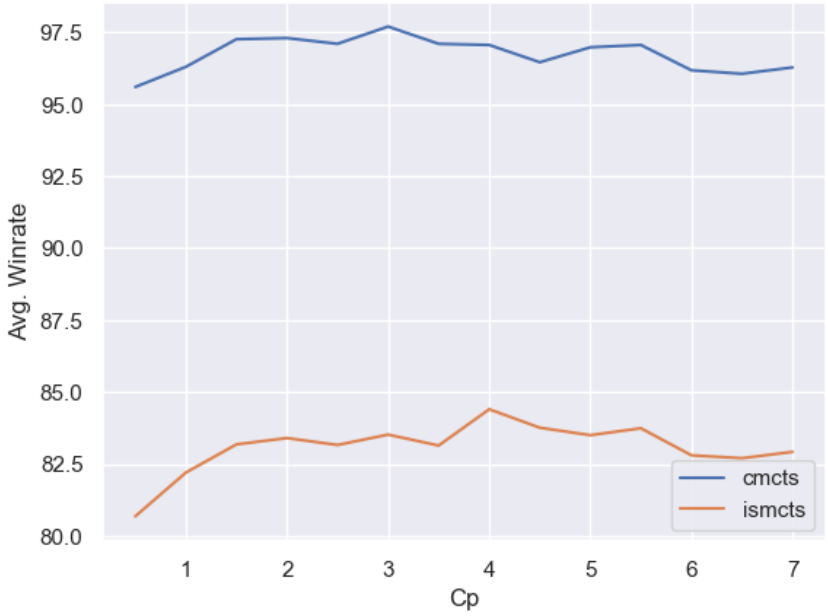


Figure 12.5: Average win rate against value of exploration constant C_p for CMCTS and ISMCTS.

as CMCTS has access to perfect information. It is also interesting to note that CMCTS prefers a lower exploration constant, suggesting that exploration is less beneficial when the correct game state is known.

12.4.3.2 POSAS Threshold

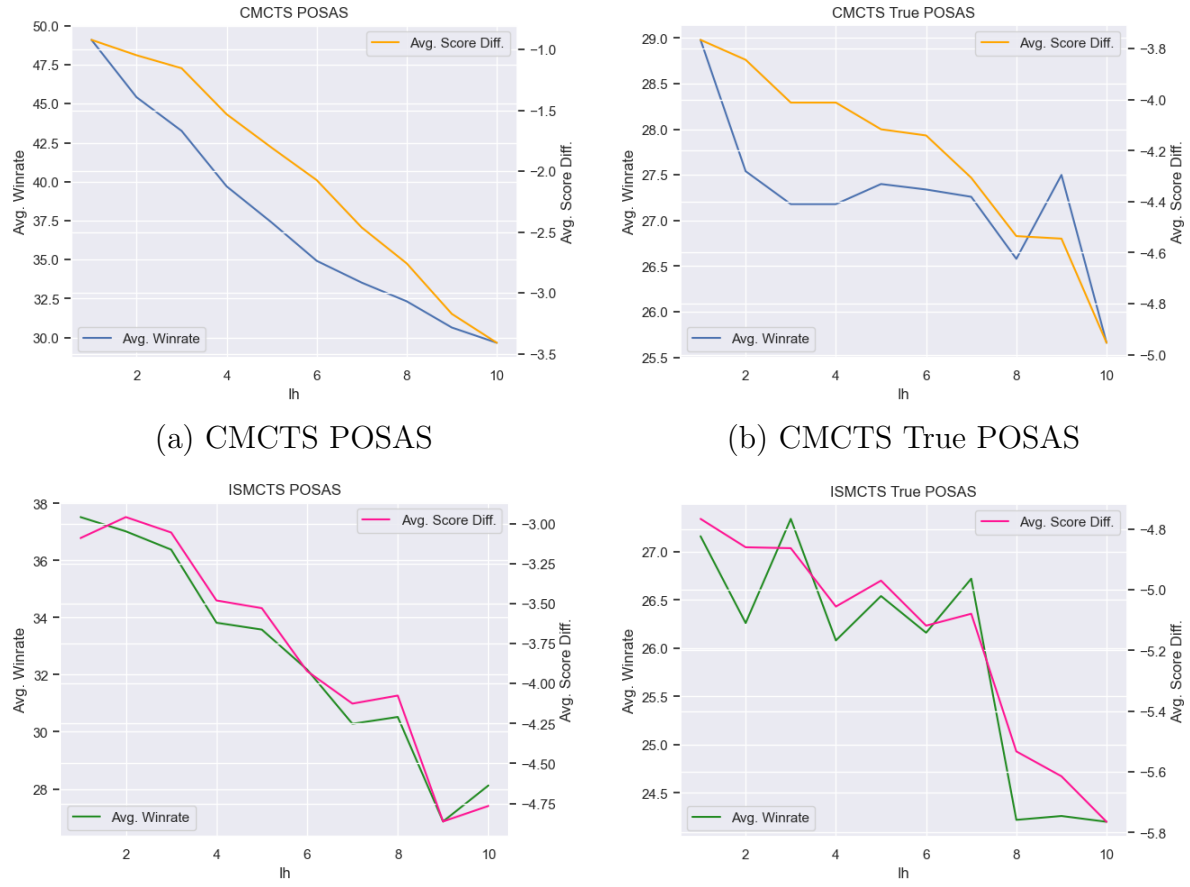
Moving onto DDA parameters, we vary the POSAS threshold I_h in

$$action = \arg\max_{a} -(|r[a].score| - I_h)^+$$

which determines the range of scores around zero in which an action has uniform chance to be selected.

We chose the range 1 to 10 for the POSAS threshold, based on the average score of non-DDA MCTS agents, which are around 15. Then, a POSAS threshold of up to a score of 10 should cover most of the actions the agents can choose from. The results are shown in Figure 12.6. For DDA agents, the performance we are looking for is an average win rate of around 50%, and an average score difference of 0. As we can see in the diagram, increasing the POSAS threshold reduces the win rate of both agents, and the closest they get to our desired performance is actually with a POSAS threshold of zero. Furthermore,

true POSAS significantly decreases agent performance, even at a small value, reflected by a 20% drop in win rate for both agents.



(a) CMCTS POSAS

(b) CMCTS True POSAS

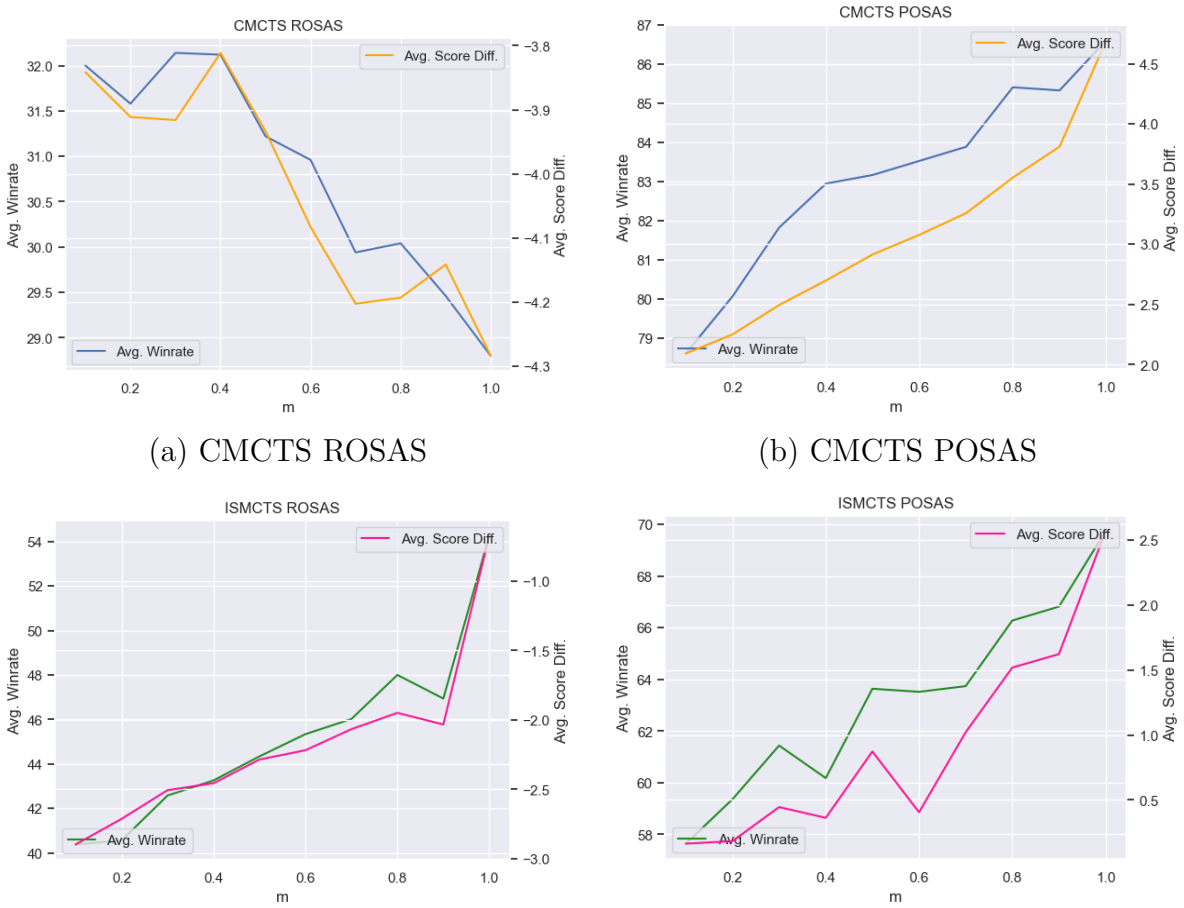
(c) ISMCTS POSAS

(d) ISMCTS True POSAS

Figure 12.6: Average win rate and score difference against value of POSAS threshold I_h for CMCTS and ISMCTS.

12.4.3.3 Momentum

Finally, we look at the effect of the momentum constant for the momentum term we devised as a modification. The higher the momentum constant, which we will refer to as m, the larger the momentum term, which means the DDA algorithm will prefer actions that minimise the score difference between itself and the previous action. Figure 12.7 shows the results of this tuning. Interestingly, CMCTS as well as ISMCTS POSAS are negatively affected by momentum, whereas ISMCTS ROSAS is improved by it. We conclude this by observing that while the other agents' win rate get further from 50% as momentum increases, ISMCTS ROSAS has win rate which increases from 40% to 50% as m increases from 0 to 1.



(a) CMCTS ROSAS

(b) CMCTS POSAS

(c) ISMCTS ROSAS

(d) ISMCTS POSAS

Figure 12.7: Average win rate and score difference against value of momentum threshold m for CMCTS and ISMCTS.

In conclusion, we have found that the optimal exploration constants are $C_p = 3$ and $C_p = 4$ for CMCTS and ISMCTS respectively. We then found that increasing the POSAS threshold decreased the DDA performance for all variants. Lastly, we found the same was true for momentum, except for ISMCTS ROSAS which is improved by increasing momentum.

12.4.4 Final Test Results

In this summary, we analyse and compare the performance between a selection of DDA agents. We do not consider the true variants in this section, since it has already been shown that their performance is much worse. For the purposes of these tests, we used a POSAS threshold of 5 and momentum constant of 0.5 as a representative middle value between those tested. Results are collected for the following agents: CMCTS ROSAS, ISMCTS ROSAS, CMCTS POSAS ($I_h = 5$), ISMCTS POSAS ($I_h = 5$), CMCTS ROSAS

($m = 0.5$) and ISMCTS ROSAS ($m = 0.5$). We also selected an agent that performed particularly well during parameter tuning, which is ISMCTS POSAS ($I_h = 5$, $m = 0.1$). The final results were collected for all 10 heuristic agents in order to test a wider range of strategies and present more reliable conclusions. The full set of result tables can be found in Appendix C.

We use two metrics here to evaluate the effectiveness of the DDA: the win rate difference is the difference between the agent and its opponent's win rate, and the average score difference is the average difference between the agent and its opponent's score at the end of the game. Absolute values of the differences have been used here.

The first interesting comparison to make is between CMCTS ROSAS and ISMCTS ROSAS, which are the two simplest forms of DDA. We predicted that CMCTS would perform better, but this was not the case: CMCTS ROSAS achieves an average win rate difference of **48.99%** and average score difference of **3.77** versus ISMCTS ROSAS which achieves **35.19%** and **3.18** respectively suggesting ISMCTS produces closer games. Despite the fact that CMCTS achieves a higher maximum playing strength, the ROSAS version is a weaker opponent compared to ISMCTS ROSAS suggesting it is somehow over compensating. A possible explanation for these observations is that the assumption made by CMCTS that the opponent also has perfect information results in inaccurate evaluation of game states, overestimating the strength of the opponent. However, CMCTS ROSAS produces a much larger draw rate at **18.77%** compared to **8.29%** which is impressive.

Generally, the DDA agents seem to be achieving lower win rates than their opponents, despite the fact they can comfortably beat all heuristic agents on their maximum strength. This mirrors the results produced by Demediuk et al. in the 2D fighting game [23]. However, it seems the results they achieved produced win rates closer to 50%, which is not surprising since TTR requires much more long term planning and the determination of the currently winning player is less clear-cut.

Now we look at how POSAS and momentum affect the performance of CMCTS. To start with, CMCTS POSAS produces significant performance improvement over the ROSAS version with a win rate difference of **34.61%** and average score difference of **1.99**, bringing its performance inline with that of ISMCTS ROSAS. It appears that the additional randomness and proactive play helped the agent, but at the expense of reducing the draw rate to **6.65%**. On the other hand, the addition of momentum slightly degraded performance of CMCTS ROSAS with win rate difference of **55.32%** and average score

difference of **3.99**, but a high draw rate of **19.48%** was maintained. This is expected since momentum slows down the rate at which the agent adapts its difficulty, but the performance decline was not large and may be a worthy trade-off if it improves the experience of the user.

POSAS and momentum had opposite effects on the ISMCTS agent, which is highly intriguing. Firstly, POSAS decreases the effectiveness of ISMCTS producing a win rate difference of **41.51%** and average score difference of **3.74**. Given that ISMCTS selects random determinzations at the start of each iteration of the algorithm, it is likely that the addition of POSAS introduces too much randomness for the effective selection of actions to keep the game in balance. If the maximum strength of ISMCTS were to be increased, perhaps we would see the same trend with CMCTS where POSAS created improvements. In contrast to CMCTS, momentum increases performance for ISMCTS ROSAS achieving a win rate difference of **30.78%** and average score difference of **2.89**.

During parameter tuning, we observed that ISMCTS POSAS ($I_h = 5$, $m = 0.1$) performed particularly well. For both CMCTS and ISMCTS, a combination of both POSAS and momentum allow the agent to beat the opponent on average in contrast to every other combination. The table in Figure 12.8 shows the full collection of results for this agent. There is still a clear relationship between the strength of the heuristic agents found during the round-robin tournament and the achieved win rate. However, on average there is only a difference of **1.90** in the final scores and a win rate of **56.47%** for the agent is achieved, demonstrating effective DDA. The average is skewed higher by the random bot, without this agent, the achieved average score difference is **1.23**.

ISMCTS POSAS (Ih=5) (m=0.1)	win %	draw %	Win % Difference	Average Score Difference
TakeRandom Turn (128)	91.3	2	84.6	6.75
TakeLastTurn (256)	68	3.9	39.9	0.951
Villain (4096)	69.2	4.2	42.6	1.977
RandomBase (3072)	65.5	4.9	35.9	2.234
HoarderTicket (6144)	59.7	5.1	24.5	0.173
Pile (7168)	42.8	5.6	8.8	0.945
PileTicket (8192)	42.8	5.6	8.8	0.945
RandomSmart (2048)	51.1	3.6	5.8	0.345
Heuristic Agent (1024)	39.3	4.7	16.7	2.169
Hoarder (5120)	35	4.8	25.2	2.536
Average	**56.47**	**4.44**	**29.28**	**1.9025**

Figure 12.8: Final results for the ISMCTS POSAS ($I_h = 5$, $m = 0.1$) agent versus ten heuristic agents.

We wanted to test how well the best performing DDA agent adjusts to heuristic agents

that change their strategies mid-game. Thus, we tested ISMCTS POSAS ($I_h = 5$, $m = 0.1$), against three swap bots. Two of these bots swap in between the fully random agent and the base heuristic agent, while the third one starts as the hoarder agent and switches to the random base agent. These specific combinations were chosen based on the diverse relative strengths of the agents, as shown in the heuristic agent ranking 12.4.

In terms of win rates, the ISMCTS agent achieved a win rate close to 50% against one of the swap bots, but it consistently outperformed the other bots. However, the average score difference in all games was relatively close to 0, signifying that the DDA is having a positive effect in creating more balanced games. The agent performs best when the swap bot starts with a worse random strategy and finishes with a heuristic strategy compared with the other way around, which in theory should yield the same win rate.

ISMCTS POSAS (Ih=5) (m=0.1)	win %	draw %	Win % Difference	Average Score Difference
128 -> 1024	47	5.1	-0.9	-0.709
1024 -> 128	83.9	3.4	71.2	3.862
5120 -> 3072	68.6	8.1	45.3	2.048
Average	**66.5**	**5.533333333**	**38.53333333**	**1.733666667**

Figure 12.9: Results for the ISMCTS POSAS ($I_h = 5, m = 0.1$) agent versus 3 heuristic swap agents

We wanted to investigate whether setting a target value above zero would improve the performance of the agents such as CMCTS ROSAS which was not playing strong enough to match its opponents. We ran the tests for this agent again but set the target value for the average reward to be three instead of zero. The table in Figure 12.10 shows the positive results of this experiment.

CMCTS ROSAS (target=3)	win %	draw %	Win % Difference	Average Score Difference
TakeRandom Turn (128)	75.9	5.8	57.6	1.052
TakeLastTurn (256)	59.1	9.8	28	1.368
Villain (4096)	63	10.3	36.3	0.37
RandomBase (3072)	51.8	6.4	10	1.132
HoarderTicket (6144)	44	4.7	7.3	3.497
Pile (7168)	40.8	7	11.4	2.042
PileTicket (8192)	40.8	7	11.4	2.042
RandomSmart (2048)	38.9	6.1	16.1	2.433
Heuristic Agent (1024)	35.2	5.8	23.8	2.82
Hoarder (5120)	31.1	7	30.8	3.052
Average	**48.06**	**6.99**	**23.27**	**1.9808**

Figure 12.10: Results for the CMCTS ROSAS agent that targets a score which is three points greater than its opponent.

This agent achieves the best win rate difference of **23.27** out of all agents investigated, but the average score difference of **1.98** is slightly higher than that of ISMCTS POSAS ($I_h = 5, m = 0.1$). This is not unexpected since the agent is now targeting a score difference of three. This approach is unlikely to generalise to opponents that fall out of the skill range of these heuristic agents. For example, a larger target score difference would need to be provided for stronger opponents like humans. Therefore, we believe the approach using ISMCTS with POSAS and momentum to be more robust.

12.5 User Acceptance Testing

The goal of user acceptance testing is to help verify that we have met our objectives relating to the experience of the users, such as enjoyment and suitable challenge. There are two aspects to this task: first, we would like to assess the intuitiveness, and overall usability of the GUI via guided test cases; then, we would like to investigate effectiveness of our DDA from a human perspective - that is, how human players feel about the behaviour of the DDA, whether it feels natural, engaging, and challenging to play against, and whether it is comparable to playing against a real opponent. We would also like to keep track of the win rate of users to see how the DDA agent performs against players of various skill levels.

To carry out the user acceptance testing, a total of 12 testers of varying skill level are recruited, and the test is conducted either in person or in an asynchronous remote setting. The skill level is defined here as the tester's self-reported proficiency in strategy/board games. The setup of the testing environment is as follows:

Table 12.9: User testing environment parameters.

Parameter	**Value**
Device	Windows or Linux machine
Map	Adaptation of TTR New York map
Adversary	CMCTS DDA agent
Number of games	3
Estimated rounds per game	20
Estimated time taken per game	15 mins

The testers are across a wide age range with the majority (66.7%) between 16–25 years old, with a general interest in board games, with half reporting they play board games "often" and the rest "sometimes". 58.3% have played Ticket to Ride previously, and half of those say that they are good at Ticket to Ride. Finally, testers all launched the application in Windows 10 or 11, and none on Linux.

12.5.1 Guided Test Cases

For the guided test, testers are asked to carry out a number of specific tasks. The goal of this is to assess the ease of use of the system - for players to be able to play against the DDA agent fairly, it is important that they are not hindered by confusing display and controls. Further, a high user satisfaction in the GUI speaks to adherence to good UI design principles, and in turn good software engineering.

Testers are prompted to set up and open the game, after which they are asked to carry out the following tasks:

- (On starting the game) Describe your player statistics (Which player are you? How many locomotives and how many train cars do you have? What tickets do you have and how many points are they worth? Can you find the route to complete one of your tickets on the map? What information do you know about your opponent?)
- Draw the required train cards to complete the route between Golden Sands and Ivory Street, and claim the route.
- Draw and keep one ticket with the highest score, then find the ticket in your hand and describe the route.
- Describe the opponent's last action.
- (Around halfway through the game) Describe your current score, including any added/deducted by completed/incomplete tickets.

For each task, testers are prompted to describe whether they felt the task was difficult or unintuitive to complete, and share any thoughts they have on the design changes that could be made to improve it. Generally, testers found the tasks easy to complete, with around 70% of responses agreeing that the tasks were "not difficult". There are some particular areas of concerns, however, which spawned a number of comments on potential improvements for the GUI.

