|School of Electronic Engineering and Computer Science

Final Report

Programme of study:
BSc Computer Science

Project Title: Implementation of Catan and Development of an Artificial Intelligence Agent for the Tabletop Games Framework

Supervisor:
Diego Perez-Liebana

Student Name:
Oliver Harrison

Final Year
Undergraduate Project 2020/21

Queen Mary
University of London

Date: 4/5/21

Abstract

This project investigates the suitability of various general AI algorithms for the modern board game Catan. To achieve this, Catan has been modelled in the Tabletop Games Framework, and a rules-based agent has been developed to test the general algorithms. The algorithms explored in this project are Monte-Carlo Tree Search, Random Mutation Hill Climber, and One-Step Look Ahead. Monte-Carlo Tree Search is also optimised using N-Tuple Bandit Evolutionary algorithm, and I explore the development of a Catan specific heuristic to aid in this.

The algorithms perform to varying degrees of success, with Monte-Carlo Tree Search performing the best. Overall while an agent that can reliably win at Catan has not been identified this project contributes to a framework for future research to utilise as well as highlighting several promising avenues of improvement regarding Catan AI.

Acknowledgements

A huge thank you to my supervisor Diego Perez-Liebana, for providing the inspiration and expertise without which this would not have been possible. I also thank everyone I have worked with on the Tabletop Games Framework; you have all been wonderful; special thanks to Martin Balla for his work on Catan and James Goodman for knowing the answer to every question I had.

I also thank Scarlett Molyneaux for listening to me rant about robots and video games for a whole year, and a final thanks to all my family for your constant love and support throughout.

Contents

Table of Figures

Table of Tables

Table of Code Snippets

Chapter 1: **Introduction**

Games of all types have long been an essential test bench for the development of artificial intelligence (AI) [1]. From Turing's 1950's chess-playing algorithm to DeepMind's AlphaStar (a highly successful StarCraft II AI player) [2], the ability for AI to play and beat human players at games has always been an essential component of AI research. This field of study can be divided into two categories; general game playing and specific game playing. General game playing involves the development of AI agents that can effectively play various games and ideally excel in games and environments they have no pre-existing knowledge of. Specific game playing is the development of AI agents that excel in a single environment but cannot play well in other environments. AlphaStar is an example of such an AI agent. To aid in the development of both general and specific AI agents, frameworks and related competitions have emerged to encourage the creation of both games for AI to play and AI algorithms to play them. The Tabletop Games Framework (TAG) is one such framework designed for modelling existing or new tabletop games [3].

The high-level aim of this project is to investigate the performance of known general AI algorithms in the tabletop board game Catan (© 1995 by Kosmos Verlag, Stuttgart); with the aim of contributing to the broader understanding of general AI performance in complex games as well as working towards a proficient Catan AI player. To achieve this, I have worked alongside PhD student Martin Balla to implement a model of Catan in the Tabletop Games Framework. Catan will be both the environment I use to carry out my experimentation and contribute to the catalogue of games TAG already provides to the broader research community.

Catan is a perfect game for continued research into AI, both general and specific, as it is a complex game with a combination of interesting characteristics. Firstly, Catan is designed for 3-4 players in its base rules, officially expanded to 6 in later extensions. This has an exponential impact on the branching factor of any algorithm that uses a tree to simulate opponents' moves. Furthermore, alpha-beta pruning, which is the traditional solution to an unmanageable branching factor, is ineffective games of more than two players [4]. In Catan, players are unable to see complete information about their opponents' cards, a traditional incarnation of hidden information or partial observability. This means players must base their decisions on assumptions about the state and motivations of their opponents. Evaluation of non-deterministic information comes naturally to human players, even if they are not particularly good at it, but reasoning based on guesswork is difficult to replicate in AI. Unlike traditional board games, the order of play in Catan is not a wholly fixed sequence. There are several instances where players must act out of turn to react, and the current stage of the game governs the number of actions a single player can make in a turn. Catan also has a high level of stochasticity (randomness), largely due to how dice rolls driving game progression, further contributing to its non-deterministic nature. Non-determinism is particularly troublesome for AI that utilises forward simulations. Not only might they be incorrectly modelling their opponents' moves but also the entire game state on which they model these moves. As such, good play in Catan requires more than just knowing a winning strategy, but also how to utilise the mechanics of the game to engineer a given position towards this winning strategy. Fortunately for the players, Catan's rules provide a trading system in which they can swap their resources with another player if both agree to the terms. Players will benefit from co-operating with one another and are able to reduce the impact of stochasticity on their strategies, but a competent player must evaluate how much a particular trade will benefit another player versus how much it benefits them; an evaluation only made more challenging by incomplete information. This zero-sum environment is interesting for AI, as an action that might provide a high personal reward could also lead to losing the game, and with incomplete information, an effective AI must know or approximate a strategy for managing this risk. From my own experience playing Catan, the opportunity for co-operation comes with loose coalitions formed between regular trading partners and potential exclusion of players who are deemed as a threat. In turn, fostering an excellent test bench for the evaluation of non-cooperative dialogue, a field in which AI research has shown some progression, and the model of Catan to be implemented could be useful for future work [5]. The last element that makes Catan uniquely interesting is deceptively simple; in most situations doing nothing is a completely valid move and sometimes can be a good long-term move. In most common adversarial games, doing nothing is a bad thing and often can mean the end of the game, such as in chess. This opens a whole avenue for exploring how AI can deal with strategies that involve weighing immediate rewards against potential but non-deterministic future rewards.

In summary, this project seeks to contribute to the platforms available to aid in AI research through the implementation of Catan in the Tabletop Games Framework and contribute new research to the field of general AI algorithms by demonstrating the use of this platform in preliminary experimentation of known algorithms. These contributions will be achieved through the following objectives:

- Identify existing digital implementations of Catan and their shortcomings to influence the design of Catan in TAG.
- Implement a model of Catan in the TAG framework.
- Implement a rules-based agent specific to Catan – this objective was added partway through the project; the motivations behind this decision are explained in Chapter 7.
- Develop a heuristic for state evaluation in Catan.
- Conduct experiments on the performance of the existing general algorithms.
- Explore parameter optimisation for promising general agents.

Chapter 2: **Literature Review**

2.1 General Game Playing Frameworks

A core challenge in the research of AI performance in the role of the player is allowing the AI agents to communicate with the game in question. Simulation of human input involving peripherals like mouses, keyboards, or controllers is an unnecessary extra layer of complexity an AI agent must handle. To resolve this challenge, researchers could work alongside the game's developer to include a special interface designed for AI research, but this would only aid in the research of specific game AI as was the case with AlphaStar [2]. A solution for general AI was the development of platforms that served as environments providing unified interfaces for specific sets of games. The Arcade Learning Environment is such a platform, providing a single interface for research on hundreds of Atari 2600 game environments, and has been used extensively by researchers to explore general AI capabilities [6]. However, this solution is still restricted by the set of games added to the platform by the developers. General game playing frameworks seek to resolve this issue by not providing a set of games (although many have a few initial games included) but instead provide a set of tools that researchers can use to create new or existing games. Such a framework will provide default building blocks as well as enforce a strict set of design principles, ensuring that all games provide a common interface for players. This allows for the development of AI agents who can play any game in framework so long as they know rules and principles of the framework. It also allows for a theoretically limitless number of games to be available to such players within the restrictions of the framework. The remainder of this section covers some existing general game playing frameworks of note.

Game Description Language (GDL) provides a framework to formally define games based on logical rules that AI players can then interpret and play with no prior knowledge [7]. The version of GDL used by Genesereth *et al.* in the AAAI general game playing competition requires discrete games with complete information, for example, Chess or Tic Tac Toe. Further work has been done on expanding this framework to video games [8] and games with partial observability [9].

GVGAI is a general game playing framework utilising a Video Game Description Language (VGDL) inspired by GDL [10]. VGDL differs from GDL in that it describes games by defining game entities and their interactions. The functionality of components must be programmed into the framework, allowing for a wider scope of possible functionality but requiring that individual functionality be implemented before use, unlike GDL, in which you can construct functionality out of logical rules. GVGAI then offers an object-orientated interface for agents to play any game defined by the underlying VGDL.

Ludi is a general game system that differs greatly from most in that its focus is not on the evaluation of AI agents but instead on the generation and evaluation of novel games through AI methods [11]. Ludi can generate combinatorial games by combining and tweaking existing rules defined in its underlying GDL and then evaluates these games in several aspects using AI play. Though not directly related to this project at this time, it poses an interesting view of the future in which general AI can learn optimal strategies for games that have been designed by another AI.

2.2 General Artificial Intelligence Algorithms

There are currently three common approaches to implementing general AI agents in games: tree-search, evolutionary computation, and machine learning (which divides into three further sub-categories) [1]. A general AI algorithm can be considered a set of instructions that will implement one or a combination of these approaches that an AI player should be able to follow to solve a given problem. The following section will explain the tree-search and evolutionary computation algorithms this project seeks to explore as well as contextualise the approaches they implement.

The general concept of tree-search based algorithms is to construct a tree structure in which the root is the state in which the search begins, and each possible action is a branch leading to a new node representing a new state with new possible actions. Theoretically, a tree-search algorithm can build an exhaustive tree in which it explores the outcome of every possible sequence of actions and therefore identify the guaranteed best path [12]. However, in any game complex enough to be worth investigating, this becomes computationally unfeasible. A wide variety of tree-search algorithms have been designed which employ a mix of different tactics to avoid this problem. The One-Step Look Ahead is a tree-search based algorithm that still performs an exhaustive search of all possible actions from the root but does not expand the tree further, instead greedily picking whichever action immediately leads it to the highest valued state. The relative performance of One-Step Look Ahead is based on its given heuristic, which is the function used to evaluate a potential state. A good heuristic will consider as much of the available game state as possible when determining how likely it is to lead to a desired outcome to allow for a very granular ranking of the available actions. The difficulty in designing such a heuristic is very dependent on the nature of the game in question; games that requires forward planning in which you may make numerous actions with no immediate reward are far more difficult to design effective OSLA heuristics for.

