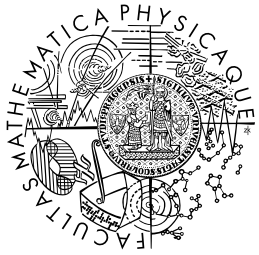

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Daniel Crha

Board game with artificial intelligence

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis: Mgr. Martin Pilát, Ph.D.

Study programme: Computer Science

Study branch: General Computer Science

Prague 2020

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

Most of all I want to thank my supervisor for his help and advice, he was always there for me whenever I needed his opinion. I also thank all of my family and friends for being there for me along the way, my journey has been long and I could not have done it without them.

Title: Board game with artificial intelligence

Author: Daniel Crha

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Martin Pilát, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: Multiplayer board games with imperfect information present a difficult challenge for many common game-playing algorithms. Studying their behavior in such games can be difficult, because existing implementations of such games have poor support for artificial intelligence. This thesis aims to implement an imperfect information multiplayer board game in a way that provides a framework for developing and testing different types of artificial intelligence for board games with the aforementioned qualities. Furthermore, this thesis explores the implementation of several algorithms for the game. This aims to showcase the artificial intelligence framework, as well as to analyze the performance of existing algorithms when applied to a board game with elements such as hidden information and multiple players.

Keywords: board game, artificial intelligence

Contents

Introduction

Foreword

In game theory, perfect information two-player games are often studied, and numerous algorithms have been designed with the purpose of playing them. This includes games like Chess and Go, which have had large breakthroughs in recent years [1]. However, real world situations do not always have perfect information, or only two parties involved. We could for example imagine multiple countries, which have only approximate information about the armies of their opponents. In this scenario, it could be useful to have tools to simulate potential enemy troop movements or placements.

Even though algorithms which are able to model imperfect information and multiple players are often useful, they are not studied nearly as often. Designing such an algorithm is not easy, and there are many pitfalls which make conventional game theory algorithms much less effective at solving imperfect information and multi-player problems. This thesis therefore aims to analyze the problems of implementing such algorithms, and to implement some of them in pursuit of that goal.

Naturally, some frameworks do already exist for the implementation of such games. However, at the time of writing, some of them only have AI (Artificial Intelligence) support as an experimental and sparsely documented feature [2], and others only focus on specific fields of AI [3]. This work aims to provide a kind of plug-and-play" experience, where AI developers have minimal barriers between cloning a git repository and having a working AI.

Goals

The main goal of this thesis is to create a multi-player board game with imperfect information states. The game's name is *Colonizers*. The game will primarily be designed with AI in mind, and it will provide a reasonable interface for the implementation of AI players.

Another goal is the implementation of several AI players for said game. This will allow us to not only explore potential problems with implementing AIs for games of this kind. We will also verify that the API (Application Programming Interface) provided by the game is sufficient for implementation of such AI players, and that the API is reasonably easy to use.

We then wish to compare the implemented AI algorithms in mutual play, and identify the algorithms which perform the best. This will establish a benchmark for future AI algorithms which can be developed for *Colonizers*.

1. Related Work

Before we discuss the design of *Colonizers* and its implementation, let us first make an overview of existing related work. This chapter will demonstrate why existing frameworks would not be a good fit for *Colonizers*. We are interested mainly in two areas here:

- Implementation of similar games, and frameworks facilitating that
- Algorithms adapted for multi-player games and algorithms adapted for imperfect information games

1.1 Game Frameworks

1.1.1 OpenAI Gym

OpenAI Gym is "a toolkit for developing and comparing reinforcement learning algorithms" [3]. It is a popular tool in the reinforcement learning field, because it is modular, and easy to work with. It features a standardized API for all of its environments (games or problems). This means that agents built for one environment can be easily transitioned to other environments, without having to structurally rebuild it. Another benefit is the fact that it is easy to create new environments, and these newly created environments can be used by anyone, since the API is standardized.

Figure 1.1 is an example (as presented in the OpenAI Gym documentation [3]) of a Python program which solves one of the simpler environments available out-of-the-box in OpenAI Gym.

```
import gym
env = gym.make('CartPole-v0')
for i_episode in range(20):
    observation = env.reset()
    for t in range(100):
        env.render()
        print(observation)
        action = env.action_space.sample()
        observation, reward, done, info = env.step(action)
        if done:
            print("Episode finished after {} timesteps".format(t+1))
            break
env.close()
```

Figure 1.1: OpenAI Gym — AI implementation.

The environment being solved (*CartPole-v0*) is a task where the AI must balance a pole by moving the cart below it left and right. The agent only performs random moves, but the example clearly illustrates how the agent interacts with the environment.

OpenAI Gym is not particularly suitable for the study of multi-player games with imperfect information for a few reasons:

- It only supports reinforcement learning agents. The API is designed with this in mind, and only provides appropriate tools to machine learning approaches. Other kinds of artificial intelligence are not very well supported.
- It does not provide any tools for determinization [1] of imperfect information states. This would force AIs to track their own information sets, and to then produce determinizations of game states on their own.

In spite of that, there is something we can take away from OpenAI Gym when designing *Colonizers*. Notably, the API is very elegant, and creating an AI which simply plays random moves is a matter of very few lines of code. We will try to achieve this with *Colonizers*.

1.1.2 boardgame.io

boardgame.io [2] is a game engine for creating turn-based games. It features many helpful features for creating board games, such as support for multiplayer, randomness, imperfect information, and a few other useful features.

Using boardgame.io for the implementation of *Colonizers* would make many things much simpler, notably the implementation of game logic would be trivial. However, it is also not suitable for the purposes of this thesis, because the AI support is poor. The engine does feature a degree of AI support, but the API is limited to using pre-existing AIs which ship with the game. The AIs which ship with the game are an MCTS (Monte Carlo Tree Search) AI and a random AI. The API for AI players only provides a method for us to list the legal moves in a given game state — it does not however provide ways to implement a fully custom AI.

[1] By the determinization of a game state, we understand the conversion of a game state with hidden information into a game state with perfect information. Determinization takes into account the information set of the given player. For example, we can imagine a poker player who has been dealt a hand which includes the Queen of Hearts. When this player is thinking about what other players may have, the Queen of Hearts is out of the question, since the player has it, and there is only one in the deck. Therefore, a rational determinization of a poker game state would be to take all cards which started in the deck, remove the ones the player is holding, and then randomly assign other cards to the other players.

1.2 Algorithms

Here we will discuss several existing algorithms which are applicable to *Colonizers*. This includes algorithms which will need to be adapted in order to be useful in our situation, and algorithms which will work mostly out-of-the-box.

1.2.1 MaxN

Most work in the field of game-playing algorithms has traditionally been done in games which involve two players, perfect information, finite games which do not feature random processes. These games are also often constant-sum, therefore they cannot feature cooperative strategies. One of the most well-known algorithms from this field is the Minimax algorithm [4]. The pseudocode in Figure 1.2 demonstrates the Minimax algorithm.

```
def minimax(node, depth, isMaximizing):
    if depth == 0 or node is terminal:
        return node.heuristicValue
    if isMaximizing:
        value = -inf
        for child in node.children:
            value = max(value, minimax(child, depth - 1, False))
        return value
    else:
        value = +inf
        for child in node.children:
            value = min(value, minimax(child, depth - 1, True))
        return value
```

Figure 1.2: Minimax algorithm [4].

The core principle of Minimax is the fact that in a two-player zero-sum game, one player's gain is the other player's loss. Therefore from the point of view of the opponent, minimizing the player's score also means maximizing their own. When evaluating the game tree, both players can use the same metric to make decisions — one player is maximizing it, and the other is minimizing it. The nodes of the tree alternate every level — in the root state, the current player is maximizing the value of nodes, and in the children of the root state, the other player is minimizing the value of nodes. In other words, if the root state is associated with player A, player A will attempt to choose a node with a maximal value at every even level of the tree (assuming the root node is level 0). Player B will then attempt to choose the least valuable child in nodes at odd levels in the tree. Minimax explores the tree of game states up to a depth d, whereby rather than deepening the tree, it uses a positional evaluation function to evaluate the leaf nodes. As found by Hoki and Kaneko [5], the quality of the positional evaluation function is very critical to the performance of Minimax.

If we want to apply a Minimax-like method to *Colonizers*, we must move away somewhat from the original Minimax algorithm. In multi-player games,

the game is often not a zero-sum game. Minimax can still be applied in this situation, though we have to make a relatively strong assumption that the goal of other players is to minimize the player's score. We often cannot afford to make this assumption, since this would mean that opponents have no regard for their own points. Since Minimax is not powerful enough for our use case, we will look to the MaxN algorithm [6].

The MaxN algorithm is not an extension of Minimax strictly speaking, but it does apply the driving principles of Minimax to games with more than two players. To introduce multiple players and a non-constant sum game to Minimax, MaxN changes the way the game is viewed. Rather than the other players trying to minimize the player's gain, each player is trying to maximize their own gain independently. Each game state has an associated payoff vector, where the i-th position of the vector contains the payoff for player i in this state. Just like in Minimax, levels in the tree correspond to players making decisions on their turns. In the context of MaxN, this means the i-th player maximizing the i-th position of the reward vector among the current node's children.

The procedure MaxN is defined recursively (as presented by Luckhardt and Irani [6]) in Figure 1.3.

(1) For a terminal node,
maxn(node) = payoff vector for node
(2) Given node is a move for player i, and
$(v_{1j}, ..., v_{nj})$ is maxn(j^{th} child of node), then
maxn(node) = $(v_1^*, ..., v_n^*)$,
which is the vector where $v_i^* = \max_j v_{ij}$.

Figure 1.3: MaxN algorithm [6].

It is important to mention that MaxN still uses a positional evaluation function, much like Minimax. However, it is not enough to only evaluate leaf nodes once. Since each node must have a value for every player, we must run the evaluation function once from the perspective of each player in the game. Notice that even though each player is only maximizing their own value and disregarding that of others, this does not necessarily extend to the positional evaluation function itself. The evaluation function could reasonably take into account the standing of other players and boost the evaluation if other players are doing poorly.

We can see an example of MaxN evaluating a state tree in a three-player game in Figure 1.4. Observe how at each level, the player on turn chooses the action which gives them the highest reward.

The MaxN algorithm itself does not solve the problem of imperfect information. Therefore we will describe the extension of MaxN to imperfect information games in Section 4.3.

1.2.2 Monte Carlo Tree Search

Monte Carlo Tree Search [7] is a class of algorithms for searching trees. In comparison with the Minimax class of algorithms, it does not feature a positional

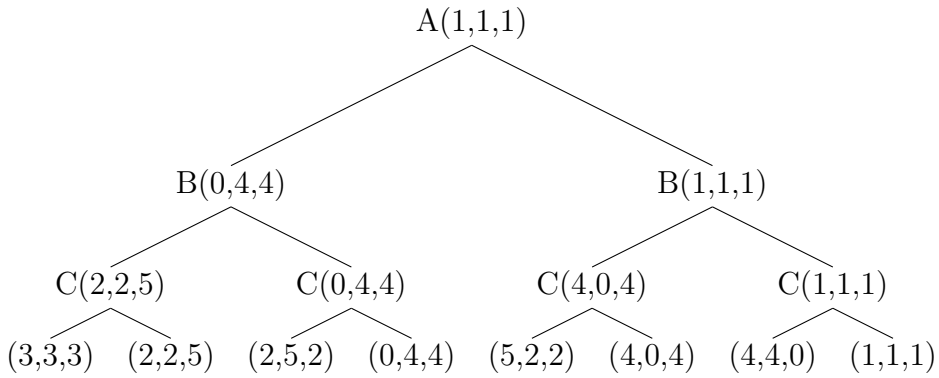

Figure 1.4: MaxN tree example [6].

evaluation function. Instead it uses simulation to evaluate positions. When we talk about MCTS in the context of game playing algorithms, MCTS consists of two parts: a moderately shallow tree, and deep simulated games. The algorithm grows its tree structure by adding one node at a time, and then performing a game simulation from the position associated with the node. The reward gained from the result of the game playout is then backpropagated up the tree. After iterating, MCTS can then choose the best move in the root node by simply choosing the node with the best accumulated reward.

We can divide a MCTS-class algorithm into four main stages:

- *Selection*: Descend the state tree until a node is reached such that at least one of the actions that leads from it has not yet been explored. We will speak more about the consideration which goes into descending the tree later in this section.
- *Expansion*: Choose an action that has not yet been explored and add its corresponding node into the tree.
- *Simulation*: Run a simulation of the game from the newly added state.
- *Backpropagation*: Propagate the reward from the result of the simulation back through the way we descended the tree.

A concept important to MCTS-class algorithms is the multi-armed bandit problem. We can best understand the problem by reasoning about the following example. Suppose that we have a slot machine with two arms, and both arms have properties which are not known to us. We start by pulling the left arm, and get a payout of 1. In this situation, it would seem reasonable to try the right arm at least once, in case its payouts were much higher than those of the left arm. Let us then assume that we pull the right arm and get a payout of 0.5. Should we keep pulling the left arm or the right arm? The answer to this is not easy, and there are multiple types of algorithms designed to solve the problem of exploration versus exploitation, including stochastic and adaptive methods [8].

We can observe that the problem of choosing whether to perform playouts in a node where we won many times previously, or whether to try new nodes is a kind of multi-armed problem. We can therefore apply algorithms designed to solve multi-armed problems in choosing which nodes to explore and which nodes to exploit in MCTS-class algorithms.

While MCTS is applicable to games with perfect information, it needs to be adapted for games with imperfect information. A popular approach is determinization, which converts states with imperfect information to states with perfect information by sampling information sets (an information set is a set of states which are possible with respect to the information available to the player) [9].

While determinization is a viable strategy, it is not without its pitfalls. Russel and Norvig [10] speculate that since all information is revealed after determinization, the resulting AI will never make information-gathering plays.