As user testing spans roughly two weeks, intermediate reviews of user feedback are conducted for the sake of iterative improvement. In practice, this means changes are made to the GUI after every batch of user feedback. This comes at the sacrifice of a slightly non-homogeneous survey summary (for example, user rating for game accessibility may improve as time goes on), but as collecting user test results takes a significant amount of time, we do not have the resources to conduct multiple rounds of user tests. Below,

we demonstrate some of the improvements made to the GUI as a result of user feedback, as the initial implementation, though complete, overlooks a number of quality of life and ease of access factors. There are many more such fixes and updates, but we only select a few to discuss for brevity.

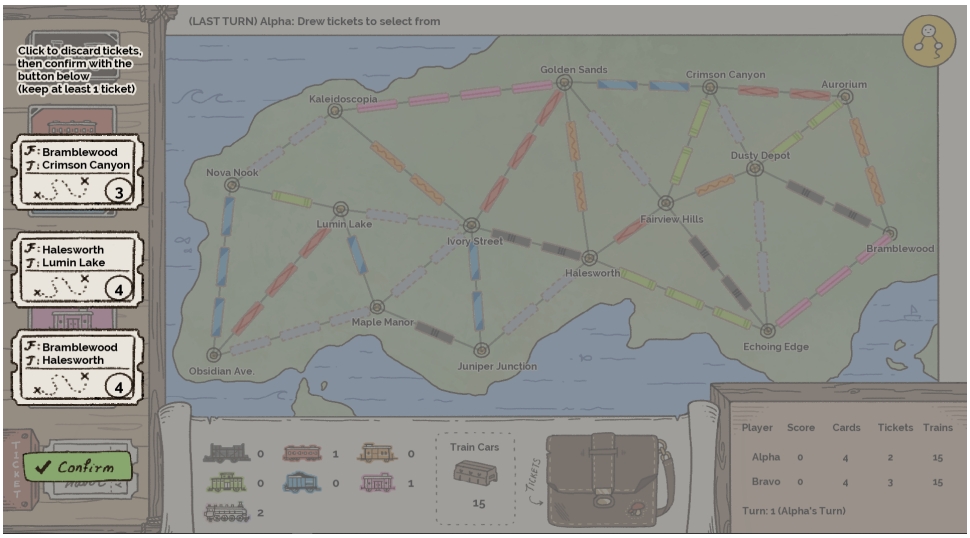
MCTS Status Display. This indicator was added after testers reported that the game would freeze after they take a turn. This is due to the MCTS running its simulations to determine the next action, but it was not obvious. The "thinking..." status now appears when the agent is taking its turn, making it obvious that the opponent is taking its turn in a natural and informal way.

Figure 12.11: Thinking status display for when the MCTS agent is taking its turn.

Removal of [Click to Continue]. A fix requested by multiple testers. Initially, the game was designed so that the user can click anywhere on the screen to let the opponent start its turn, after the player finishes their turn. This is often forgotten by players, who then waste minutes waiting for the opponent. Now, after a player carries out their turn, the game automatically progresses, which results in a much smoother play experience.

Ticket Selection Screen Initially, upon drawing tickets the player is greeted with a ticket selection view, which presents the three drawn tickets and allows the player to pick ones to keep (Figure 12.12a). This was a pleasant interface, but testers found it difficult to strategise as tickets blocked the view of the map. In the updated version (Figure 12.12b), tickets are instead contained in the card track, which leaves a full view of the map. The background remains greyed out to remind players that they cannot interact with the rest of the game until ticket selection is complete.

(a) Old ticket selection screen.



(b) Updated ticket selection screen.

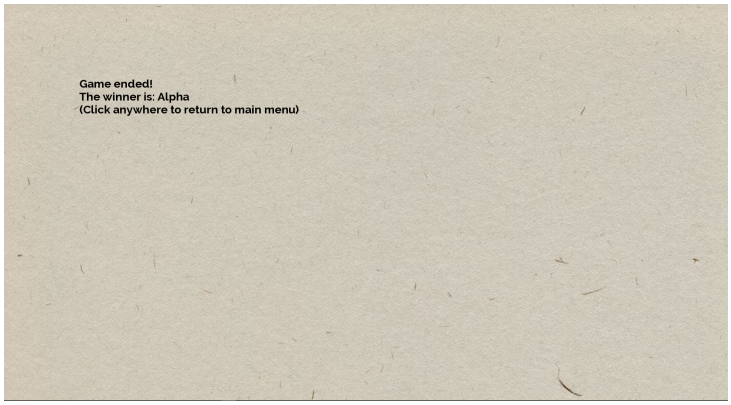
Action Banner Improvements. After testers reported that the action banner is sometimes difficult to understand, the actions were updated to be more user-friendly. For example, drawing one card from the table and one from deck was previously "Draw one from table (blue)", which confused many testers, and is now updated to "Drew one card from deck, and one from table (blue)". The route claim action was also not user-friendly, and was changed to "Claimed route from A to B, using colour: blue".

Final Round Signal. After a player gets to less than 3 train cars, the other player gets to take another turn, and the game ends. In a real life setting, the finishing player would make an announcement, but this was not obvious in the game as the turns are controlled by `GameController`. Testers were often caught off-guard by the sudden ending of the game, so to address this, an indicator is displayed when the human player is taking their

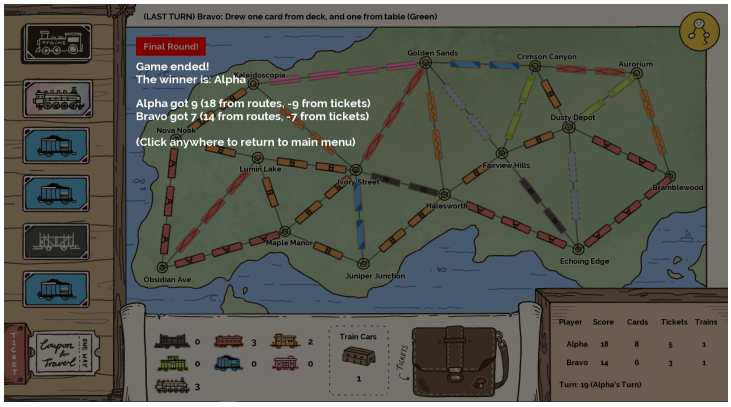
last turn.

Figure 12.13: Enter Caption

End Game Display. At the end of the game, the application initially had an extremely bare-bones display (Figure 12.14a), only showing which player has won. Players were not able to look at the final state of the map, nor are they informed of player scores. This was updated in Figure 12.14b, which now shows player scores, as well as a simple breakdown, detailing the exact points gained from routes and from tickets.



(a) Old game end display.



(b) Updated game end display.

12.5.2 Freeplay

The freeplay section of user testing contributes to the assessment of whether we were able to fulfil our first objective, stating that the DDA agent should offer a balanced and appropriate level of challenge to players of any skill level, and that players should find the DDA agent enjoyable to play against based on key products indicators such as turn time, human-likeness, and challenge level.

Initial user testing showed that ISMCTS is significantly weaker than CMCTS, rarely able to win against human players, even at full strength (i.e. non-DDA version). This is reflected in initial casual plays, where the ISMCTS agent (both DDA and non-DDA) lost by more than 20 points in all 5 games. Hence, ISMCTS was not extensively tested in wider user tests, as the team wanted to focus on CMCTS, the stronger version of the two, as well as the effectiveness of modifications made (POSAS, momentum).

The results for three DDA agents, CMCTS ROSAS, CMCTS POSAS, and CMCTS with momentum, are summarised in Table 12.10. For each agent, users are asked to rate four perceived attributes out of 10: their overall enjoyment of the game, whether the turn length is too long, whether the agent felt human-like, and whether they felt that the agent was cheating. Finally, we also track the score differences of each agent against users (a negative score means the agent lost on average).

Table 12.10: Performance and user ratings for CMCTS DDA agents.

	ROSAS	**POSAS**	**Momentum**
Enjoyment	8.64	9	8.78
Turn Length	5.36	4.88	3.44
Human-like?	4.64	5.63	4.67
Cheating?	1.36	2	2.22
Avg. Score Diff.	-10.45	-12.88	-13.11

Overall, the DDA agents won 2, drew 2, and lost 26 games against testers. We found that POSAS with momentum made the CMCTS agent weaker against testers, but this did not necessarily decrease enjoyment. In fact, both modifications slightly increased the average user enjoyment of the game. Testers had mixed opinions on whether the agent took too long on its turn, and though we see a decrease in perceived turn length, we have shown previously that each of the agents take the same amount of time to run, hence it may be the case that testers become more tolerant of the wait as time goes on. Testers were also mixed on whether the agent felt human-like, with an average of 4.6 to 5.6 for each agent. Finally, testers reported that they did not feel the agent was cheating.

Furthermore, as we aim to provide balanced play experiences to all players, we separate the players into 3 groups based on their self-reported skill level in Ticket to Ride: good, okay, and never played. Interestingly, the average score of the agent against players who reported they were good at Ticket to Ride is -9.89, whereas it is -15.92 against those who reported they were okay at Ticket to Ride, and -8.8 against those who have never played Ticket to Ride before. This suggests the DDA agent might not strictly perform worse against better players.

As an extension, we asked testers to try their skill against the full-strength, non-DDA version of the CMCTS agent, and asked for their best score. A total of 8 testers participated in this trial, of which 4 were able to win against the agent, resulting in an average agent score of 0.25. However, testers felt that the agent was obviously cheating (6.56/10), as it played very aggressively and often moved to block off its opponent.

12.5.3 User Testing Summary

Overall, users found Coupon for Travel aesthetically pleasing (9.58/10), adequately intuitive (7.15/10) and accessible (7.25/10), easy to install (8.3/10), and most thought that they could see themselves playing more of the game (8.75/10). Testers were largely able to win against the DDA agent, and generally enjoyed the games played. Testers did not feel that the DDA agent was very human like, but they also did not feel like the agent had an unfair advantage. Finally, the self-reported skill level of testers does not have a direct correlation with the game outcomes.

There were a number of limitations to our approach to user testing, the most prominent of which is the sample size. Due to the multitude of aspects to be tested and the length of the game itself, the survey takes testers a long time to complete ($\sim$1hr). Combined with the requirement for desktop installation, and the shorter timeframe for user testing due to project crunch, we received only 12 responses. Given the subjective interpretations of "enjoyment", "human-likeness", and "difficulty", while user testing results so far indicate promising results, many more responses will need to be collected before we are able to formally make claims about the efficacy of the application and agent performance.

Chapter 13

Evaluation

Following on from the testing chapter, we carry out an evaluation based upon our test results. Specifically, we will summarise and analyse the performance, DDA efficacy and user testing results, and evaluate this against our project objectives to assess the success of the project. Finally, we look back on the management methodologies put into place over the course of the project, in an effort to evaluate the effectiveness of these methodologies and improve upon them in future projects.

13.1 Results

We were able to implement our DDA agents in a game environment that allowed for fast MCTS iterations with times in the order of 10^4ns. This has allowed us to feasibly use up to 10^6 iterations per turn against players, whilst maintaining sub minute turn times that user testers found somewhat long but acceptable. The large number of iterations is sufficient to provide challenge for players against full strength CMCTS on a map with 16 routes and provide a fair challenge on a map with 30 routes. We are pleased with these results since they validate our choice of technologies, game environment design, implementation and optimisation.

In addition, we have found that DDA modifications have not greatly impacted performance, with average game time increasing by at most 47%, with the use of ISMCTS with the POSAS modification. Consequently, we did not have to make performance considerations when evaluating our best DDA agent. Our game simulation results are fully reproducible due to careful consideration during design and implementation. We provide all the parameters and in some cases the whole commands are included. We encourage

readers to reproduce results of our tests or run test cases of their own.

During DDA efficacy testing, we analysed how different combinations of modifications and changes in parameters affect the performance of the agents. We found that all the true variants of the DDA modifications performed worse. In general CMCTS did not perform as well as ISMCTS which was unexpected, but potential reasons are discussed such as the assumption of perfect information. We found that POSAS and our proposed modifciation momentum had opposite effects: for CMCTS, POSAS increased efficacy, but momentum decreased it and vice versa for ISMCTS. A particular combination that was found to be effective was ISMCTS POSAS ($I_h = 5$, $m = 0.1$) for which the synergy of modifications helped it produce the lowest average score difference seen compared to other agents. However, other agents provided optimal results for other metrics: CMCTS ROSAS with a target score difference of three achieved the lowest win rate difference seen and CMCTS ROSAS ($m = 0.5$) produced the largest draw rate. It is clear there this no one best approach, but some are better than others. More work should be done on how these different metrics translate into engaging gameplay experienced by the user.

For user acceptance testing, we focused on the stronger variant, CMCTS, after initial performance comparisons of the two agents. We tested three variants of CMCTS: ROSAS, POSAS, and with momentum. We found that testers enjoyed playing against all the variants, especially CMCTS POSAS, despite the fact that the agents almost always lost. Testers thought that the agents had somewhat human-like behaviours, but this was not fully convincing. Despite the fact that CMCTS DDA has perfect information, testers did not feel like the agent was cheating, which is significantly better than CMCTS at full strength, where testers felt the game was very unfair. We observed that the skill levels of testers were not directly correlated with their scores against the DDA agents, which suggests the DDA works well against different opponent skill levels, however as the skill statistic is self-reported, and we only have a few samples, it is not enough to draw conclusive results. ISMCTS was less explored during user testing due to the lower maximum playing strength being too weak for humans.

In terms of the user application, testers really enjoyed its style but believed accessibility features could be improved. We received a large number of feedback regarding improvements, some of which we were able to implement. Overall, we believe that we have delivered a polished product that users gain enjoyment from and demonstrated that the explored techniques can achieve DDA behaviour that improves the balance of the game in strategy games like TTR.

13.2 Fulfilment of Objectives

In this section, we review our project objectives that we outlined in Chapter 3, by evaluating them against project outcomes and discussing the extent to which each objective has been fulfilled.

1(a). The user survey indicates that testers of all skill levels enjoyed playing against the MCTS agent, with an average enjoyment rating of 8.81 out of 10. Many testers agreed that they can see themselves playing more of the game, which is a further indicator of enjoyment. These results largely fulfils the objective, however we do not have enough play data to verify whether players are able to improve by playing against the DDA agent.

1(b). As shown in our final results, we produced agents with average win rates close to 50%. However, it is clear the more work will be required in order to achieve a truly consistent win rate across skill levels. We discussed further metrics which show success including draw rate and win rate difference, with different agents showing strengths in these areas.

1(c). On top of the discussion in 1(a), testers reported that the agent's turn times are somewhat long (6/10), and that the agents feel somewhat human-like (5/10) in their plays. They did feel the agent's plays were fair (8/10). Based on the previous metrics, we believe the agent has performed satisfactorily.

2(a). Testers rated the user interface 7.15 out of 10 for intuitiveness, 7.25 out of 10 for accessibility, and 9.58 out of 10 for style. Though there are areas of improvement, many testers have left positive comments about the interface, which all point to the successful fulfilment of this objective.

2(b). The application can be launched via an executable on Windows machines, and testers found this installation process easy (8.3/10). Although there were no responses from Linux users, Linux installation has been tested and is straightforward for those with prior Linux experience. In addition, we were able to develop the application simultaneously on Windows, Linux and Mac devices. Overall, the game was able to fulfil the criteria of being able to be installed on several platforms.

All project deliverables are successfully fulfilled, with the exception of settings and sound for the game application. Though these features would improve the polish of the game, they were ultimately dropped due to time constraints. However, we were able to deliver on

all additional deliverables, including a testing application for game simulation, heuristic agents for DDA efficacy testing, and variations of the base MCTS DDA algorithm.

13.3 Methodology Evaluation

In this section we will evaluate the project methodologies and discuss whether they were successful, and how they could be improved. As we outlined in Section 7.6, we are broadly content with the management techniques employed.

The scrum based management methodology has proven to be a suitable structure to base our work about. In particular, we found regular sprint reviews with our supervisor to be important milestones to work toward, and scrum artefacts such as the sprint board to be useful ways of visualising scope. We did find this structure became less important towards the termination of the project, in particular Sprint Retrospectives outlived their usefulness. The flexibility afforded by our methodology has allowed us to continually refine scope and adapt our user application, which has led to a better product. The supplementary Gantt chart and planning spreadsheets were a useful guide to help us meet key deadlines such as the poster presentation.

We were disappointed by the availability of our client, who was less available than we had hoped. However, we were able to partially mitigate the gap left by the client, getting feedback directly from end users throughout development.

We were caught out by the time taken on software engineering tasks, in particular the time taken to implement the first MCTS model. However, we are proud that we have not had to compromise on our core features, and have been able to successfully implement and analyse many MCTS modifications due to additional hours committed by each team member. Our feasibility study helped us to better anticipate the project complexity and informed us of unrealistic features. Now we have completed the project, we are grateful that we decided not to investigate reinforcement learning agents, as it would require too much development time given our constraints.

The division of responsibilities were particularly effective tools. Each team member found that single points of responsibility allowed specialisms to develop, which allowed richer features to be developed. This had the greatest utility during the first half of the project, when research could be carried out on DDA agents in parallel with the development of the game environment.

The focus of generalisation during design has allowed us to effortlessly integrate new maps. For example, the MiniMap used to validate MCTS behaviour was implemented in under ten minutes. It also allows users to produce their own maps, and allows us room to expand in the future on the user application. Furthermore, the philosophy of generalisation allowed the simple addition of new heuristic agents and configurations of existing agents, with minimal modifications to the game environment.

External code profiling tools allowed us to check memory safety and identify regions for optimisation. We believe our data-driven optimisation strategy is the reason behind our success in producing fast MCTS simulation. We developed a variety of testing tools contained within the test application. Game recording and replaying functionality was useful to validate game behaviour and debug difficult bugs. Furthermore, we were able to easily write testing scripts that ran many different game configuration with differing parameters due to the configurable command flag system of the test application. These tools greatly reduced time spent testing, and allowed AI developers to be less concerned with the implementation details.

Chapter 14

Conclusion

In this project, we aimed to develop a Dynamic Difficulty Adjustment agent based on the Monte-Carlo Tree Search algorithm for Ticket to Ride, a board game with complex strategies and imperfect information. Our proposal has considerable novelty - though the MCTS algorithm has been a popular area of research, DDA using MCTS has been explored to a lesser extent and this is compounded in the imperfect information setting. To achieve our objectives, we built a standalone application that allows users to play a replication of Ticket to Ride - *Coupon for Travel* - against DDA agents, which is cross-platform between Windows, Linux, and Macintosh devices. We implemented two types of MCTS agents, a Cheating Monte-Carlo Tree Search agent which transforms an imperfect information game into a perfect information game by cheating, and an Information-Set Monte-Carlo Tree Search agent which treats nodes as information sets. The team used a scrum-based methodology with 2-week sprint cycles, holding sprint planning meetings at the start and sprint reviews at the end of each sprint.

Despite being a small team, we were able to complete all of our deliverables on time. This included our fully fledged user application and several DDA MCTS agents with modifications like POSAS and momentum, the latter of which we designed ourselves. Furthermore, we developed several rule-based agents and testing tools that can run game simulations. DDA efficacy testing has been successfully conducted. We ran a round-robin testing on several heuristic agents to provide evidence of their varying skill levels. Subsequently, we have tested the DDA against these bots, demonstrating that it can effectively adapt to different strategies including ones which change mid game. We were not able to get client feedback on the final product, but a number of testers were recruited to complete user acceptance testing, for which we have received positive feedback in the

form of both a high enjoyment rating (8.81/10) and a high style rating (9.58/10). Main areas of improvement includes strength of the DDA agents, as both CMCTS and ISMCTS agents typically lost against human players - in the case of ISMCTS, the performance ceiling is low, but in the case of CMCTS, a different DDA approach could be explored for it to play slightly more aggressively. When comparing these factors against our original objectives, we feel that the project outcome achieves its initial goal, and that we have produced a novel and fun piece of software to be proud of.

14.1 Future Work

Finally, there are a number of recommendations for future development on both the agents and the user application.

We propose a few suggestions for future MCTS DDA that builds on our findings:

1. Explore techniques to improve the maximum strength of ISMCTS to see if there is performance changes in the DDA. For example parallel MCTS [10] is one of many extensions that could be looked at.

2. Investigate the performance implications of ISMCTS + Partially Observable Moves and Muliple-Observer ISMCTS.

3. Explore the idea of applying momentum across games so that the agent remembers the opponent's skill level.

4. Explore the use of dynamic score difference targets. For example, an agent could aim for a large winning score difference near the start of the game and taper towards a draw at the end of the game.

Further, the recommendations for future user application work are as follows:

- The game application can be further polished with the implementation of a game settings page and sound effects, as well as a background music track.

- Extra mechanics that are in the original Ticket to Ride game, such as tunnels and ferries, as referred to in Section 2.3.2.

- The accessibility of the game can be improved by adding train car patterns on the map to the corresponding coloured train cards, in order for those affected by

colour-blindness to easily match them up.

- An in-game help menu with terminology and score guide will help new users get familiar with the game quickly.

- More game maps will bring variety to the game, and larger maps in particular could be explored as both players and agents are likely to perform differently.

Bibliography

[1] M. Csikszentmihalyi, *Flow: The psychology of optimal experience* (Harper Perennial Modern Classics), eng, Nachdr. New York: Harper [and] Row, 2009, ISBN: 978-0-06-133920-2.

[2] Y. Hao, S. He, J. Wang, X. Liu, J. Yang, and W. Huang, "Dynamic Difficulty Adjustment of Game AI by MCTS for the game Pac-Man," Aug. 2010, pp. 3918–3922. DOI: 10.1109/ICNC.2010.5584761.

[3] S. Ganzfried and T. Sandholm, "Game Theory-Based Opponent Modeling in Large Imperfect-Information Games," en, pp. 533–540, 2011.

[4] P. Gaubil, "Coveted German Spiel des Jahres (Game of the Year) 2004 awarded to Ticket to Ride, the Train Adventure Game.," en,

[5] F. De Mesentier Silva, S. Lee, J. Togelius, and A. Nealen, "AI-based playtesting of contemporary board games," Aug. 2017, pp. 1–10. DOI: 10.1145/3102071.3102105.