Monte-Carlo Tree Search is a tree search algorithm which rose to fame for its role in solving Go. Originally described by Rémi as a combination of a traditional tree-search and the Monte-Carlo method to efficiently approximate a min-max tree without needing min-max's exhaustive exploration [13]. The core loop of MCTS involves a selection phase, in which, by starting at the root node, a formula is used (examples are UCB1, epsilon-greedy, and Bayesian bandits [1]) to traverse down the tree till a node with unexplored children is selected. This node is then expanded through the selection of one possible child node (traditionally selected at random), and from this node, a rollout is performed. A rollout is the Monte-Carlo element of MCTS, in which the algorithm simulates random actions until a terminal game state is reached. The outcome of this rollout is then backpropagated up through the tree from the child node, and depending on its value and current tree structure, this may adjust the path which the next loop of the MCTS algorithm explores. There are now many variations on this core loop, introducing a wide range of variable hyper-parameters to suit different problems. For example, modern games like Catan have potentially indefinite length through random play, while real-time games like video games have strict time budgets to react in. In both these scenarios rollouts are unlikely to reach a terminal state in the required time, so a rollout budget is enforced with a heuristic to value the state the rollout finishes at. Other common hyper-parameters are the policy used in traversing the tree, the maximum tree depth, or the policy used to select the final node. This makes MCTS a candidate for automated optimisation, and in this report, I will explore to what extent the N-Tuple Bandit Evolutionary Algorithm can be used to optimise MCTS for Catan. NTBEA was designed for noisy and expensive optimisation problems, which Catan's stochasticity and branching factor create for any agent optimisation problem. NTBEA has also already proved to significantly outperform the traditional grid search for hyper-parameters in-game agents [14].

Evolutionary algorithms are a broad family of algorithms that utilise evolutionary computation based on the process of biological evolution to solve optimisation problems [15]. From the perspective of games, this typically involves the selection of one or more series of actions representing solutions. These solutions are then valued using a fitness function, much like a heuristic, before a mutation operation is carried out to change a sub-set of the solution. The value of this mutated solution is then compared with the parent, and the fittest series is kept. This is a very general explanation, and there are many variations between implementations. In this project, I will investigate a simple implementation of the Rolling Horizon Evolutionary Algorithm (RHEA) called a Random Mutation Hill Climber (RMHC). RMHC evolves a series of actions of variable length from its current state using a rollout and then evaluates this solution with a heuristic. RMHC then randomly selects a point in its current action sequence and performs another rollout, generating a mutated sequence. RMHC will keep the fitter of the two solutions and then repeat this process until its budget is exhausted, at which point it will perform the first action in its fittest solution.

2.3 Catan implementations

JSettlers is the digital implementation of Catan that was used in many of the studies that informed this project. *JSettlers*, while a very faithful implementation of Catan, was not designed for AI research. Implemented in Java, it uses a server-client model so that users can host games and play with other people over the internet. This makes *JSettlers* very slow when used for the collection of data regarding AI players, as the architecture becomes a bottleneck rather than the speed of the players. Effective research into Catan AI needs a fast platform for progress to be made. *SmartSettlers* is an implementation of Catan by a group of researchers for the purpose of experimenting with an MCTS agent [16]. It was designed to remedy *JSettlers* speed problem and is much faster. *SmartSettlers* is not a fully faithful implementation of Catan, with no hidden information, and the player agent is unable to trade, nor is it publicly available therefore could not be used as the basis for my research. Neither of these platforms or any other existing digital implementations I could identify provide a Catan environment in which general AI players can also be tested across a variety of games without needing specific interfaces.

2.4 Artificial Intelligence in Catan

There has already been some research into the performance of AI in Catan, as well as research into games with similar characteristics. MCTS has shown promise by beating the popular *JSettlers* AI, which uses hand-crafted rules on the *SmartSettlers* platform [16]. By injecting Catan specific domain knowledge into the weighting of actions, an MCTS agent was able to match or beat *JSettlers* depending on its hyper-parameters. However, there are some limitations to this research. *SmartSettlers* deviations have already been outlined, only 100 games were played with each agent, and the budgets of the agents in question were defined by MCTS iterations per move, either 1000 or 10000. One hundred games is a relatively small sample size for a game with as much stochasticity as Catan, and the budgets given to the MCTS agents are very large and fail to give a notion of how long the agents actually take to perform a move; an important part of performing at human level is also performing at human speed. Therefore, though promising, the results from *Szita et al.* are not conclusive of good performance from a general MCTS agent. Learning-based agents have shown that they can learn better strategies with regards to trading in Catan, but this study suffers from a similar issue sample size of games due to using *JSettlers* [5]. For general Catan strategy, learning algorithms have shown to be effective in improving agents play, and with the use of a heuristic-based high-level strategy policy (to manage mid/long-term objectives), agents implementing learned behaviours for low-level strategies were able to beat competent human players in a small sample [17]. There has been experimentation with several variants of MCTS in a model of Catan defined in GDL as part of a Bachelor's thesis that may be relevant in informing the optimisation of MCTS later in this project [18].

2.5 Summary

Research into general game playing is an important field for the development and improvement of AI methods. So far, there has been little conclusive research on general game playing in games that mirror modern board games with complex features like partial observability and non-determinism. Catan is the archetypal modern board game, providing an excellent point to expand research into the general playing of modern board games. Furthermore, research into Catan AI is also ongoing, with no known ideal solutions. This chapter has identified ongoing research worth contributing to on general AI in Catan and the need for a platform to support it.

Chapter 3: **Catan**

In this chapter I will give a brief introduction to Catan's base [19] rules for 4 players to contextualise the decisions made in implementation.

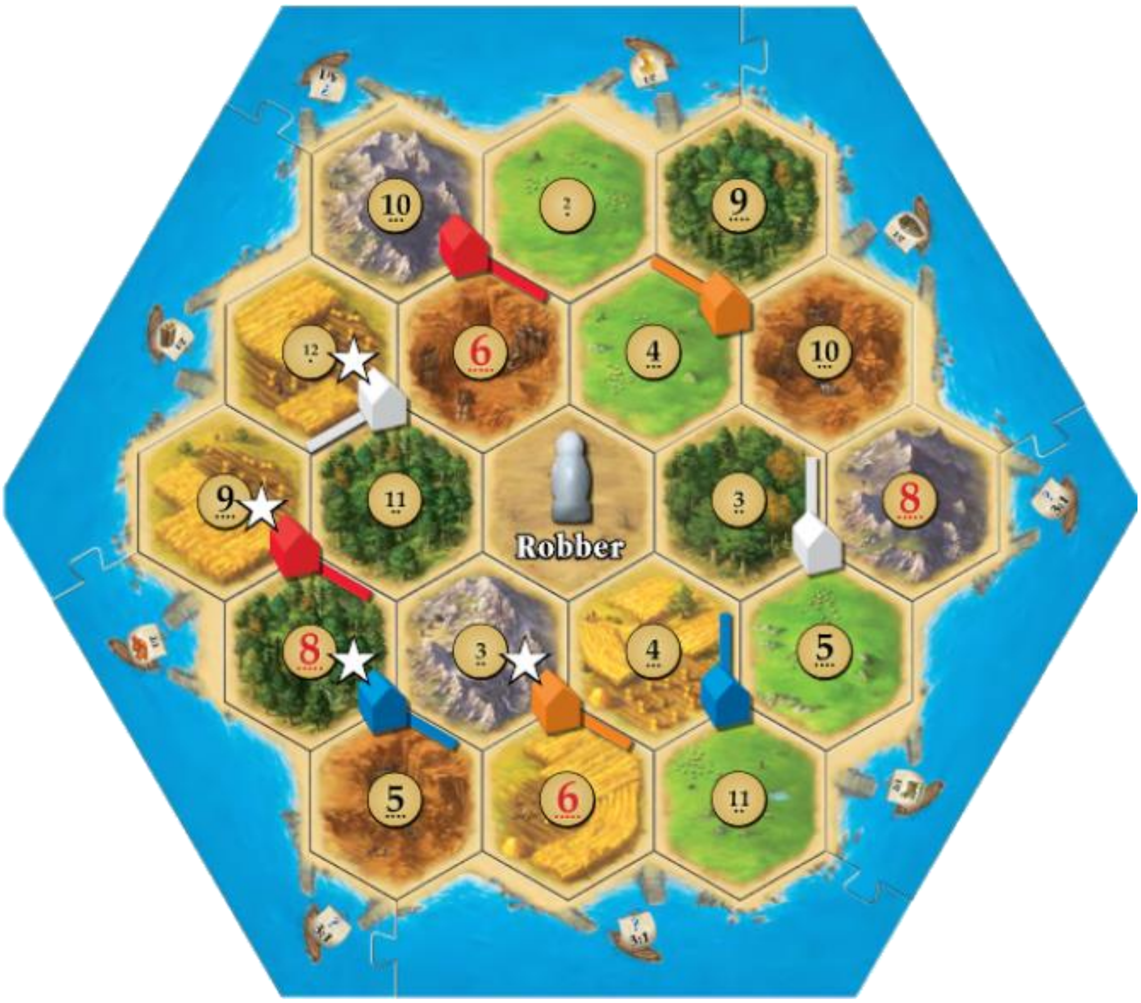


Figure 1: Example Catan Board

The Catan board (*Figure 1*) is constructed out of 19 terrain hexes (tiles) which all represent a specific resource from brick, grain, lumber, ore and wool; there is also a single desert tile. The board is surrounded by sea, with nine harbours attached to vertices of the hexes. Each resource tile has a number token placed on it, which correlates to a possible dice roll. The layout of the board is not fixed. The tiles can be arranged in any manner as long as it forms a hexagon surrounded by sea.