Another potential issue is the fact that determinization does not take into account the fact that opponents have a degree of uncertainty about the player's own hidden information. Whitehouse, Powley and Cowling [11] point out that "Determinization does not randomise the player's own cards, and information set trees are built solely from the point of view of the root player. In a sense this is a worst case assumption, but it does mean that these algorithms can never exploit the opponents' lack of information".

Other potential problems include two mentioned by Frank and Basin [12]. The first is *strategy fusion*. This occurs whenever an algorithm attempts to combine strategies from particular worlds to produce an optimal strategy for all worlds. Quoting Frank and Basin [12]: "*The flaw in this approach occurs because of the property of incomplete information games that the exact state of the world at any given point of play may not be known to a player. This imposes a constraint on a player's strategy that he must behave the same way in all possible worlds at such points; a constraint typically broken when combining strategies designed for individual worlds*". The second issue they identified is *non-locality*, whereby certain determinizations may be essentially irrelevant, since players have the ability to avoid them with gameplay decisions.

Some variants of MCTS try to use determinization in clever ways to avoid its drawbacks. We will be looking at ISMCTS (Information Set Monte Carlo Tree Search) [9], in particular we are interested in the SO-ISMCTS variant. In order to overcome the obstacles associated with determinization, SO-ISMCTS tree nodes correspond to information sets rather than game states. Specifically, they correspond to information sets from the root player's point of view. This means that if we choose a determinization for a SO-ISMCTS tree node, it is likely that many of that node's edges will not be valid moves in the context of the determinized state. Therefore, SO-ISMCTS limits the tree into the subtree of valid moves with respect to a given determinization when descending the tree.

Cowling, Powley and Whitehouse [9] also present two other versions of ISMCTS, namely SO-ISMCTS + POM and MO-ISMCTS. We discuss their merits in Section 4.4.

Figure 1.5 shows high-level pseudocode for the SO-ISMCTS algorithm, as presented by Cowling, Powley and Whitehouse [9].

```
def SO-ISMCTS([s_0]^~1, n):
    create a single-node tree with root v_0 corresponding to the
        root information set [s_0]^~1 (from player 1's viewpoint)
    for n iterations do:
        choose a determinization d at random from [s_0]^~1, and
        use only nodes/actions compatible with d this iteration

        # Selection
        repeat
            descend the tree (restricted to nodes/actions compatible
            with d) using the chosen bandit algorithm
        until a node v is reached such that some action from v leads
        to a player 1 information set which is not
        currently in the tree or until v is terminal

        # Expansion
        if v is nonterminal:
            choose at random an action a from node v that is
            compatible with d and does not exist in the tree
            add a child node to v corresponding to the player
            1 information set reached using action a and set
            it as the new current node v

        # Simulation
        run a simulation from v to the end of the game using
        determinization d

        # Backpropagation
        for each node u visited during this iteration do
            update u's visit count and total simulation reward
            for each sibling w of u that was available for
            selection when u was selected, including u itself do
                update w's availability count

    return an action from the root node v_0 such that the
    number of visits to the corresponding child node is maximal
```

Figure 1.5: SO-ISMCTS algorithm [9].

2. Game Design

This chapter's purpose is to discuss the considerations which went into designing the game's rules, and to describe said rules in detail.

The game is set on Mars with futuristic themes. In the game universe, Mars is only just starting to be settled by humans, and there was a precious mineral found under the surface - Omnium. This triggered a rush of colonists, who are eager to make some profit. In the game, they compete for resources, and they all want to build the largest colony, because the person with the largest colony can extract the most Omnium and get rich.

2.1 High Level Design

Colonizers has a few design decisions which are inherently set in stone by the premise of this thesis:

- The game must have more than two players
- The game must feature hidden information
- The game must support AI

AI support is only tangentially related to the design of the game rules, therefore we will not discuss it at length in this chapter. We will focus on the other two requirements.

Colonizers is a four-player game. Four was chosen as a sweet spot for complexity, since with five players, the game would start to get prohibitively expensive to compute. It could be argued that three would accomplish the same goal, but four makes more sense with respect to having enough design space. Four players is also a very common player count for board games.

Hidden information is somewhat more tricky to get right. It can be implemented in many ways, but even the simplest inclusions make the game much more tricky to process with AI. There are two elements of hidden information in *Colonizers*:

- Players' hands and the Deck
- Players' colonists

These elements will be explained in more detail in the following section. It is worth noting however that it is possible to make information-gathering plays, even though only in very limited ways and in rare circumstances. Information-gathering plays are not a large design focus of *Colonizers*.

The game also features interaction between players — both malicious and cooperative. This naturally means that it is possible for multiple players to cooperate in order to gain an advantage, or to conspire against another player in order to damage that player's chances of winning. Many traditional AI algorithms are not capable of cooperation or conspiracy, which provides room for specialized AIs to shine.

Colonizers is technically not finite, since there exists a strategy which, if employed by all players, will lead to the game never ending. The game also has potentially infinite states. In practice this is incredibly unlikely, since the strategy involves players intentionally passing up plays which would move them closer to the victory condition.

2.2 Game Rules

The game is played in rounds, which consist of turns. Turns then consist of phases. The four players start the game in a given order, and they always take turns in this order for the entire game.

Each player has a colony where they can build modules. Each module has a point value, and the goal of the game is to build the most valuable colony. When any player builds eight modules in their colony, the game will end at the end of that round, after the remaining players have taken their turns. When the game ends, the values of all modules in each player's colony are added up, and the resulting value is that player's final score. Players can also get a bonus to their score if they reached eight modules in their colony before the game ended — the first player to build eight modules gets four bonus points, and other players to build eight modules get two bonus points each. [1] The final ranking of the players when the game ends is determined by points — players with more points rank higher. If multiple players are tied in points, the player whose position according to the player order is earlier ranks higher. More information about modules and ways to interact with them follow in subsequent subsections.

There is also a rare game end condition, which is triggered by players attempting to draw from an empty deck [2]. This immediately ends the game with a draw, giving all players zero points and a rank of zero.

2.2.1 Colonist Pick

At the start of each round, players take turns picking colonists. A colonist is a character with special powers, and the player controls a given colonist for one turn. A player's colonist is hidden from the other players. There are six colonists in the game (see Section 2.2.3 for a list of available colonists). At the start of the colonist picks, a random colonist is secretly removed for play for the rest of the round. Then, players take turns picking from the remaining colonists one by one. This means that after the last player picks, there is one colonist left over. This colonist is then removed from play for the rest of the round.

[1] As an example, assume that all four players have seven modules in their colony. When player 1 takes their turn, they do not build a module, ending the turn with seven modules. Player 2 then builds a module on their turn, taking them to eight modules in their colony. This triggers the game end condition, but the game is not over until all players have taken their turn this round. Since player 2 was the first to reach eight modules, they get four bonus points. Player 3 then also builds a module, taking them to eight. Since player 3 was not the first to build eight modules, they get two bonus points. Player 4 then does not build a module, ending the game with seven modules and zero bonus points. When player 4 finishes their turn, the game ends.

[2] The deck is discussed in more detail in Section 2.2.4

The colonist pick phase creates a situation where players have asymmetrical information. For example, the first player knows which colonist was removed at the start, but has no information about the other players apart from knowing the four colonist he is passing on. In contrast, the last player has relatively little information about the players before them, but they know which colonist is removed from play after being left over.

2.2.2 Proper Turns

After each player has chosen a colonist, the players take their actual turns, in order of first to last. Each player acts in all phases of their turn before passing the turn to the next player.

Each turn consists of the following phases:

- *Draw Phase*. The player has two options in this phase. They may acquire two Omnium, which is the game's currency. The player's Omnium count persists between rounds. Omnium is used to build modules in the player's colony. The player may also opt to draw 2 modules from the deck. The player must then keep one of the modules, and place the other at the bottom of the deck. The drawing action is not available if the player's hand is full (five modules). If the player controls a colonist with a passive ability, this ability is automatically performed at the start of the draw phase. The player's colonist is also revealed to other players during this phase.

- *Power Phase*. The player may choose to use their colonist's active ability if the colonist has one.

- *Build Phase*. The player may choose to build one module from their hand. To build a module, the player must spend the Omnium amount required by the module's build cost. Building the module adds it to the player's colony and removes it from their hand.

2.2.3 Colonists

The following colonists and their respective abilities are available in *Colonizers*:

- Visionary
  - Passive Ability: Draw a card if the player's hand is not full (maximum hand capacity is five).
  - Active Ability: None
- Ecologist
  - Passive Ability: Gain 1 Omnium for each green module in his colony.
  - Active Ability: None
- Miner
  - Passive Ability: Gain 1 Omnium for each blue module in his colony.
  - Active Ability: None
- General
  - Passive Ability: Gain 1 Omnium for each red module in his colony.
  - Active Ability: None
- Opportunist
  - Passive Ability: None
  - Active Ability: Steal up to 2 Omnium from a chosen colonist. If no player controls the chosen colonist, this ability has no effect.
- Spy
  - Passive Ability: None
  - Active Ability: Swap hands with a chosen colonist. If no player controls the chosen colonist, this ability has no effect.

2.2.4 Modules

At the start of the game, all modules start in a deck in a random order. The order of modules in the deck is hidden. Whenever a module is drawn by a player, it is taken from the top of the deck. Whenever a module is discarded, it is placed at the bottom of the deck. The only exception to this is the situation where a player is discarding a card after drawing two in the draw phase. If the player's colonist is the Visionary, it is possible for the player to overdraw (draw more modules than the hand capacity would allow). In this situation, overdrawn modules are removed from play entirely for the rest of the game.

There are 52 modules in the game. Modules have no special effects on the game board, the only interaction the player has with them is building them. The modules built in the player's colony also influence the passive ability of the following colonists: Ecologist, Miner, General. The colored modules are intentionally less efficient than modules without a color. This is because they enable synergies with certain colonists. As a consequence, the game has a dynamic where in the early game, it is often beneficial to build colored modules for synergy in order to establish a good Omnium economy for future turns. As the game draws closer to the end, it is often best to start disregarding synergy and simply build the most efficient buildings points-wise. Table 2.1 is a table of all modules available in the game. Note that the module names are a cosmetic feature which would be welcome in a physical copy of the game, but due to screen space constraints on a computer screen, module names were omitted from the game's user interface.

Name	**Build Cost**	**Value**	**Color**	**Quantity**
Oxygen Generator	4	4	Green	4
Water Reservoir	5	6	Green	4
Hydroponics Facility	6	8	Green	4
Eco-Dome	8	11	Green	1
Marketplace	2	2	Blue	4
Warehouse	3	3	Blue	4
Quarry	5	6	Blue	4
Omnium Purification Plant	8	10	Blue	1
Garrison	1	1	Red	4
Barracks	2	2	Red	4
Military Academy	3	3	Red	4
Planetary Defense System	6	7	Red	1
Housing Unit	1	1	None	4
Spaceport	4	5	None	4
Research Lab	6	8	None	4
Mass Relay	12	16	None	1

Table 2.1: Available modules.

2.3 Branching Factor

The branching factor of a game is an important statistic which has heavy influence on the performance of many game-playing algorithms [13]. Therefore it is useful

to analyze the branching factor present in *Colonizers*.

Firstly, we can analyze the branching factor present during the colonist pick phase. There are six colonists in the game, and one of them is always randomly removed. Then, players take turns picking from the remaining five colonists. This means that the first player may choose from five colonists, and the last player may choose from only two. We can view the distribution of colonists during the pick phase as a permutation of colonists, therefore there are $6! = 720$ different outcomes for the colonist pick phase. An interesting case are permutations where the first and last colonist (the colonists removed from play) are swapped. Even though both end up out of play for the round, their presence during the pick phase has an observable effect on the players' information sets. Therefore we must consider these permutations as distinct.

During the draw phase, a player may either gain Omnium, or draw two cards and discard one of them. This means that the draw phase for a single player has 3 different outcomes.

In the power phase, branching factor is only relevant for colonists with active abilities, namely Spy and Opportunist. Players controlling either of these colonists may choose to either not use their power, or to target one of the five remaining colonists. This gives us 6 possible choices. For other colonists, the branching factor is 1. Since two out of the six colonists have an active ability, we can say that the average branching factor for the power phase is $\frac{16}{6} \approx 2.67$.

Lastly, we will consider the build phase. A player may always choose to build nothing. Since the hand size is limited to five, a player may choose to perform up to six different actions during the build phase.

Altogether, this gives us an approximate branching factor of $3 * \frac{16}{6} * 6 = 48$ per player turn taken. Since each player takes a turn during a round, we have a branching factor of $48^4 = 5308416$ per round. If we also consider the colonist pick phase, we have an approximate branching factor of $5308416 * 720 = 3822059520$ for a complete round — nearly four billion outcomes. For reference, game length usually ranges between from about 15 to 40 rounds. [3].

[3]We did not conduct exact measurements for game length, these values are simply approximate observations.

3. AI Framework

This chapter contains a high-level overview of the AI Framework in *Colonizers*. More concrete descriptions are available in Attachment A.1 and Attachment A.2.

3.1 Design

Firstly, the choice of technologies used for creating *Colonizers* warrants some explanation. The architecture of the application consists of three layers:

- *Game Engine*. The game's backend is implemented in C# and targets .NET Core 3.1. The game logic itself is separated into a library, and this library is then hosted behind a web API with ASP.NET Core.
- *UI Layer*. The UI for the application is a single-page application (SPA) made with Angular.
- *AI Scripts*. The AI scripts are implemented in Python 3.7. The project contains a base class `AIBase` for other AIs. This base class provides the AIs with the necessary API for communicating with the game engine.