[6] C. Huchler, "AN MCTS AGENT FOR TICKET TO RIDE," en,

[7] A. Brodnik and M. Grgurovič, "Solving all-pairs shortest path by single-source computations: Theory and practice," *Discrete Applied Mathematics*, Algorithmic Graph Theory on the Adriatic Coast, vol. 231, pp. 119–130, Nov. 2017, ISSN: 0166-218X. DOI: 10.1016/j.dam.2017.03.008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0166218X17301415 (visited on 04/30/2024).

[8] I.-L. Wang, E. L. Johnson, and J. S. Sokol, "A Multiple Pairs Shortest Path Algorithm," *Transportation Science*, vol. 39, no. 4, pp. 465–476, 2005, Publisher: INFORMS, ISSN: 0041-1655. [Online]. Available: https://www.jstor.org/stable/25769268 (visited on 04/30/2024).

[9] R. Coulom, "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search," en, in *Computers and Games*, H. J. van den Herik, P. Ciancarini, and H. H. L. M. (Donkers, Eds., Berlin, Heidelberg: Springer, 2007, pp. 72–83, ISBN: 978-3-540-75538-8. DOI: 10.1007/978-3-540-75538-8_7.

[10] C. B. Browne, E. Powley, D. Whitehouse, *et al.*, "A Survey of Monte Carlo Tree Search Methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, Mar. 2012, Conference Name: IEEE Transactions on Computational Intelligence and AI in Games, ISSN: 1943-0698. DOI: 10.1109/TCIAIG.2012.2186810. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6145622 (visited on 04/09/2024).

[11] L. Kocsis, C. Szepesvari, and J. Willemson, "Improved Monte-Carlo Search," en,

[12] P. I. Cowling, E. J. Powley, and D. Whitehouse, "Information Set Monte Carlo Tree Search," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 2, pp. 120–143, Jun. 2012, Conference Name: IEEE Transactions on Computational Intelligence and AI in Games, ISSN: 1943-0698. DOI: 10.1109/TCIAIG.2012.2200894. [Online]. Available: https://ieeexplore.ieee.org/document/6203567 (visited on 11/18/2023).

[13] R. Kelly and D. Churchill, "Comparison of Monte Carlo Tree Search Methods in the Imperfect Information Card Game Cribbage," en,

[14] S. Yang, M. Barlow, T. Townsend, *et al.*, "Reinforcement Learning Agents Playing Ticket to Ride–A Complex Imperfect Information Board Game With Delayed Rewards," en, *IEEE Access*, vol. 11, pp. 60 737–60 757, 2023, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2023.3287100. [Online]. Available: https://ieeexplore.ieee.org/document/10154465/ (visited on 10/25/2023).

[15] M. Bowling, N. Burch, M. Johanson, and O. Tammelin, "Heads-up limit hold'em poker is solved," *Science*, vol. 347, no. 6218, pp. 145–149, Jan. 2015, Publisher: American Association for the Advancement of Science. DOI: 10.1126/science.1259433. [Online]. Available: https://www.science.org/doi/10.1126/science.1259433 (visited on 04/11/2024).

[16] G. Tesauro, "Temporal difference learning and TD-Gammon," *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, Mar. 1995, ISSN: 0001-0782. DOI: 10.1145/203330.203343. [Online]. Available: https://dl.acm.org/doi/10.1145/203330.203343 (visited on 04/11/2024).

[17] L. Strömberg and V. Lind, *Board Game AI Using Reinforcement Learning*, eng. 2022. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:oru:diva-99988 (visited on 10/24/2023).

[18] I. Smith and C. Anton, "Artificial Intelligence Approaches to Build Ticket to Ride Maps," en, *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, no. 11, pp. 12 844–12 851, Jun. 2022, Number: 11, ISSN: 2374-3468. DOI: 10.1609/aaai.v36i11.21564. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/21564 (visited on 10/24/2023).

[19] F. de Mesentier Silva, S. Lee, J. Togelius, and A. Nealen, "Evolving maps and decks for ticket to ride," in *Proceedings of the 13th International Conference on the Foundations of Digital Games*, ser. FDG '18, New York, NY, USA: Association for Computing Machinery, Aug. 2018, pp. 1–7, ISBN: 978-1-4503-6571-0. DOI: 10.1145/3235765.3235813. [Online]. Available: https://doi.org/10.1145/3235765.3235813 (visited on 10/25/2023).

[20] M. P. Silva, V. do Nascimento Silva, and L. Chaimowicz, "Dynamic Difficulty Adjustment through an Adaptive AI," in *2015 14th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*, ISSN: 2159-6662, Nov. 2015, pp. 173–182. DOI: 10.1109/SBGames.2015.16. [Online]. Available: https://ieeexplore.ieee.org/document/7785854 (visited on 10/20/2023).

[21] S. Kennedy, "ChallengeMate: A Self-Adjusting Dynamic Difficulty Chess Computer," en, [Online]. Available: https://web.stanford.edu/class/aa228/reports/2018/final9.pdf (visited on 10/20/2023).

[22] K. Fujita, *AlphaDDA: Strategies for Adjusting the Playing Strength of a Fully Trained AlphaZero System to a Suitable Human Training Partner*, arXiv:2111.06266 [cs], Sep. 2022. DOI: 10.48550/arXiv.2111.06266. [Online]. Available: http://arxiv.org/abs/2111.06266 (visited on 10/22/2023).

[23] S. Demediuk, M. Tamassia, W. L. Raffe, F. Zambetta, X. Li, and F. Mueller, "Monte Carlo tree search based algorithms for dynamic difficulty adjustment," in *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, ISSN: 2325-4289, Aug. 2017, pp. 53–59. DOI: 10.1109/CIG.2017.8080415. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8080415 (visited on 10/20/2023).

[24] *Ticket to Ride Universe*, en-US. [Online]. Available: https://www.daysofwonder.com/ticket-to-ride/ (visited on 04/28/2024).

[25] *Exceptions to copyright*, en, Jan. 2021. [Online]. Available: https://www.gov.uk/guidance/exceptions-to-copyright (visited on 04/29/2024).

[26] E. Gibson, M. D. Griffiths, F. Calado, and A. Harris, "The relationship between videogame micro-transactions and problem gaming and gambling: A systematic review," *Computers in Human Behavior*, vol. 131, p. 107219, Jun. 2022, ISSN: 0747-5632. DOI: 10.1016/j.chb.2022.107219. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0747563222000413 (visited on 04/29/2024).

[27] J. Hari, *Stolen focus*, First edition. New York: Crown, 2021, ISBN: 978-0-593-13852-6.

[28] J. P. Zagal, S. Björk, and C. Lewis, "Dark Patterns in the Design of Games," en, pp. 533–540, 2011.

[29] W. Hoffmann-Riem, "Artificial Intelligence as a Challenge for Law and Regulation," en, in *Regulating Artificial Intelligence*, T. Wischmeyer and T. Rademacher, Eds., Cham: Springer International Publishing, 2020, pp. 1–29, ISBN: 978-3-030-32361-5. DOI: 10.1007/978-3-030-32361-5_1. [Online]. Available: https://doi.org/10.1007/978-3-030-32361-5_1 (visited on 04/29/2024).

[30] Atlassian, *What is Scrum? [+ How to Start]*, en. [Online]. Available: https://www.atlassian.com/agile/scrum (visited on 10/25/2023).

[31] *Unity Real-Time Development Platform — 3D, 2D, VR & AR Engine*, en. [Online]. Available: https://unity.com (visited on 10/25/2023).

[32] G. Engine, *Godot Engine - Free and open source 2D and 3D game engine*, en. [Online]. Available: https://godotengine.org/ (visited on 10/25/2023).

[33] *SFML*. [Online]. Available: https://www.sfml-dev.org/ (visited on 10/25/2023).

[34] *Visual Studio: IDE and Code Editor for Software Developers and Teams*, en-US. [Online]. Available: https://visualstudio.microsoft.com (visited on 04/29/2024).

[35] *CMake - Upgrade Your Software Build System*, en-US. [Online]. Available: https://cmake.org/ (visited on 04/29/2024).

[36] *Procreate® – Sketch, Paint, Create.* en-US. [Online]. Available: https://procreate.com/ (visited on 04/29/2024).

[37] *Valgrind Home*. [Online]. Available: https://valgrind.org/ (visited on 04/29/2024).

[38] *Valgrind*. [Online]. Available: https://valgrind.org/docs/manual/cl-manual.html (visited on 04/29/2024).

[39] *QCacheGrind (KCacheGrind) Windows build*, en, Jun. 2014. [Online]. Available: https://sourceforge.net/projects/qcachegrindwin/ (visited on 04/29/2024).

[40] S. Gelly and D. Silver, "Monte-Carlo tree search and rapid action value estimation in computer Go," *Artificial Intelligence*, vol. 175, no. 11, pp. 1856–1875, Jul. 2011, ISSN: 0004-3702. DOI: 10.1016/j.artint.2011.03.007. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S000437021100052X (visited on 04/21/2024).

[41] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995, ISBN: 978-0-201-63361-0.

[42] T. Lugrin, "Random Number Generator," en, in *Trends in Data Protection and Encryption Technologies*, V. Mulder, A. Mermoud, V. Lenders, and B. Tellenbach, Eds., Cham: Springer Nature Switzerland, 2023, pp. 31–34, ISBN: 978-3-031-33386-6. DOI: 10.1007/978-3-031-33386-6_7. [Online]. Available: https://doi.org/10.1007/978-3-031-33386-6_7 (visited on 04/30/2024).

[43] F. Panneton and P. L'Ecuyer, "On the xorshift random number generators," en, *ACM Transactions on Modeling and Computer Simulation*, vol. 15, no. 4, pp. 346–

361, Oct. 2005, ISSN: 1049-3301, 1558-1195. DOI: 10.1145/1113316.1113319. [Online]. Available: https://dl.acm.org/doi/10.1145/1113316.1113319 (visited on 04/29/2024).

[44] *C++ Core Guidelines: Rules for Smart Pointers – MC++ BLOG*, en-US, Dec. 2017. [Online]. Available: https://www.modernescpp.com/index.php/c-core-guidelines-rules-to-smart-pointers/ (visited on 04/29/2024).

[45] *Memory and Performance Overhead of Smart Pointers – MC++ BLOG*, en-US, Dec. 2016. [Online]. Available: https://www.modernescpp.com/index.php/memory-and-performance-overhead-of-smart-pointer/ (visited on 04/29/2024).

[46] M. Matsumoto and T. Nishimura, "Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator," en, *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 1, pp. 3–30, Jan. 1998, ISSN: 1049-3301, 1558-1195. DOI: 10.1145/272991.272995. [Online]. Available: https://dl.acm.org/doi/10.1145/272991.272995 (visited on 04/29/2024).

[47] *On C++ Random Number Generator Quality*, en, Jun. 2018. [Online]. Available: https://arvid.io/2018/06/30/on-cxx-random-number-generator-quality/ (visited on 04/29/2024).

[48] *Using Faster Psudo-Random Generator: XorShift*, en-US.UTF-8. [Online]. Available: https://codingforspeed.com/using-faster-psudo-random-generator-xorshift/ (visited on 04/29/2024).

[49] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, Jul. 2019, Conference Name: IEEE Std 754-2019 (Revision of IEEE 754-2008). DOI: 10.1109/IEEESTD.2019.8766229. [Online]. Available: https://ieeexplore.ieee.org/document/8766229 (visited on 04/29/2024).

[50] *Methods of computing square roots - Wikipedia*, Apr. 2024. [Online]. Available: https://archive.ph/OKC1L (visited on 04/29/2024).

[51] *Archive of Interesting Code*. [Online]. Available: https://www.keithschwarz.com/interesting/code/?dir=zeckendorf-logarithm (visited on 04/29/2024).

[52] S. Milton and H. W. Schmidt, "Dynamic Dispatch in Object-Oriented Languages," en,

[53] *Clang C Language Family Frontend for LLVM*. [Online]. Available: https://clang.llvm.org/ (visited on 04/29/2024).

[54] *GCC, the GNU Compiler Collection - GNU Project*. [Online]. Available: https://gcc.gnu.org/ (visited on 04/29/2024).

[55] *Visual Studio C/C++ IDE and Compiler for Windows*, en-US. [Online]. Available: https://visualstudio.microsoft.com/vs/features/cplusplus/ (visited on 04/29/2024).

Appendix A

Enumerated Types

In this chapter we detail the values of enumerated types referenced throughout this section. These are used for game configuration, colours codes and DDA variants in implementation.

TRAIN_RESOURCE

```
BLACK = 0
WHITE = 1
RED = 2
ORANGE = 3
YELLOW = 4
GREEN = 5
BLUE = 6
PURPLE = 7
LOCOMOTIVE = 8
NONE = -1
```

AGENT_CODE

```
CONSOLEPLAYERAGENT = 1
GUIPLAYERAGENT = 2
REPLAYAGENT = 32

// Heuristic AI Agents
```

```
SWAP_BOT = 64
SWAP_BOT2 = 70
SWAP_BOT3 = 96

TAKERANDOMTURN_AIAGENT = 128
TAKELASTTURN_AIAGENT = 256
TAKELOCOMOTIVE_AIAGENT = 512

HEURISTIC_AGENT = 1024
RANDOM_SMART = 2048
RANDOM_BASE = 3072
VILLAIN_AGENT = 4096
HOARDER = 5120
HOARDER_TICKET = 6144
PILE = 7168
PILE_TICKET = 8192

// Cheating MCTS Agents
CHEATING_MCTS_AIAGENT = 16384
CHEATING_MCTS_DDAAGENT = 32768
CHEATING_MCTS_DDAAGENT_TRUE_ROSAS = 32769
CHEATING_MCTS_DDAAGENT_POSAS = 32770
CHEATING_MCTS_DDAAGENT_TRUE_POSAS = 32771

// Information Set MCTS Agents
INFORMATION_SET_MCTS_AIAGENT = 131072
INFORMATION_SET_MCTS_DDAAGENT = 262144
INFORMATION_SET_MCTS_DDAAGENT_TRUE_ROSAS = 262145
INFORMATION_SET_MCTS_DDAAGENT_POSAS = 262146
INFORMATION_SET_MCTS_DDAAGENT_TRUE_POSAS = 262147
```

PLAYER_EVENT_CODE

```
BADPLAYEREVENT = -1

DRAWTICKETPLAYEREVENT = 0
DRAWANDDISCARDTICKETPLAYEREVENT = 1
```

DRAWLOCOMOTIVEPLAYEREVENT = 3

DRAWFROMPILEPLAYEREVENT = 4

DRAWFROMTABLEPLAYEREVENT = 5

DRAWONEFROMEACHPLAYEREVENT = 6

CLAIMROUTEBYNAMEPLAYEREVENT = 7

CLAIMROUTEBYIDPLAYEREVENT = 8

PLAYEREVENT = 1000

TICKETPLAYEREVENT = 1001

ISTICKETCOMPLETED = 1002

CLAIMROUTEPLAYEREVENT = 1003

DRAWTRAINCARDSPLAYEREVENT = 1004

DDA_VARIANT

DISABLED = 0

ROSAS = 1

POSAS = 2

TRUE_ROSAS = 3

TRUE_POSAS = 4

Appendix B

Performance Graphs

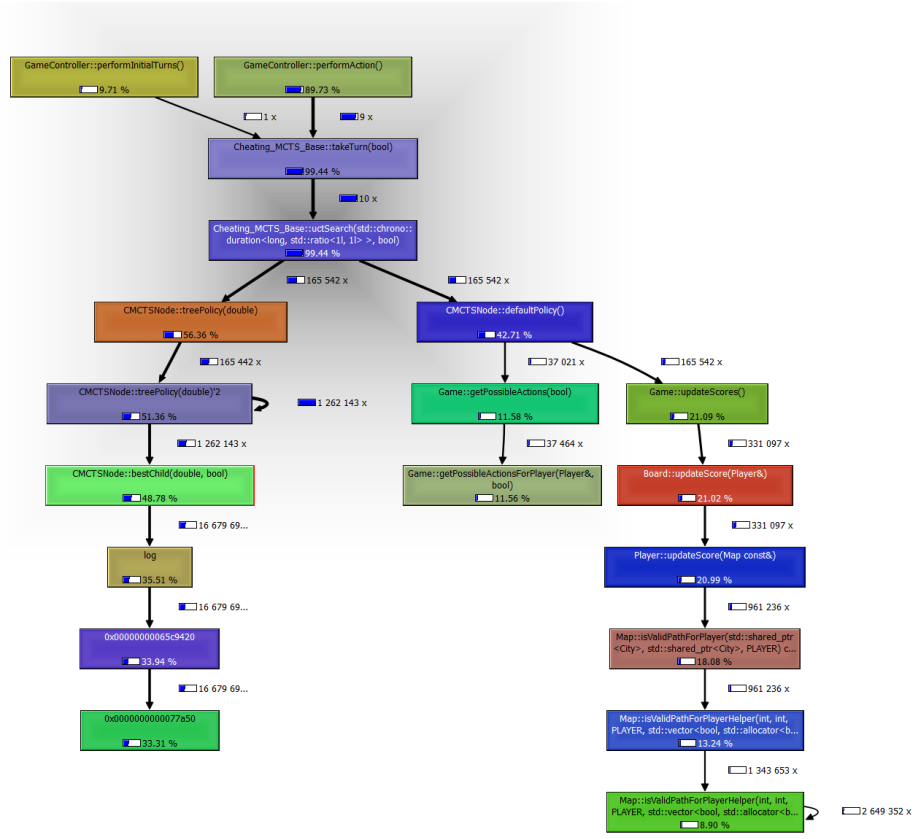


Figure B.1: Earliest callgrind code profile of CMCTS, viewed using QCacheGrind. Note the mass of computation falls under log. This was largely due to errors in MCTS base algorithm implementation leading to less default policy than expected, but was also due to the nature of the log operation itself.

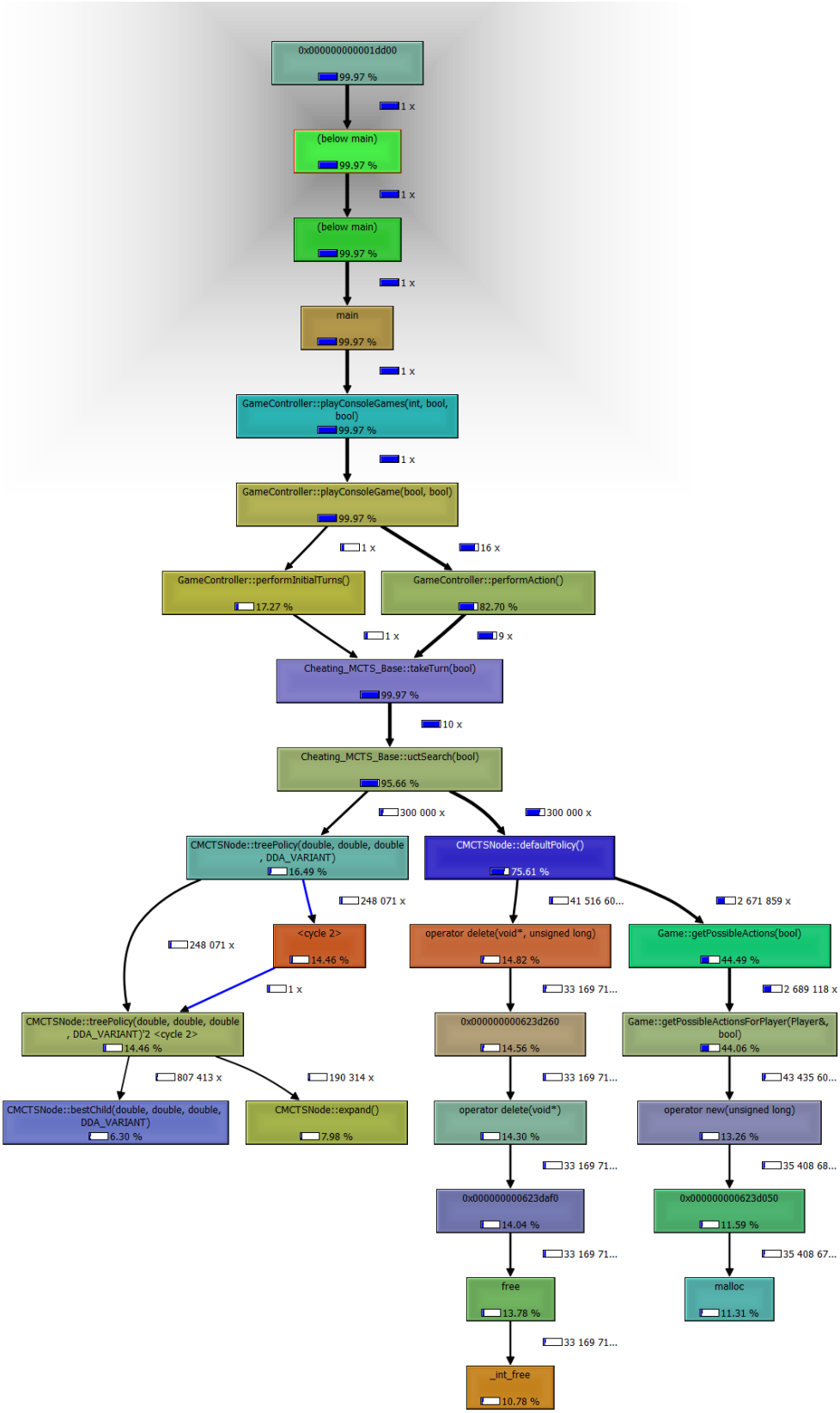


Figure B.2: Final callgrind code profile of CMCTS, viewed using QCacheGrind.