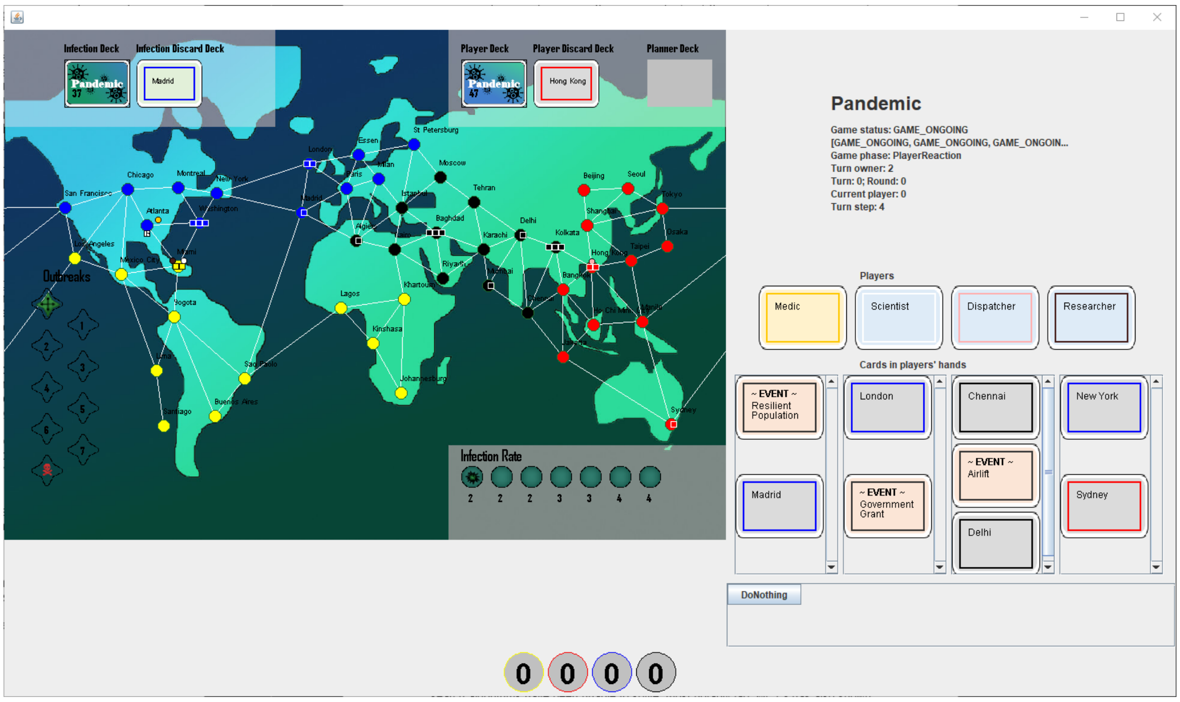
After the board has been constructed, players take turns placing a settlement on a vertex of the tiles, as well as an attached road that runs along the edge of the hex until each player has two settlements. Settlements cannot be placed on vertices directly adjacent to a vertex where a settlement already exists; this rule holds for the entire game. Once each player has placed their settlements, players choose which of the two settlements will provide their starting resources. The players receive a single resource card of the type that their chosen settlement touches, so up to 3 different resources.

Play can now begin. Catan is sequentially turn-based, with no way to miss or lose turns. Each player first rolls two dice, the sum of which correlates to the number tokens placed on the resource tiles. Any resource tile with the rolled sum produces resources for that turn. If any player (not just the rolling player) has a settlement or city on a vertex of a producing tile, they receive 1 (for a settlement) or 2 (for a city) resource cards of the type of said tile. If the value of the dice totals to 7, a special event chain occurs called the robber. First, any player with more than seven cards in their hands must choose half of their current card count rounded down to discard to the bank; which helps prevent hoarding of finite resources. The player who rolled the seven is now allowed to move the robber piece, which begins on the desert (refer to figure N). The robber piece must be moved to another tile (the rolling player cannot elect to leave it where it is); any tile it is placed on is blocked from producing resources if the number on it is rolled. The rolling player is also allowed to steal one resource card at random from any single player who has a settlement on a vertex of the tile where the robber has been placed. The act of moving the robber can also be triggered by playing a knight development card (covered below), but this does not trigger the discard event. After the dice roll is resolved, the rolling player now has the option to buy, trade or use a single development card. The player may spend resources to build new settlements, roads and cities, or purchase a development card. All of these actions have different resource requirements, as well as certain conditional requirements. For example, cities can only be built by upgrading an existing settlement; these costs and conditions are covered in more depth in the rules almanac. The player can also negotiate a trade of resource cards with any other player in the game; there are no rules governing these trades other than that they must involve the current turn owner. For example, the current player can facilitate trade between two other players by acting as a proxy, but the two players whose turn it is not cannot trade directly. The player can also trade 4 of any resource for 1 of another type with the bank. Ownership of the harbours provides special rates for trades with the banks, and these rates are either 3:1 any resources or 2:1 for a specific resource offer. At any point during their turn, the player can play a single development card if it was not purchased on the current turn. These cards provide a variety of immediate bonuses that can be used to help the player towards another goal and can also affect the other players in the game. Once the current player is ready, they can end their turn, passing it to the next player. This gameplay loop continues until one player reaches the victory condition.

Victory in Catan is achieved through a points-based system, and a player must reach ten victory points to win the game. There are several ways to earn victory points: settlements and cities are worth one and two victory points, respectively, certain development cards are worth one point and are hidden by the owning player until they reach ten points. The first player to build a continuous road at least five roads long receives two points for the longest road. Any player completing a longer road steals these two points from the current longest road holder. Finally, the first player to play three knight development cards receives two points for the largest army, any player that plays more knight development cards than the current holder steals these two points.

Chapter 4: **Tabletop Games Framework**

TAG is a Java-based general game playing framework designed around several key concepts that make it ideal for this project [3]. TAG offers a common skeleton for structuring games designed to simplify development and abstract responsibility to the underlying framework, which is then accessed through a common API for AI agents providing a common interface for all games. It encourages the development of reusable object-oriented components for the representation of game objects, with many common components available for use out of the box or further extension. It includes built-in support for partial observability, forward modelling, various turn order styles, and variable player counts, all important components required for this project. Finally, it offers researchers a useful suite of tools, including game analysis reports and automated parameter optimisation. There are several games already implemented within TAG. One of these is *Pandemic (Figure 2),* a cooperative board game that shares the features of partial observability, stochasticity, and a reactive turn-order with Catan.



Pandemic

Game status: GAME_ONGOING
[GAME_ONGOING, GAME_ONGOING, GAME_ONGOIN...
Game phase: PlayerReaction
Turn owner: 2
Turn: 0; Round: 0
Current player: 0
Turn step: 4

Figure 2: Pandemic's GUI representation from TAG

Every game in TAG has several core classes it must implement as well as many optional classes and interfaces it can utilise. Each game must implement classes that extend the *AbstractForwardModel, AbstractGameState, TurnOrder,* and *AbstractParameters* classes provided by the framework.

The *ForwardModel* class contains the core game logic for a specific game. It is responsible for initialising the game state, action generation, and the action execution loop. *GameState* is responsible for any stateful information relating to a game and managing access to it. *TurnOrder* handles tracking whose turn it is and handing the turn over to the correct person. TAG provides several different variations of *TurnOrder* for different scenarios. The final of the core classes is *Parameters,* which class contains customisable rules for the game in question.

Games can also define components; these represent real game space objects or conceptual components for grouping or managing other components. TAG provides general components like dice, cards, and boards that can be utilised out the box by games and provides patterns and interfaces for implementing custom components. These components track state and offer functionality that would mirror how these components would act in real life. *GameState* exposes access to these components, as components can be hidden from players in game with partial observability; it is *GameState* that manages the protection and distribution of hidden information.

In TAG, AI agents interact with the game through actions. Actions extend the *AbstractAction* class, which requires a constructor that sets up the initial state with any information the action needs to be aware of, and an *execute* method that carries out the effect on the action on the game state if it is selected by a player.

TAG also offers several already implemented AI players as well as a simple pattern for implementing new players. *Code Snippet 1* is the Java class for a random player in the TAG framework, a succinct example of the basic pattern for implementing players. All that is required for implementing a player is the *getAction* method responsible for selecting which action to take.

```java
public class RandomPlayer extends AbstractPlayer {
    private final Random rnd;

    public RandomPlayer(Random rnd) {
        this.rnd = rnd;
    }

    public RandomPlayer()
    {
        this(new Random());
    }

    @Override
    public AbstractAction getAction(AbstractGameState observation,
        List<AbstractAction> actions) {
        int randomAction = rnd.nextInt(actions.size());
        return actions.get(randomAction);
    }

    @Override
    public String toString() {
        return "Random";
    }
}
```

Code Snippet 1: RandomPlayer as an example of the player design pattern in TAG

TAG also allows for the design of heuristics for specific games or to be used by a single player by adding an instance of the heuristic to the constructor of said player. TAG also offers the *TunableParamaters* abstract class, which provides a pattern for the design of both heuristic and player parameters that can be optimised. Parameters can also be loaded into the framework through JSON files, allowing for storing and editing of parameter profiles.

The existing agent algorithms implemented in TAG at the time of writing are the OSLA, MCTS, and RMHC algorithms. All these algorithms follow the pattern defined in *Code Snippet 1* with complex action selection methods.

Chapter 5: **Implementation**

The following section is not an exhaustive description of the codebase; please refer to the supporting evidence for the full codebase.

Because project has been implemented as part of the larger framework of TAG, it is stored in the same repository. The work on *core.games.catan* was shared between myself and another researcher, Martin Balla. See *Appendix B* for a full description of my contributions to the project, while all work on the *CatanRuleBasedPlayer* was solely my own. All other code is provided by TAG.