The application is then hosted in Electron in order to run as a desktop application. This is facilitated by the C# library Electron.NET, which allows the hosting of ASP.NET Core applications inside Electron. There are multiple reasons why we chose Electron instead of a more traditional GUI like WPF or WinForms, and instead of rendering graphics for the game ourselves:

- Electron is cross-platform. The attentive reader has probably noticed that all three of the aforementioned components are made with cross-platform technologies, and this is very much by design. In principle, nothing prevents *Colonizers* from being cross-platform, which is advantageous considering many AI researchers have Linux as their primary platform. In practice however, *Colonizers* only supports Windows due to the way that communication between the game engine and AI scripts is implemented. This communication channel could be easily replaced with a cross-platform solution, in fact the classes which are responsible for communication could easily be swapped out. The only reason why this has not been done is simply prioritization — the application has other features which had a higher priority.
- Electron provides easy ways of bundling and installing applications. It has a convenient installer which makes installing the application easy for end users.
- Electron is a stable and tested technology, which is used by many widely-used applications.
- A turn-based board game lends itself well to being drawn in a web page and being controlled by a SPA. If *Colonizers* were an action game or a real-time strategy game, this solution would no longer be viable.

C# was chosen for the implementation of the game engine because it is a language well-suited for writing backends. There are many solid libraries, and the language is fast thanks to a well-optimized JIT (Just In Time) compiler. It would have been somewhat easier to implement the game's backend in Python, since the communication between the game engine and the AI scripts would have been trivial. However, C# was chosen mostly because it is a statically-typed language, which offers protections against many types of bugs and errors by catching them at compile-time. When writing game logic, many errors are prevented simply by clever usage of types, as opposed to them coming out during runtime with Python.

The choice of Python for AI scripts was an easy one, considering most machine learning is done with Python today. It is also a rather easy language to learn and understand.

Angular was chosen as the SPA framework because it is a popular and robust solution. Its state management system capabilities lend themselves well to implementing a UI with many pieces which depend on one another in complex ways. One downside of Angular is that since it is a web application framework, its real-time performance is less than ideal. This negative is essentially void when applied to *Colonizers* however, since our game is a turn-based board game. Therefore our focus when choosing a framework was mainly on having a robust solution which is easy to develop and maintain. There are multiple such frameworks, notably React and Vue, which would have also been viable choices. However, Angular is simply the SPA framework the author is the most familiar with.

Another design consideration is the design of the API used to communicate between the game engine and AI scripts. The chosen API is subclassing — a new AI must simply subclass a provided base class (`AIBase`), and implement an abstract method. The AI script is then started in a separate process by the game engine, and the implemented method is called when the game engine requires the AI to make a move. The AI then chooses a move, and returns an identifier corresponding to the chosen move. This API is simple and easy to understand, as well as easy to implement.

3.2 Interface

The API for communication consists of the AI subclassing the `AIBase` class, and implementing the `messageCallback(self, gameState)` abstract method. This method is invoked by the game engine when it requires a move to be chosen by the AI. The `gameState` parameter contains a dictionary representing the current game state, as well as a list of available actions in this state.

The AI base class also contains various utilities for AI scripts to use. The most important of these utilities is the `simulate(self, boardState, move)` method. This method will simulate the given move via the game engine, and return the new game state. This allows AIs to simulate playouts without having to implement them internally. Another useful method is `determinize(self)`, which returns a determinized version of the current game state. This determinization is performed by the game engine. The AI also does not need to track information sets, since the game engine does this internally. This means that for example if the AI is playing second and choosing a colonist, the game engine will automatically

track the information the AI has about the previous player's colonist (one of two possible colonists). Other utility methods included simplify working with the game state dictionary by providing methods for commonly performed operations.

Figure 3.1 shows the implementation of an AI for *Colonizers*. This AI simply chooses random moves.

```
from AICore import *

import sys
from random import seed, randint

class RandomAI(AIBase):
    def __init__(self):
        super().__init__()

    def messageCallback(self, gameState):
        # important to return string, not number
        return str(self.pickRandomAction(gameState))

    def pickRandomAction(self, gameState):
        actionCount = len(gameState["Actions"])
        return randint(0, actionCount - 1)

if __name__ == "__main__":
    if len(sys.argv) != 2:
        # AI Script must have 1 argument - name of named pipe
        raise Exception('Invalid arguments')
    seed(42) # Seed AI for reproducibility
    ai = RandomAI()
    ai.run(sys.argv[1])
```

Figure 3.1: Random AI implementation.

As seen in Figure 3.1, creating a new AI is very simple. About half of the shown code is boilerplate code, and the actual AI class is very simple.

Note that the game engine invokes the AI script via command line, which means that the AI must contain the `__main__` boilerplate code.

3.3 Adding new AIs

There are two supported kinds of AI scripts: a stand-alone Python file, and a folder with potentially multiple Python files and a complicated subdirectory structure. The second option exists to facilitate more complicated AIs, where implementing the whole AI in a single Python file would not be reasonable.

The game looks for AI files in an internal directory inside its installation folder. While it is possible to add and remove scripts this way, the GUI provides a way to do this easily. In order for an AI script to be recognized by the game, it must not only be located in the internal directory, but it must also follow certain conventions.

For stand-alone Python files, the only convention is that the file must follow the naming convention of `<Name>Intelligence.py`. An example of this is `RandomIntelligence.py`.

For folder-based AIs, the folder itself must follow the naming convention of `<Name>Intelligence`, for example `NestedIntelligence`. This folder must also directly contain a file named `main.py`, which will be used as the AI's entrypoint. All added AI files must also contain a `__main__` function, since they are invoked as the entrypoint. See Figure 3.1 for an example of how to add a new AI.

The AI needs to reference the `AICore.py` file in some way, since it contains the AI base class. This file is contained in the aforementioned internal directory, which means that stand-alone AI scripts placed in this directory can import the file without issues. The problem comes with folder-based AIs, since Python unfortunately does not provide a convenient way to import files which are higher in the file hierarchy. Therefore when copying AI folders with the GUI, the `AICore.py` file will automatically be copied into the new AI's directory. This means that folder-based AIs for *Colonizers* cannot contain a `AICore.py` file in the top level of their folder, since the file would be overwritten during copying. Alternatively, the folder can be manually copied into the game files. In that situation however, the `AICore.py` file must be copied manually as well.

4. Used Algorithms

Colonizers has four different kinds of AI implemented out-of-the-box. This chapter describes their implementations, and discusses the design decisions taken when creating them.

4.1 Random Decisions

The random decision algorithm is rather primitive — when it is presented with a choice of actions, it simply picks a random one. It is meant to be the bottom baseline for other algorithms, as well as being a proof-of-concept. Figure 4.1 shows the AI class which implements this logic.

```
class RandomAI(AIBase):
    def __init__(self):
        super().__init__()

    def messageCallback(self, gameState):
        # important to return string, not number
        return str(self.pickRandomAction(gameState))

    def pickRandomAction(self, gameState):
        actionCount = len(gameState["Actions"])
        return randint(0, actionCount - 1)
```

Figure 4.1: Random choice algorithm.

An interesting property of this AI is the fact that if four of them play against each other, it is possible for the game to never end, since the random decision making does not have to converge towards an end state. This situation is extremely unlikely however.

4.2 Heuristics

The heuristic AI is intended to be the real baseline for other implemented AI algorithms. It consists of a number of rules which determine the action to perform in a given game state. If no rules are applicable to a given state, the AI simply falls back to random choice. First, let us reason about why certain heuristic rules were chosen and what they mean.

Colonist Pick Phase

Colonists with active abilities have an inherent risk associated with them, since they might either hit an unintended target with their ability, or they might miss altogether. This is why their payoffs are higher that other colonists — it represents a payoff for the risk taken. The approach we took with the colonist pick

heuristics is one that makes consistently good decisions — going for synergy-based strategies and getting guaranteed value. For example, picking the General when the player has four red modules in their colony is not only a safe play, but it also provides even more value than the risk-based colonists, were they to hit their intended target.

An argument could be made for picking risk-based colonists more often when the player is in positions 3 or 4, since the earlier players reveal their colonist on their turn, essentially removing the risk portion of the colonist. However, in many situations players do not hold a large amount of cards or Omnium for a long time. Since players are incentivized to be able to build eight modules before the game ends by bonus points, players often build modules as soon as it is possible. This strategy also reduces the risk of being targeted by other players. In summary, we considered such rules to be of little use, considering the situations where they apply are so rare.

Draw Phase

The draw phase is not an especially deep part of the game, since both actions have reasonable value at most times. We simply added a few rules which prevent the AI from either drawing too many cards, or having too much Omnium. These resources are worth relatively little when the player is unable to spend them.

Discard Phase

Since the bonus for building eight modules is non-trivial, we added a few rules across the game phases to ensure that if the game is about to end, the AI will do its best to reach eight buildings as fast as possible. This will usually have the AI keeping cheap cards, and disregarding color synergy in favor of point efficiency.

If such a situation is not near however, we implemented a rule for creating color synergies. Colored synergies are an important part of the game, since they can provide players with a steady income of Omnium. Therefore the heuristic AI tries to go for color synergies with its modules before the late game.

Power Phase

Only two colonists in the game have interactions in the power phase, but it is a very important part of gameplay nonetheless. These two colonists can be extremely powerful and they can cause massive swings in tempo [1].

Both decision making rules for these colonists are similar in nature — they attempt to find the target with the highest expected return for the power usage. This is done by counting the potential reward (resources of player we are trying to target) and subtracting from that the measure of uncertainty. The less information we have about the target's colonist, the more risk is involved in the play, and the expected return is lower. If the best expected return is small enough, the AI passes its turn. This is because of the risk of hitting an unintended target and actually making its own situation worse than if it had simply passed.

[1] Tempo is an unofficial term in card games referring to the flow of the game. If a single player takes the lead early and stays in the lead consistently, we would call that a play with a lot of tempo.

Build Phase

The rules for the build phase are in essence similar to those in the discard phase. This is not a coincidence, in fact the discard phase and the build phase are probably the most tightly coupled phases in the game. Since there is a risk of having modules stolen from the hand by a Spy if the player lets the modules sit in his hand, many players will try to keep as few resources possible. They will instead try to spend them as fast as possible, since modules in the colony are theirs permanently. Things we discussed in the discard phase apply here as well.

Figure 4.2 shows high-level pseudocode for the implemented heuristic AI.

```
if game phase is "ColonistPick":
    if player has at least 3 modules of the same color in their colony:
        pick color synergy colonist if available
    if player has 0-1 modules in hand:
        pick Visionary if available
    pick randomly
else if game phase is "Draw":
    if player has 0 Omnium or at least 4 modules in hand:
        acquire Omnium
    if player has 0 modules in hand:
        draw modules
    pick randomly
else if game phase is "Discard":
    if any player has 7 or 8 modules in their colony:
        keep the highest value module the player can afford
    if the player has 5+ modules in their colony:
        keep the module with the highest difference of value − cost
        that the player can afford
    keep the module with the most color synergy with the player's
    colony if possible

    pick randomly
else if game phase is "Power":
    if player's colonist is Opportunist:
        choose the most valuable player to steal from,
        where value is calculated as
        Omnium − number of possible colonists for that player + 1
        randomly choose a colonist this player could have,
        then steal from this colonist
    if player's colonist is Spy:
        find player with more cards than the current player
        and with sufficient information about their colonist
        if this player exists:
            swap hands with them
        else:
            do nothing
    pick randomly
else if game phase is "Build":
    if any player has 7 or 8 modules in their colony:
        build the module with the highest possible value
    if the current player has 5+ modules:
        build the module with the highest difference of value − cost
    build the module with the most color synergies if possible

    build randomly
```

Figure 4.2: Heuristic algorithm pseudocode.

4.3 MaxN

The principles of the MaxN algorithm have been discussed in Section 1.2.1. However, the algorithm as presented in Section 1.2.1 would not be applicable to a game with hidden information. Therefore this section will focus mainly on describing the adaptations made in order for it to function in an environment with imperfect information.

As adaptation to imperfect information, we used determinization to transform imperfect information states into perfect information states. Figure 4.3 shows a code excerpt with the determinization usage. The adapted algorithm samples `DETERMINIZE_COUNT` determinizations, and runs the regular MaxN algorithm for that determinization. This means that during each determinization, our modified MaxN algorithm behaves exactly as the existing MaxN algorithm would. It traverses a perfect-information game tree and uses a positional evaluation function to decide the optimal moves. Just like the regular MaxN, for each determinization, we get a final suggested move from the algorithm. The moves suggested by these runs are saved, and after all sampled determinizations are processed, the move chosen by the adapted algorithm is the move which was chosen as the best move the most often by the regular MaxN runs.

```
def pickAction(self, actualGameState):
    # Small optimization when we don't have a choice
    if len(actualGameState["Actions"]) == 1:
        return 0

    # Counter for number of times the given action was the best
    actionValues = [0 for x in range(len(actualGameState["Actions"]))]
    for i in range(self.DETERMINIZE_COUNT):
        gameState = self.determinize()
        rootNode = MaxnNode(gameState, 0)
        payoffs = self.maxnPayoffs(rootNode, self.SEARCH_DEPTH)

        bestIndex = max(range(len(payoffs)), key=lambda i: payoffs[i])
        actionValues[bestIndex] += 1

    return max(range(len(actionValues)), key=lambda i: actionValues[i])
```

Figure 4.3: Determinization used for MaxN.

Values for the mentioned constants were selected as `DETERMINIZE_COUNT = 10` and `SEARCH_DEPTH = 7`. These were empirically selected to allow for decent response time when a move is requested.

We also have not mentioned the used positional evaluation function that we used in the implementation of `MaxnIntelligence`. The chosen evaluation function is rather simple — it tries to maximize the amount of points that the player has. It assigns the highest value to building modules, and to having the bonus for completing all eight buildings in the player's colony. This bonus, along with victory values of modules in the player's colony, count 1:1 towards the heuristic evaluation. Somewhat less value is given to having modules in the hand, and

having Omnium, with their exact value being such that the value gained by having a module built is always greater than having it in the hand, and having the Omnium to build it. Figure 4.4 shows the code for evaluation of a single player.

```
def evaluatePlayer(player):
    value = 0

    for module in player["Colony"]:
        value += module["VictoryValue"]
    value += 0.6 * len(player["Hand"])
    value += 0.3 * player["Omnium"]
    if len(player["Colony"]) == 8:
        value += 4

    return value
```

Figure 4.4: Positional evaluation function used for MaxN.