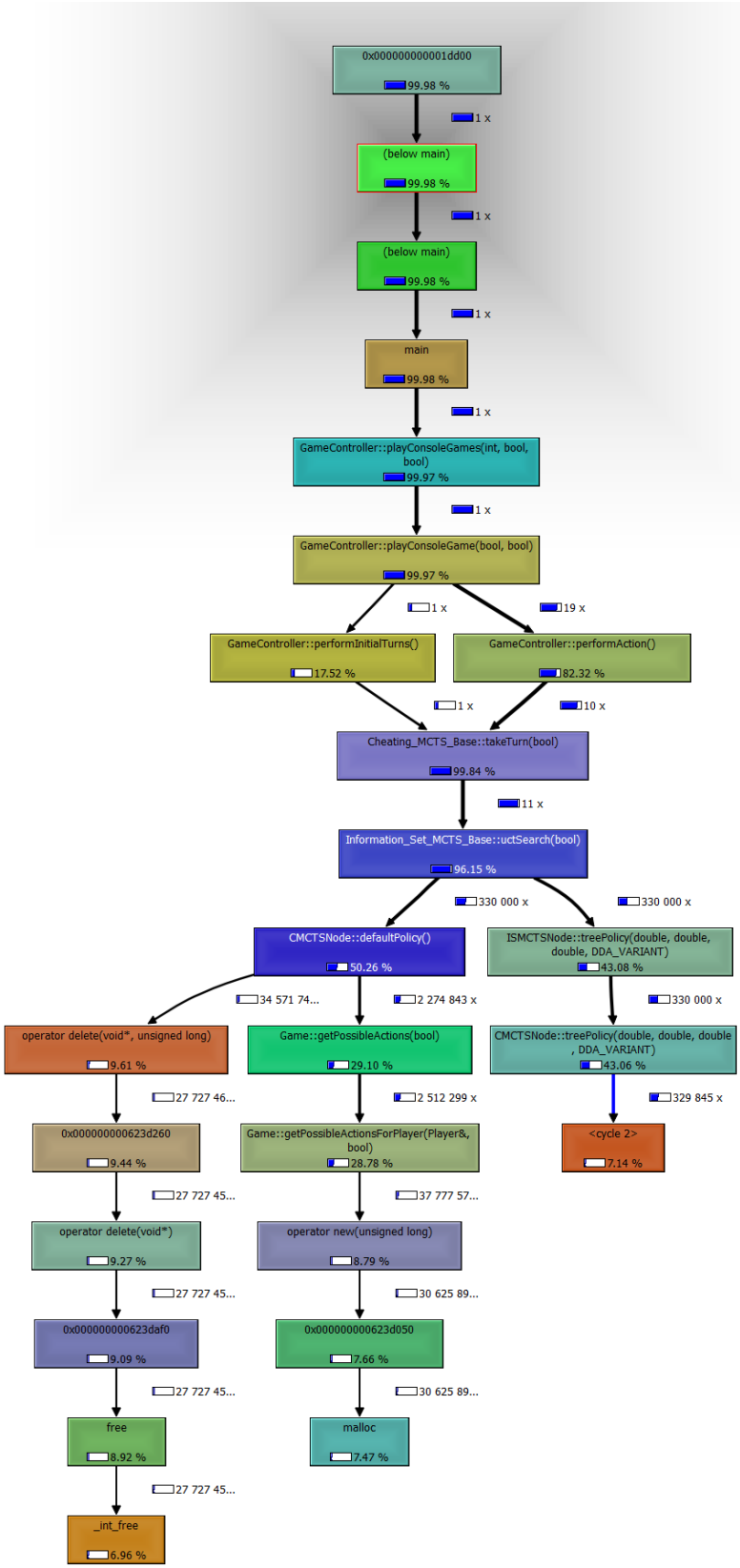


Figure B.3: Final callgrind code profile of ISMCTS, viewed using QCacheGrind.

Appendix C

DDA Efficacy Test Results

CMCTS ROSAS	win %	draw %	Win % Difference	Average Score Difference
TakeRandom Turn (128)	31.9	27.8	8.4	1.324
TakeLastTurn (256)	16.3	28.3	39.1	2.727
Villain (4096)	14.9	26.8	43.4	1.945
RandomBase (3072)	17.1	18.7	47.1	3.451
HoarderTicket (6144)	12.3	17.7	57.7	5.569
Pile (7168)	17.4	13.8	51.4	3.985
PileTicket (8192)	17.4	13.8	51.4	3.985
RandomSmart (2048)	15.7	15.4	53.2	4.359
Heuristic Agent (1024)	9.1	12.9	68.9	5.276
Hoarder (5120)	9.1	12.5	69.3	5.027
Average	**16.12**	**18.77**	**48.99**	**3.7648**

CMCTS POSAS (lh=5)	win %	draw %	Win % Difference	Average Score Difference
TakeRandom Turn (128)	61.8	8.5	32.1	1.609
TakeLastTurn (256)	39	8.8	13.2	1.489
Villain (4096)	38.8	8	14.4	0.436
RandomBase (3072)	35.9	6.1	22.1	1.215
HoarderTicket (6144)	28.4	3.9	39.3	3.024
Pile (7168)	26.9	6.2	40	2.12
PileTicket (8192)	26.9	6.2	40	2.12
RandomSmart (2048)	27.2	6.4	39.2	2.128
Heuristic Agent (1024)	21.8	5.9	50.5	2.923
Hoarder (5120)	19.1	6.5	55.3	2.865
Average	**32.58**	**6.65**	**34.61**	**1.9929**

CMCTS ROSAS (m=0.5)	win %	draw %	Win % Difference	Average Score Difference
TakeRandom Turn (128)	27.8	28.3	16.1	1.308
TakeLastTurn (256)	12.3	27.7	47.7	3.045
Villain (4096)	11.8	28.1	48.3	1.308
RandomBase (3072)	14.5	18.6	52.4	3.869
HoarderTicket (6144)	10	15.1	64.9	6.019
Pile (7168)	12.1	15.8	60	4.511
PileTicket (8192)	12.1	15.8	60	4.511
RandomSmart (2048)	11.3	15.8	61.6	4.678
Heuristic Agent (1024)	7.6	14.9	69.9	5.354
Hoarder (5120)	6.5	14.7	72.3	5.318
Average	**12.6**	**19.48**	**55.32**	**3.9921**

ISMCTS ROSAS	win %	draw %	Win % Difference	Average Score Difference
TakeRandom Turn (128)	79.1	7.1	65.3	4.168
TakeLastTurn (256)	43	11.8	2.2	1.227
Villain (4096)	38.7	11.1	11.5	0.983
RandomBase (3072)	44.1	7.8	4	0.945
HoarderTicket (6144)	30.7	13.4	25.2	2.485

Pile (7168)	23.2	6.6	47	4.249
PileTicket (8192)	23.2	6.6	47	4.249
RandomSmart (2048)	31.4	7	30.2	2.603
Heuristic Agent (1024)	17.8	5.3	59.1	5.491
Hoarder (5120)	16.7	6.2	60.4	5.435
Average	**34.79**	**8.29**	**35.19**	**3.1835**

ISMCTS POSAS (lh=5)	**win %**	**draw %**	**Win % Difference**	**Average Score Difference**
TakeRandom Turn (128)	73.4	4.2	51	3.536
TakeLastTurn (256)	42.3	5.6	9.8	1.577
Villain (4096)	39.6	6.7	14.1	1.133
RandomBase (3072)	34.9	4.4	25.8	2.344
HoarderTicket (6144)	29.2	6.2	35.4	3.391
Pile (7168)	22.6	3.9	50.9	4.441
PileTicket (8192)	22.6	3.9	50.9	4.441
RandomSmart (2048)	25.7	3.2	45.4	4.084
Heuristic Agent (1024)	16.2	3.2	64.4	6.015
Hoarder (5120)	14.3	4	67.4	6.409
Average	**32.08**	**4.53**	**41.51**	**3.7371**

ISMCTS ROSAS (m=0.5)	**win %**	**draw %**	**Win % Difference**	**Average Score Difference**
TakeRandom Turn (128)	82.3	6	70.6	4.734
TakeLastTurn (256)	48	11	7	0.833
Villain (4096)	43.9	13.8	1.6	0.34
RandomBase (3072)	45.2	9.9	0.3	0.679
HoarderTicket (6144)	36.2	13.3	14.3	2.302
Pile (7168)	26.5	6.3	40.7	3.988
PileTicket (8192)	26.5	6.3	40.7	3.988
RandomSmart (2048)	35.2	7.8	21.8	2.02
Heuristic Agent (1024)	18.9	7.1	55.1	5.094
Hoarder (5120)	18.5	7.3	55.7	4.88
Average	**38.12**	**8.88**	**30.78**	**2.8858**

ISMCTS POSAS (lh=5) (m=0.1)	**win %**	**draw %**	**Win % Difference**	**Average Score Difference**
TakeRandom Turn (128)	91.3	2	84.6	6.75
TakeLastTurn (256)	68	3.9	39.9	0.951
Villain (4096)	69.2	4.2	42.6	1.977
RandomBase (3072)	65.5	4.9	35.9	2.234
HoarderTicket (6144)	59.7	5.1	24.5	0.173
Pile (7168)	42.8	5.6	8.8	0.945
PileTicket (8192)	42.8	5.6	8.8	0.945
RandomSmart (2048)	51.1	3.6	5.8	0.345
Heuristic Agent (1024)	39.3	4.7	16.7	2.169
Hoarder (5120)	35	4.8	25.2	2.536
Average	**56.47**	**4.44**	**29.28**	**1.9025**

CMCTS ROSAS (target=3)	win %	draw %	Win % Difference	Average Score Difference
TakeRandom Turn (128)	75.9	5.8	57.6	1.052
TakeLastTurn (256)	59.1	9.8	28	1.368
Villain (4096)	63	10.3	36.3	0.37
RandomBase (3072)	51.8	6.4	10	1.132
HoarderTicket (6144)	44	4.7	7.3	3.497
Pile (7168)	40.8	7	11.4	2.042
PileTicket (8192)	40.8	7	11.4	2.042
RandomSmart (2048)	38.9	6.1	16.1	2.433
Heuristic Agent (1024)	35.2	5.8	23.8	2.82
Hoarder (5120)	31.1	7	30.8	3.052
Average	**48.06**	**6.99**	**23.27**	**1.9808**

Appendix D

Game Instructions

Attached are game installation instructions and game play instructions that were provided to users.

TicketToRide

Hello, and thank you for showing interest in our TicketToRide game.

If you are a user tester and want to run the game on a windows device, without installing code from source, we recommend to go to section I.

If you are a member of the University of Warwick Department of Computer Science (DCS), we recommend you play the game on the DCS systems due to guaranteed compaibility. For these steps we recommend you go to section II.

For people who would like to build and install the game from code who do not satisfy the previous conditions, we recommend you go to section III and follow the steps for your specific device.

(I) Installing and Running from Windows from Binaries

STEPS:

1. Unzip the folder `TicketToRide_win64.zip`
2. Open the extracted foldler.
3. In the route node there should be an executable `TicketToRide.exe`
4. Click this executable and the game should open

(II) Installing and Running from Warwick DCS Systems for Computer Science Students

IN PERSON INSTRUCTIONS

STEPS:

1. Unzip the folder `TicketToRide_src.zip`, which contains the source files for the program
2. Navigate in the terminal to the extracted folder `... /TicketToRide/`
3. **Generate build files**. Run the command `cmake -S . -B build -D CMAKE_BUILD_TYPE=Release -D CMAKE_CXX_FLAGS="O3"`. This should set up the folder for building.
4. **Building Binaries**. Navigate to the newly created folder, `build` (this should be `/TicketToRide/build/`), and run the command `make`. This should build the program.
5. Once the make process has finished, you can run the executable by typing (still in the directory `/TicketToRide/build/` in the terminal) `./bin/TicketToRide/`. The game should now open and you should be ready to play.

REMOTE INSTRUCTIONS

To access the TicketToRide GUI remotely, you need to create a Virtual Network Computing (VNC) connection. To do this you will need to follow the instructions outlined in the DCS guide here: https://warwick.ac.uk/fac/sci/dcs/intranet/user_guide/remote-login/vnc-windows/

1. Open a VNC window and log in to DCS system remotely
2. Now follow the steps as outlined in the "in person instructions"

(III) Further Installation Instructions

If you are not using the previous two forms of installation, you can build from source, but you may have to install additional dependencies.

Linux

NOTE: only verified for Ubuntu based Linux systems.

1. *DEPENDENCIES*

- Make sure CMake is installed (`sudo apt-get -y install cmake`)
- Make sure a C++17 or later compiler is installed. We recommend GCC for guaranteed results. (`sudo apt-get install build-essential`)
- You may find that CMake fails to generate build files. You will need to install additional dependencies in order for CMake/make to successfully generate and build (these libraries are needed for SFML). Here follows the instructions for (Ubuntu/Debian) for which there are 3 ways

    1. **Install SFML library (which will install dependencies by default):**

        - run `sudo apt-get install libsfml-dev`

    2. **Install all dependencies in one go:**

        - run `sudo apt-get install libx11-dev libgl1-mesa-dev libudev-dev libalut-dev libvorbis-dev libsndfile1-dev libxrandr-dev libxcursor-dev libfreetype-dev libfreetype6 libfreetype6-dev`

    3. **Manually install specific missing dependencies, responding to specific CMake error messages.**

        - Could NOT find X11 => `sudo apt-get install libx11-dev`
        - Could NOT find OpenGL => `sudo apt-get install libgl1-mesa-dev`
        - Could not find UDev library => `sudo apt-get install libudev-dev`
        - Could NOT find OpenAL => `sudo apt-get install libalut-dev`
        - Could NOT find VORBIS => `sudo apt-get install libvorbis-dev`
        - Could NOT find FLAC => `sudo apt-get install libsndfile1-dev`
        - During Makefile these libraries needed installing for me too
            - `sudo apt-get install libxrandr-dev`
            - `sudo apt-get install libxcursor-dev`
            - `sudo apt-get install libfreetype-dev libfreetype6 libfreetype6-dev`

2. *INSTALLATION*

1. Download source files. Either run `git clone https://github.com/CS407-DDA-Group-Project/TicketToRide/` (make sure git is installed (`sudo apt-get install git-all`)), or download source files as a zip and extract.
2. **Generate build files**. Navigate to the `/TicketToRide/` folder. Run CMake as follows: `cmake -S . -B build -D CMAKE_BUILD_TYPE=Release`.

    - NOTE: if you want to include testing application in your build configuration, add the `-D TEST_BUILD=On` flag, leading to full install instruction `cmake -S . -B build -D CMAKE_BUILD_TYPE=Release -D TEST_BUILD=On`.
3. **Building Binaries**. Navigate to `/TicketToRide/build/` folder. Run `make`.

3. *RUNNING THE APP*

1. Navigate to `/TicketToRide/build/` folder.
2. Execute the command `./bin/TicketToRide` to run the GUI version, and (if installed) execute `./bin/TicketToRideTest -h` to get you started with the test console app.

Mac

NOTE: only verified for arm64 chip devices.

1. *DEPENDENCIES*

- Make sure CMake is installed
- Make sure a C++17 or later compiler is installed. We recommend GCC for guaranteed results.

2. *INSTALLATION*

1. Download source files. Either run `git clone https://github.com/CS407-DDA-Group-Project/TicketToRide/` (make sure git is installed), or download source files as a zip and extract.
2. **Generate build files**. Navigate to the `/TicketToRide/` folder. Run CMake as follows: `cmake -S . -B build -D CMAKE_BUILD_TYPE=Release`.
    - NOTE: if you want to include testing application in your build configuration, add the `-D TEST_BUILD=On` flag, leading to full install instruction `cmake -S . -B build -D CMAKE_BUILD_TYPE=Release -D TEST_BUILD=On`.
3. **Building Binaries**. Navigate to `/TicketToRide/build/` folder. Run `make`.

3. *RUNNING THE APP*

1. Navigate to `/TicketToRide/build/` folder.
2. Execute the command `./bin/TicketToRide` to run the GUI version, and (if installed) execute `./bin/TicketToRideTest -h` to get you started with the test console app.

Windows

NOTE: only verified for Windows 10 systems.

1. *DEPENDENCIES*

- Make sure CMake is installed (https://cmake.org/download/ for your operating system)
- Make sure a C++17 or later compiler is installed. We have used Visual Studio and Visual Studio Compiler. If you have other C++ compiler set ups the following steps will not apply, and you will have to install as you would install any other CMake program in your setup.
- Download Visual Studio Community Edition 2022:
    - Link: https://visualstudio.microsoft.com/vs/community/

- Install the C++ development extension (don't modify extras)
- Be warned it is a large IDE (~10gb), since it downloads Visual C++, has real time debugging and other developer functions

2. *INSTALLATION*

1. Download source files. Either run `git clone https://github.com/CS407-DDA-Group-Project/TicketToRide/` (make sure git is installed), or download source files as a zip and extract.
2. **Generate build files**. Navigate to the `/TicketToRide/` folder. Run CMake as follows: `cmake -S . -B build -D CMAKE_BUILD_TYPE=Release`.
    - NOTE: if you want to include testing application in your build configuration, add the `-D TEST_BUILD=On` flag, leading to full install instruction `cmake -S . -B build -D CMAKE_BUILD_TYPE=Release -D TEST_BUILD=On`.
3. **Building Binaries Using Visual Studio**.
    - Right click on the desired executable inside of the *Solution Explorer*(this will be TicketToRide unless you want to use the test infrastructure), and select "*Set as Startup Project*".
    - On the top bar, change the build mode from Debug to Release.
    - Press the green triangle *Local Windows Debubgger* or press F5 to build and run the app, or Ctrl+Shift+B to just build the binaries.

3. *RUNNING THE APP*

1. In Visual Studio, press F5.
2. Alternatively, if built in visual studio navigate to the install directory and navigate the `build` directory (`/TicketToRide/build`) and run the command `/bin/Release/TicketToRide.exe`. This should open up the game ready to play. To close the game just exit the window or press Ctrl+C in the terminal.
3. If you have issues loading resources, copy the resources folder directly into the same directory which contains the TicketToRide.exe file, and press the executable from within this containing directory.

Game Instructions

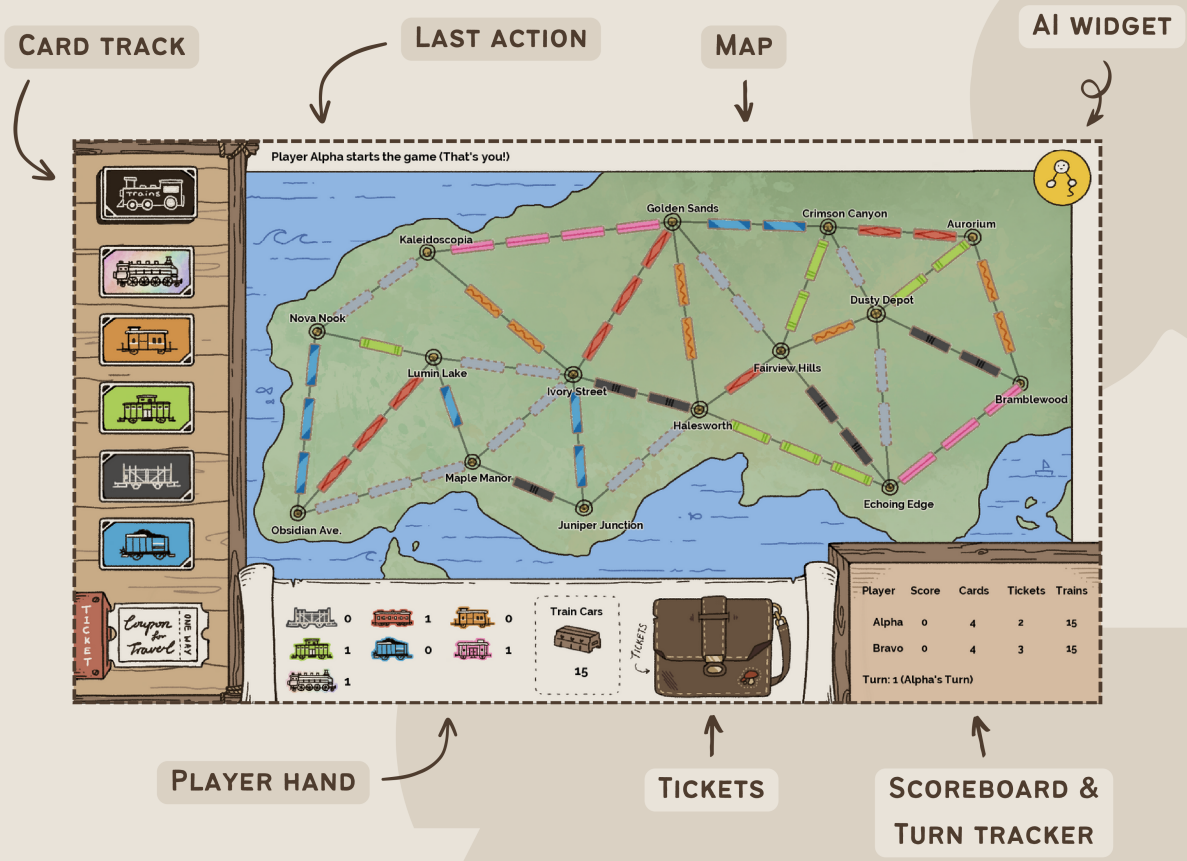
This is a two-player adaptation of the game Ticket to Ride, named Coupon for Travel. Upon launching the game, you will be greeted with a start menu. Here, you can choose between two maps.

In Coupon for Travel, your goal is to win the game by scoring more points than your opponent. To do this, you must claim routes on the map using train cards of the corresponding colour.

Navigating the Game

On the left, you will find the card track where both you and your opponent can draw train cards and tickets from. The banner at the top displays the last player's action.