There has been a single intentional major deviation from the Catan rules in this implementation. Trading has been simplified into a system allowing players to offer any quantity of a single type of resource for any quantity of a different type of resource. The players can then negotiate these quantities a set number of times defined in *CatanParameters* before the trade is automatically declined. Customisable limits have also been introduced for the number of actions per turn stage and the number of rounds to prevent the game from getting stuck in loops but can also be changed through *CatanParameters* along with several other rules.

5.1 Catan Implementation

The *CatanTurnOrder* class is responsible for controlling the flow of the game and ensuring that players are acting and reacting at appropriate times; as such, I am covering it first as it impacts most other components of the game. *CatanTurnOrder* is an extension of the *ReactiveTurnOrder* class, which is, in turn, an extension of TAG's base abstract *TurnOrder* class. This provides *CatanTurnOrder* with the ability to allow players to act out of the order defined at the beginning of the game to respond to events. This is vital for Catan as actions like trading or development cards trigger events that can require reactions from players who are not the current turn owner. This is implemented through a queue of id codes; if an event triggers a reaction from one or more players, their id codes are added to the *reactivePlayers* queue; the underlying *ReactiveTurnOrder* class ensures the players in this queue resolve their actions before returning to normal play. *CatanTurnOrder* also utilises the *IGamePhase* interface through the enumerator *CatanGamePhase*, shown in *Code Snippet 2.*

```java
public enum CatanGamePhase implements IGamePhase {
    Setup,
    Trade,
    Build,
    Robber,
    Discard,
    Steal,
    TradeReaction,
    PlaceRoad,
}
```

Code Snippet 2: Catan's game phases

The actions available to a player of Catan are determined by both the game state and what phase of play the current game is in. A single player's turn can consist of multiple phases with different action choices, such as the Trade phase, where the current player can initiate trades, followed by the Build phase, where the current player can purchase game objects with their resources. The *CatanGamePhase enum* is used to track which phase of play the game is in, and *CatanTurnOrder* is responsible for ensuring that the game state transitions between these phases in the proper order. *Code Snippet 3* is a selection statement demonstrating how *CatanTurnOrder* transitions from the *Main* play phase, a default phase provided as part of the *DefaultGamePhase enum* that *CatanGamePhase* extends, to either the *Trade* or *Robber* phase according to Catan's rules.

```
IGamePhase gamePhase = gameState.getGamePhase();
if (gamePhase.equals(AbstractGameState.DefaultGamePhase.Main)){
    if (((CatanGameState)gameState).getRollValue() == 7){
      gameState.setGamePhase(CatanGameState.CatanGamePhase.Robber);
    } else {
        gameState.setGamePhase(CatanGameState.CatanGamePhase.Trade);
    }
}
```

Code Snippet 3: Transitioning between game phases in Catan

CatanGameState extends *AbstractGameState* provided by TAG and is responsible for tracking all state information for a given game. *Code Snippet 4* is the class attributes for *CatanGameState,* representing all information that must be managed over a game of Catan.

```
protected CatanTile[][] board;
protected Graph<Settlement, Road> catanGraph;
protected Card boughtDevCard;
protected int scores[];
protected int victoryPoints[];
protected int knights[];
protected int exchangeRates[][];
protected int largestArmy = -1;
protected int longestRoad = -1;
protected int longestRoadLength = 0;
protected OfferPlayerTrade currentTradeOffer = null;
int rollValue;
```

Code Snippet 4: Catan's game state

CatanGameState is also responsible for exposing access to game components to the other classes in the game and as such contains many getter and setter style methods and helper methods for retrieving, checking or manipulating state information. For example, *CatanGameState*'s *swapResources* method takes two player’s id codes and an array representing trade values and uses these to swap the actual cards in the decks of these players in instances of trade. Every class in TAG implements a *_copy* method to allow for players to perform forward modelling without impacting the actual game state. *Code Snippet 5* is *CatanGameState*'s *_copy* function, provided as an example of a deep copy for a complex object; all classes follow a similar pattern for their copy methods but vary in complexity based on the underlying state for each class.

```
protected AbstractGameState _copy(int playerId) {
    CatanGameState copy = new CatanGameState(getGameParameters(),
        getNPlayers());
    copy.gamePhase = gamePhase;
    copy.board = copyBoard();
    copy.catanGraph = catanGraph.copy();
    copy.areas = copyAreas();
    copy.gameStatus = gameStatus;
    copy.playerResults = playerResults.clone();
    copy.scores = scores.clone();
    copy.knights = knights.clone();
    copy.exchangeRates = new
        int[getNPlayers()][CatanParameters.Resources.values().length];
    for (int i = 0; i < exchangeRates.length; i++){
        copy.exchangeRates[i] = exchangeRates[i].clone();
    }
    copy.victoryPoints = victoryPoints.clone();
    copy.longestRoadLength = longestRoad;
    copy.rollValue = rollValue;
    if (currentTradeOffer == null){
        copy.currentTradeOffer = null;
    } else {
        copy.currentTradeOffer = (OfferPlayerTrade)
                this.currentTradeOffer.copy();
    }
    return copy;
}
```

Code Snippet 5: CatanGameState's _copy function as an example of a complex deep copy

CatanGameState's final responsibility is exposing a basic heuristic function that can be utilised by players, shown in *Code Snippet 6* I elected for a heuristic bound between -1 and 1 representing a loss or win, respectively, and each score point moves the heuristic proportionally closer to 1.

```
protected double _getHeuristicScore(int playerId) {
    if (getPlayerResults()[playerId] == Utils.GameResult.LOSE)
        return -1.0;
    if (getPlayerResults()[playerId] == Utils.GameResult.WIN)
        return 1.0;
    return ((double)((scores[playerId])+(victoryPoints[playerId])))
        /((CatanParameters)gameParameters).points_to_win;
}
```

Code Snippet 6: Catan's default heuristic

The core game logic for advancing play is in the *CatanForwardModel* class, an extension of the *AbstractForwardModel* class. The first method of note is the _*setup* method, responsible for the initialisation of the games initial state and the components contained within.

```
for (int i = 0; i < state.getNPlayers(); i++) {
    Area playerArea = new Area(i, "Player Area");
    Deck<Card> playerHand = new Deck<>("Player Resource Deck",
        CoreConstants.VisibilityMode.VISIBLE_TO_OWNER);
    playerHand.setOwnerId(i);
    playerArea.putComponent(playerHandHash, playerHand);

    Deck<Card> playerDevDeck = new Deck<>("Player Development Deck",
        CoreConstants.VisibilityMode.VISIBLE_TO_OWNER);
    playerDevDeck.setOwnerId(i);
    playerArea.putComponent(developmentDeckHash, playerDevDeck);
    playerArea.putComponent(roadCounterHash, new Counter(0, 0,
        params.n_roads, "Road Counter"));
    playerArea.putComponent(settlementCounterHash, new Counter(0, 0,
        params.n_settlements, "Settlement Counter"));
    playerArea.putComponent(cityCounterHash, new Counter(0, 0,
        params.n_cities, "City Counter"));

    state.areas.put(i, playerArea);

}
```

Code Snippet 7: Example of setup performed by CatanForwardModel

Code Snippet 7 is an example of this initialisation. This code is responsible for the initialisation of the *Areas* for each player in the game. These *Areas* are components which group together other game objects to manage partial observability and ownership, such as their individual decks which are also initialised in this loop. This method is called once at the start of every new game and initialises several other parts of game state in the manner outlined in *Code Snippet 8.* Another key method of the *CatanForwardModel* is the *_next* method, this method has several responsibilities. Firstly, *_next* executes the action selected by the player, demonstrated in *Code Snippet 8.* The effects of the action on the games state are implemented in the action itself, such that the forward model's role in the process is quite simple.

```
if (action != null){
    action.execute(gs);
} else {
    throw new AssertionError("No action selected by current  player");
}
```

Code Snippet 8: Catan's action execution

The forward model then evaluates the impact of the action on the player's scores, updating the game state if necessary and checking if any player has reached the win condition. The forward model then decides whether to end the game if the win or max game length conditions have been met or delegate to *CatanTurnOrder* to identify which game phase and whose turn it is now. If the game is not over, the forward model then simulates the dice roll and handles the outcome before returning to the core game loop defined in TAG's *Game* class. The *CatanForwardModel* also contains the *_computeAvailableActions* function, which contains a series of selection statements that decide which function of the *CatanActionFactory* is called to generate the available actions based on the current game phase. It is TAG's core *Game* loop that calls *_computeAvailableActions* by calling *computeAvailableActions. comuputeAvailabeActions* is defined in the superclass *AbstractForwardModel* and makes a call to the specific underlying implementation of *_computeAvailableActions*. Polymorphism like this is what enables TAG's general game playing, allowing game unique underlying implementations to be accessed from generic superclasses.