Our adapted version of MaxN still has some potential issues however. Let us consider the issue of *non-locality* — the problem whereby certain game states are explored in spite of the fact that they will most likely be avoided by players. An example of this could be the following: our MaxN algorithm is picking a colonist, and after it picks, the Miner is among the colonists passed on to other players. Let us also assume that the other players have no blue modules built among them. In this situation, we might explore determinizations where a player has picked Miner as their colonist — even if this would be the suboptimal play, since other colonists have more useful abilities in this scenario.

We can also consider a situation where the best play would be denying an opponent an opportunity, rather than gaining value ourselves. An example of this in *Colonizers* could be the following situation: our MaxN algorithm is picking a colonist, and it sees that all players except one have full hands, but one player's hand is empty. For the player with an empty hand, picking Spy as their colonist would be a very high value play, since it could mean up to a 10 card swing in their favor. A possible strategy to consider would be for MaxN to pick Spy itself, to prevent the other player from making this high value play. However, our version of MaxN cannot explore the game tree with enough depth to be able to make these kinds of plays.

4.4 Information Set Monte Carlo Tree Search

The chosen variant of ISMCTS is SO-ISMCTS. This variant's pseudocode has already been shown in Figure 1.5 Since the actual implementation is mostly faithful to the pseudocode presented in the original ISMCTS paper [9], we will not be repeating the pseudocode in this section. Many of the algorithm's strengths and weaknesses have also been discussed in Section 1.2.2, therefore this section will mostly highlight some implementation details.

First of all, the choice of SO-ISMCTS warrants some explanation. Cowling, Powley and Whitehouse also presented two other variants of ISMCTS: SO-

ISMCTS + POM and MO-ISMCTS [9]. SO-ISMCTS + POM (Single Observer Monte Carlo Tree Search + Partially Observable Moves) aims to solve the issue of strategy fusion [2]. A partially observable move is a move which is observable by the player, but there is some hidden information associated with the move. SO-ISMCTS can be vulnerable to this, since it treats all opponents' moves as fully observable. SO-ISMCTS + POM alleviates this issue by making actions which are indistinguishable from the point of view of the player share a single edge in the tree. It does this at the cost of significantly weakening the opponent model, since it makes the assumption that the opponent chooses randomly between moves indistinguishable to the player. MO-ISMCTS (Multiple Observer Information Set Monte Carlo Tree Search) addresses this issue by maintaining a separate tree for each player, representing information sets from the point of view of that player.

The reason SO-ISMCTS was chosen over the two other variants is because in *Colonizers*, strategy fusion is not as big of a problem as it might seem. This is because partially observable actions like drawing cards or swapping hands often lead to the same strategy being played regardless. The only potential problematic area is the colonist pick phase, because using active colonist abilities then creates an instance of strategy fusion. SO-ISMCTS was ultimately chosen since it has the benefit of being simpler, and the choice is a conscious tradeoff with respect to the identified instance of strategy fusion.

SO-ISMCTS uses a multi-armed bandit algorithm to balance exploration and exploitation when traversing the information set tree. In our implementation, we have chosen UCB1 as the multi-armed algorithm. Kuleshov and Precup [14] note that "UCB1 achieves the optimal regret [3] up to a multiplicative constant, and is said to solve the multi-armed bandit problem". UCB1 calculates the score of a node as

$$\overline{X}_j + c\sqrt{\frac{\ln n}{n_j}}$$

where $\overline{X}_j$ is the average reward of the playouts which passed through the node j, n is the count of visits of the parent of j, and n_j is the number of times the node j was selected during the algorithm. c denotes the exploration constant, and for our implementation, we chose a value of 0.7, which was found to be a reasonable value by Cowling, Powley and Whitehouse [9].

Our implementation of SO-ISMCTS also uses the heuristic algorithm described in Section 4.2 to perform game playouts during the simulation stage of the algorithm. This helps achieve more accurate playouts, and therefore more accurate rewards than when using random playouts.

The number of iterations we use for ISMCTS is 200. This is an empirically selected value which allows moves to be computed within a reasonable amount of time (tens of seconds at most), while having enough room to properly expand the tree. We also considered imposing a hard time limit on the AI's decision time, and simply running iterations of ISMCTS until the time expired. We chose not to do this for the sake of the AI behaving consistently on different machines.

[2] Strategy fusion is discussed at more length in Section 1.2.2.

[3] Regret in this context means not choosing the optimal decision in retrospect.

5. Experiment Description

There are two qualities which we want to analyze with respect to the game and the implemented AI algorithms:

- Identify potential asymmetries in game balance
- Compare methodologies used by the AI algorithms

To this end, we conducted five experiments, split according to their purpose. The following sections elaborate on the experiments and their results.

5.1 Game Balance Experiments

As mentioned in Chapter 2, the game features a degree of asymmetry. The order in which players take their turns inherently changes the viability of certain strategies, because players in different positions have different information sections available to them. For example, the player in the first position has perfect information about which colonist was removed from play during the colonist pick phase, while the second and third players do not have such certainty.

Most importantly however, a player's colonist is revealed at the start of their turn. This means that if the player in the fourth position is a Spy or an Opportunist, they will know all the other players' colonists when their turn comes around. This means that this player will be able to target any player with their targeted ability without the fear of missing or hitting an unintended target.

With these things in mind, we can hypothesize that players in the earlier positions have an easier time achieving synergy-based strategies, since they get priority when picking colonists. On the other hand, we can also hypothesize that players in later positions will benefit from play based around using targeted colonist abilities.

In Chess, it is widely agreed that the white player has an advantage [15]. Similarly, we aim to discover whether player ordering confers a measurable advantage to any player in *Colonizers*. We will conduct this experiment with the null hypothesis — we assume that there is no significant advantage for any player ordering.

5.1.1 Description

We conducted two experiments in this section. In both of them, four identical AIs played 1000 games against each other. In the first experiment, the AI in question was `RandomIntelligence`, and in the second experiment it was `HeuristicIntelligence`.

All random events were seeded, and the results of the games were captured in JSON (JavaScript Object Notation) files. The results were then parsed and analyzed. The JSON result files can be found in the attached source code, refer to Attachment A.2.6 for more information on their location and semantics.

The random seeds used by application components during the experiment were as follows. Note that the chosen seeds do not have any special meaning, and they were selected at random by the author. There is no particular reason for the algorithms to have different seeds.

- `RandomIntelligence`: seed 42
- `HeuristicIntelligence`: seed 97
- `GameConstants`: seed was changed every game to prevent the same game from being played 1000 times. The seeds were generated by a C# random number generator seeded with 42.

The algorithms' positions were shuffled at the beginning of each game with the Fisher-Yates Shuffle [16], using the game engine's random number generator. In this experiment this is not necessary since we have four instances of the same algorithm. This experiment was performed using the same method as other experiments where the shuffling is necessary, therefore the shuffling is present here. We mention this since in order to exactly reproduce the experiment, the shuffle must be performed with the game engine's random number generator.

5.1.2 Findings

Experiment 1

First off, let us focus on the experiment runs with `RandomIntelligence`. Results of the 1000 runs can be seen in table 5.1 Note that by "Losses" we mean fourth-place finishes, not failing to finish first.

Position	**1**	**2**	**3**	**4**
Wins	310	213	251	226
Losses	197	261	279	263
Average rank	2.3	2.572	2.553	2.575

Table 5.1: Results of `RandomIntelligence` play.

The most notable result we have is the fact that AIs in the first position seem to be winning the most often. AIs in the first position also lose (place fourth) less, and they have a better average ranking overall.

We can try to verify the significance of these results mathematically. We will employ the χ^2 test to check whether the number of wins for position 1 follows

the binomial distribution $B(1, 0.25)$. The null hypothesis in this case is that the win rate follows the aforementioned binomial distribution. With 1000 trials, we would expect 250 of them to succeed and 750 of them to fail. We can compute χ^2 as follows

$$\chi^2 = \frac{(310 - 250)^2}{250} + \frac{(690 - 750)^2}{750} \approx 19.2$$

This gives us the distribution $\chi^2(1)$ with 1 degree of freedom. Our χ^2 test statistic is 19.2, which gives us a p-value of 0.00001. If we consider a significance level of 0.05, we can reject the null hypothesis.

This may indicate a potential balance issue in the rules of the game, with the first position being more powerful than the other ones, which appear to be similar in power. However, measurement on randomly choosing AIs does not necessarily indicate imbalance, since random agents do not play optimal strategies. Therefore we cannot conclude anything about game balance just yet, but this statistical difference is worth keeping in mind.

Experiment 2

The other experiment in this section is very similar to the first one, except we have four instances of `HeuristicIntelligence` instead of four instances of `RandomIntelligence` playing against each other. Results of the 1000 runs can be seen in table 5.2. Note that by "Losses" we mean fourth-place finishes, not failing to finish first.

Position	**1**	**2**	**3**	**4**
Wins	230	202	282	286
Losses	415	298	152	135
Average rank	2.8	2.67	2.302	2.228

Table 5.2: Results of `HeuristicIntelligence` play.

If we compare these results to those in table 5.1, we can see almost exactly the opposite results. With random AIs playing, we saw that the AI in the first position had a statistically significant advantage. With heuristically driven AIs, it is obvious on first glance that earlier positions are less powerful and later positions are more powerful. We can verify this statistically by performing the χ^2 test on the number of wins and losses for positions 1 and 4, similarly to Experiment 1.

We can start with wins for position 1

$$\chi^2 = \frac{(230 - 250)^2}{250} + \frac{(770 - 750)^2}{750} \approx 2.14$$

This gives us a p-value of 0.14350. If we consider a significance level of 0.05, we cannot reject the null hypothesis that the number of wins for the first position follows the distribution $B(1, 0.25)$.

If we move on to wins for position 4, we proceed as follows

$$\chi^2 = \frac{(230 - 250)^2}{250} + \frac{(770 - 750)^2}{750} \approx 6.912$$

This gives us a p-value of 0.00856, therefore we can reject the null hypothesis if we consider a statistical significance of 0.05.

We can also analyze the loss statistics, starting with position 1

$$\chi^2 = \frac{(415 - 250)^2}{250} + \frac{(585 - 750)^2}{750} \approx 145.2$$

This gives us a p-value of nearly 0, therefore we can reject the null hypothesis when considering a statistical significance of 0.05.

For losses at position 4, we have

$$\chi^2 = \frac{(135 - 250)^2}{250} + \frac{(865 - 750)^2}{750} \approx 70.54$$

This also gives us a p-value of nearly 0, therefore we can again reject the null hypothesis when considering a statistical significance of 0.05.

While the differences between wins per position are notable, the most interesting are the loss statistics. It would appear that the earlier ranks (particularly the first one) are susceptible to being targeted by players in other ranks. Since every player's colonist is revealed at the start of their turn, this makes the first position an easy target for all other players. The game does have counter-balances for this situation — notably the fact that the first player to build their colony to full gets four extra victory points. However, it would seem that the heuristic AI does not have the necessary tools to deal with being targeted down by others. This could possibly be due to an implementation bias inherent in the chosen heuristics, but it could also signal a game balance issue.

5.2 Algorithm Comparison Experiments

We have implemented four algorithms in this thesis — `RandomIntelligence`, `HeuristicIntelligence`, `MaxnIntelligence` and `ISMCTSIntelligence`. In order to determine the qualities of said algorithms, we will analyze their differences, along with their advantages and disadvantages. We will also look at how the algorithms perform in play against each other, with the hopes of determining which algorithm is the most suitable for a game like *Colonizers*.

To start with, we would not expect `RandomIntelligence` to perform well in any kind of mutual play. It is present simply as a benchmark for the performance of other AIs.

The more important benchmark is `HeuristicIntelligence`, since it represents rules which were created by observing humans play the game [1]. Therefore we consider this AI to be a minimum benchmark for other AIs to be competent.

`MaxnIntelligence` is based on the MaxN algorithm [6], which is itself based on Minimax [4]. This AI was adapted for imperfect information games, and it spends a non-trivial amount of computing power on simply exploring possible determinizations of the current game state. If we take that into consideration, along with the fact that the branching factor for *Colonizers* is non-trivial [2] , we

[1] The rules were designed by the author after the author played the game several times with his friends, who had not played a similar game before

[2] This is explored in depth in Section 2.3.

would expect `MaxnIntelligence` to perform relatively poorly. The depth of the search trees used could not be reasonably increased beyond 7, due to performance concerns. We expect that any kind of long-term strategy could not be achieved by it since it lacks the necessary exploration depth. The Minimax family of algorithms does however offer very solid insight into the few turns it examines, therefore we hypothesize that this AI will be primarily good at tactics-based play. The performance of `MaxnIntelligence` against `HeuristicIntelligence` is uncertain, therefore we will follow the null hypothesis and assume that their performances are statistically similar. We also hypothesize that since this AI has a strong foundation for tactical prowess, it should win more often when in later positions (namely third and fourth).

The final AI tested is `ISMCTSIntelligence`. This AI is well-adapted to imperfect information and multiple player environments. Therefore we would expect it to outperform the three aforementioned AIs in most situations. We expect it to play well in most circumstances, regardless of player permutation.

5.2.1 Description

We conducted three experiments in this section. In the first experiment, one of each implemented intelligence played 50 games against each other. This experiment is meant to assess the general playing ability of the AIs. In the second experiment, we let two instances of `HeuristicIntelligence` and two instances of `MaxnIntelligence` play against each other for 50 games. Lastly in the third experiment, we performed the same thing as in the second experiment, but we replaced `MaxnIntelligence` instances with `ISMCTSIntelligence` instances. These two experiments are meant to benchmark the adapted algorithms against the heuristic solution.

All random events were seeded, and the results of the games were captured in JSON files. The results were then parsed and analyzed. The JSON result files can be found in the attached source code, refer to Attachment A.2.6 for more information on their location and semantics.

The random seeds used by application components during the experiment were as follows. Note that the chosen seeds do not have any special meaning, and they were selected at random by the author. There is no particular reason for the algorithms to have different seeds.