The map displays all the routes and the cities they connect to, and the scoreboard gives a summary of how each player is doing. Finally, at the bottom you will find your hand. This display shows how many trains you have and what colours they are, as well as the number of the carriages you have left. You can find your tickets in the leather bag!



Playing a Turn

On your opponent's turn, simply click to continue. On your turn, you must choose one of three actions:

1. Draw train cards: Click on train cards to draw them. Two train cards can be drawn, either both from the deck (top), both from the face-up cards on the table, or one from each. But note that if you choose to draw a locomotive (rainbow card) from the table, you cannot draw anything else.

2. Draw Tickets: You can draw three tickets by clicking on the ticket machine at the bottom of the track. Click on the tickets you would like to discard, and the rest will be added to your wallet.

3. Claim route: Once you have enough train cards, you can claim a route of the same colour by first clicking on the route, then clicking on a combination of trains in your hand. Blank routes can be claimed by any one set of colour, and locomotives can be used in place of any coloured train. E.g. To claim the route from Celestia to Goldenreach, Alpha would need to click on the route, click the yellow train in their hand once, and click the locomotive once.

Tickets

Tickets are special cards that contain two cities and a number. If at the end of the game you have a route connecting these two cities, you gain the number of points. Otherwise, you lose that number of points. You can see your tickets by hovering over the wallet, and scroll between them by clicking the next button.

Game End & Scoring

The game ends when one of the players have less than 3 trains left. Then, the other player gets to take one more turn, and the game is scored.

Each player gains their ticket scores, as well as the following for each route they own:

1 point for each route of length 1;
2 point for each route of length 2;
4 point for each route of length 3;
7 point for each route of length 4;
10 point for each route of length 5;
15 point for each route of length 6.

AI Widget

If you are interested, click on the yellow icon in the top right corner to see the Monte-Carlo tree search statistics of your AI opponent!

Appendix E

Sample Sprint Review

During the project we produced several sprint review presentations, which we used to accompany our presentation to key project stakeholders. In this appendix, we provide a sample of these presentation slides, which were created using Google Slides. The following pages contain the slides for the eight sprint review.

CS407 Group Project:
Dynamic Difficulty Adjustment AI for Board Games

Sprint Review: Sprint 8

28/02/24 - 12/03/24

Admin:

Internal Sprint Review and Coursework

Overall State of Software

- What is Done
  - Game logic (apart from longest path)
  - Majority of SFML display
  - CMCTS agents; heuristic agent
  - Recording/Replay of game state
  - …
- What is not Done
  - AI agents! Information set; further iterations of MCTS; further heuristic agent variations
  - GUI interactions
  - Testing framework; able to be run on batch compute
  - MCTS widget
  - …

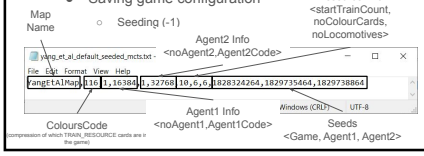
Random Seeding

- Old way:
  - (Now called `Random::static_shuffleIterator(...)`)
  - Creates new random device each time => not reproducible or traceable
- New way
  - `Random::shuffleIterator(...)`
  - Instanced; fixed seed
  - This seed can be noted down (`Random::getSeed()`) for game repeatability purposes, and to reseed (`Random::reset()`)

```
class Random {
private:
    int m_seed;
    std::random_device m_randomDevice;
    std::default_random_engine m_randomGenerator;
public:
    Random();
    Random(int seed);
    Random(const Random&);
    Random& operator=(const Random&);

    // Reseed the random device with its starting seed
    void reset();
    // Returns the seed that the random device was initially seeded with
    int getSeed() const;


    // Shuffle with random seed
    // Static shuffling method, that shuffles an iterator (eg. std::vector)
    using a new
    // random device seeded with provided seed
    template<class T_Iterator>
    static void static_shuffleIterator(T_Iterator& iterator, unsigned int
    seed);

    // Static shuffling method, that shuffles an iterator (eg. std::vector)
    using a new
    // random device seeded with time-based seed
    template<class T_Iterator>
    static void static_shuffleIterator(T_Iterator& iterator);

    // Member function, that shuffles an iterator (eg. std::vector), using
    the member
    // random device (i.e. random device persists after use and is not
    reseeded)
    template<class T_Iterator>
    void shuffleIterator(T_Iterator& iterator);
};
```

GameController

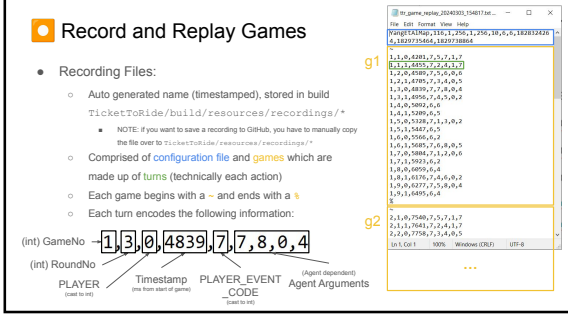
- Setting Up Games
  - Deprecated constructors
  - `GameConfig` (C-style struct; not class)
  - From Configuration File
- Saving game configuration
  - Seeding (-1)




```
struct GameConfig {
    std::string mapId;
    int coloursCode;
    // seed of -1 => time seeded
    int shuffleSeed;

    int noAgentType1;
    AGENT_CODE agentType1;
    // seed of -1 => time seeded
    int agentSeed1;

    int noAgentType2;
    AGENT_CODE agentType2;
    // seed of -1 => time seeded
    int agentSeed2;

    int noStartingTrainCars;
    int noCardsPerColour;
    int noWildcards;
};

enum class AGENT_CODE {
    // [ types encoded as powers of 2 ]

    // Human Agents
    CONSOLEPLAYERAGENT = 1,        // 2^0
    GUIPLAYERAGENT = 2,            // 2^1

    //// Replay Agent
    //REPLAYAGENT = 32,            // 2^5

    // Heuristic AI Agents
    TAKERANDOMTURN_AIAGENT = 128,  // 2^7
    TAKELASTTURN_AIAGENT = 256,    // 2^8

    // MCTS Agents
    CHEATING_MCTS_AIAGENT = 16384,  // 2^15
    CHEATING_MCTS_DOAAGENT = 32768  // 2^16
};
```

Record and Replay Games

- Recording Files:
  - Auto generated name (timestamped), stored in build `TicketToRide/build/resources/recordings/*`
    - NOTE: if you want to save a recording to GitHub, you have to manually copy the file over to `TicketToRide/resources/recordings/*`
  - Comprised of configuration file and games which are made up of turns (technically each action)
  - Each game begins with a ~ and ends with a %
  - Each turn encodes the following information:



Record and Replay Game

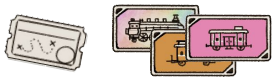
- Wrappers that derive from GameController and act as GameController, but additionally record or replay the game
  - `RecordGameController(std::string configFile);`
    - Use a `RecordGameController` instead of a GameController when you want a replay to be recorded. Otherwise treat the same as GameController.
  - `ReplayGameController(std::string replayName, bool& isReplayOver);`
    - Use a `ReplayGameController` instead of a GameController when you want to run a replay. NOTE: the replay will eventually terminate; this is signified by the `isReplayOver` flag.
- `ReplayAgent` handles the replay for each player, and is passed a pointer to the replay file.

Game UI, media, and SFML

- Progress summary:
  - Game is almost fully rendered, bar ticket display
  - All (core) sprites are done
- What's next?
  - Game interactions
  - Widget for MCTS display
- Then the game prototype will be finished!
  - A little behind schedule

Game UI

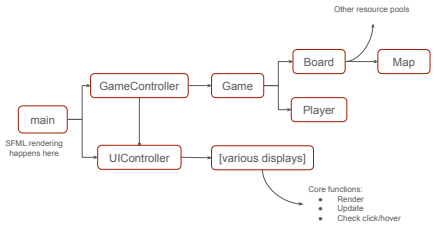
- Spent most of my time on it this term
  - Setting up console game to be SFML renderable, making sprites, coding UI etc.
- Ticket display is tricky as it requires an expandable UI
- UI render needs polishing but otherwise done
- Need to program in actions, which will take time but shouldn't be too technically challenging…



Media

- New title screen :)
- Sounds, music, maybe?

Code Structure



Interacting with the game

How the render is updated

1. UIController is passed to GameController
2. On each turn, GameController queries game components for current state
3. GameController calls UIController.update…() with info (int, string, map, etc.)

How we check for hover

1. Main calls UIController.checkHover() at (60 times/s) with mouse position
2. Each UI component checks if the mouse is within its sprite's bounds
   a. If yes, do something

Interacting with the game

- Why was that relevant?

- … I'm going to take some time away from project to work on courseworks
  - Game actions may have to be implemented by someone else if we want it done now
- How might actions work?
  - Well, should be similar to a combination of hover and update!
  - For each required component, implement checkClick() based on hover
    - Can refer to start button on the menu
  - Then, use the return values from checkClick() to determine player action, and send to game
  - Complications - might need SFMLPlayerAgent…

Heuristic Agents

- Progress:
  - Base Heuristic Agent completed
  - Cost functions and helper functions implemented
- Next:
  - Create more heuristic agents by making adjustments to the base agent and/or adding new heuristic rules.

Actions Rework for Ticket Drawing

- Reworked actions for ticket drawing
- Agents now decide to pick up tickets and then choose them in separate actions
- Agents can now select their tickets at the start of the game
- Work can be done now on developing more AI agents

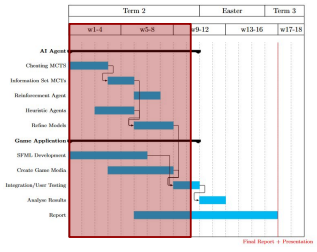
Next Steps in AI Development

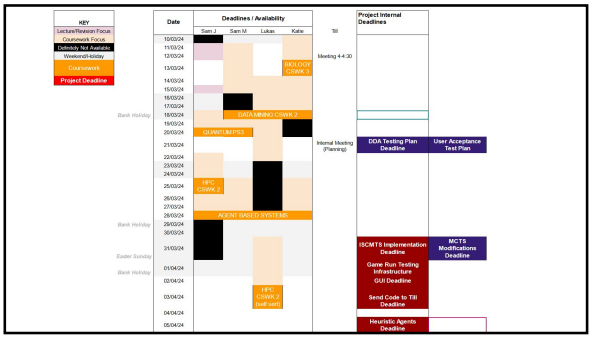
- Implement Information Set Monte Carlo Tree Search
- Research and decide which MCTS extensions/modifications to implement
  a. E.g. parallel MCTS

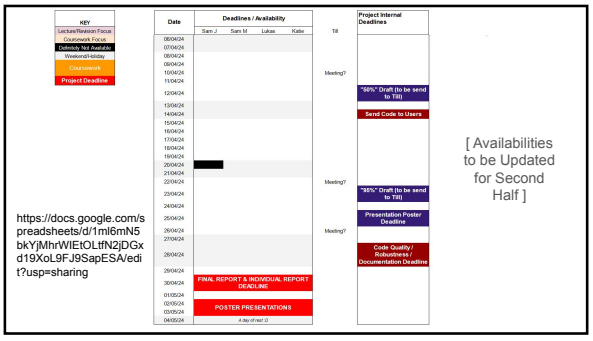
Progress & What's Next

Our Gantt Chart has outlived its utility as a planning device, as we are quite close to the deadline now.

We have recently conducted detailed planning for the next few weeks.







https://docs.google.com/spreadsheets/d/1ml6mN5bkYjMhrWIEtOLtfN2jDGxd19XoL9FJ9SapESA/edit?usp=sharing

[Availabilities to be Updated for Second Half]

Questions

1. Please can we arrange a meeting during the Easter break, and an additional meeting for when we get back (Between April 21-30)
   a. Do you want/need to see progress updates over Easter?
2. Report reviewing: what do you want to see?
   a. One or two intermediate report drafts (for example background, implementation stuff first, and second result and evaluative stuff?)
3. What are your thoughts on our Easter schedule?

Appendix F

Planning Spreadsheet

KEY		Date	Deadlines / Availability					Project Internal Deadlines			Progress
Lecture/Revision Focus			Sam J	Sam M	Lukas	Katie	Till				
Coursework Focus		10/03/24									
Definitely Not Available		11/03/24									
Weekend/Holiday		12/03/24					Meeting 4-4:30				
Coursework		13/03/24									
Project Deadline		14/03/24									
		15/03/24									
		16/03/24									
		17/03/24									
	Bank Holiday	18/03/24		DATA MINING CSWK 2							
		19/03/24									
		20/03/24				BIOLOGY CSWK 3 (self cert)					
		21/03/24					Internal Meeting (Planning)	**DDA Testing Plan Deadline**	**User Acceptance Test Plan**		
		22/03/24									
		23/03/24									
		24/03/24									
		25/03/24									
		26/03/24									
		27/03/24	QUANTUM PS3 (self cert)	QUANTUM PS3 (self cert)							
		28/03/24	AGENT BASED SYSTEMS								
	Bank Holiday	29/03/24									
		30/03/24									
	Easter Sunday	31/03/24									
	Bank Holiday	01/04/24									
		02/04/24						**GUI Deadline**	**ISCMTS Implementation Deadline**	**MCTS Modifications Deadline**	
		03/04/24	HPC CSWK 2 (self sert)		HPC CSWK 2 (self sert)			**Send Code to Till Deadline**			
		04/04/24					Internal Meeting (Heuristic Agents, discuss, distribute work)				
		05/04/24	**busy 1-4pm**				7:30pm Heuristic Agents Discussion	**Heuristic Agents Deadline**	**Game Run Testing Infrastructure**		
		06/04/24					2pm Report delegation meeting				
		07/04/24					8pm CODE HANDOVER	[NEW DEADLINE]			
		08/04/24									
		09/04/24									
		10/04/24									
		11/04/24									
		12/04/24						**"50%" Draft (to be send to Till)**			
		13/04/24									
		14/04/24									
		15/04/24									
		16/04/24									
		17/04/24						**AI Finished**	THOROUGH DDA TESTING PLAN, COMBOS ETC		
		18/04/24									
		19/04/24									
		20/04/24									
		21/04/24									
		22/04/24					Meeting?				
		23/04/24						**"95%" Draft (to be send to Till)**			
		24/04/24									
		25/04/24						**Presentation Poster Deadline**			
		26/04/24					Meeting?				
		27/04/24									
		28/04/24						**Code Quality / Robustness / Documentation Deadline**			
		29/04/24									
		30/04/24	**FINAL REPORT & INDIVIDUAL REPORT DEADLINE**								
		01/05/24									
		02/05/24	**POSTER PRESENTATIONS**								
		03/05/24									
		04/05/24	*A day of rest :D*								

Appendix G

User Survey Results

In this appendix we list exhaustively the comments provided from the user survey we conducted.

G.1 Test Scenarios

Scenario 1. Describe your stats (Which player are you? How many locomotives do you have, how many train cars, what tickets do you have and how much are they worth? Can you find the route to complete a ticket on the map? What information do you know about your opponent?)

Question 1: Did you find the last task difficult to perform? Why or why not?

Responses:

not difficult

no.

Wasn't too difficult to perform, couldn't find which player I was at first. Other than that, very easy.

Finding the route is not that easy, everything else is easy, but I do know how to play

The tasks were fine to achieve, things are laid out logically

Took me a while to discover that I'm player Alpha, it wasn't immediately obvious where I should look to find my player name until I saw it at the top. Instinctively I looked right to see the players but it was not immediately obvious which player I was. I believe I have 15 locomotives? I think the change in terminology makes me unsure. Tickets were quite easy to find as there is a distinctive label and the value is easy to find. Finding a route for a particular ticket requires extra steps, looking at the map.

Finding information about the opponent is simple using the scoreboard in the bottom right.

With limited experience at the boardgame it was a little challenging, however once I'd read the rules things became clearer. The actual in game elements were slightly confusing to begin with as there are no tooltips. I was alpha, I had 15 train cars, a mix of green, blue, pink, and any colour carriages.

It was fine

No, all seems fairly logical and clearly laid out.

No (I have already used the gui)

Intuitive

I have 15 train cars, I have 3 locomotives (rainbow card), I have a ticket from golden sands to obsidian avenue worth 8 and a ticket from dusty depot to obsidillian avenue worth 8. I can find the possible routes although with my current hand they are obviously impossible. I know Bravos score, cards, tickets and trains. I was initially confused but I am stupid. I got a little confused with the terms traincar vs carriage vs train vs locomotive.

Question 2: Are there any design changes that would make the last task easier to perform?

Responses:

Player Info Title for scoreboard

indicate on scoreboard which player you are

More obvious which player you are, maybe in a contrasting colour somewhere small on the GUI

Maybe highlight route on mouseover, would be nice to know how many tickets I have in total otherwise I just kinda cycle forever lmaoo. Also in ticket to ride (unless I'm wrong) cards are immediately when you take them, which is not the case here. Routes should be checked off if you have done them

Highlighting the cities of you routes when viewing them would help new players

For the player name, it would be helpful to have a popup when you start the game that says your player name or perhaps allows you to input your name. Also your name could be highlighted in the list of players on the right (maybe a different colour). I think finding a route for a given ticket could be simplified - perhaps you could highlight the specific to and from places in a different colour and making the font larger. I don't think you should show a route because part of the task is finding that yourself, but highlighting the start and end points would be helpful.

Tooltips would help to describe the elements on screen, rather than having to reference the rules pdf.

Maybe make it more clear which player you are

It easy to miss the number of trains the opponent has as it just a numerical value. Perhaps a column of train cars that decreases in height as each car is used?

No, all very clear and cool.

Prevent me choosing loco as 2nd ticket if I've already selected a colour ticket. I didn't realise train cars were the pieces until end game. There could be icons of 15 cars that reduce instead of a total. The black car icon is not black, so it's too similar to the loco icon which both appear white.

Easiest one first: train car and carriages are equivalent terms(?)and I think you should just chose one and use it throughout (train car can be abbreviated to T.C). (Note my final suggestion is about getting rid of the term train car entirely, but I'll still leave this up).

Second one: the locomotive refers to the rainbow card only (?) - I'd just refer to this card as a rainbow train card. The distinction between locomotive and train in the game is not analogous to the distinction between locomotive and train in real life. Its function in game is the same as a coloured train card really, it just is more powerful, but this is already indicated by the term 'multicolored'.

Also, the locomotive card in the players hand should be more obviously multicoloured.

Finally, a less clear suggestion: Select the terms (i.e. train, traincar cards etc) you chose when referring to the different elements of the game more carefully to improve the distinction between train cars and trains and/or improve the distinction between what's actually in your hand and what you have available to play with. Maybe it's just easiest to re-label 'Train Cars' (e.g. centre bottom of the game) as 'cards used (n/15)'. So the number starts from 0 and then in the rules you explain that you can use a maximum of 15 train cards in the game. I don't have to explain my reasoning sorry but basically, the game rules uses trains and train cars interchangeably sometimes. and other issues.

Scenario 2. Draw the required train cards to complete the route between Golden Sands and Ivory Street, and claim the route.

Question 1: Did you find the last task difficult to perform? Why or why not?

Responses:

not difficult

no

No, good instructions.

Yeah actually it wasn't easy, it's not clear you have to click the route first

Generally logical, the process allows you to choose how many locomotives to use each time

Getting the cards needed from the deck was a bit challenging. It's been a while since I've played ticket

to ride so I did not realise I needed to pick two cards. It was not obvious that something had happened after I chose a card from the deck and especially after choosing the rainbow locomotion card. It was not immediately obvious how you place a route down either.

At first I thought I had to click the trains deck at the top, and then click the card I wanted to add, but later in the exercise I realised I was drawing 1 random and 1 from the table. This was slightly difficult as it wasn't clear what my clicks were achieving.

Fine - can be a little confusing how to claim the route

No. Taking a card that is face-up its easy to tell you've selected it as it fades. When you take one from the pack, it is not so clear as there is no feedback. Sometimes the cards do not appear in my deck until I click that my go is complete

Easy to intuit - but that's only because I've played T2R so I don't know how easy for a newb. Why does bravo player only draw one card often?

No it was easy to execute once I got my head around the different elements of the game.

Question 2: Are there any design changes that would make the last task easier to perform?

Responses:

update train count before opponent's turn (i dont know what random card i got)

make it clearer when a card is claimed

No.

It would take a big redesign, but clicking the trains you have and then highlighting valid routes on the map to spend them on would feel a lot more natural imo because you're spending the tickets. Also tickets do not draw one at a time, really confusing when you're drawing two from the top of the deck. I get why it's like that, but drawing a card should just grey out rainbow cards instead. Actually, to return to the route idea, clicking on a route could auto-suggest the tickets you have (i.e. maximise the color and fill in with locos) which you could then confirm by clicking on the route. This would allow you to auto-complete routes by double-clicking them!

A notice to show you what train cards you have drawn from the deck would be nice

I think adding a number or some text saying that you have 1 card remaining after you click on a card on the left would have given the extra information needed. Also making the card transparent whilst this signals that it was selected, I think maybe a thick border and bringing the card forward would have been clearer?

I think some animations would have made it clearer what was going on regarding moving the cards from

the deck. If the coloured cards visually moved to where all your train cards are that would be nice. Feedback with the locomotion would be good because it was not obvious that anything had changed to the card.

When filling a route, again I think a border rather than transparent route would have been clearer. It also was not obvious that I needed to click on the icons on the bottom to fill the route. What would help is perhaps making the applicable cards appear bigger (larger image and text, maybe even a border) which may make it clearer that we need to click on these. It would also be cool to make it so that when you hover over a route, the appropriate cards are highlighted so we can check if we can make it - maybe some validation could be added that highlights whether you can do a route so we don't attempt one that we cannot make.

Lastly, perhaps a walkthrough on first playthrough would be helpful to get the user accustomed with the UI.