The *CatanActionFactory* is not a default class provided by TAG but is a useful abstraction pattern to prevent cluttering the forward model in games with complex logic for generating actions. Another game utilising an action factory within the framework is *Pandemic (Figure 2)*. The *CatanActionFactory* exposes several static functions that contain the logic for generating sets of possible actions based on the current game state. All of *CatanActionFactory's* functions share a common pattern in which each static function refers to a given game phase and contains one or more nested functions that return lists of a specific type of action to be added to the total list of possible actions. Then each nested function consists of a series of loops that produce individual unique instances of possible actions, with the main difference between each function being what the loops are iterating over. As such, I will only cover one example in this report, the function which contains the most complex action generation logic in Catan, *getDiscardActions.*

getDiscardActions is called by *CatanForwardModel* during the *Discard* game phase. *Code Snippet 9* shows the first scenario for discard actions, in which the player has not surpassed the discard limit (nominally 7 in Catan but customisable through parameters in our implementation), and as such, only receives a *DoNothing* action.

```
int deckSize = playerResourceDeck.getSize();
if (deckSize <= ((CatanParameters)gs.getGameParameters())
        .max_cards_without_discard)
{
    actions.add(new DoNothing());
    return actions;
}
```

Code Snippet 9: Action generation for Discard phase if player has less than max cards

If the player has more than the max number of cards, then the action factory must now generate all possible discard combinations, without producing combinations that are functionally identical. For example, the action to discard the cards *{Wool, Wool, Lumber, Lumber}* is functionally identical to discarding {*Lumber, Lumber, Wool, Wool}*, so even though both are valid combinations the *CatanActionFactory* is not allowed to generate and return both as this will break the hash tables TAG uses for comparing objects. To solve this issue, *CatanActionFactory* does not try to generate combinations of cards but instead generates combinations of quantities of each owned resource that summate to the quantity that must be discarded. This prevents duplication of actions and greatly cuts down on the time and memory required to generate all possible combinations. Refer to *Appendix C* for the algorithm that handles this computation, in which each value in the array *resources*, which contains integer representations of the player's owned quantity of each resource (*Brick, Lumber, Ore, Grain, Wool),* is iterated over in a decreasing fashion. In each loop, if the quantity to discard is equal to the quantity required, then this combination is added to the list of possible combinations. Then if the quantity to discard is larger than or equal to the quantity required, then the current loop continues to the next iteration without going deeper as an efficiency gain. This is because each loop decrements, so if the total quantity is already higher than or equal to the required quantity, then the quantity of the current resource being iterated over needs to decrement further before looking at the next resource. Otherwise, the total quantity available to discard at his layer must be lower than the quantity required to discard, so the next resource loop layer is entered. This process repeats for all five resources in a nested fashion, providing an efficient method for generating combinations of resources and avoiding JVM memory issues that come from combinations problems with objects. Finally, all combinations are iterated over, and *DiscardCards* actions are generated for each combination by creating an array of *Card* objects based on the values in the combination. The complete list is then returned up to the core *Game* loop to be passed to the player.

The final element that handles core game functionality is the individual game actions for Catan. There are currently fifteen unique actions in Catan that all fulfil the general pattern outlined in the previous TAG section. To be concise, I will give a general overview of how the actions function combined with specific examples from a single action. All Catan actions extend the *AbstractAction* class provided by TAG, which demands a series of functions be implemented. Several actions in Catan have attributes, but I have designed all actions state to be immutable as this is preferable for performance and simplicity.

The example action I will use is *AcceptTrade*. This action represents a player accepting a trade that another player has proposed to them through the related action *OfferPlayerTrade*. *Code Snippet 10* is *AcceptTrade's* attributes and constructor. The two integer values represent the id codes of the involved players, while the integer arrays are representations of the quantity of each resource type being traded by each player. As mentioned, all attributes are final, meaning immutable.

```
public final int offeringPlayer;
public final int receivingPlayer;
public final int[] resourcesRequested;
public final int[] resourcesOffered;

public AcceptTrade(int offeringPlayer, int receivingPlayer,
int[] resourcesRequested, int[] resourcesOffered) {
    this.offeringPlayer = offeringPlayer;
    this.receivingPlayer = receivingPlayer;
    this.resourcesRequested = resourcesRequested;
    this.resourcesOffered = resourcesOffered;
}
```

Code Snippet 10: AcceptTrade's immutable state and constructor

The first method every action must override from *AbstractAction* is the execute method. This method is responsible for carrying out the effects of the action on the game state if it was chosen by a player. *Code Snippet 11* shows this method for *AcceptTrade*, in which the *swapResources* method is used to trade the resources that *AcceptTrade* stored as state between the two players. All actions and much of the core logic makes use of *AssertionError*. This is because the generic nature of the framework combined with allowing the development of players creates a high risk of an error resulting from actions being used at inappropriate times, null pointers, or incorrect types. I adopted a philosophy of *'fail-fast'* to aid in testing as many of these errors will otherwise not cause the program to crash outright but instead just cause strange and unintended behaviour that can be difficult to diagnose.

```
public boolean execute(AbstractGameState gs) {
    if(CatanGameState.swapResources((CatanGameState) gs, receivingPlayer,
        offeringPlayer, resourcesRequested, resourcesOffered)){
        return true;
    } else {
        throw new AssertionError("A partner did not have sufficient
                resources");
    }
}
```

Code Snippet 11: AcceptTrade's execution method

The next overridable method from *AbstractAction* is the *copy* method required of every class. I have intentionally made the state of all actions in Catan immutable so that the copy functions for all actions can return pointers to themselves, as there is no way for the state to change if it is immutable. The next two methods required by *AbstractAction* is the *equals* and *hashCode*. *equals* provides a facility to check if two actions are equal by comparing if each attribute of the action is equal. *hashCode* is responsible for generating a hash for the given action; like *equals,* two functionally identical actions should generate two identical hashes. *AcceptTrade's* implementation of these methods is given in *Code Snippet 12.*

```
public AbstractAction copy() {
    return this;
}
public boolean equals(Object obj) {
    if (this == obj) return true;
    if (obj instanceof AcceptTrade){
        AcceptTrade otherAction = (AcceptTrade)obj;
        return otherAction.offeringPlayer==offeringPlayer
                && otherAction.receivingPlayer == receivingPlayer
                && Arrays.equals(otherAction.resourcesRequested,
                    resourcesRequested)
                && Arrays.equals(otherAction.resourcesOffered,
                    resourcesOffered);
    }
    return false;
}
public int hashCode() {
    int retValue = Objects.hash(offeringPlayer,receivingPlayer);
    return retValue + 41 * Arrays.hashCode(resourcesOffered) + 163
        * Arrays.hashCode(resourcesRequested);
}
```

Code Snippet 12: AcceptTrade's copy, equals, and hashCode methods, which are required by every class in TAG

The final functional elements of Catan are its components. Catan predominantly uses default components provided by TAG with a few custom components. *Road* and *Settlement* represent their in-game counterparts; they are responsible for managing the state regarding ownership and id codes. A *Graph* is another custom component comprised of *Edge* components in a standard node graph design. They are used to represent the layout of the board during computation. The final part of Catan, while not strictly a component, is the *CatanTile. CatanTile* represents the individual tiles of a Catan board. Each tile utilises an x and y coordinate for positioning and is responsible for managing state information for roads, settlements and harbours attached to it. The board space is constructed from a *2D* array comprised of these tiles.

Martin Balla has developed a GUI, shown in *Figure* 3. I did not use this GUI in my research, and as such, I will not cover how it functions in this report. Refer to the supporting evidence for its implementation.

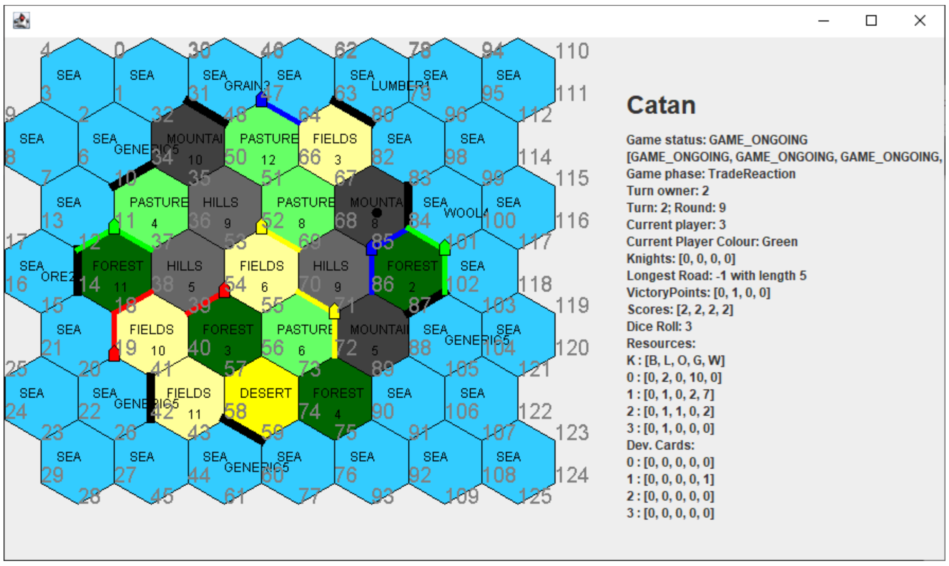

```
Catan
Game status: GAME_ONGOING
[GAME_ONGOING, GAME_ONGOING, GAME_ONGOING,
Game phase: TradeReaction
Turn owner: 2
Turn: 2; Round: 9
Current player: 3
Current Player Colour: Green
Knights: [0, 0, 0, 0]
Longest Road: -1 with length 5
VictoryPoints: [0, 1, 0, 0]
Scores: [2, 2, 2, 2]
Dice Roll: 3
Resources:
K : [B, L, O, G, W]
0 : [0, 2, 0, 10, 0]
1 : [0, 1, 0, 2, 7]
2 : [0, 1, 1, 0, 2]
3 : [0, 1, 0, 0, 0]
Dev. Cards:
0 : [0, 0, 0, 0, 0]
1 : [0, 0, 0, 0, 1]
2 : [0, 0, 0, 0, 0]
3 : [0, 0, 0, 0, 0]
```

Figure 3: Catan's GUI in TAG

5.2 AI and Heuristic Implementations

The *CatanRuleBasedPlayer* is a hand-coded AI player extending the *AbstractPlayer* framework; I cover the motivations for developing this agent in Chapter 7. At its core, the *CatanRuleBasedPlayer* uses switch statements to branch based on the types of action and game phase, with each branch then containing logic for evaluating each individual action of a given type. This evaluation ranks actions into a priority list based on a set of rules regarding that type of action. The agent then randomly selects an action from its highest priority list. *Code Snippet 13* demonstrates this core loop, with the outer switch for game phase and then a fragment of the inner switch for *Build* phase action types.

```
switch (gamePhase){
        […]
    case Build:
        for (AbstractAction action : possibleActions) {
            actionType =
               ActionType.valueOf(action.getClass().getSimpleName());
            tempResources = currentResources.clone();
            switch (actionType){
                case PlayKnightCard:
                    if(KnightCardCheck(cgs, action)){
                        actionPriorityLists.get(0).add(action);
                    }
                    break;
                case BuildCity:
                    actionPriorityLists.get(1).add(action);
                    break;
                case BuildSettlement:
                    actionPriorityLists.get(2).add(action);
                    break;
               […]
                default:
                    // add to lowest priority list as default
                    actionPriorityLists.get(actionPriorityLists.size()-1)
                       .add(action);
            }
        }
```

Code Snippet 13: An example of CatanRuleBasedPlayer's core execution loop

Development of CRB was not originally intended to be carried out in this manner but was instead necessitated by Catan's characteristics. As such, its action selection logic is very simple. Nonetheless, it provided an important improvement over the random players, and I will evaluate its results and future potential in the following chapters.