- `RandomIntelligence`: seed 42
- `HeuristicIntelligence`: seed 97
- `MaxnIntelligence`: seed 99
- `ISMCTSIntelligence`: seed 15
- `GameConstants`: seed was changed every game to prevent the same game from being played 1000 times. The seeds were generated by a C# random number generator seeded with 42.

The algorithms' positions were shuffled at the beginning of each game with the Fisher-Yates Shuffle [16], using the game engine's random number generator. We mention this since in order to exactly reproduce the experiment, the shuffle must be performed with the game engine's random number generator.

5.2.2 Findings

Experiment 3

In Experiment 3, we had one of each type of AI play against each other in 50 games. Results of the 50 runs can be seen in table 5.3 Note that by "Losses" we mean fourth-place finishes, not failing to finish first.

AI	Random	Heuristic	MaxN	ISMCTS
Wins	0	5	8	37
Losses	40	1	9	0
Average rank	3.8	2.38	2.54	1.28

Table 5.3: Results with one of each AI.

We can immediately see that there is no point in statistically checking whether all AIs are equally good, since the results are so extreme. ISMCTS is the clear winner, having won the majority of games and not having lost a single one. An interesting data point is of the 13 instances where ISMCTS didn't win, it places second 12 times and third 1 time. The more interesting part is the comparison of the heuristic AI with MaxN. It would appear that the heuristic AI is much less likely to lose than MaxN. This makes sense, considering that the heuristic AI is a collection of rules designed to always move the AI towards gaining score, even if it is not optimal.

We can take a look at position-based data in order to gain insight into which AIs are strong in which positions. Table 5.4 shows positions where the AIs scored their wins.

AI	Random	Heuristic	MaxN	ISMCTS
Position 1	0	2	5	11
Position 2	0	1	2	6
Position 3	0	2	1	9
Position 4	0	0	0	11

Table 5.4: Positions of wins in Experiment 3.

We can try to check whether AIs are similarly likely to win in any position. We will not perform the statistical test on the results for the heuristic or MaxN AIs, since the sample size for wins is too small. We can, however, perform the χ^2 test for ISMCTS wins by position. As the null hypothesis, we will assume that the AIs are equally likely to win in any position. We can check the hypothesis that the win distribution for ISMCTS between positions follows the distribution $B(1, 0.25)$. ISMCTS won 37 times in total, giving us an expected 9.25 wins per position.

$$\chi^2 = \frac{(6 - 9.25)^2}{9.25} + \frac{(31 - 27.75)^2}{27.75} \approx 1.52$$

This gives us a p-value of 0.21762, therefore we cannot reject the null hypothesis if we consider a statistical significance of 0.05. This means that we cannot reject the hypothesis that ISMCTS is equally likely to win in any position.

Experiment 4

In Experiment 4, we had two instances of `HeuristicIntelligence` and two instances of `MaxnIntelligence` play against each other in 50 games. Results of the 50 runs can be seen in table 5.5 Note that by "Losses" we mean fourth-place finishes, not failing to finish first.

AI	**Heuristic**	**MaxN**
Wins	35	15
Losses	10	40
Average rank	2.13	2.87

Table 5.5: Results with heuristic AI versus MaxN.

We can use the χ^2 test to check whether there is a statistical difference between these AIs' likelihoods to win. We will assume that the heuristic AI's wins follow a binomial distribution $B(1, 0.5)$. Then

$$\chi^2 = \frac{(35-25)^2}{25} + \frac{(15-25)^2}{25} = 8$$

This gives us a p-value of 0.00468. We are considering a statistical significance of 0.05, therefore we can reject the null hypothesis. As a consequence, we have established that in this scenario, the heuristic AI performs significantly better than MaxN.

Since we earlier hypothesized that MaxN would be more likely to win in later positions, we can also look at positional results as shown in Table 5.6

AI	**Heuristic**	**MaxN**
Position 1	13	13
Position 2	15	2
Position 3	4	0
Position 4	2	0

Table 5.6: Positions of wins in Experiment 4.

The results here are interesting in multiple ways. The first is the fact that our earlier hypothesis that MaxN would be strong at playing later positions was completely wrong, with MaxN not achieving a single win in positions 3 or 4.

The other way these results are interesting is the fact that in position 1, both AIs had the same likelihood of winning, even though MaxN is obviously statistically inferior to its heuristic counterpart. We do see a similar trend to the results of Experiment 3, where most wins seem to happen at earlier positions. This could potentially indicate the existence of overpowered strategies available only to players in early positions.

In summary, results for Experiment 4 were rather surprising, considering we rejected both of our prior hypotheses based on the experiment results.

Experiment 5

In Experiment 5, we had two instances of `HeuristicIntelligence` and two instances of `ISMCTSIntelligence` play against each other in 50 games. Results of the 50 runs can be seen in table 5.7 Note that by "Losses" we mean fourth-place finishes, not failing to finish first.

AI	**Heuristic**	**ISMCTS**
Wins	5	45
Losses	48	2
Average rank	3.26	1.74

Table 5.7: Results with heuristic AI versus ISMCTS.

We can immediately see that there is not much point in examining these results statistically, since it is obvious that ISMCTS is significantly better. We will therefore analyze the positional results as shown in Table 5.8 in more detail, since they paint a less obvious picture.

AI	**Heuristic**	**ISMCTS**
Position 1	2	9
Position 2	2	13
Position 3	0	14
Position 4	1	9

Table 5.8: Positions of wins in Experiment 5.

We will perform a χ^2 test to check whether the positional wins for ISMCTS follow a binomial distribution $B(1, 0.25)$, focusing on wins in position 3

$$\chi^2 = \frac{(14 - 11.25)^2}{11.25} + \frac{(31 - 33.75)^2}{33.75} \approx 0.896$$

This gives us a p-value of 0.3439. If we consider a statistical significance of 0.05, we cannot reject the null hypothesis. This means that we cannot reject the hypothesis that ISMCTS is equally likely to win in any position.

The results of Experiment 5 are consistent with the hypotheses we had prior to performing the experiment. We confirmed that the performance of ISMCTS is much better than that of the heuristic algorithm.

Conclusion

We have implemented a multi-player board game with imperfect information elements. The game also features an AI framework, which makes it easy to add new AI algorithms. This provides an environment where both new and existing algorithms can be compared.

By implementing several AI algorithms using the aforementioned framework, we have verified that the designed API for AI algorithms is easy to use and contains all the needed functionality.

In the experimental part of this thesis, we identified potential design flaws in the game rules. Namely we speculate that players in earlier positions have an advantage, much like the player going first has an advantage in Chess. We also confirmed the hypothesis that the ISMCTS algorithm is well-suited for solving games like *Colonizers*.

Future work includes writing new AI algorithms for the game, and further analysis of how balanced the game is.

Bibliography

[1] David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 01 2016.

[2] boardgame.io [online]. `https://boardgame.io`. Accessed: 2020-05-31.

[3] Openai gym [online]. `https://gym.openai.com`. Accessed: 2020-05-31.

[4] I. Millington and J. Funge. *Artificial Intelligence for Games*. Second Edition. Morgan Kaufmann, 2009.

[5] K. Hoki and T. Kaneko. Large-scale optimization for evaluation functions with minimax search. *Journal of Artificial Intelligence Research*, 49:527–568, 2014.

[6] C. A. Luckhardt and K. B. Irani. An algorithmic solution of n-person games. In *AAAI*, 1986.

[7] G. Chaslot. *Monte-Carlo Tree Search*. PhD thesis, Maastricht University, 2010.

[8] A. Slivkins. Introduction to multi-armed bandits. *Foundations and Trends in Machine Learning*, 12(1-2), 2019.

[9] P. I. Cowling, E. J. Powley, and D. Whitehouse. Information set monte carlo tree search. *IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES*, 4(2), 2012.

[10] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2009.

[11] D. Whitehouse, E. Powley, and P. Cowling. Determinization and information set monte carlo tree search for the card game dou di zhu. In *2011 IEEE Conference on Computational Intelligence and Games, CIG 2011*, pages 87–94, 10 2011.

[12] I. Frank and D. Basin. Search in games with incomplete information: A case study using bridge card play. *Artificial Intelligence*, 100(1-2):87–123, 1998.

[13] Z. Michalewicz and D. B. Fogel. *How to Solve It: Modern Heuristics*. Springer, second edition, 2004.

[14] V. Kuleshov and D. Precup. Algorithms for the multi-armed bandit problem. *Journal of Machine Learning Research*, 1(48), 2000.

[15] W. F. Streeter. Is the first move an advantage? *Chess Review*, page 16, 05 1946.

[16] D. E. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, third edition, 1998.

[17] Electron.net [online]. `https://github.com/ElectronNET/Electron.NET`. Accessed: 2020-06-04.

List of Figures

List of Tables

List of Abbreviations

AI Artificial Intelligence
API Application Programming Interface
DI Dependency Injection
GUI Graphical User Interface
ISMCTS Information Set Monte Carlo Tree Search
JIT Just In Time (Compiler)
JSON JavaScript Object Notation
MCTS Monte Carlo Tree Search
POM Partially Observable Moves
REST Representational State Transfer
SPA Single Page Application
SO-ISMCTS Single Observer Monte Carlo Tree Search
MO-ISMCTS Multiple Observer Monte Carlo Tree Search
UCB1 Upper Confidence Bounds 1 (Algorithm)
UI User Interface

A. Attachments

A.1 User Documentation

This attachment serves as a guided tour of the game and its features. It is written in such a way that it can be understood even by persons without a technical background. These persons may wish to simply play the game without necessarily developing AI for it, and this attachment is meant to give them the necessary knowledge.

A.1.1 Requirements

To run *Colonizers*, the Windows 10 operating system is required. Also required is the following software:

- .NET Core 3.1 Runtime
- ASP.NET Core 3.1 Runtime
- Python 3.7

The game's UI is designed for a minimum screen resolution of 1920x1080 at 100% zoom level. It is not recommended to play the game on lower resolution screens, since graphical errors may occur.

A.1.2 Installation

Colonizers is distributed via an installer application, which is shown in Figure A.1.

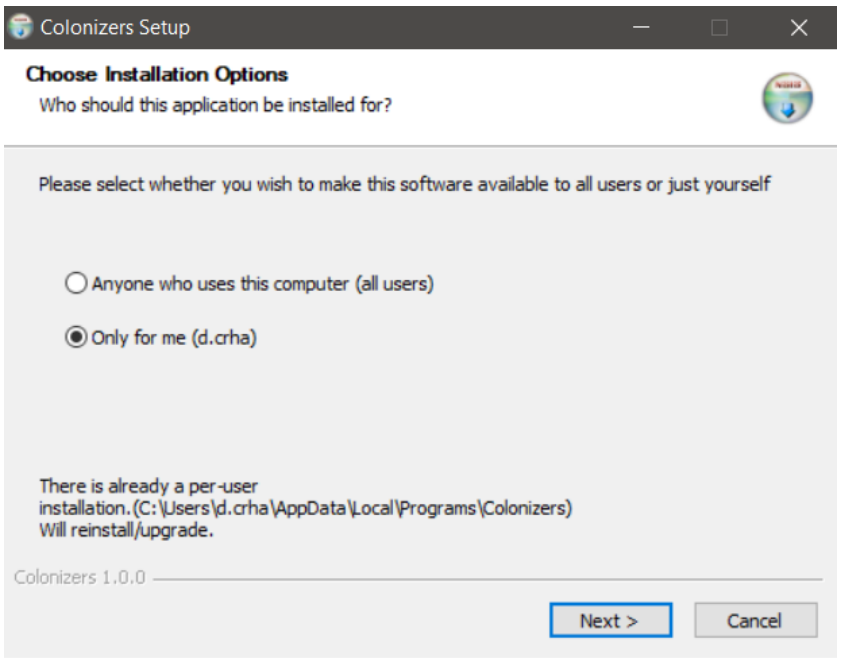

Figure A.1: *Colonizers* installer.

The installer is an executable named `Colonizers Setup` $x.y.z$`.exe`, where x, y and z are placeholders for application versions. During installation, the user will be asked whether they wish to install the application only for themselves, or for all users of the computer. We recommend installing the application only for the current user, since it does not require elevation. The installer also allows the user to configure the installation directory, as shown in Figure A.2.

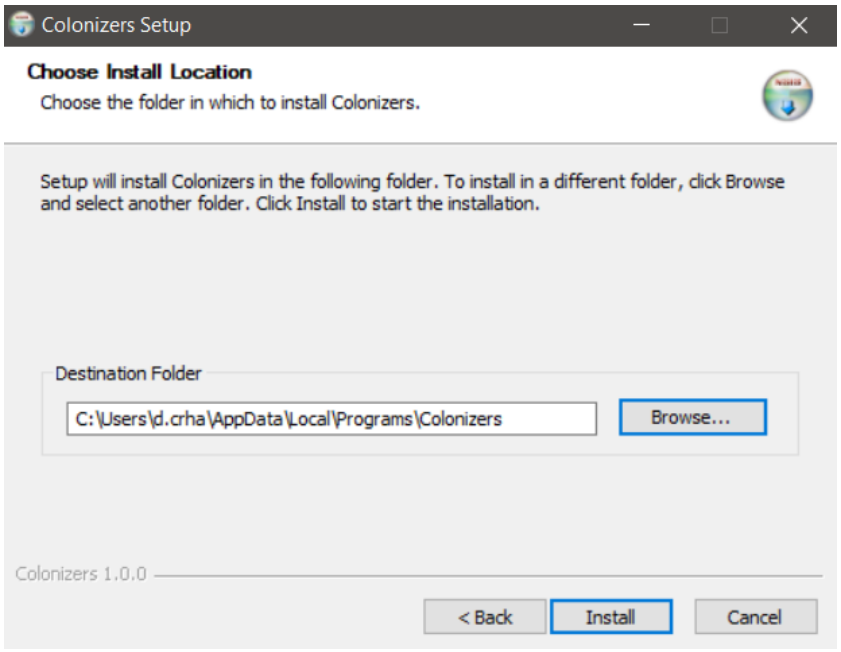

Figure A.2: Choosing the installation path in the installer.