Perhaps a step by step tutorial (arrows pointing at the different elements, telling you how to interact with them.)

An animation of a card going from the pack to my deck would provide feedback. Instant update of cards in deck once cards selected

Perhaps a highlight over the train card area once selecting a route to indicate where you should press next. But this is mostly useful only when first learning the game.

The GUI could suggest combos of tickets valid for the route, or give message if none. Maybe a popup with the choices? But it does work well as is.

Nothing other than the locomotive -¿ coloured train suggestion that I made above. Note an alternative is to state the fact that "the multicoloured card is the locomotive card" earlier in the rules when explaining the players hand. You could rightfully argue that this fact is already present in the rules but it is still easy to miss.

Scenario 3. Draw and keep one ticket with the highest score, then find the ticket in your hand and describe the route.

Question 1: Did you find the last task difficult to perform? Why or why not?

Responses:

easy. would like to investigate inventory while opponent thinks

no

No, good visual instruction on how to discard tickets.

Not difficult, echoing edge -¿ nova nook, 8 pts

It would be more intuitive to select the cards to keep I think, but the UI makes it very clear how the process works, so this is not an issue

This was quite simple to do as the ticket drawing UI is really clean!

Bramblewood to Lumin Lake, rewards 8 score. It requires me to claim routes between Bramblewood ¿ Echoing Edge ¿ Haleswood ¿ Ivory Street ¿ Lumin Lake. I could also go to Dusty Depot ¿ Fairview Hills ¿ Haleswood ¿ Ivory Street ¿ Lumin Lake

Easy to find route with highest score. Tricky to see the route on the map as the tickets obscure the centre of the map and you cannot see all the destinations.

Easy. I selected a ticket but saw it was discarding with an X but was easy to undo it

Route dusty depot -¿ juniper junction. Was easy to perform.

Question 2: Are there any design changes that would make the last task easier to perform?

Responses:

no it was ok

if you could see the tickets you already have while picking

No.

But why click to discard??? yuck. surely click to keep. I think generally greying out invalid actions would be helpful (like not keeping any/discarding all). Should be able to hide the tickets to view the gamestate before deciding.

My only thought about changing here is that we are performing a negative action (discarding cards). Whilst this is communicated, I feel like instinctively I would select the cards that I want to keep so I would have selected one or two cards that I want and discarded the other. That might just be how my brain works though!

Perhaps an image of a mouse next to the word "Tickets" by the bag, indicating that you have to mouseover.

Display the tickets to the side or the bottom of the page so that the full map can be seen.

Discard button or a keep button?

No it was fairly obvious. I think accepting cards is more intuitive than rejecting cards when playing a game like this, but that's potentially a personal choice and it's fine either way.

Scenario 4. Describe the opponent's last action.

Question 1: Did you find the last task difficult to perform? Why or why not?

Responses:

easy

no

Little bit weird to read the last action as it is by ID and not name, and the route changed colour.

It's ok, AI only ever seems to draw one card, and I get the claim action but only after puzzling it out for a while. I don't get the [by ID], and I didn't realise the locations were alphabetised until later

It is fairly clear what cards the opponent has chosen to draw, but a bit less clear what route they have claimed

This was easy to do as the last action is helpfully highlighted on the top.

My opponent claimed Kaleidoscopia to Nova Nook with 2x Orange cars. Without remembering what I saw change on the screen, the information displayed at the top "(LAST TURN)" isn't clearly decipherable to me as a new player of "ticket to ride".

It drew a blue cafd from the table

No

I can understand it easily. But it is the most raw computery-like part of a very polished/professional looking gui and so isn't the most readable/digestable. Unsure what [by ID]'s significance is.

Some actions say [By ID] and I don't know that that means

His last turn he claimed a route from juniper junction to maple manor. It was not particularly difficult to perform as it can be easily understood from the sentence at the top.

Question 2: Are there any design changes that would make the last task easier to perform?

Responses:

could be nicer output format

last turn description could be easier to understand

Have the action described with the station's full name, and keep the route the same colour but have the

A/B written over the top in a bold/brighter colour.

Highlight the opponent's claim if they made one. Also drawing from the top of the deck isn't in included in the action, which is weird to me! Actually only if you draw once. Explains the one AI action tho.

Either highlighting the last played route or expanding the description to include full city names and length

Perhaps an option to scroll through actions would be nice. It would be nice when the AI places some trains to highlight what was last placed when hovering over the text?

Refinement of the "(LAST TURN)" text to clearly indicate which route your opponent claimed, using what cars, etc...

I don't know what the second card was (I assume from the deck?)

Cards taken from the deck are not reported

Perhaps more natural wording and explanation

The claimed routes could stand out more with brighter colour and/or bolder outline. Minor prob once I spotted the As and B's.

I didn't actually see the turn happen. Potentially making it so you see his turn actively happen (e.g. include animations). Also the option to replay bravo's turn again would be helpful.

Scenario 5. Describe your current score, including any added/deducted by tickets.

Question 1: Did you find the last task difficult to perform? Why or why not?

Responses:

asy

no

Quite easy to do, but had to click through the tickets

No. Why is there a number in brackets? Do the claimed routes count as points as in TTR? It's really hard to tell what the point contributions are.

The score board makes sense

Assuming that the score calculates all this for me then this is easy to find.

My score is 4. It says 3 tickets but I have no reference to know if that is adding or subtracting.

-11

The score only accounts for points from trains laid down on the map. No credit seems to being made for tickets / routes completed. I didn't look at my score during play.

score: 4. Easy peasy.

Question 2: Are there any design changes that would make the last task easier to perform?

Responses:

none

see all tickets at once

might be good to include a 'view all tickets' screen if you're holding a lot of them

Please present one total score, with a breakdown provided

At the end of the game it could give more info about which points are from routes

If the score does not have the total score including added or deducted by tickets then perhaps next to the score you could signal (in green or red) the added or subtracted from the base score. A tooltip could be then added to explain what this means. E.g. 4 (+2)

Some kind of tutorial to walk you through each UI element, step-by-step may help.

Add completed tickets to the score

Not sure if this is the best place to say it, but alpha and bravo could be replaced with eg 'player' and 'bot' to make it fully clear. Again, not very important.

Current score could be more prominent

No

G.2 Final Questions

Question 1: Were there any game components you found particularly difficult to read or interpret, or actions you found difficult to perform?

Responses:

lack of animations

having to click to continue after an action is not needed, not obvious, and easy to forget.

Nothing particularly difficult after reading the instructions the first time

Black train being white but I think that's it

It was a little slow to work out where the opponent placed their tracks

I found it difficult when navigating the GUI test to go back to the main menu, I had to close the game. I think most of the game was easy to interpret once I was used to the UI, maybe a tutorial would have helped. I think the game could have more clearly showed what the AI did in the last turn to make it obvious as the text on the top is not particularly clear when the AI places a route.

Not in particular. Most of it was solved by trial and error. Tooltips would help

Drawing from the deck twice - it wasn't clear if you'd taken one or two and if the turn had finished. Might be nice if you can look at your tickets while the opponent is thinking

I could not see the bottom of the screen unless I adjusted the size of the window - the windows bar at the bottom got in the way. When B draws a card from the table, that is announced, however there is no mention that B has also drawn a card from the pack.

When picking new route cards some of the map is obscured by the ui. It doesn't seem like there would be any easy solution to this however and it has been positioned well so basically only Ivory Street is covered. Similarly, if it is possible to open the ticket pouch while choosing tickets that would also be useful, but it's easy to remember so again, no big problem. Could there be a way to exit to main menu mid game rather than closing the whole executable. Sorry my only suggestions are little insignificant things like this...

The ticket choices obscure the map so you cannot see locations and make the right choice. I drew a wrong card this way and struggled to choose every time.

Honestly just the stuff at the beginning about understanding what the terms refer to. But it's a great game! Perhaps f you had an in-game tutorial round or a video to explain then that would also make things even better!

Question 2: Could you see a scenario where you might play this version of the game instead of an actual physical version of Ticket to Ride? Why or why not?

Responses:

it is more accessible and less commitment

no, want to play with people

No, I much prefer the social aspect of board games to the actual gameplay

Right now, never, because it is actually still harder to use, but if full advantage is taken of it being digital I could see it being quicker to play. A hard, fast AI would be fun (gonna do killbot in a sec)

It is nice to have a single-player version

Yes! It's really fun to play against the computer and to get some practice at how to get better at the game. The only reason I wouldn't would be to play it with actual people.

I play a lot of video games and using this to play virtually would be great.

Yes because it's quick and you can play it on your own. No because it's still more fun with real people

Yes, as it allows you to play solo. To play the physical game requires 2 or more people. However, I miss the social aspect of playing the physical version of the game

Yes, it has a shorter game time so easy to play as a bit of fun. The hard bots are also a nice challenge to repeat to try to beat.

I prefer playing against people.

I'd rather play this version as the AI doesn't judge me. I genuinely enjoyed the game!

Question 3: Any final comments?

Responses:

colourblind accessibility isn't entirely complete

yes

9/10 - IGN

It's just little things but they make a big difference! Would be cool if you could see the full game history, but really that's just a skill issue that I haven't memorised every turn. At least have the last two turns visible so I can think during my opponents turn

Fun to play. Processes generally feel intuitive. The use of sound, animation, or highlighting could help improve clarity at times. A few more maps or some music and sounds would increase the charm of the game, but are not needed. Graphics are nice and I couldn't find any bugs (I did see if it would let me pick up a locomotive as a second card)

You've all done a brilliant job! The game is a lot of fun and the AI works really well. The only comment I would have is that the game freezes when the AI is thinking. Perhaps this could be done on a separate thread to allow the UI to still be interacted with. Especially early on in the game, the AI does take a while to think (15seconds) which is tolerable but perhaps the AI could do some thinking whilst the player is deciding their moves? Not sure if that would be possible though.

Something to guide the players through the interface, and perhaps a sample turn of what to do to play a round would be helpful

Love the look of the game and really enjoyed playing with agent 4! Great job

Scoring summary is not easy to understand (beyond reading whar A and B have scored). Not sure why the number is repeated in brakets, e.g. 27(27). It would be better if the score was followed by a breakdown, e.g 27 (15 trains + 12 tickets), or 4 (15 trains - 11 tickets). Good fun to play and a nice mini-version of the game.

The gui is very cool (of course not the main point of the project but still it was a really good job)

Exellent GUI! Only real problem is choosing tickets when you cannot see the map

Appendix H

Project Specification

Attached is the initial project specification, as written 26th October, 2023.

Dynamic Difficulty Adjustment AI for Board Games

CS407 Group Project Specification

Sam James, Sam Medwell, Lukas Rutkauskas, Katie Yang

Supervisor: Professor Till Bretschneider

Year of Study: 4

October 26, 2023

WARWICK
THE UNIVERSITY OF WARWICK

Contents

1 Introduction

For a game to be engaging, it must provide a suitable level of challenge. If it is too difficult, the player will feel frustrated, whereas if the game is too easy, the player can become bored. Dynamic Difficulty Adjustment (DDA) is the process of learning the skill level of a player and adjusting the difficulty of a game as they play it [1]. In the context of board games, the game difficulty is directly linked to the strength of the opponent's play. Static difficulty levels (e.g. easy and hard) fail to match the skill level of every player and players will not know beforehand their current skill level in the game [2]. Moreover, players can more effectively improve their skill in a game when suitably matched against opponents at or slightly above their skill level [2].

We aim to create an AI agent to play the board game Ticket to Ride, an imperfect information game, which utilises DDA and an application that will allow human players to play against the agent developed. We will evaluate the performance of our AI using both simulated and human games. Our client, a member of the Tabletop Games society at the University of Warwick, will work with us to formalise requirements for the system based on their personal interest, as well as representing the general end users to ensure the project is able to deliver value in the ways of intellectual stimulation and enjoyment. Furthermore, we hope the project will build upon existing solutions and lay out further groundwork for future developments on DDA in board games.

2 Background

Machine players in games have already been explored in great detail, from rule-based agents to ones that incorporate machine learning or reinforcement learning. The purpose of this section is to outline the motivation behind the project, as well as to give a brief overview of the existing literature on DDA in games. Finally, we will introduce Ticket to Ride, the board game chosen for this project, and explore existing AI solutions that have been applied to the game.

2.1 Motivation

There are a number of motivations behind creating an intelligent machine player for board games. Firstly, board games generally require the participation of two or more players, and this option may not be available to everyone. The inclusion of one or many AI players allows people to enjoy a game at any time. For example, a dedicated player may wish to practice their strategy against a strong opponent, or someone without a suitable social group may wish to spend time playing against a machine player that behaves intelligently or authentically human-like. An AI player could also provide opportunities for new players to learn a board game without the pressure of having to play against more experienced players, which is especially relevant for board games with complex rule sets, as the game environment as well as the AI player will ensure the game is played correctly.

It is important to ensure the AI player provides an enjoyable experience for the human player. Aside from social aspects, the entertainment value of board games are typically tied to players' abilities to strategise in both short and long term in order to ultimately win the game. If the AI opponent is too weak, the game can feel boring and unengaging; if the AI opponent is dominating the game, it might cause frustration and demotivate players from playing the game altogether. Hence, it is important for any AI players implemented to provide a balanced game experience regardless of the player's skill level, which lead to the conception of Dynamic Difficulty Adjustment.

2.2 Related Work

In this section, we explore some previous work carried out on DDA AI in games. Obtaining an initial understanding of the area is important for planning the research direction of the project.

There has been significant work carried out in the area of DDA in video games but,

surprisingly, little has been done on board games. Many video games are classed as real-time where players make actions simultaneously and continuously. Therefore, there is difficulty in selecting the correct time to adjust the difficulty of the game without breaking the immersion of the experience [2]. Since board games are turn-based, an agent can only make actions at specified times which reduces the complexity in that respect.

On the other hand, difficulty of video games can be altered without changing the behaviour of any opponents, for example, a player could deal extra damage or the spawn rate of beneficial items could be altered [1]. This option is not available in board games since it would involve bending the rules of the game which would not provide a satisfactory experience. Instead, the difficulty of the game must be controlled by altering the intelligence of the opponent. Hao et al. points out that, even in video games, a player would not enjoy the feeling of loosing due to the opponent receiving unfair advantages as opposed to making more intelligent decisions [3].

2.2.1 Video Games

Silva et al. demonstrates DDA AI in the video game Defence of the Ancients (DotA) [1]. It works using a heuristic function containing three variables, one of which being the number of deaths incurred by the player, to estimate the strength of the player. By taking the difference in this function at consecutive time steps, the change in player strength can be estimated. The AI then attempts to match the progression or regression in player strength by changing between three defined skill levels. This is a simplistic approach but it highlights the fact that a DDA AI needs to be able to accurately estimate the skill level of the player.

Hao et al. uses Monte Carlo Tree Search (MCTS) to control two ghosts in the game of Pac-Man [3]. MCTS is an algorithm that uses random simulations of games to find best moves. They control strength of the AI by adjusting the computation time given to the algorithm. By carrying out many simulations using different computation times, they estimated an equation which maps computation time to the win rate of Pac-Man. However, these results were collected using a computer player for Pac-Man which used a very limited strategy. They also propose using artificial neural networks (ANNs) that are trained on games generated by MCTS in order to estimate the behaviour of the algorithm without the computational overhead. The use of ANNs represent an interesting research direction.

The work by Demediuk et al. uses MCTS to play a real-time fighting game [2]. They point out that the approach taken by Hao et al. requires a new formula that determines the computation time allocated to MCTS to be refined for each game. Moreover, this approach is sensitive to the hardware used, meaning different win rates would be achieved on different computers. Instead, they propose a few small modifications to the algorithm itself which causes it to search for moves with an expected value of zero as opposed to those with maximum value. The result is that agent will play better moves when it is ahead and worse ones when behind. They also explore how to make an agent play more proactively by setting a threshold around zero in which actions would be selected randomly.

Another previous work Demediuk et al mention is that by Andrade et al. [4] which uses Q-Learning, a reinforcement learning algorithm, to estimate the value of game states. These values are then used to create a rank order for a set of available actions in any given game state. To start with, an agent would select the action in the 50th percentile of value (level 0.5); the level is updated to reflect the current strength of the player. However, Demediuk et al. says that rank ordering is not a good way to determine the strength of a given action, since a certain state may result in a disproportionate number of good or bad actions [2].

2.2.2 Board Games

One piece of existing work on DDA for board games is a dynamic difficulty chess AI created by Kennedy [5]. It makes use of the Stockfish static board analyzer and a basic minimax algorithm. A hand crafted function is used to map the board evaluation to a probability that the agent will blunder and search depth such that the agent will increase its search depth and decrease it blunder probability in losing positions. In this example, player strength is based purely on the current game state; it could be interesting to investigate ways to take into account the full game history.

AlphaDDA, proposed by Fujita [6], is an adaption of AlphaZero that includes DDA. AlphaZero is an algorithm that combines MCTS and neural networks to produce super human play in games such as Chess and Go. AlphaDDA uses the exact same architecture as AlphaZero except it selects actions proportional to its chance of winning the game in a similar way discussed previously.

2.3 Ticket to Ride

Ticket to Ride is a popular modern board game where players aim to build the most successful railway network across a specific region of the world. The game was designed by Alan R. Moon with its first edition released in 2004, and was highly successful, being widely known in the board game community, with the first edition winning the prestigious Spiel des Jahres prize [7].

2.3.1 Rules

Ticket to Ride can be played by between 2 and up to 5 players, depending on the edition.

As displayed in Figure 1, the game board represents a section of the world covering regions such as Europe, North America or Asia, and is comprised of a set of cities and possible connections between these cities. In the most simple editions, players interact

Figure 1: Ticket to Ride Europe Board. [CREDIT: Days of Wonder]

with train pieces, train cards and ticket cards. Train pieces are used to claim a connection for the given player, train cards are resources that must be collected in order to claim a connection and ticket cards specify a route between two cities (that do not have to be neighbouring cities).

At the beginning of the game, players are dealt destination ticket cards, which players aim to complete by the end of the game (and should be kept secret from other players). Completing a destination ticket grants points, but failure to complete it deducts points.

In each turn a player can either (i) draw train cards, (ii) claim a route, or (iii) draw additional destination tickets.

i Train cards are drawn from two piles, either a blind deck or from a set of 5 visible cards that everyone can view.

ii In order to claim a route, a player must play a set of train cards that are of the same color and quantity as the spaces marked by the connection on the board. For example, a connection that requires four pink train cards must be claimed by playing four pink cards from the player's hand. Once claimed, the player places their train pieces onto the corresponding route, that shows the player's ownership of that route.

iii A player can draw up to 3 additional ticket cards, but must keep at least one of these routes (they may discard the remaining routes if they should like).

The game continues until at one player has 2 or fewer train pieces remaining, from which every player has one final turn. The player with the most points wins the game.

The game is scored as follows:

- Players earn points for each connection they claim (longer connections yield proportionally higher score)
- Players earn points for every destination ticket they complete (and lose the same amount of points for those they do not).
- Additional points are provided to the player with the longest continuous path on the board.

1

2.3.2 Using Ticket to Ride to Demonstrate DDA in Board Games

Ticket to Ride, like many other board games, combines strategic planning, adversarial interactions, hidden and visible information, resource management as well as an element of luck. This creates an interesting environment for developing a DDA AI which appears to not have been explored very deeply.

We are interested in exploring AI in modern classic games such as Catan or Carcassonne which are less popular than classics such as chess but still have a wide player base. Since we want to start with exploring DDA in two-player board games, we discarded Catan which didn't appear to have an engaging two-player experience. For example, the trading mechanic is diminished given the game becomes zero-sum and random chance in receiving resources is magnified. It will be important to start with a two-player game since it reduces the complexity. While Carcassonne is a good two-player game candidate, it is perfect information unlike Ticket to Ride and we thought that it would be exciting and novel to explore DDA in imperfect information board games.

Moreover, work has been done previously on creating Ticket to Ride maps that are balanced and produce engaging gameplay [8]. This is important since we will need to use a reduced sized map due the large state of Ticket to Ride; in fact the smallest official game of Ticket to Ride (New York) has a larger state space than Chess [9]. Being able to create maps of suitable size will help us collect more valuable results.

Thus, we settled on Ticket to Ride, which demonstrates a diverse set of game strategies whilst maintaining a strong balance between ubiquity and novelty.

2.3.3 Physical Game Editions

Ticket to Ride has several versions, that cover different regions of the world and may have small differences in rule-sets. Ticket to Ride Europe, the modern edition, is more balanced [10] compared to the first edition which covers North America. However, there are also smaller maps, such as Ticket to Ride London, medium size maps such as Ticket to Ride Switzerland on top of the larger maps such as Ticket to Ride Europe.

Different game editions may have additional rules. For instance, Ticket to Ride Europe

[1]The full rules of Ticket to Ride can be read at the ticket to ride website. The official rule book for the Ticket to Ride Europe edition can be found at https://ncdn0.daysofwonder.com/tickettoride/en/img/t2re15-rules-EN.pdf.

has different connection types, tunnels and ferries, which require different train card conditions, as well as train stations that can unblock a route for the player that played the station.

We plan to investigate Ticket to Ride Europe, Ticket to Ride Switzerland and Ticket to Ride London. The latter two are better suited to 2 player, whereas the former is an improved version of the first edition game.

Finally, members of the board game community and academic circles have developed custom maps [8]. We plan to start DDA agent development using such custom maps, before working up to the official London, Switzerland and Europe maps,

2.3.4 Prior AI work on Ticket to Ride

There has been recent work on AI in Ticket to Ride by Yang et al. [9] which uses a state-of-the-art reinforcement learning algorithm known as Proximal Policy Optimisation (PPO). As well as this, it provides analysis of the state space of the game and a survey of algorithms used to play imperfect information games. This paper will provide a useful starting point for investigating the techniques that can be applied to play Ticket to Ride. One algorithm they mention is called Information Set Monte Carlo Tree Search (ISMCTS) which applies MCTS to games of imperfect information. Therefore, one approach may be to combine ISMCTS with modifications made to MCTS discussed in the previous section. They also explain and compare rule-based AI created in previous work against the one they develop.