The last part of implementation for this project is the design and implementation of a tuneable heuristic for evaluating states in Catan, to be utilised by AI players that rely on state evaluation like MCTS or RMHC. *CatanHeuristic* is the class for this, extending *TunableParameters* and implementing *IStateHeuristic*; this class provides methods for evaluating a game state from a players perspective as well as optimising the weights assigned to the state evaluated. *Code Snippet 14* shows the default weights used in this iteration of *CatanHeuristic*.

```
double playerScore = 0.4;
double playerResources = 0.05;
double playerDevelopmentCards = 0.05;
double playerCities = 0.25;
double playerSettlements = 0.15;
double playerPorts = 0.1;
double opponentsScore = -1;
```

Code Snippet 14: Parameters being evaluated by CatanHeuristic

Each weight is used to multiply the outcome of an evaluation of a certain element of the state in question. For example, *Code Snippet 15* demonstrates how the *playerScore* and *opponentsScore* weights are used, in which they multiple the calculated value of the current player's score and all their opponents score. Opponents scores are given a negative default value, as the higher a player's opponents score the closer the player is to losing.

```
// value each player’s score then divide by total required to win, for
opponents divide by number of opponents
if(playerScore != 0.0 || opponentsScore != 0.0){
    int[] scores = state.getScores();
    for(int i = 0; i < scores.length; i++){
        if (i != playerId){
            stateValue += opponentsScore * ((((scores[i] +
               state.getVictoryPoints()[i]))/
               (double)((CatanParameters)state.getGameParameters())
               .points_to_win)/(double) state.getNPlayers());
        } else {
            stateValue += playerScore * (((scores[i]))
               / (double)((CatanParameters)state
               .getGameParameters()).points_to_win);
        }
    }
}
```

Code Snippet 15: Example of CatanHeuristic evaluating a section of the game state

This heuristic intentionally only evaluates the most obviously important elements of the state. I would not consider myself more than an average Catan player, and to prevent my relative inexperience from overrepresenting elements of the state that are not truly valuable, I designed a simple but easy to improve heuristic. By measuring its future performance, I will be able to iteratively improve it without requiring too much prior game knowledge. Eventually introducing advanced elements like Longest Road once I am sure the obvious elements are being evaluated correctly.

Chapter 6: **Results**

The following chapter will describe the results of this project. Every experiment was divided into four sub-sets of games, shifting the order of play in each sub-set (player one → player two, player four → player one, etc.) to remove the bias introduced by turn order in Catan [16]. Individual games had a max round of 500 to prevent infinite games and make experiments much faster; 500 was chosen as games over 2000 rounds never went on to finish without interruption, and almost all games over 500 rounds would carry on to 2000 as per *Table N*. Every game ran had four players, meaning win rates higher or lower than 0.25 indicates an agent winning respectively more or less than its share of games in a set. All tests were carried out on a personal desktop computer with a 4.3GHz processor.

Finishing Round	Max Round	
	2000	500
Round < 100	4831	4669
100 < Round < 200	2717	2740
200 < Round < 300	414	565
300 < Round < 400	166	145
400 < Round < 500	62	44
500 < Round < 1000	9	n/a
1000 < Round < 1500	0	n/a
1500 < Round < 2000	0	n/a
Draw	1801	1837

Table 1: Finishing rounds in Catan over 10000 games with random players

Table 2 is a selection of results from The MCTS agent that used TAG's default parameters (*Appendix D)* with a time budget of 40ms. Experiments not including an MCTS agent consisted of 10000 games while those including the MCTS agent consisted of 1000 games.

Player	Win Rate	Player	Win Rate	Player	Win Rate	Player	Win Rate
Random	0.24	CRB	0.261	MCTS	0.375	MCTS	0.154
Random	0.191	CRB	0.248	Random	0.157	CRB	0.292
Random	0.201	CRB	0.238	Random	0.135	CRB	0.279
Random	0.194	CRB	0.247	Random	0.168	CRB	0.258
Draw	0.174	Draw	0.006	Draw	0.165	Draw	0.017

Table 2: Win and draw rates for several different algorithms in Catan

The CRB agent displays a reasonable improvement in performance against MCTS compared to the random agent, but its performance is largely underwhelming for reasons discussed in Chapter 7. However, CRB is effective at its core goal of reducing the frequency of draws and speeding up game simulation: the average game time for four random agents was recorded as 319ms compared to 226ms with CRB agents.

NTBEA was used to optimise TAG's existing MCTS agent towards winning in Catan using the search space in *Table 3* based on an example provided by James Goodman and with these parameters; 10 runs, 100 iterations, 20 evaluation games, an exploration constant of 1, and CRB opponents.

Parameter	Values
K (UCB exploration constant)	**0.01**, 0.1, 1.0, 10.0, 100.0
Rollout Length	**0**, 3, 10, 30, 100
Max Tree Depth	1, 3, **10**, 30, 100
Open Loop	false, **true**
Selection Policy	**Robust**, Simple
Redeterminise	**false**, true
Tree Policy	UCB, **AlphaGo**, EXP3, RegretMatching
Opponent Tree Policy	SelfOnly, Paranoid, **MaxN**
Explore Epsilon	0.01, 0.03, 0.1, **0.3**

Table 3: Parameter space for MCTS agent. Optimal values are in bold. Refer to [3] for parameter definitions.

While the performance of these parameters is disappointing in later experiments, their selection is explainable and is readily improvable in future. The 40ms time budget enforced was too short for MCTS to make any rational action, evidenced by the poor performance in *Table 3*. As NTBEA was optimising for wins and performance was so poor, it is unlikely many successful generations evolved to improve the solution. The estimated value of the chosen parameter set was 0.250 +/- 0.0437, and such a low estimated value indicates a poor optimisation outcome.

Following NBTEA optimisation, the *CatanHeuristic* was implemented, completing the set of agents intended for the first round of experiments: OSLA; RMHC; MCTS with TAG's default parameters (*Default)*[1]; MCTS with optimised parameters *(Optimised)*; and both optimised *(O-Heuristic)* and unoptimised (*U-Heuristic)* MCTS utilising the *CatanHeuristic*. In each test, the agent plays 1000 games against three CRB players. A 40ms time budget is used for RMHC, and all MCTS variations and the OSLA agent has the same heuristic as the MCTS agents.

[1] These are the same results from *Table N*

Player	Win Rate	Player	Win Rate	Player	Win Rate
OSLA	0.158	RMHC	0.169	Default	0.154
CRB	0.762	CRB	0.751	CRB	0.829
Draw	0.08	Draw	0.08	Draw	0.017

Player	Win Rate	Player	Win Rate	Player	Win Rate
Optimised	0.144	O-Heuristic	0.099	U-Heuristic	0.201
CRB	0.592	CRB	0.633	CRB	0.779
Draw	0.264	Draw	0.268	Draw	0.02

Table 4: Win and Draw rates across several agents against 3 CRB opponents

Table 4 is the win and draw rates from these experiments; performance across the board is poor but not surprising. As outlined, Catan poses many difficulties for existing general AI algorithms, and most agents have equivalent performance to the random agent (who has a 0.153 win rate against 3 CRB players). Both MCTS agents using the optimised parameters perform poorly due to the ineffective parameter search outlined earlier.

Player	Random	CRB	OSLA	RMHC	Default	Optimised	O-Heuristic	U-Heuristic
Avg. score	3.458	3.954	3.479	3.270	3.598	3.367	3.458	4.106

Table 5: Average score in games lost against 3 CRB opponents

Also gathered from these experiments was the average score for each agent in games it did not win, shown in *Table 5* As Catan has more than two players, it is possible to play well and still lose, so score provides an easily accessible but more granular understanding of a player's performance. Most average scores reflect the win rates of the agents, but some are of note. RMHC has the lowest average score despite having the second-highest win rate, potentially indicating RMHC (and by extension RHEA) is effective at winning games from an advantageous position but not employing a strategy to turn a losing game into a winning one. On the other side, despite having the worst win rate by far, the optimised MCTS agent with *CatanHeuristic* has a solidly average score, indicating it can identify short term strategies to earn score but stalls after early play possibly explaining the large increase in draws seen from both optimised algorithms. Overall, the unoptimised heuristic performed best in both categories, outperforming the CRB on average score, affirming [16] findings on the performance of MCTS with domain knowledge.

Identifying the 40ms budget time as overzealously restrictive led to a small set of further experiments to evaluate how this had impacted agents, again one agent in question against three CRB agents.