When the installation is finished, the application may be launched from the specified directory. The installer also adds *Colonizers* into the Start menu, and creates a desktop shortcut for the game.

A.1.3 Game Configuration

After launching *Colonizers*, the user will be presented with a configuration screen. On this screen, it is possible to configure the game and AI.

The first notable portion of this screen is the player selection, as shown in Figure A.3.

This section contains four dropdowns, each corresponding to a player. Each dropdown lists all the available AIs which are present in the user's game installation. By default, these dropdowns will have the following values:

- `Human Player`
- `HeuristicIntelligence`
- `ISMCTSIntelligence`
- `MaxnIntelligence`
- `RandomIntelligence`

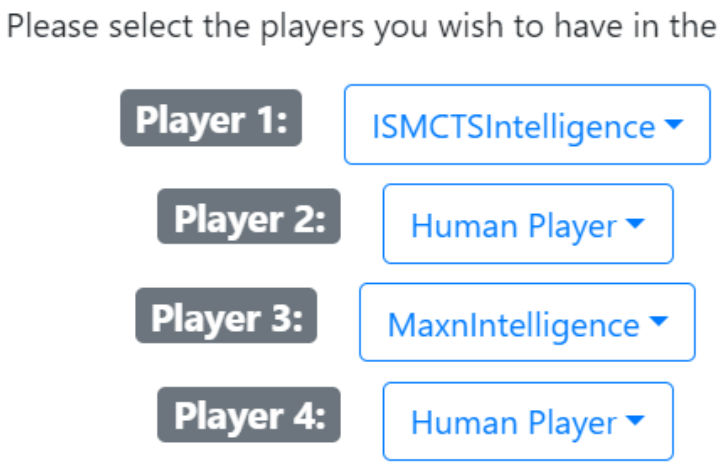

Figure A.3: Player selection.

`Human Player` means that on this player's turn, the UI will become interactible and the user must choose action to perform. The other player options are AIs which are bundled with the game. Table A.1 shows a comparison of the aforementioned AIs, based on the result of this thesis' experiments. Based on this table, the user can choose the AI opponents which suit their needs.

AI	**Strength**	**Evaluation speed**
`RandomIntelligence`	Weak	Fast
`HeuristicIntelligence`	Moderate	Fast
`MaxnIntelligence`	Moderate	Moderate
`ISMCTSIntelligence`	Strong	Slow

Table A.1: Simplified AI comparison.

There is an option to not enable information hiding. Normally, certain information would be hidden on the screen, since that information is hidden in the game. However, for purposes of testing AI, the option shown by Figure A.4 can be left unchecked. This will reveal all hidden information in the UI. Note that this option does not affect the AI in any way, this option is purely a cosmetic one.

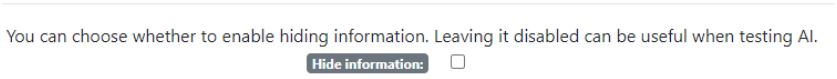

Figure A.4: Checkbox for hiding information.

The next section of configuration is are the buttons for adding new AI into the game, as shown in Figure A.5.

These buttons open file select dialogs, allowing the user to add new AIs. When adding an AI script, the script must follow the naming convention of `<Name>Intelligence.py`, for example `CleverIntelligence.py` [1]. If the se-

[1]If an AI is added this way which shares a name with and existing AI, the existing AI will be replaced by the new one. This also applies to AIs which are bundled with the game

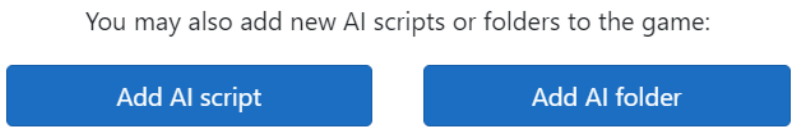

Figure A.5: Buttons for adding new AI scripts.

lected file does not follow this convention, it will not be recognized by the game. When adding an AI folder, the folder must follow the naming convention of `<Name>Intelligence`, and this folder must contain a `main.py` script. If the folder does not follow these conventions, it will not be recognized by the game. This is explained in more depth in Section 3.3.

Lastly, this screen allows the configuration of the Python executable used to execute AI scripts, as seen in Figure A.6. The shown button will open a file select dialog, where the user can select their installed Python executable.

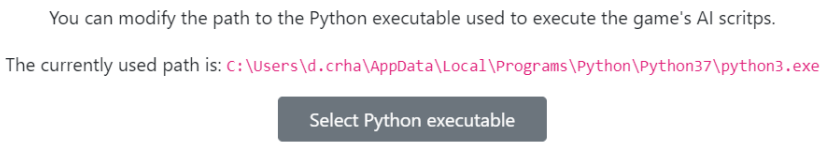

Figure A.6: Configuration of Python executable.

After the user is done configuring the game, they may start a game by clicking the *START GAME* button. Note that this button is disabled if the user has not specified a Python executable to use.

A.1.4 Gameplay

If the reader has not yet read Section 2.2 before reading this section, it may be wise to do so now, in order for them to be familiar with the game's rules.

Board Overview

After launching the game, the user will be greeted with a view similar to the one depicted in Figure A.7.

This is the main game screen, and it is where all gameplay will be happening. First off, we can focus on the buttons to the right of the screen, as shown in Figure A.8.

These buttons control the flow of the game. The game does not start until the user presses the *Start Game* button. When the user does press it, the game starts and the buttons becomes disabled. The other button, *Abandon Game*, may be pressed at any time during gameplay to immediately end the current game and return to the configuration screen.

On the left side of the screen, we can also see miscellaneous information about the game state written in plain text. This information includes the amount of modules left in the deck, the player on turn, and the current game phase.

The player overview areas are a core part of gameplay, you can see an example of this area in Figure A.9.

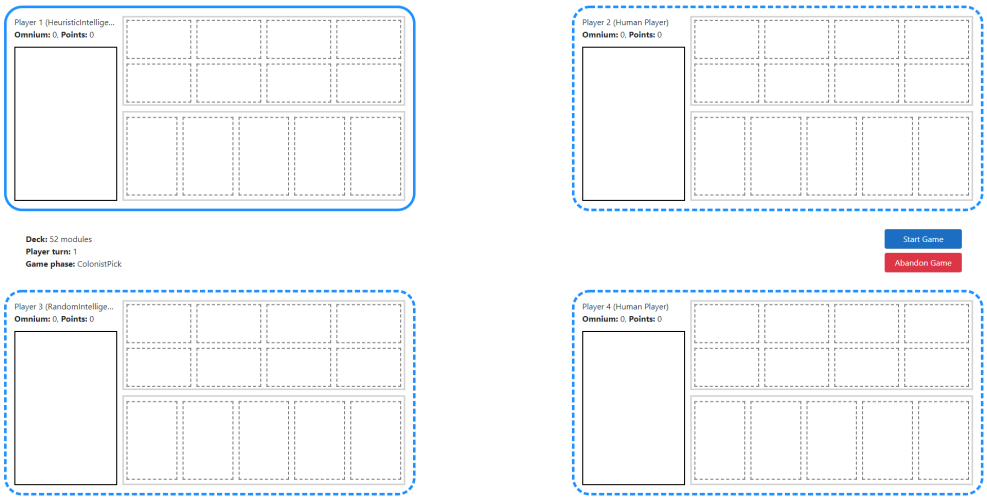

Figure A.7: Overview of main game screen.

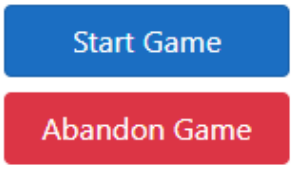

Figure A.8: Game control buttons.

This area contains all information about a given player. In the top left, it shows the player's position and their name (meaning either *Human Player*, or the name of the AI playing this player). In the top left we can also see the amount of Omnium the player has (the game's currency) and the number of points the player has. The number of points determines the ranking at the end of the game.

On the left side of the player overview is the player's colonist card. This shows the colonist this player has selected. A colonist is a character controlled by the player for a single turn, and the colonist provides the player with special abilities to use at specific times of the game. The colonist is easily identifiable by the icon and large text below it. For a full description of what abilities colonists have, please refer to Section 2.2.3.

It is possible for this colonist card to be hidden, instead displaying a card featuring a large question mark. This can happen when hidden information is enabled during game configuration. Specifically, a player can only see another player's colonist if the other player has already taken their turn.

Another feature shown by Figure A.9 is the colony overview, seen in Figure A.10. This area contains the modules the player has built during the game. The colony area has eight possible slots to build on. A square with a grey dashed border is an empty slot. When any player builds eight modules in their colony, the game will end at the end of the round, after all players have taken their turn.

The elements with colored borders and two numbers inside them are called

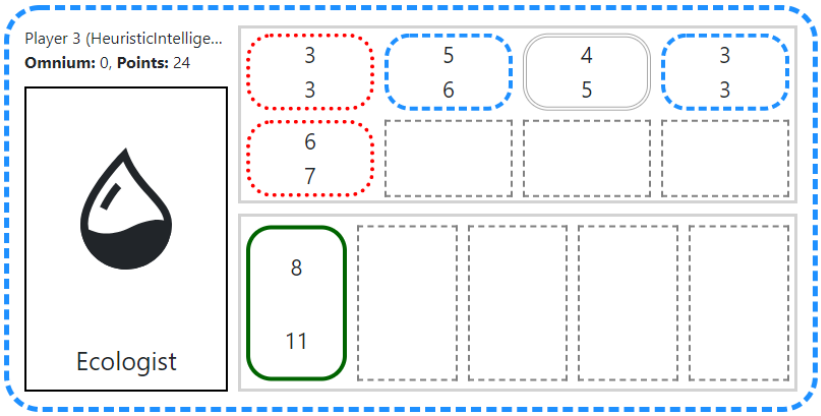

Figure A.9: Player overview.

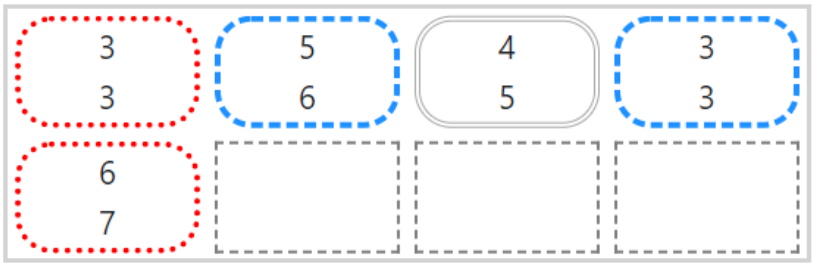

Figure A.10: The colony overview.

modules. They are drawn by players from the deck, and they can be built in a player's colony. In order to build a module, the player must pay its build cost. A module's build cost is the number shown in the upper part of the module. When build in a player's colony, modules count towards that player's score. A module's contribution to a player's score is the number found in the bottom part of the module. Modules also have a color, depicted by the colored border [2] . This color is relevant during interactions with certain colonist abilities. The modules in a player's colony are not hidden, and are visible to other players even when information hiding is enabled.

The player overview seen in Figure A.9 also contains the player's hand, as seen in Figure A.11.

Figure A.11: A player's hand.

[2]In order for the game to be accessible to colorblind persons, the borders are also distinguished by the border pattern. A solid line means green, a dashed line means blue, a dotted line means red, and a double line means a module without a color.

Whenever a player draws modules from the deck, they go into their hand. They then remain in their hand until they are built, or removed by other means. The hand has a size limit of five, it is not possible to draw more modules when a player is at five modules in hand. The modules in other players' hands are among the parts of the game board affected by information hiding. If information hiding is enabled, modules in the hands of other players will be displayed without color, and with question marks instead of their real values.

The last feature of the player overview is its blue border. In Figure A.9, this border is dashed, meaning it is currently another player's turn. The player on turn always has a solid border around their player overview.

Taking Turns

Firstly, it should be noted that AIs take turns autonomously without any need of input from the user. Whenever the turn is passed to an AI, it will immediately start looking for the best move, and as soon as that move is found, it is played and the game continues. This means that with AIs that make decisions particularly fast, if the game is configured to have four of these AIs, the game can potentially end in mere seconds. Therefore we will be discussing only features related to human players taking turns for the rest of this section.

The first phase of a turn is the colonist pick phase. If you set up a game where the first position has a human player, you will immediately be greeted with a selection similar to the one seen in Figure A.12.

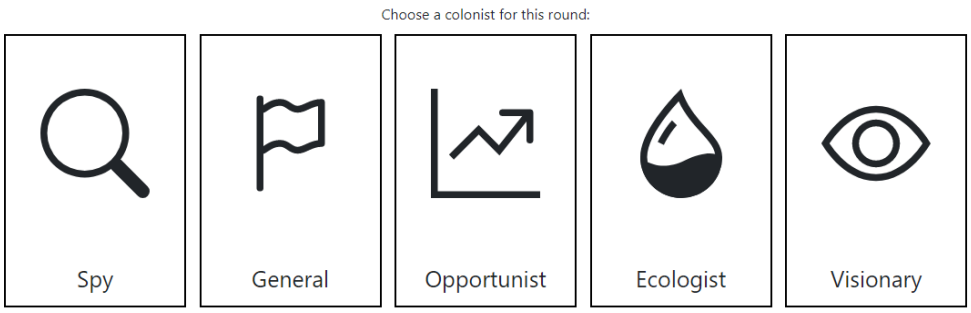

Figure A.12: Colonist selection.

At the start of each round, all players will take turns picking a colonist, in order of first player to last. Note that only five out of the total six are available for selection at any given time, since one is randomly removed from play every round. Selecting a colonist is done by clicking on the corresponding colonist card.

After all players have picked their colonist, players will each take their turn, in order from first to last. The first phase of a player's individual turn is the draw phase, as seen in Figure A.13. The shown buttons will be present in a card overlaying the game board, similarly to the colonist pick phase. Colonist passive abilities trigger automatically during the draw phase.