Another work by Strömberg et al. uses a Double Deep Q-Network (DDQN) to play Ticket to Ride [11]. This is another reinforcement learning approach suggesting further research on how DDA could be applied in reinforcement learning will be beneficial.

2.4 Conclusion

From the initial research phase, adaptions to the Minimax and MCTS algorithms seem like promising approaches for creating a DDA AI for Ticket to Ride as long as they can be adapted for games with imperfect information. As well as this, reinforcement learning algorithms that make use of artificial neural networks should be researched further. A suitable approach should be able to produce very strong play since the difficulty of an AI is capped by its maximum strength [2]. As well as this, a stronger AI would be able to evaluate the opponents skill level more accurately.

3 Objectives

1. Build upon existing research on Dynamic Difficulty Adjustment (DDA) to implement and analyse AI models for two player Ticket to Ride, an imperfect information board game.

   (a) The AI should offer a balanced and appropriate level of challenge to player of any skill level, facilitating the player's improvement in the game.

   (b) The AI should consistently achieve a provided target win rate against opponent players.

   (c) Players should find the AI enjoyable to play against, based upon their assessment of key product indicators such as the AI's turn-time, "human-likeness" and challenge.

2. Develop an application that allows a player to play Ticket to Ride against the developed AI.

   (a) The user interface should be accessible, intuitive, and enjoyable to interact with.

   (b) The application should be able to be easy to install, across several platforms.

4 Deliverables

1. Develop the *game environment*: the core game system which encapsulates all the game mechanics of Ticket to Ride and maintains the game state.

   (a) Define an interface that allows communication between the game environment, game application front end and the AI.

   (b) Build the core game system that implements Ticket to Ride game rules.

   (c) Build a console user interface that can affect the game environment, for internal testing and verification purposes.

2. Develop an AI agent to play two-player Ticket to Ride which dynamically alters its skill level to match the opponent.

   (a) Produce a report assessing the efficacy of various approaches to developing a DDA AI based upon the results of implementations for each approach.

   (b) Complete the back-end system of the game application by integrating the agent with the game environment.

3. Develop the front-end of the game application.

   (a) Design and implement a graphical user interface, which dynamically displays the game state as well as providing the player with ways to interact with the game environment.

   (b) Implement additional game elements including a game menu, settings, and media such as art and sound.

4.1 Extensions

Below we have outlined possible extensions to the project which could be explored in addition to the completion of the core objectives.

4. Develop a human-friendly feedback system to help the player improve their gameplay, based upon information learnt by the DDA.

5. Investigate how DDA AI agents interact in a multiple player or multiple AI environment.

5 Architecture

This section aims to provide an insight into the preliminary planning that has taken place thus far. Since a large part of the DDA development requires the resolution of difficult questions that must be rooted in further research, this section mainly concerns decisions made regarding the technical implementation of the game environment, as well as plans for how the game environment, game application front-end and the AI will interact and be integrated together as part of a larger system.

5.1 Game Environment

The game environment is the application's foundation. It should implement the rules and mechanics for the chosen game, store game states, and provide operations players can perform to alter the game state. It will provide a common interface for the UI and AI alike, and is responsible for receiving, executing and passing on relevant information of the game state.

We have chosen to develop a desktop application rather than a web application. The main rationale behind this is that a number of additional complexities come with a web application, such as hosting and a more restricted set of utilities for front-end development. For this project, we would like to focus our efforts on developing the DDA AI, hence we want to minimise unnecessary time spent on the game environment itself.

Two approaches have been considered for the development of the desktop game environment. The first approach we considered for the game is to use a game engine, with good candidates being Unity or Godot. As game engines are tailored to game development, we will benefit from the efficiency granted by using built-in tools and features, as well as visual editors which can speed up level design. However, productivity will take an initial hit due to a steeper learning curve for these tools. Furthermore, we are concerned by software bloat induced by many unnecessary features that come with game engines, since Ticket to Ride, as a board game, requires a simple 2D implementation, whereas game engines provide tools for 3D features such as lighting, collision detection and particle effects. This bloat could lead to negative performance and storage implications.

The second approach explored is to directly implement the game in our chosen language, using media APIs, such as those provided by the Simple and Fast Multimedia Library (SFML). SFML is a cross-platform software development library designed to provide a simple API to multimedia components, aiming to ease the development of games and multimedia applications. Compared to a full-fledged game engine, SFML is

much more lightweight in nature while still providing hardware-accelerated 2D graphics with OpenGL. Whilst SFML is a library developed for C++, it provides bindings for Python and Java, which provides flexibility to our development stack. However, it has limited built-in features compared to most established game engines.

Further investigation will be conducted to determine which approach is more appropriate for this project. The team has some basic experience with the aforementioned game engines, hence an initial step would be to experiment with designing a simple game in SFML to test the feasibility of the second approach. Regardless of which approach we will take, a common intermediate step will be the implementation of a console interface by the end of week 10. A simple console interface at an earlier stage of development will provide means for testing basic game mechanics as well as allowing for early AI testing. The final UI will take significantly more time to develop to ensure the user experience satisfies our objectives and will require significant client engagement for validation.

The game system should also be capable of reading and writing a log of game turns, including the events that have transpired each turn such as each players' actions. This can be used as a part of the evaluation process, where each game can be effectively "replayed" for analysis of the AI's rationality. Additionally, being able to play an entire game from a capture can be used for automated testing. Furthermore, this log can be used as a basis to provide feedback to the player at the end of the game, one of the extensions we intend to explore for the project.

5.2 User Interface and Interaction between Components

The goal of the user interface is to let human players engage with the game environment, hence facilitating the interaction between human and AI players. As the user-facing component of the system, the UI aims to resemble the original game in its board game form, as well as provide a user-friendly experience for the human player. It will render the game's graphics, and feedback the player's operations to the game environment. Specific inspiration for the UI will come from the board game design itself, as well as a previous official digital adaptation of the game[2].

A common interface should be developed for the AI. As a number of models will be implemented and trialed, it makes sense to establish a common schema that details the information that will be given to the AI, as well as their type, and lastly any actions

[2]The digital version of the game, which has been discontinued pending a refresh, can be found on several platforms, from Asmodee Digital (https://www.asmodee-digital.com/en/ticket-to-ride/)

should be standardised. Ideally, the system should be able to switch between different AI models by simply replacing the name of the model loaded in.

6 Legal, Social, and Ethical Considerations

Legal, social and ethical issues have been considered in order to ensure the validity of the project. These issues inform key considerations which must be taken into account during development, whilst they also set the boundaries of what can be done.

6.1 Legal Issues

We must consider the issues of copyright, software licenses and data protection, which raise potential legal implications for the project.

6.1.1 Copyright

Ticket to Ride is a copyrighted board game, which is owned by Days of Wonder[3] Replicating its mechanics, artwork, or any other copyrighted elements without proper authorization could lead to copyright infringement claims.

Ticket to Ride has trademark protection for its name and logo. Unauthorized use of the game's name or logo can lead to trademark infringement claims.

However, these will not present an issue for our project, as our use case falls under Fair Use, since the project falls under non-commercial research and private study.[4] There is negligible financial impact to the copyright owner due to a lack of distribution. Furthermore, several team members own physical copies of the game. We will have to obtain proper permission and licensing agreements to ensure compliance with intellectual property rights if we intend distribute the application publicly, online, or commercially.

6.1.2 Tools and Technologies

Software and tool licenses must be adhered to. The technologies referenced in Section 9 are all free to use and some are open source. Public hosting of the application may breach licences of any new technology or data that may be considered during research stages.

[3] Additionally, Ticket to Ride has registered trademarks in the UK's Intellectual Property Office, an example can be found at https://trademarks.ipo.gov.uk/ipo-tmcase/page/Results/1/UK00003682261.

[4] A a guide produced by the Government of the United Kingdom's copyright law exceptions can be found at https://www.gov.uk/guidance/exceptions-to-copyright.

6.2 Social Issues

When designing the application, we need to take into account different disabilities and accessibility preferences. Measures to achieve this could include providing colour settings for people with visual impairments such as forms of colour blindness and alternative control options for people without full motor control of both hands.

Ticket to Ride features several locations and cultures, which should be represented with sensitivity and accuracy by avoiding stereotypes and misrepresentations to maintain respect and avoid potential backlash from users.

6.3 Ethical Issues

Research involving data obtained from human participants raises ethical considerations. However, as we are collecting feedback to evaluate our software as opposed to using user data to develop our models, we do not need to apply for university approval.[5]

[5]The University of Warwick's policy for Ethical consent can be found at https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs310/ethics.

7 Project Management

The success of a project relies on a foundation of project management. We have chosen and modified varying management methods and tools that are best suited to our project conditions. In this section we will specify in detail the procedures that we will follow and the motivations behind them.

7.1 Methodology

We have used Scrum [12] as the basis of our approach. We have chosen this methodology to respond to our team arrangement, as well as our project conditions. We have a small development team comprised of 4 individuals that must be responsive to both a customer and a project supervisor. We are students who have interwoven schedules and significant external commitments and varying capacities week to week. Our project has hybrid characteristics, with both research and software development components, and requirements can change based upon results from research or from discussion with the customer. We have immovable deadlines, which include generating a specification, progress presentation, final report and final presentation.

A project with changing customer requirements and ambiguity does not suit a formal waterfall approach. Furthermore, the varying work capacity and schedule make it difficult to plan long term goals in granular detail. Based on these restraints, an agile approach will most easily adapt to our project conditions. However, we do have long term immovable deadlines that constrain flexibility which must be taken into account in our decision to choosing Scrum.

7.1.1 Scrum

Scrum is known for its adaptability, making it ideal for dealing with changing client requirements, and will also help us respond to the results of our AI research: scrum allows for continuous feedback and the re-prioritization of tasks. The core Scrum roles of Scrum Master and Product Owner help facilitate collaboration between our small team and maximize productivity. Furthermore, Scrum is well suited to accommodating projects with our hybrid characteristics: the iterative nature allows for the continuous integration of research findings. Finally, several team members have professional experience working in Scrum teams, which will make it more natural to follow the methodology.

We will be running 2 week sprint cycles, which will allow a suitable volume of work to be completed, whilst allow for meaningful contact with our supervisor and other stake-

holders.

7.1.2 Drawbacks and Modifications

However, there are some drawbacks to choosing Scrum, given our conditions. Firstly, varying team capacities and external commitments could make it difficult to allocate and maintain fixed times for specific Scrum ceremonies, such as daily stand-ups or sprint planning sessions, which will impede maintenance of a rhythm. As a result, we have had chosen not to perform Daily Stand Ups, and we will have to introduce flexibility for meeting times (except for the Planning meeting).

Secondly, immovable deadlines hinder Scrum's flexibility: Scrum is designed to accommodate changing requirements and adapt to feedback in an iterative manner. Being deadline-driven will impose restrictions and pressures onto scope management. In order to work towards the key document deadlines, we will use a Gantt chart to map an overall high-level work structure which will be taken into consideration during planning meetings.

Thirdly, the research-intensive nature of our project could develop issues. Whilst Scrum can continuously integrate research findings into development, Scrum does not provide a proper framework for managing research-driven activities, or the ambiguities spawned from such. As a result we will have to mix in additional strategies in order to make our project work. This includes an unusual density of sprint-length spike[6] tasks, especially for the AI development side, due to significant remaining research. Furthermore, the hybrid project characteristics have led to a dual track development approach to scope distribution, as discussed in Section 7.1.3.

7.1.3 Dual Track Development

Our project is hybrid in nature, with significant software engineering and research components. During our first few sprints, we will split our work into these two areas, with two team members working on each component. We will adjust the break down of responsibility according to the outcomes of retrospectives.

In a typical Scrum, an increment is produced at the end of each cycle, that everyone's work is integrated into. This will not work in our setting and is why we opted for a dual track approach. The work done in each sprint in game development track will be reflected in the increment, whereas the AI work will only be integrated at key points throughout

[6]Spikes are time-boxed research activities that can spawn other tasks.

the project. This is to reduce time spent integrating partial or unfinished components.

7.1.4 Code Review

For non-trivial tasks, we will require the review of a team member before merging the changes into the main branch of the code. This additional layer of quality control will help to catch potential bugs in code that might have been missed, as well as help to improve code readability and maintainability, helping to enhance the overall quality of work completed. Furthermore, we will employ peer programming and research sharing sessions in order to gain an understanding of another team member's area of the work, encouraging collaboration and knowledge sharing of code.

7.1.5 Meetings

We plan for 7 meetings per sprint, which can be supplemented by ad-hoc meetings as necessary. See Section 7.2 for an explanation and justification of Sprint Planning, Sprint Review and Sprint Retrospective meetings, which are timetabled as in Table 1.

Table 1: Sprint Team Meetings Timetable. Each sprint cycle starts on Friday. Meetings written in bold text are essential meetings, those not in bold are optional.

Week	**Day**						
	Fri.	Sat.	Sun.	Mon.	Tue.	Wed.	Thur.
1	**Sprint Planning**	-	-	General Meeting Slot		Team Time	-
2	Refinement	-	-	General Meeting Slot		**Sprint Review + Sprint Retro-spective**	-

7.2 Ceremonies

We have chosen to perform three out of the four scrum ceremonies: sprint planning, review and retrospective meetings.

Sprint Planning In the Sprint Planning meeting, we assess what work we plan to complete during the next sprint cycle. This includes determining team member availability for the current sprint to ensure our work rate is realistically adapted to external factors. New tasks may be created and added to the backlog. In this meeting we assess

and prioritise our existing backlog of tasks, including any newly created tasks. Then we will assess any unassessed tasks (including determining a task's scope using story points). We then will draw an appropriate volume of work from the Product Backlog to match the team availability for the given sprint, assign them to a team member(s) and move the tasks to the Scrum Backlog column in our scrum board. We plan to do our sprint planning meetings on Friday of the first week in each sprint cycle.

Sprint Review In the Sprint Review, we will demonstrate the work completed during the active sprint to the relevant stakeholders. This will take the form of a fortnightly meeting with the project supervisor. Afterwards we will provide a summary of the sprint to the client. The Sprint Review acts as a framework to enhance the structure and purpose of these meetings, with each meeting acting as a demonstration for all the work since the last. We plan to put together a simple presentation for each sprint that showcases what tasks we have completed and how they fit in to our overarching goals. This ceremony is important not only to keep our stakeholders in the loop with our project and receive feedback, but also to document our work for future reference in the team.

Sprint Retrospective The Sprint Retrospective is a Scrum ceremony used by team members to reflect on what went well and what what can be improved in a sprint. We will conduct this meeting immediately after our Review (see Table 1) to reflect on how we want to change methodology and resolve any team or workload issues. We intend for this meeting to be brief, and plan to append team time where we can work through any larger issues raised in the Review. This meeting is a way to integrate continuous improvement to our workflow.

(Not a) Daily Stand Up In Scrum, a daily stand up is a quick daily meeting designed to quickly inform everyone of what's going on across the team. This is not appropriate for our conditions. Due to reasons outlined in Section 7.1.2, we will replace formal daily stand up with in person or digital correspondence, varying on the day, and arrange a hoc meetings to address any team concerns.

7.3 Artefacts

We have chosen to implement three main Sprint artefacts: the Product and Sprint Backlogs as well as the Increment.

Product Backlog Contains all planned features that are prioritized against each other within the backlog, that is managed by the Product Owner.

Sprint Backlog and Board The Sprint Backlog maintains the features that are to be worked on during the current sprint. We plan to use a scrum board which will track the progress of these tasks. The scope of the sprint backlog is fixed for the duration of the sprint, based on the availability and commitment of the team. If a team member has finished their assigned tasks in the sprint backlog, they can draw tasks from the product backlog.

In order to visualise and monitor task progress, we will use a scrum board. This is a useful tool for the team to monitor whether a given task is in the sprint backlog, in progress, awaiting a review or done (as seen in Figure 2). Furthermore, it allows the team to know what each other member is working on.

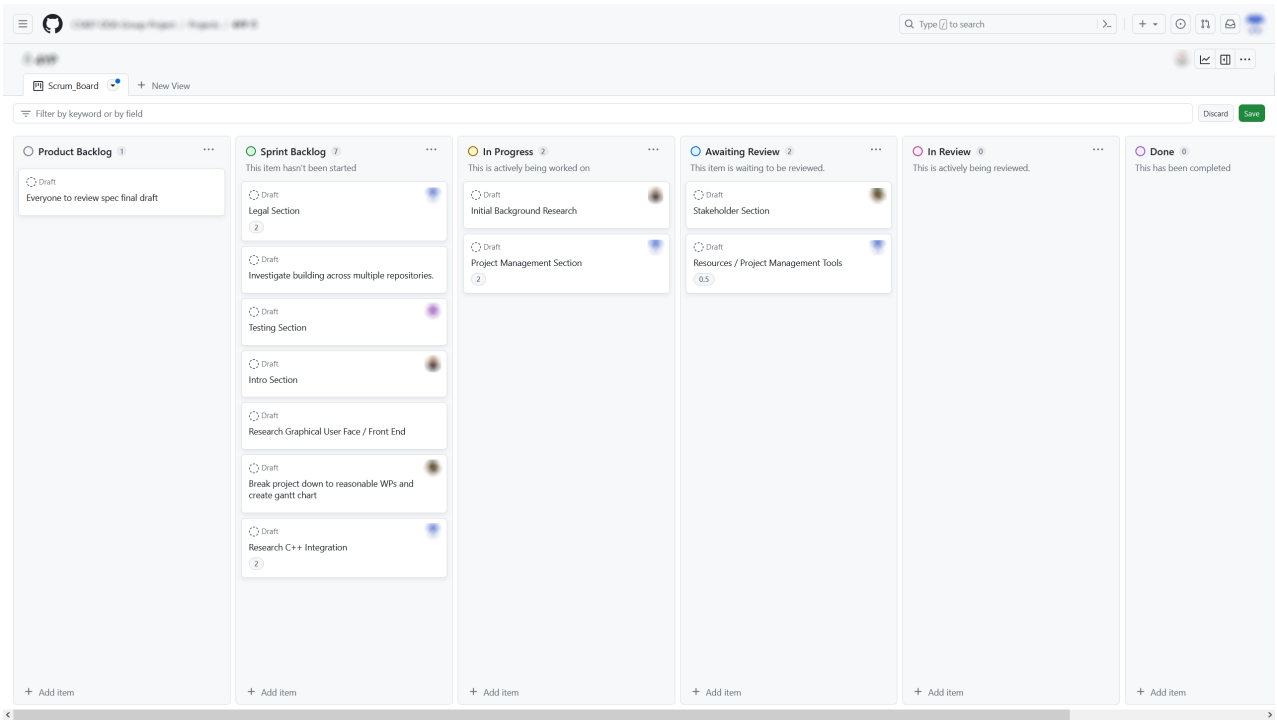


Figure 2: Product Backlog, Sprint Backlog and Scrum Board, hosted on Github.

Gantt Chart We will use a Gantt chart to work toward deadlines and show tasks at the epic-level, but this is meant to be used as a guide, since the sprint planning meetings will override this, as is the way with agile development. (See Figure 4 for our current long term development plan).

7.4 Roles and Responsibilities

As a team of four working on a software-research hybrid project, every member will take the role of developer and researcher. However, it is necessary for every member to take distinct additional roles to properly divide responsibility.

Defined roles will ensure the project is delivered successfully, with the rationale being that each member will be able to individually push the project forward, and a single point of contact for each major aspect of software development will ensure decisions can be quickly, or if needed, unilaterally reached.

For this project, we have chosen to assign a defined role to each team member, including Scrum and non-Scrum specific roles.

Scrum Master Sam James

For the purposes of our project, the Scrum Master will have a reduced responsibility set compared to their responsibilities within a multi-team larger project. Our Scrum Master will lead Scrum ceremonies, with exception of Sprint Planning, for which the Product Owner will be responsible. The Scrum Master is responsible for overseeing the progress of the sprint and coordinating between team members and resolving contacts. The Scrum Master is also responsible for ensuring the well-being of each team member, making sure that they are working at an appropriate load and are in a healthy mental and physical state.

For the purposes of our project, the Scrum Master will have the additional responsibility of managing key documents, such as the project's specification and final report.

Product Owner Katie Yang

The Product Owner ensures the client and other important stakeholders' needs are met. They are responsible for defining and communicating the overall vision for the project to the team, which should primarily align with customer interests. In this case, the Product Owner will maintain the team backlog on both game and AI development, ensuring any stories created accurately reflects the vision for the project and contributes to the planned roadmap, as well as re-prioritising items for sprints to ensure continuous value flow. There should always be meaningful work that developers can pull from the backlog. For this project, they will also be the primary customer and admin contact in order to bridge the team's development efforts and client expectations, and to collect and apply

any feedback.

Technical Lead Sam Medwell

The technical lead is expected to have an understanding of all of the technical aspects of the project and how they fit together. This is important as work is split up between different components such as the game and the AI which need to be integrated together. The technical lead will:

- Lead decisions that concern the technical direction of the project such as the choice of algorithms to implement;
- Be responsible for the research direction for the DDA AI;
- Manage and organise the collection of sources, e.g. papers, that are used throughout the project.

Testing Lead Lukas Rutkauskas

The Testing Lead is responsible for overseeing the quality assurance process. This includes developing a comprehensive testing strategy and defining test cases that evaluate the core functionalities of the game environment and DDA. Most importantly, they will safeguard the quality and stability of the project by employing a testing framework that is able to both efficiently test the consistency of the DDA AI and maintain the integrity of the game's mechanics and rules.