Budget (*ms*)	RMHC		Default		U-Heuristic	
	Win Rate	Avg. Score	Win Rate	Avg. Score	Win Rate	Avg. Score
40	0.16	3.270	0.15	3.598	0.20	4.106
100	0.15	3.599	0.34	4.465	0.28	4.630
1000	n/a	n/a	0.31	4.597	0.29	4.938

Table 6: The effects of various budgets on different agents in Catan

Visible in *Table 6,* a higher computational budget led to an expected significant improvement in performance in both average score and win rate for the MCTS agents tested. RMHC, however, saw no improvement; this suggests that its limitations in Catan are not caused by budget constraints, but the sample of games run for this experiment was limited to 100, so this result is not conclusive.

Chapter 7: **Evaluation**

This project identifies a clear benefit from a freely available digital implementation of Catan, which is not limited by a specific interface. The implementation of Catan in TAG is a close replica of the tabletop game that provides a strong base for future expansion and is satisfactory for AI research in its current state, demonstrated by the experiments carried out in this report. The implementation of many of Catan's rules as tuneable parameters provides an avenue for investigation into the impact of single-game variation on AI performance in future work. Trade is the only instance of significant deviation from Catan's base rules as outlined in implementation. There are a few remaining minor elements that require tweaking, particularly the GUI. I have consistently applied fail-fast principles as well as ensured my code is both efficient and readable by following TAG's established development patterns, resulting in a stable solution. This project has addressed the gaps identified in existing Catan implementations, and while not perfect, it is implemented in a fashion that will make future improvements accessible.

The first obstacle encountered in this project was the exceptionally poor performance of random play. Random play is unlikely to be a winning strategy in any game; however, the ability to do nothing combined with the ability to take actions that directly harm the player's own position while aiding their opponents is relatively unique among board games. Gathering data on agents playing against random players was both unpredictable and time-consuming. CRB's development was in direct response to this issue. CRB was based solely on my own Catan experience and some informal research of Catan strategies as it was not the focus of this project. While CRB did solve the issues of speed and indeterminate games to a satisfactory extent, its performance was still disappointing. Two identifiable areas of weakness for CRB are that it does not evaluate starting positions, and it does not weight build actions differently based on their possible outcomes [20]. Substantial improvement to CRB will require a complete redesign as its greedy action analysis is not suitable for Catan.

The *CatanHeuristic* was designed to implement TAG's pattern of capturing all game state that might be relevant and then using tuneable parameters to allow the optimisation of the weights assigned to each element of said state. The benefit to this method is that the designer does not need perfect knowledge of the game, just enough to identify the parts that are important. Time constraints prevented me from optimising the heuristic, but even with its default weights, agents that utilised it performed better than without, a promising result encouraging further development.

This project successfully investigates the performance of several general AI algorithms in Catan. The relatively good performance of the MCTS agents with a heuristic specific to Catan supports the conclusions on domain knowledge in [16]. Promisingly, the performance of MCTS with no heuristic ranged from close to better than its knowledgeable counterpart at larger computational budgets; with more effective optimisation, general MCTS will be able to outperform the existing heuristic. RMHC was the best general algorithm with a time budget of 40ms but still did not come close to winning its share of games. Moreover, RMHC performance did not improve with a budget increase like MCTS, indicating it is unlikely to see further improvement in Catan just through optimisation. Subsequently, if RMHC failings are due to rollouts being an ineffective estimator in Catan, it reasons to believe that RHEA will perform poorly as well. Without injecting domain knowledge to control evolution, I do not believe an evolutionary computation-based algorithm will be more likely to succeed than tree-search algorithms in Catan or games like it. OSLA's performance is surprisingly average, I was unable to identify a reliable method of improving its performance further. However, a key limitation of these evaluations is the questionable certainty of the results they rely on. Catan's highly stochastic environment demands a considerable quantity of games to ensure results are not just a fluke. Coupling this with the long average game length leads to a challenging environment to collect data under time constraints. Through the course of this project, a very wide variety of data has been generated. Early experimentation with the random player would often produce confusing results of no real value except to contradict the previous set of results. This project aimed to resolve the time blocker other projects utilising *JSettlers* had run into, which I can confidently say it has, but the time required to gather data to reach this point was underestimated. Criticism of existing research in this area is a lack of sufficient sample data, and while the conclusions based on the data I have seem logically sound, more research is needed to confirm the validity of this data.

While this project has demonstrated the use of NTBEA for optimisation of a general algorithm for Catan within TAG, the chosen parameters were not practical for generating a competent agent. Score or heuristic should be optimised over wins in future, and computational budget should begin at the maximum viable and work inwards if possible, not the minimum and outwards when necessary, as accidentally occurred in this project. Furthermore, the number of runs and iterations were too low to explore Catan thoroughly enough to receive an effective optimisation.

Chapter 8: **Conclusion**

General AI is an ongoing field of study with countless *possible* practical applications but no existing *successful* practical application. This project has succeeded in contributing an easily accessible implementation of Catan with a wide scope for extension to the TAG framework. This implementation can now be used by researchers in conjunction with the many other games implemented in TAG to continue to push the boundaries of what AI agents can achieve in general scenarios. The trading system still requires development to be a truthful Catan replication, and there is more work to do on streamlining the code base, but overall, I am satisfied that this implementation fills the gap identified in Chapter 2. Beyond the refinements already planned for Catan, there are many future possibilities, such as several official expansions that introduce more complex mechanics and strategies and many unofficial variations. TAG's component-based structure allows anyone to take the existing implementation and re-use any part they wish, potentially for Catan expansions but also just for games of a similar nature.

I developed the first iteration for the Catan heuristic, making use of parameters optimisable through the same framework as player parameters in TAG. I have not yet had the opportunity to attempt optimisation, so there exists some immediate future work. Furthermore, the heuristic only considers the most prominent essential elements of a state at this time, potentially this will be the perfect heuristic for Catan, but it is much more likely that future work can be done on evolving its complexity. As more people use TAG this should begin to inform some of these designs that require experience to perfect.

A significant shortcoming of this project was the optimisation of the MCTS agent using TAG's NTBEA. After several difficulties in getting the parameter search to run, with James Goodman's assistance, I identified an error present before I joined development on TAG that I had then copied to every class I implemented for the Catan package. This required the reimplementation of several methods in almost every class used in Catan. Following that, the first attempt at NTBEA ran for four consecutive days before the machine crashed, spurring CRB development beyond just a design. Armed with CRB but out of the time I had planned to spend on NTBEA I rushed an attempt for the general MCTS agent, I did not allow for enough iterations or runs, and I optimised for an unsuitable goal with a very restrictive computational budget. The resulting set of MCTS parameters, while rationally selected based on my NTBEA parameters, performed worse than the default MCTS parameters in TAG. The optimisation output was a failure because I was attempting to learn and implement concepts simultaneously; despite the agent's failure, this informed how I approached the rest of the project in a beneficial way and still demonstrated the capabilities of TAG and Catan. I intend to return to this soon to try optimising a general MCTS again, and there will continue to be lots of possible future work regarding the use of NTBEA in TAG for heuristic and player optimisation.

The creation of CRB is both a strength and weakness of the project. CRB had initially been envisioned as a carefully hand-crafted agent capable of a competent baseline level of play but was relegated to prioritise the game's development.

Once experimentation fully started, I developed CRB as I needed a predictable player to prevent draws, confusing results, and unnecessarily long games. CRB fills this role excellently, almost always lowering the draw rate, providing a consistent level of difficultly for agents, and speeding up games. But the idea of a competent baseline level player was lost to haste, and CRB ended up not much better than a random player in terms of quality. I would like to revisit CRB with a proper design methodology in mind, utilising the work of [20] to inform this, but it is something that has already been academically explored, so it will probably not be an avenue for future academic work. There is the possibility of future work in hand-crafted Catan AI if it becomes necessary to train or test more general players, but the current research focus is on the use of AI algorithms to approximate hand-coded AI behaviour.

The successful implementation of Catan has been evidenced through my investigation into the performance of the general AI algorithms currently available in TAG. These investigations did not provide any conclusive solutions to Catan AI, nor did I really expect them to; solving Catan AI, especially in a general nature, is beyond the scope of this project. These investigations instead aimed to provide preliminary conclusions to focus the direction of future research on this topic. RHMC performed poorly in Catan, and I expect any general evolutionary computation algorithm that does not make effective use of domain knowledge to struggle. MCTS showed promise as with an adequate budget, it beats the CRB player, and combining it with a heuristic only improves it. OSLA was examined out of completeness but was predictably bad; Catan requires a player to look more than one step ahead for effective play. Excellent heuristic design could theoretically improve OSLA, but it would require near-perfect knowledge of Catan. The final weakness of this report lies in the amount of data I have been able to collect for some of the agents. All experiments involved enough games to be significant in drawing conclusions, but from the variation in results I have seen while working on this project, it would be unwise to guarantee any of these conclusions based on the number of games I have had the time to simulate. As such, there is plenty of immediately attemptable avenues of future work investigating general AI for Catan; MCTS and RMHC, in particular, require a substantial time dedication to allow a necessarily large set of games to run to reduce the impact of Catan's stochasticity.

As highlighted by this chapter, the contributions and conclusions of this project has led to several potential areas of further research and thanks to TAG it is possible that if these areas are not explored by myself, that they can be explored by someone else in the field.

Chapter 9: **References**

[1] G. N. Yannakakis and J. Togelius, Artificial Intelligence and Games, Springer, 2018.

[2] The AlphaStar Team, "AlphaStar: Grandmaster level in StarCraft II using multi-agent reinforcement learning," [Online]. Available: https://deepmind.com/blog/article/AlphaStar-Grandmaster-level-in-StarCraft-II-using-multi-agent-reinforcement-learning. [Accessed 26 November 2020].

[3] R. D. Gaina, M. Balla, A. Dockhorn, R. Montoliu and D. Perez-Liebana, "TAG: A Tabletop Games Framework," in *Experimental AI in Games (EXAG), AIIDE 2020 Workshop*, 2020.