At this point, a player may choose to either gain two Omnium, or draw two modules from the deck and discard one of them. Note that the draw action will be unavailable if the player's hand is full. If the player chooses to draw modules, they will be presented with an additional dialog as seen in Figure A.14. The

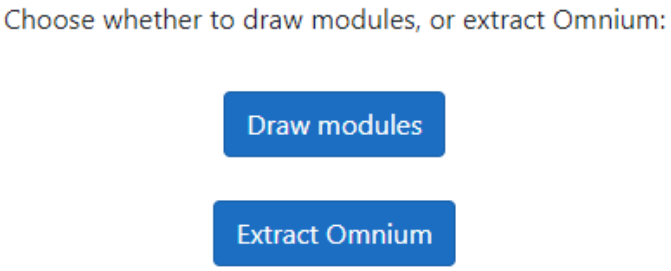

Figure A.13: Draw phase selection.

player must choose which module they want to keep, and which they want to discard. The choice is made by clicking on the module the player wishes to keep.

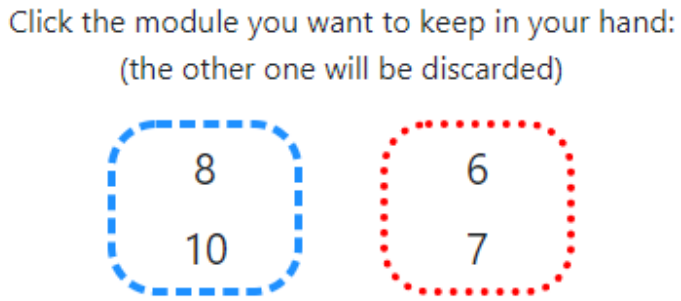

Figure A.14: Choice of module to discard.

The next phase is the colonist power phase. If the player controls a colonist with an active ability (Opportunist or Spy), they will be presented with a choice similar to that shown in Figure A.15. The player may choose to target a given colonist by clicking on their respective card, or the player may choose to not use their colonist's ability by clicking the *Do nothing* button to the right.

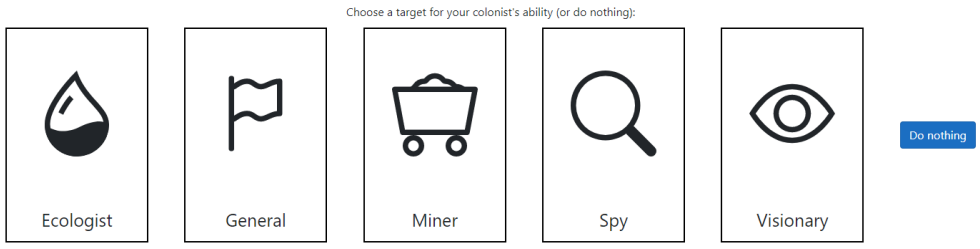

Figure A.15: Selection of colonist active ability target.

Players controlling colonists without active abilities will instead be presented with a small dialog containing a single button which passes the phase.

The last phase of a turn is the build phase. In this phase, players may build up to one module from their hand by spending the required amount of Omnium. Modules which the player can afford to build will have a hammer icon present. Clicking this icon will build the module in the player's colony. If the hammer icon is not present, it means that the player cannot afford the module, or the game is not in the build phase. The hammer icon is shown in Figure A.16, next to a module without the hammer icon.

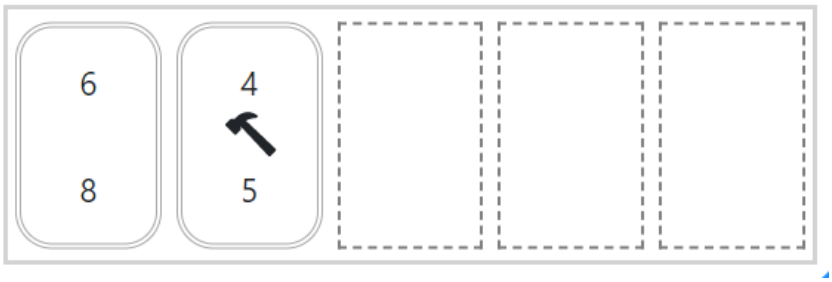

Figure A.16: Hammer icon used to build modules from the hand.

After all players end their turns, the round will end and a new round will begin, starting again with the colonist pick phase. New rounds will keep starting after the previous round ends until a player has built eight modules in their colony. The first player to reach eight modules receives four bonus point on game end, and subsequent players to reach eight modules receive two points each. When the game ends, a dialog will open showing the user the final ranking and final scores. This final score table is shown in Figure A.17.

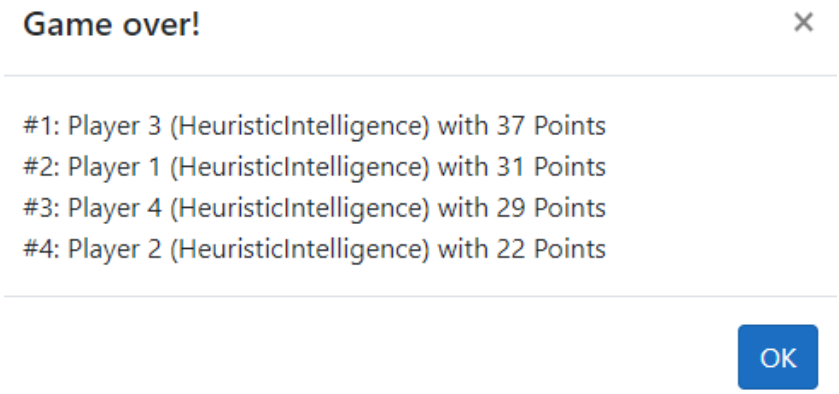

Figure A.17: Game over screen with final scores.

Here, the player may either click *OK* to go back to the game configuration screen, or they may simply close the dialog and spend time looking at the final game board state. When the user wishes to return to the game configuration screen, they may press the *Abandon Game* button to do so.

A.2 Developer Documentation

A.2.1 Prerequisites

In order to do development work on *Colonizers*, the following software is required:

- .NET Core 3.1 SDK
- Python 3.7
- Node.js 10 and NPM 6.4.1

The game's UI is designed for a minimum screen resolution of 1920x1080 at 100% zoom level. It is not recommended to play the game on lower resolution screens, since graphical errors may occur.

In order to build and run *Colonizers* in Electron, the Electron.NET CLI package is required. You may install this package as a .NET Core tool by running the `dotnet tool install ElectronNET.CLI -g` command. This gives you access to the `electronize` command.

It is also highly recommended to use Visual Studio 2019 for development work on the game engine or the UI. Visual Studio 2019 provides support for debugging both the UI and the game engine in the same window, which makes for a seamless development experience. Visual Studio 2019 is also capable of attaching to an external process for debugging, which turns out to be extremely useful with Electron.NET.

For developing and debugging Python AI scripts, the author used Visual Studio Code, but other software capable of debugging Python scripts is viable as well.

The project can be run by navigating to the project directory of the `Desktop` project and running the command `electronize start`. This will build the application and start it inside Electron. Building the application is also done with the `electronize` command-line tool. If we want to build *Colonizers* for Windows 64-bit, we would use the command `electronize build /target win`. Further documentation for the `electronize` command-line tool is available in the documentation for the Electron.NET project [17].

A.2.2 Project Structure

The entire game is contained in a .sln (solution) file, which is a file type used to organize projects in Visual Studio. This solution contains five projects:

- `AICore` — project with the API for AI scripts and the AICore scripts themselves.
- `Desktop` — project with the UI, consisting of an Angular web application and an ASP.NET Core Web API. The Web API is the *ClientApp* subdirectory of this project's directory.
- `Game` — C# library project containing the game logic and code responsible for communicating with Python AIs.

- `Experiments` — console application project containing the experiment scenarios explored in this thesis.
- `ColonizersTests` — unit test project using xUnit as the testing framework. Tests can be run with the command `dotnet test`, or through Visual Studio 2019's test explorer.

The flow of data in the application starts with the UI, since all initiative starts with the user. The UI then uses the ASP.NET Core Web API to execute game logic. If required, game logic then talks to processes executing Python AI scripts. This flow can be seen in Figure A.18.

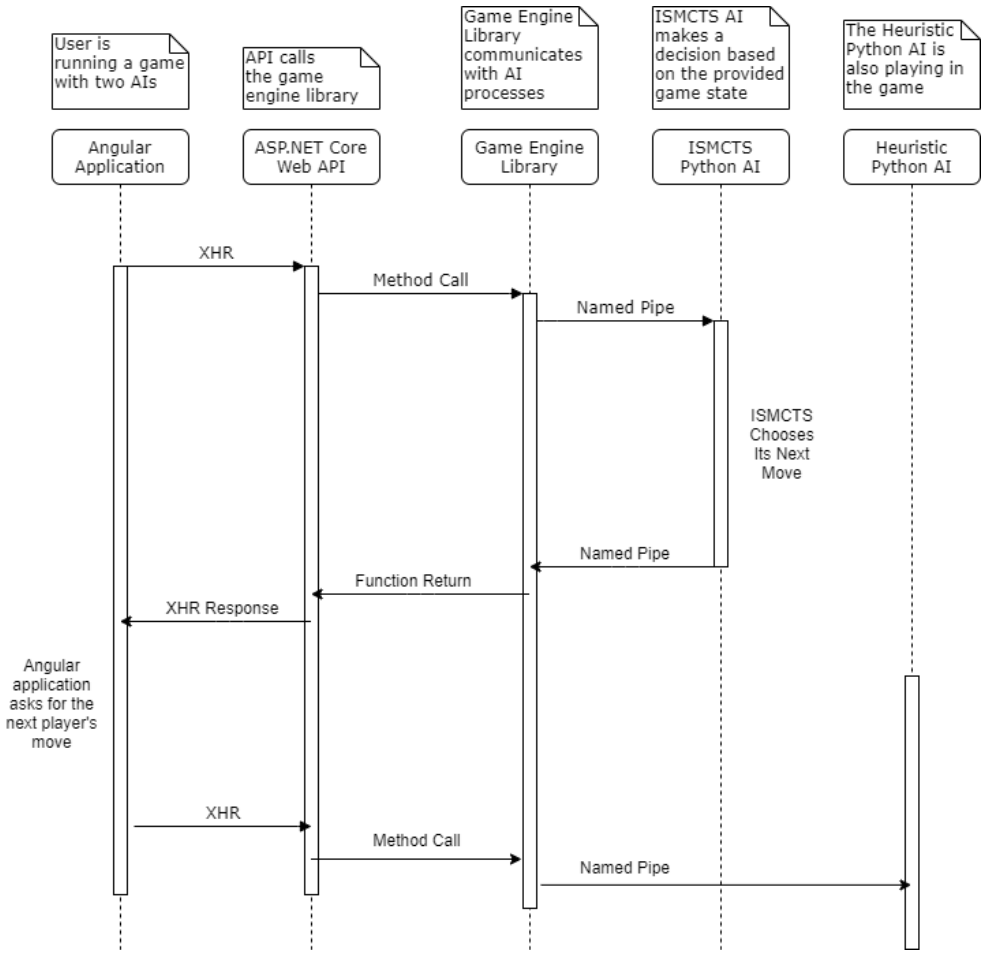

Figure A.18: *Colonizers* sequence diagram.

The aforementioned components will be discussed in more detail in the following subsections.

A.2.3 Game Engine

This subsection will discuss not only the C# library implementing the game logic, but also the ASP.NET Core Web API, since the library is provided to the user Interface through Web API calls.

ASP.NET Core Web API

The Web API consists of controllers, which define REST (Representational State Transfer) API endpoints. These endpoints are then called by the Angular application. In order to promote code reusability, functionality was extracted from controller methods into separate services, which perform more complex operations such as formulating method calls to the game engine. The following controllers are present in the project (in the `Controllers` directory at the top level of the project directory):

- `GameController` is responsible for manipulating game state. It contains endpoints for creating new games, performing actions during gameplay, and for cleaning up after a game is finished.
- `AIController` provides methods for configuring the AI scripts in *Colonizers* — it facilitates adding new AIs and changing the Python executable path used to execute AI scripts.

The following services are present in the Web API project (in the `Services` directory at the top level of the project directory):

- `FileDialogService` uses the Electron API to open file dialogs. These are used to select AIs to add and to configure the Python executable path.
- `GameService` makes calls to the game engine library. It is responsible for configuring and managing.
- `PlayerService` is responsible for the creation of player objects. Since there are three different types of players in *Colonizers* (human player, AI script player and an AI folder player), their creation is not trivial. Therefore this logic was extracted to this service.
- `PythonExecutableService` is responsible for managing the path to the used Python executable. Selecting the path every time the application restarts would be very inconvenient, therefore the application remembers the configured path. This path is stored in on disk in the game's installation folder.
- `StateService` only serves to store game state information in between API calls from the UI application. This avoids unnecessary transfer of JSON data containing the entire game state with every API call.

The ASP.NET Core Web API also sets up DI (dependency injection) for both itself and the game engine. ASP.NET Core provides its own DI framework which is used in this application. The DI is configured in the `ConfigureServices` method of the `Startup` class. The game engine provides an extension method `AddColonizersGame` for easy registration of all its components into DI.

Game Engine Library

All game logic is implemented in the `Game` C# library. This makes game logic a reusable unit, which was useful during the implementation of the experiment project.

First, we will discuss the representation of game state. The root of the model class hierarchy is the `GameState` class, shown in Figure A.19. This class may be found in the root of the `Game` project.

```
/// <summary>
/// Indicates whether the game is over in this state
/// </summary>
public bool GameOver { get; set; } = false;

/// <summary>
/// If the game is over, contains game results
/// </summary>
public GameEndInfo GameEndInfo { get; set; }

/// <summary>
/// The state of the game board
/// </summary>
public BoardState BoardState { get; set; }

/// <summary>
/// Things the current player can do on their turn
/// </summary>
public IList<IGameAction> Actions { get; set; }
```

Figure A.19: GameState class from the game engine library.