7.4.1 Dual Track Roles

Each team member has development and research responsibilities, which will be focused in either track, as explained in Section 7.1.3. To begin with, Sam Medwell and Lukas Rutkauskas will be working in the DDA AI track, whereas Sam James and Katie Yang will be working in the game application track.

These roles will be assessed in retrospective meetings, since one track may require more team members than the other during different stages of development. Furthermore, a rotation of members can help familiarise the members each track with the work of the other.

8 Stakeholder Analysis

This section identifies the main stakeholders for the project, evaluates their interests, and provides engagement strategies that aim to fulfil respective expectations.

A key stakeholder in this project is the client, who is a student at the University of Warwick, as well as a member of the Tabletop Games society. They are curious to explore the application of DDA to a board game, as well as to provide a end-user perspective as a representative from the potential user base of the system. During the project, the team and the client will work closely together to ensure that the client's feedback is incorporated, and that there is transparency throughout the development process. In this project, we want to place an emphasis on the iterative aspect of the agile methodology, which will ensure the final product fulfils the client's expectations and ultimately delivers value to them. This is done at the end of every sprint where the client will be updated with the project progress, which allows the client to feedback to the team. This then allows the team to shape requirements between sprints to better fit client needs, and to continuously improve upon the existing solution.

Other stakeholders in the project include the module organiser, the project supervisor, the team, and the end users of the system. The module organiser's primary interest is to ensure the project aligns with the modules' goals, and it is important they are informed of major changes in the project. The supervisor is a subject expert that works much more closely with the team, as the team will need to seek out their technical expertise, as well as advice on project research and the development process. The end users represent the general target demographic of the final product, and it is important that the team provides a solution that fulfils the client's requirements, while also factoring in the needs of the wider audience.

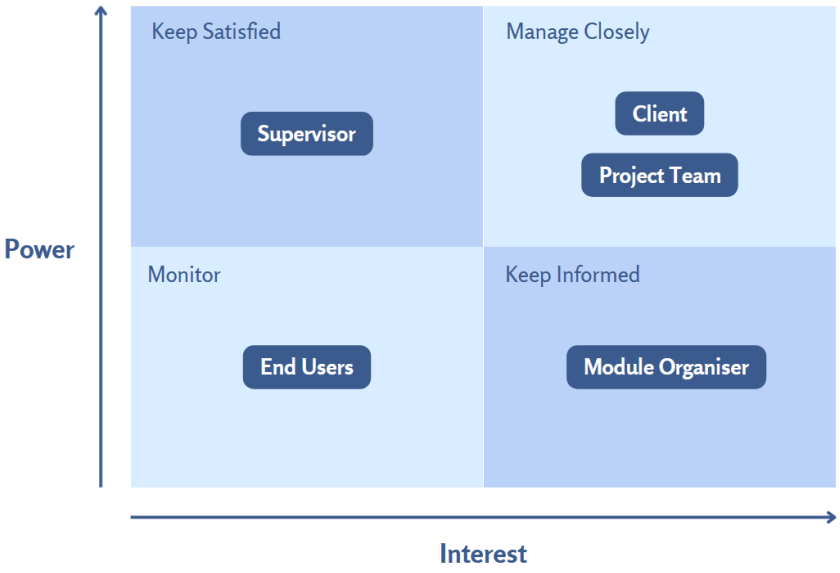


Figure 3: Power-interest grid for stakeholder analysis.

The engagement strategy is largely based on the stakeholders' interest, power, as well as availability. The power-interest grid shown in Figure 3 segments the aforementioned stakeholders into four groups, each with a respective management expectation. This diagram informs the general engagement strategies: stakeholders who hold more power in the project should be kept satisfied with its direction and progress, while stakeholders who are more interested in the project should be kept up-to-date with project's status. Table 2 below expands on the actual engagement strategies employed in this project based on previous analyses.

Table 2: Table of stakeholders and engagement strategies.

Name & Role	**Description**	**Engagement Strategy**
Client (Robert Crawley)	Has high interest and influence, informs the general direction of the project.	Manage closely with fortnightly meetings detailing project progress, and should review key design documents.
End Users	An extension of the client, represents the general users of the system.	Feedback should be taken into account when considering the system's designs.

Supervisor (Till Bretschneider)	Has high influence in the project, is a subject expert and an assessor of the project.	Keep up to date with project status. Regular fortnightly meetings with the project team are scheduled, during which discussion of the project will take place.
Module Organiser (Sara Kalvala)	Sets the framework of the project itself.	Keep informed of substantial changes to the project.
Project Team	The developers working on the project.	Team leads to manage members closely, team keeps each other informed of updates and hold weekly sync-up meetings for discussion of ideas and project direction.

9 Resources

This section is dedicated to detailing the tools and technologies that will be utilised throughout the project. We will outline the resources at our disposal and explain their role in enabling our team to effectively plan, develop, and monitor our project.

9.1 Technologies

We will be using Visual Studio Code [13] as our primary code editor for the development of the system. It is streamlined and provides all the necessary tools for our relatively simple workflow, such as debugging and version control through Git. Specific technologies for developing each component of the system will be detailed below.

9.1.1 AI Development

To develop the AI, we would like to use Python [14]. Firstly, Python is the most popular language for machine learning and deep learning, meaning we will have plenty of well developed tools if we choose an approach that uses neural networks. As well as this, the team members that will be working on the AI are familiar with Python which reduces development time. The technology choice for the game environment may mean Python cannot be integrated, in which case a different language may be used.

Computational resources will inevitably be a limiting factor in the strength of the AI so we should aim to use the best ones available to us. We can make use of the department's batch compute system and compute nodes to train deep learning models and run game code respectively.[7] Cloud compute services could be explored but the success of the project should not rely on this, since the map size can also be altered to reduce the complexity of the game.

9.1.2 Game Environment and Front-End

As discussed in the Architecture Section, the game environment would either be developed with a full-fledged game engine, or directly through a graphics API such as the Simple and Fast Multimedia Library (SFML) [15]. Further experimentation is required to conclude which option is more fit for purpose. For SFML, development would be done in C++ and we can explore other language bindings if necessary. For game engines, our choices would be Unity [16], which is more feature-rich, and Godot [17], which is more lightweight and works well with 2D games, both of which primarily support C#.

[7] The User Guide for the DCS Systems, including instructions for the use of the batch compute system, can be found at https://warwick.ac.uk/fac/sci/dcs/intranet/user_guide/.

We will aid the development of game art through the use of a variety of digital illustration tools such as Photoshop.

9.2 Project Management Tools

9.2.1 Code Management — Github

We chose to use Git for version control due to its ubiquity and team familiarity. We use Github as a the host for our Git repository to allow the team to store their code in a centralized location, as well as to allow for online backups of the codebase.

9.2.2 Progress Monitoring — Github Board

To monitor the progress of tasks, we chose to use a Scrum Board. We have chosen to use Github Board due to its light feature set and integration with code repositories. More heavyweight Project Management solutions such as Jira are excessive for the size of the team and the project, and would only burden team members with learning new software and workflows.

9.2.3 Documentation Management — Google Drive, Overleaf and Zotero

We will store documentation materials such as meeting minutes and notes in a Google Drive. A free, cloud based solution, Google Drive (and Google Workspace by extension) allows for simultaneous collaboration. However, we must be careful to take offline backups and plan contingencies for periods of poor internet connectivity.

Key milestone documents such as the Specification and Final Report will be written in Latex. We will use Overleaf, an online Latex editor, that allows the real time document co-authoring of a single Latex document, which is needed due to the group nature of the project.

Zotero will be used as our reference management system. Zotero is well suited for this role since the reference list can be accessed by and contributed to by more than one person simultaneously. Furthermore, Zotero can integrate with Overleaf (and thus our documentation). Tags will be used to organise the sources, by categorising them by algorithm and genre and marking which ones have been read.

9.2.4 Communication — Email, Teams and Discord

We will use Teams to schedule formal video meetings with stakeholders, whereas for informal internal team communications we will use Discord.

10 Testing and Evaluation

This section outlines the types of testing that will be carried out throughout this project's development. We will use a multi-faceted testing approach, encompassing functional and non-functional testing.

10.1 Functional Testing

Functional testing is vital for our quality assurance strategy. It focuses on verifying that all functions and features of our software perform in accordance with the specified requirements.

10.1.1 Testing the Game Implementation

Throughout the game environment's development process, we will extensively employ unit testing as a fundamental quality assurance measure. These unit tests will be comprehensive, checking each software component on their intended functionality. Additionally, they will test whether further development has inadvertently influenced the core game system in a way that allows the rule set and mechanics of the board game to be broken. We will achieve this by running the unit testing suite each time we incorporate changes into the main branch of the application. In essence, these unit tests will serve as a safeguard to guarantee the game's consistency and adherence to the intended design.

10.1.2 Testing the Interface

Integration tests will be created with the purpose of evaluating whether the UI functions as intended and affects the game state correctly. Integration testing will be mainly carried out during the development of the game environment in conjunction with the interface. Additionally, further integration testing will be employed before any playtesting takes place, as it is important that the playtesters have access to a completely accurate and fully functional version of the game.

10.1.3 Testing the DDA

The most important quantitative feature the DDA needs to achieve is consistency in adjusting its difficulty/skill level. To ensure this functionality, the testing will primarily require the DDA AI agent playing the game against other AI agents with fixed difficulty. The tests will be satisfactory if the DDA AI agent successfully achieves a win rate provided prior to the simulation when playing against opponent of any skill level.

To achieve this, we will create multiple AI agents with varying levels of expertise, spanning from beginner to more advanced. We will automate the testing process by writing a script that, upon execution, simulates hundreds of games where the DDA AI agent plays against AI agents set at fixed difficulty levels.

Concurrently with these simulations, we will also collect various valuable analytics. For instance, by collecting data about all the moves the DDA AI makes, we can gain insights into its decision making within the board game, providing us with valuable information for further refining the design of the DDA AI.

10.2 Non-functional Testing

In addition to functional testing, non-functional testing is equally significant. This type of testing evaluates aspects such as performance, usability and enjoyment.

10.2.1 Performance Testing

Performance testing is indispensable as it can provide useful insights such as the capacity to simulate a specific number of games in a given time frame. Based on the assessed efficiency, this information can guide the decision-making process, either leading to optimisations or, conversely, directing developing time toward other areas.

Another important aspect to test is turn time efficiency in terms of DDA AI. Testing this aspect can be easily integrated in the general performance testing as it inherently involves simulating games. Useful metrics can be derived from the game data such as average turn time. Additionally it is essential to analyze how rapidly the turn time increases in correlation with the placement of more pieces on the game board or the accumulation of additional cards in the AI's hand.

10.2.2 Playtesting

Multiple human players will be granted access to the game environment equipped with the DDA AI agent. Each participant will engage in multiple gameplay sessions against the DDA AI several times and provide feedback on several aspects, such as UI's intuitiveness and clarity, AI's resemblance to a human player, and users' overall enjoyment of the experience.

The main purpose of playtesting will be to analyse the feedback provided by human players which will be collected via a feedback form. By carefully considering and incorporating this feedback, we aim to ensure that the game offers an optimal and enjoyable

experience to players.

10.3 Testing Framework

We have not yet made choices regarding which testing tools and frameworks will be employed for each testing category. Therefore we will prioritise looking into these in the early stages of the project, ensuring that unit tests can promptly implemented and utilised as part of the development process.

11 Risk

Within this section, we acknowledge and address the major risks linked to our project. Our goals are first, to mitigate these risks proactively, and second, to establish clear and effective strategies for managing them should they take place. By taking a proactive approach to risk management, we aim to reinforce our project's resilience and ensure that we are well-prepared to deal with any obstacles that may arise during development.

11.1 Risk Assessment

Table 3: Risk Assessment Table.

N°	Risk Overview	Likelihood	Impact	Risk Prevention	Risk Response Plan	Residual Risk
1	Ticket to Ride is too complex	Mid	High	Allow different maps to be loaded into the game so one of a suitable size can be used.	Switch to a game with lower complexity.	Mid
2	Development of AI does not finish in time for user testing	Mid	Mid	Start developing the AI only after having researched the topic thoroughly and hence acquiring a good understanding of the technical details so that development time would be utilised efficiently.	Dedicate an extra team member to work on AI. Remove or amend lower importance requirements so that more time can be spent on AI development.	Low
3	DDA is effective but does not increase player enjoyment	Mid	Mid	Start user testing early, ideally as soon as the AI can be integrated into the game environment. Define quantifiable metrics for "enjoyment", incorporate them into the user survey.	Focus on specific target metrics and identify how each metric could be improved.	Low
4	Loss of a team member	Low	High	Pair programming/research to ensure knowledge is not lost.	Spend more time on developing the AI at the expense of UI.	Mid
5	Legal issues with board game	Low	High	Good understanding of proprietorship of the board game, maintain conditions of Fair Use.	Resolve by contacting board game developers/owners.	Low

12 Project Schedule

As specified in the Project Management section, we use a Gantt chart as a necessary tool to guide work at a high level to conform to rigid assessment deadlines. The Gantt chart is also important to ensure major milestones are reached in time for the next stage in the project. Our Gantt chart is high-level and focused mainly on project epics; it aims to not tie developers to an immutable schedule but act as a guideline to compare progress against.

As shown in the Gantt chart, the project will start off with dual-track development, with the team focusing on building the game environment and conduct research on the AI agent simultaneously. This parallelisation of tasks will minimise time potentially spent trying to continuously integrating the AI and the game together. Once both systems are sufficiently mature, the team can start working together on combining the two subsystems into the final game.

Holidays are accounted for via two-week gaps in the schedule. We believe it is important for group members to have opportunities to take a break from the project, in order to prevent overworking and burnout. Furthermore, other coursework and revision often also need to take place during holidays, which will take up productive hours in these periods.

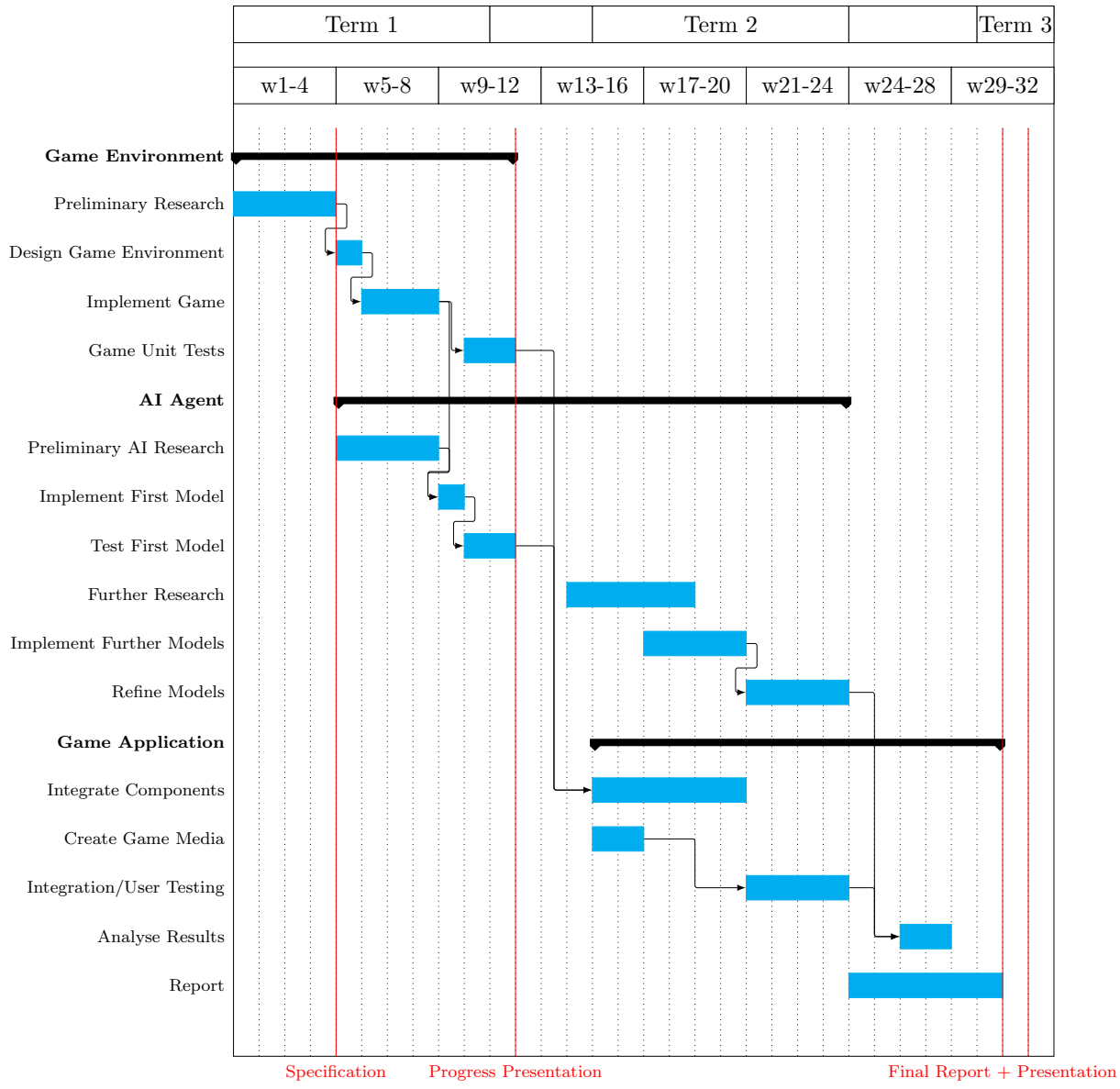


Figure 4: Gantt chart with weekly breakdown of project progression plan.

References

[1] M. P. Silva, V. do Nascimento Silva, and L. Chaimowicz, "Dynamic Difficulty Adjustment through an Adaptive AI," in *2015 14th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*, ISSN: 2159-6662, Nov. 2015, pp. 173–182. DOI: 10.1109/SBGames.2015.16. [Online]. Available: https://ieeexplore.ieee.org/document/7785854 (visited on 10/20/2023).

[2] S. Demediuk, M. Tamassia, W. L. Raffe, F. Zambetta, X. Li, and F. Mueller, "Monte Carlo tree search based algorithms for dynamic difficulty adjustment," in *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, ISSN: 2325-4289, Aug. 2017, pp. 53–59. DOI: 10.1109/CIG.2017.8080415. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8080415 (visited on 10/20/2023).

[3] Y. Hao, S. He, J. Wang, X. Liu, J. Yang, and W. Huang, "Dynamic Difficulty Adjustment of Game AI by MCTS for the game Pac-Man," Aug. 2010, pp. 3918–3922. DOI: 10.1109/ICNC.2010.5584761.

[4] G. Andrade, G. Ramalho, H. Santana, and V. Corruble, "Challenge-sensitive action selection: An application to game balancing," in *IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, Sep. 2005, pp. 194–200. DOI: 10.1109/IAT.2005.52. [Online]. Available: https://ieeexplore.ieee.org/document/1565536 (visited on 10/17/2023).

[5] S. Kennedy, "ChallengeMate: A Self-Adjusting Dynamic Difficulty Chess Computer," en, [Online]. Available: https://web.stanford.edu/class/aa228/reports/2018/final9.pdf (visited on 10/20/2023).

[6] K. Fujita, *AlphaDDA: Strategies for Adjusting the Playing Strength of a Fully Trained AlphaZero System to a Suitable Human Training Partner*, arXiv:2111.06266 [cs], Sep. 2022. DOI: 10.48550/arXiv.2111.06266. [Online]. Available: http://arxiv.org/abs/2111.06266 (visited on 10/22/2023).

[7] P. Gaubil, "Coveted German Spiel des Jahres (Game of the Year) 2004 awarded to Ticket to Ride, the Train Adventure Game.," en,

[8] F. de Mesentier Silva, S. Lee, J. Togelius, and A. Nealen, "Evolving maps and decks for ticket to ride," in *Proceedings of the 13th International Conference on the Foundations of Digital Games*, ser. FDG '18, New York, NY, USA: Association for Computing Machinery, Aug. 2018, pp. 1–7, ISBN: 978-1-4503-6571-0. DOI: 10.1145/3235765.3235813. [Online]. Available: https://doi.org/10.1145/3235765.3235813 (visited on 10/25/2023).

[9] S. Yang, M. Barlow, T. Townsend, *et al.*, "Reinforcement Learning Agents Playing Ticket to Ride–A Complex Imperfect Information Board Game With Delayed Rewards," en, *IEEE Access*, vol. 11, pp. 60 737–60 757, 2023, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2023.3287100. [Online]. Available: https://ieeexplore.ieee.org/document/10154465/ (visited on 10/25/2023).

[10] F. De Mesentier Silva, S. Lee, J. Togelius, and A. Nealen, "AI-based playtesting of contemporary board games," Aug. 2017, pp. 1–10. DOI: 10.1145/3102071.3102105.

[11] L. Strömberg and V. Lind, *Board Game AI Using Reinforcement Learning*, eng. 2022. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:oru:diva-99988 (visited on 10/24/2023).

[12] Atlassian, *What is Scrum? [+ How to Start]*, en. [Online]. Available: https://www.atlassian.com/agile/scrum (visited on 10/25/2023).

[13] *Visual Studio Code - Code Editing. Redefined*, en. [Online]. Available: https://code.visualstudio.com/ (visited on 10/25/2023).

[14] *Welcome to Python.org*, en, Oct. 2023. [Online]. Available: https://www.python.org/ (visited on 10/25/2023).

[15] *SFML*. [Online]. Available: https://www.sfml-dev.org/ (visited on 10/25/2023).

[16] *Unity Real-Time Development Platform — 3D, 2D, VR & AR Engine*, en. [Online]. Available: https://unity.com (visited on 10/25/2023).

[17] G. Engine, *Godot Engine - Free and open source 2D and 3D game engine*, en. [Online]. Available: https://godotengine.org/ (visited on 10/25/2023).