[4] R. E. Korf, "Multi-player alpha-beta pruning," *Artificial Intelligence,* vol. 48, no. 1, pp. 99-111, 1991.

[5] S. Keizer et al, "Evaluating Persuasion Strategies and Deep Reinforcement Learning methods for Negotiation Dialogue agents," in *Association for Compuational Lingustics*, Valencia, Spain, 2017.

[6] M. G. Bellemare, Y. Naddaf, J. Veness and M. Bowling, "The Arcade Learning Environment: An Evaluation Platform for General Agents," *Journal of Artificial Intelligence Research,* vol. 47, 2013.

[7] M. Genesereth, N. Love and B. Pell, "General Game Playing: Overview of the AAAI Competition," *AIMag,* vol. 16, no. 2, p. 62, 2005.

[8] M. Ebner, J. Levine, S. M. Lucas, T. Schaul, T. Thompson and J. Togelius, "Towards a Video Game Description Language," *Dagstuhl Follow-Ups,* vol. 6, pp. 85-100, 2013.

[9] M. Thielscher, "A General Game Description Language for Incomplete Information Games," *AAAI,* vol. 24, no. 1, Jul. 2010.

[10] D. Perez-Liebana, S. Samothrakis, J. Togelius, T. Schaul and S. Lucas, "General Video Game AI: Competition, Challenges and Opportunities," *AAAI,* vol. 30, no. 1, Mar. 2016.

[11] C. Browne and F. Maire, "Evolutionary Game Design," *IEEE Transactions on Computational Intelligence and AI in Games,* vol. 2, no. 1, pp. 1-16, Mar. 2010.

[12] A. M. Turing, "Digital computers applied to games," in *Faster than thought*, London, Pitman, 1953, pp. 286-311.

[13] R. Coulom, "Efficient selectivity and backup operators in Monte-Carlo tree search," in *Computers and Games*, Turin, Italy, Springer, 2006, pp. 72-83.

[14] S. M. Lucas, J. Liu and D. Perez-Liebana, "The N-Tuple Bandit Evolutionary Algorithm for Game Agent Optimisation," in *2018 IEEE Congress on Evolutionary Computation (CEC)*, 2018.

[15] D. B. Fogel, "What is evolutionary computation?," *IEEE Spectrum,* vol. 37, no. 2, pp. 26-32, Feb. 2000.

[16] I. Szita, G. Chaslot and P. Spronck, "Monte-Carlo Tree Search in Settlers of Catan," in *Advances in Computer Games*, Pamplona, Spain, 2009.

[17] M. Pfeiffer, "Reinforcement learning of strategies for Settlers of Catan," in *International Conference on Computer Games: Artificial Intelligence, Design and Education 2004*, Wolverhampton, 2004.

[18] G. Roelofs, *Monte Carlo Tree Search in a Modern Board Game Framework B.S thesis,* Maastricht: Maastricht University, 2012.

[19] CATAN GmbH, “Catan - Game Rules & Almanac (3-4 Players),” 2020. [Online]. Available: https://www.catan.com/files/downloads/catan_base_rules_2020_200707.pdf. [Accessed 02 05 2021].

[20] M. Guhe and A. Lascarides, "Game strategies for The Settlers of Catan," in *2014 IEEE Conference on Computational Intelligence and Games*, 2014.

[21] R. E. Fikes and N. J. Nilsson, "Strips: A new approach to the application of theorem proving to problem solving," *Artificial Intelligence,* vol. Volume 2, no. Issues 3–4, pp. Pages 189-208, 1971.

Appendix

Appendix A: Abbreviations

AI – Artificial Intelligence

RHEA – Rolling Horizon Evolutionary Algorithm

RMHC – Random Mutation Hill Climber

OSLA – One-Step Look Ahead

CRB – Catan Rule Based [Player]

MCTS – Monte-Carlo Tree Search

TAG – Tabletop Games Framework

NTBEA – N-Tuple Bandit Evolutionary Optimisation

Appendix B: Contributions Table

The following table includes every file from the *core.games.catan* package from TAG, a contextualising description of the file and a brief summary about my contributions. The ReadMe and GUI package in Catan have not been included as I was not involved, any other omitted files will also have not been worked on by myself. This table has been included to reduce the potential risk of plagiarism as I have worked alongside PhD student Martin Balla for part of this project, I have tried to clearly define where my contributions end and have aired on the side of caution. If necessary, a full git log can be provided to support this table and I am totally happy to remove anything that is in the table erroneously. Once again sincere thanks to Martin Balla who laid the groundwork for this project long before I had joined the team.

After James Goodman helped me identify the source of a bug which was hash code functions that had been implemented incorrectly before I joined that I had unfortunately then copied onto some of my own work. I was required to perform a series of fixes on every single file in the actions package to resolve this. To save repeating the process in the table, I will explain it here and link the git commit as proof below. For every single action I re-implemented the state as immutable, this was lots of work for most actions, some also requiring large changes to the CatanActionFactory while for others it meant just adding a final modifier. Once all actions had immutable state, I had to reimplement the *hashCode, copy, equals* and *getString* method for each one, some of these already had partially or fully correct functions but all required some amount of change. This process provided the opportunity to consolidate some classes into one single class and delete unused ones and I also introduced error throwing in many places the actions in Catan to promote stability. This turned into many small commits, which I squashed into one before merging with remote, the commit can be seen here:

https://github.com/GAIGResearch/TabletopGames/commit/8c46f09c113105bdbec44263478bec39784de3ac

File	Description	My Contributions
CatanActionFactory	Handles the generation of actions for players	Introduced trade and buy stages; Tracking development card played; Player to player trade offer generation; Accept trade framework; Trade renegotiation action generation; DiscardCards action generation; Several bug fixes
CatanForwardModel	Responsible for execution of game logic	Early turn stage management; Added any new action I created to *_computeAvailableActions*; Added functionality to

		check for trade offer and add correct player to reactive players list; Stopped searching tiles for production if 7 was rolled; Max round system; Added multi-action turn system;
CatanGameState	Stores state and exposes components and info to other parts of the framework	Added a function for swapping cards between players for trades; Added a function for getting a players settlements; Implemented default heuristic score function; Handful of bug fixes;
CatanHeuristic	Parameterised heuristic for state evalutation	Entirely implemented by me;
CatanParameters	Customisable parameters that define Catans rules	Added parameters for negotiations, max round and multi-action turns;
CatanTurnOrder	Responsible for transferring turn ownership and tracking the turn stages	Turn stage management before transition to game phases; Multi-action turn implementation
	components	
Settlement	Represents settlement or cities in game world	A handful of changes to make harbours work;
CatanTile	Represents a board hex	Small change to make harbours work;
	Actions	
AcceptTrade	Accepts a player trade offer	Entirely implemented by me;
BuildSettlement	Places a settlement piece on the board	Small changes to make harbours work;

OfferPlayerTrade	Sends an offer to a player	Entirely implemented by me;

Table 7: Table outlining my contributions to the core.games.catan package

Appendix C: Discard Combinations Algorithm

```
for( int brickIndex = resources[0]; brickIndex >= 0; brickIndex--){
    if(brickIndex == r){
        combinations.add(new int[]{brickIndex, 0, 0, 0, 0});
    }
    if(brickIndex >= r){
        continue;
    }
    for( int lumberIndex = resources[1]; lumberIndex >= 0; lumberIndex--){
        if (brickIndex + lumberIndex == r) {
            combinations.add(new int[]{brickIndex, lumberIndex, 0, 0, 0});
        }
        if (brickIndex + lumberIndex >= r) {
            continue;
        }
        for( int oreIndex = resources[2]; oreIndex >= 0; oreIndex--){
            if (brickIndex + lumberIndex + oreIndex == r) {
                combinations.add(new int[]{brickIndex, lumberIndex,
                        oreIndex, 0, 0});
            }
            if (brickIndex + lumberIndex + oreIndex >= r) {
                continue;
            }
            for( int grainIndex = resources[3]; grainIndex >= 0;
               grainIndex--){
                if (brickIndex + lumberIndex + oreIndex + grainIndex == r){
                    combinations.add(new int[]{brickIndex, lumberIndex,
                       oreIndex, grainIndex, 0});
                }
                if (brickIndex + lumberIndex + oreIndex + grainIndex >= r){
                    continue;
                }
                for( int woolIndex = resources[4]; woolIndex >= 0;
                       woolIndex--){
                    if (brickIndex + lumberIndex + oreIndex + grainIndex +
                       woolIndex == r) {
                        combinations.add(new int[]{brickIndex, lumberIndex,
                       oreIndex, grainIndex, woolIndex});
                    } else if (brickIndex + lumberIndex + oreIndex +
                       grainIndex + woolIndex < r) {
                        break;
                    }
                }
            }
        }
    }
}
CatanParameters.Resources[] values = CatanParameters.Resources.values();
for (int[] combination : combinations){
    CatanParameters.Resources[] cardsToDiscard
        = new CatanParameters.Resources[r];
    int counter = 0;
    for (int i = 0; i < combination.length; i++){
        for (int k = 0; k < combination[i]; k++){
            cardsToDiscard[counter] = values[i];
            counter++;
        }
    }
    actions.add(new DiscardCards(cardsToDiscard, gs.getCurrentPlayer()));
}
```

Appendix D: Default TAG AI Parameters

Parameter	Values
K (UCB exploration constant)	*sqrt(2)*
Rollout Length	10
Max Tree Depth	10
Epsilon	1e-6
Rollout Type	Random
Open Loop	false
Redeterminise	false
Selection Policy	Robust
Tree Policy	UCB
Opponent Tree Policy	Paranoid
Explore Epsilon	0.01

Table 8: TAG's default MCTS parameters

Parameter	Values
Horizon	10
Discount Factor	0.9

Table 9: TAG's default RHMC parameters