This object represents all game state data during a single game of `Colonizers`. Therefore it (or modified versions of it) is used for communicating the game state to other components, be it the UI or AI scripts. Note that it also contains information about whether the game ended and how it ended. Lastly, it also contains a list of possible actions that could be taken from the current game state. This allows AIs to easily work with the game state without having to worry about enumerating the action space themselves. We also include a simplified version of the `BoardState` class for reference, since it contains all data concerned with the actual state of the board, as seen in Figure A.20. This class is also located in the root folder of the `Game` project.

A noteworthy property of the `BoardState` class is `DiscardTempStorage`. This property is used to temporarily hold modules after a player has chosen to draw modules from the deck, but before they have decided which one to keep.

Another interesting part of the game engine is the way the game logic is implemented. The game has multiple turns in every round and multiple phases in each turn, organized in a way that resembles a state machine. Maintaining such a class hierarchy would not be a viable strategy, since cyclic references would be an inevitability. In order to avoid dependency hell and make the code structure easier to maintain and understand, we have employed the mediator pattern, facilitated by the `MediatR` library. This allows us to separate logic pertaining to particular game phases into their own, self-contained units without complicated external dependencies. The game logic classes are separated into three categories:

```
public enum Phase { ColonistPick, Draw, Discard, Power, Build }
public IList<PlayerInfo> Players { get; set; }
public IReadOnlyList<Colonist> PlayableColonists { get; set; }
public IList<Colonist> AvailableColonists { get; set; }
public List<Module> Deck { get; set; }
public IReadOnlyList<Module> StartingDeck { get; set; }
public IList<Module> DiscardTempStorage { get; set; }
public int PlayerTurn { get; set; }
public Phase GamePhase { get; set; }
```

Figure A.20: BoardState model class (simplified).

- `Commands` — the basic models of actions to perform on a particular board state. Every kind of action taken by players in the game has its own `Command` associated with it, marked by the `IGameAction` interface. These commands may be dispatched via the mediator. Located in the `Commands` directory.
- `CommandHandlers` — classes which implement logic mutating the game board. For example, if we have a command representing the action of building a module, a `BuildModule` command will be dispatched via the mediator, and then handled by the `BuildModuleCommandHandler`. Command handlers only mutate the game state based on their input command, they do not enumerate possible actions. Located in the `CommandHandlers` directory.
- `ActionGetters` — these classes are responsible for enumerating the actions space. They receive an input game state, and from it they generate a list of actions which are legal in this state. Located in the `ActionGetters` directory.

Each `CommandHandler` has a reference to its associated `ActionGetters`. If we imagine this situation in a state machine context, the `CommandHandlers` handle movement between states, and `ActionGetters` are responsible for finding out which states we can move to next afterwards. The whole game logic system consists of pure classes and functions, meaning for the same input, they provide the same output. This is a crucial property of this design, considering the game logic is quite complex in certain places. It gives the code consistency between runs, and makes it easy to debug and understand.

We can examine a high-level method which processes a single turn of the game in Figure A.21. We can see that given a game state and a list of players, we first ask the current player to choose a move. After they have selected a move, the appropriate `Command` is passed into the game logic structure through the mediator. The structure then returns a new game state along with a list of actions possible in this new state.

Lastly, the game engine library contains the code responsible for communicating with the Python AI implementations. This communication is done via named pipes. When a game is starting, a named pipe is created for each AI. The AI is then started as a separate process using the configured Python executable path.

```
public async Task<GameState> ProcessTurn(GameState gameState,
    IReadOnlyList<IPlayer> players)
{
    var boardState = gameState.BoardState;
    IPlayer currentPlayer = players[boardState.PlayerTurn - 1];
    int moveId = await currentPlayer.GetMove(gameState, resolver);
    var selectedMove = gameState.Actions[moveId];
    return await resolver.Resolve(selectedMove);
}
```

Figure A.21: Processing of a single game turn.

This process is passed its pipe name as an argument. When the AI starts up, it connects to the pipe and starts listening. The game engine will send a request for a decision to be made, and the AI may start deciding. When the AI chooses a move, the move is returned through the same pipe. The pipe is also used for other communication between the AI and the game engine, notably for requesting determinization and simulating moves. It is worth mentioning that due to the fact that the named pipes have pre-defined names, it is not currently possible to run multiple instances of the game on the same machine. We say currently, since implementing a mechanism for randomizing the pipe names would not be a very complicated extension of the library.

A.2.4 User Interface

Colonizers uses Electron as a means to run the game as a desktop application, since the game is developed using web technologies. Specifically, it uses the Electron.NET library, which provides an access to the Electron APIs to C# applications, and it also facilitates usage of Electron's build tools to package C# applications. C# and the whole .NET platform in general still do not have a widely-used cross-platform UI framework, therefore Electron was a good fit for this project.

The UI for *Colonizers* is an Angular application, which is then served inside Electron. This application is located in the `ClientApp` directory in the `Desktop` project directory. Electron then uses the Chromium rendering engine and Node.js in the background. The Angular application allows configuration of the game, it handles presentation of game state to the user, and it is responsible for communicating with the ASP.NET Core Web API.

In order to do development work on the UI, it is recommended that you use Visual Studio 2019. It has a very useful feature whereby it allows you to debug both JavaScript and C# code at the same time in the same project. Since the Angular application is located in the same project as the Web API, this is an invaluable feature. Since the UI is an Angular application, naturally it is possible to run and debug it without running it in Electron. The source code for this project contains launch settings pre-configured for Visual Studio 2019 in order to accomplish exactly this. A sidenote is that while running outside Electron, the application does not have access to Electron APIs. *Colonizers* uses the File

Dialog API multiple times, therefore this functionality will be unavailable in this case. Interaction with the Electron APIs is always preceded by a check whether the application instance is running inside Electron, therefore calls to methods which use Electron APIs will not cause exceptions. We can see such a guard for Electron presence in Figure A.22.

It is also possible to debug the application while it is running inside Electron. The command `electronize start` will launch the application inside Electron when run. It is then possible to attach to the application's process in Visual Studio 2019. This allows us to debug code which uses Electron API calls, which is not possible when not running in Electron. The mentioned command also has a useful option — `electronize start /watch` which will watch application files for changes and re-compile only changed application files.

```
public async Task<bool> AddSingleScript()
{
    if (HybridSupport.IsElectronActive)
    {
        BrowserWindow mainWindow = Electron.WindowManager
            .BrowserWindows.First();
        OpenDialogOptions options = new OpenDialogOptions
        {
            Properties = new OpenDialogProperty[] {
                    openDialogProperty
                }
        };

        string[] files = await Electron.Dialog.ShowOpenDialogAsync(
            mainWindow, options);
    }

    return false;
}
```

Figure A.22: Electron API call guarded by check for Electron presence.

The source code for the Angular application is written in TypeScript, CSS and HTML. The HTML used is not pure HTML, rather the HTML files are Angular templates. Angular Templates are a way to insert data into markup seamlessly. The TypeScript files are transpiled to JavaScript at build-time. We use Angular in a client-side mode, whereby the source files are compiled ahead of time, and are delivered to the client on-demand. Angular also offers a server-side rendering option, allowing to offload some work from clients onto servers. However, due to the fact that both client and server run on the same machine in *Colonizers*, this option was not used.

The most important building blocks of Angular are *Components*. Components control the view presented to the user, and they prepare data for presentation by the view. We can see the source code for a component in Figure A.23

We can see a few important component features in Figure A.23:

```
@Component({
    selector: 'app-discard',
    templateUrl: './discard.component.html',
    styleUrls: ['./discard.component.css']
})
export class DiscardComponent implements OnInit {

    @Input() gameState: GameState;
    @Output() onPick = new EventEmitter<number>();

    constructor() { }

    ngOnInit() {
    }

    getModules(): Module[] {
        // Find the appropriate modules in the temp discard storage
        return this.gameState.actions.map(
            x => this.gameState.boardState.discardTempStorage
                .find(y => y.name === x.module));
    }

    keep(module: Module) {
        this.onPick.next(this.gameState.actions.findIndex(
            x => x.module == module.name));
    }

}
```

Figure A.23: An Angular component.

- The definition of the component's template and styles in the `@Component` decorator. Notably, the styles specified in this scope only apply to this component's template.
- The component has an `@Input()` and an `@Output()`. These are the ways other components interact with this one. Keeping component interaction to only inputs and outputs makes components pure, meaning they will output the same data when provided with the same inputs. In a complex application, this is a very desirable property, since it makes debugging easier and bugs more rare.
- The component defines a selector — `app-discard`. Using this selector, other components can include this one in their templates.

We can see an example of the aforementioned component being used in Figure A.24. The excerpt is from `GameComponent`'s template, and it demonstrates how it binds one of its own fields as an input for the `DiscardComponent`, and that it is listening for events emitted by it.

```
<div *ngIf="isWaitingForHumanPlayer && isDiscardPhase()">
    <app-discard [gameState]="gameState"
                 (onPick)="onHumanPlayerAction($event)">
    </app-discard>
</div>
```

Figure A.24: Usage of Angular component in another component's template.

These features are the core of how the UI application is built — it is based on a divide-and-conquer principle, where the entire view is composed of smaller components, which are in turn composed of even smaller components.

Another notable building block of `Colonizers` is the usage of Angular services. A service in Angular is meant to be a way to abstract data manipulation and API calls away from components. Therefore, all code pertaining to communication with the ASP.NET Core Web API is contained within the two service classes — `GameService` and `ScriptsService`.

A.2.5 AI framework

In the previous subsection, we have already discussed the communication between game engine and the AI from the game engine's point of view. In this subsection, we will examine the other side. The API provided to AIs is contained within the `AICore.py` source file (located in the root folder of the `AICore` project), in the abstract class `AIBase`. This class is meant to be a base class for all AI implementations added to the game. It provides the AIs with the following functionality:

- Communication with the game engine. The AIs do not need to manually read from and write to pipes, this is handled by base class methods. Whenever the game engine requests an action from the AI, the base class will read this message from the named pipe, and invoke the AI code to get a response. It then writes this response back to the named pipe.
- Other communication with the game engine — determinization and simulation. Determinization, provided a game state with hidden information, will produce a game state with perfect information. This is done by the game engine using information set data it tracks internally. This is used in algorithms like ISMCTS or our adapted version of MaxN, both of which we examined in the experimental part of this thesis.
- The `AICore.py` source file also contains various utility functions for working with game state. Game state is provided to AIs as a dictionary which copies the structure of the `GameState` object we discussed in Attachment A.2.3. Among these functions are utilities like counting modules in a player's colony on a per-color basis.

Since the game engine executes the AIs by running them from the command line with a Python interpreter, it is crucial that the `AICore.py` file be accessible to them. To this end, whenever an AI is added to the game, it is copied into a folder in the game files containing the `AICore.py` file. With folder-based AIs the

situation is a little more tricky, since importing files higher in a folder hierarchy is not straightforward in Python. Therefore, when copying in folder-based AIs, a copy of `AICore.py` is copied along with the new files into the new folder. This way, the AI has access to this file at runtime.

Observe that during application build or publish, the AIs which ship with the game are copied into the publish folder. This is accomplished through the .csproj files for the `Desktop` and `Experiments` projects.

While creating AIs for the game, debugging the AI is relatively straightforward. Simply set a breakpoint at the game engine location where the AI is being run with the Python interpreter, then skip the line creating the actual process and instead run your own process in debug mode in your development environment of choice.

The following is a description of the API provided by the `AIBase` class:

- `messageCallback(self, gameState)` — abstract method, must be implemented in descendants. This method is called by the framework when it is the AI's turn in the game, and its response is required. The gameState object represents the current game state (the object will be described later in this section). The expected return value is a string containing a single number, corresponding to the index of the chosen action. The by index we mean the zero-based index of the chosen action in the `gameState["Actions"]` list.

- `determinize(self)` — uses the information set stored by the game engine to produce a determinized version of the current game state. The return value is the determinized version with all hidden information revealed.

- `simulate(self, boardState, move)` — simulates the given move on the given board state, and returns the new game state. The `move` parameter must be a string representation of an action, following the specific format returned by the `getActionString(action)` function of the `AICore.py` file.

- `run(self, pipeName)` — initializes the communication with the game engine. The `pipeName` parameter is passed to the AI script as the only argument, therefore it is located at `sys.argv[1]` in the main AI file. A typical usage pattern is shown in Figure A.25

```
if __name__ == "__main__":
    ai = ISMCTS_AI() # Inherits from AIBase
    ai.run(sys.argv[1])
```

Figure A.25: Common usage scenario of the `run(self, pipeName)` method.

There are a number of other utility functions in the `AICore.py` file, however most of them are not worth mentioning here, since they are simply one line shorthands for common operations. There is one worth mentioning however — `getActionString(action)`. This function converts an action (as obtained from the `gameState["Actions"]` list) into the shorthand string format used by the `simulate(self, boardState, move)` method.

A.2.6 Experiments

The experiments performed in this thesis are implemented in the `Experiments` project of the `Colonizers` solution. It is a C# console application project. It is invoked via the command line with two arguments — the first is a number between 1 and 5, corresponding to the experiment number, and the second is the path to the Python executable to use when executing AI scripts. Note that the attachments to this thesis do not contain a binary for the `Experiments` project, therefore you will have to build and run it yourself. This may be done by installing the required software for developing *Colonizers* mentioned in this chapter, navigating to the `Experiments` project folder, and running the command `dotnet run`, followed by the parameters required by the program.

After the experiment is run, it will produce a JSON file containing the results of the experiment. This file is generated in the directory where the `Experiments` project is run from. Figure A.26 shows the structure of these JSON files, with less important fields omitted for brevity. The JSON files associated with the five experiments are also located in the `Results` folder of the `Experiments` project. They were added there for reference, since some experiments may take tens of hours to run even on reasonably fast machines. Be aware that a simple diff of the JSON result files is not sufficient for determining whether or not a given experiment was successfully replicated. This is because the files contain running times of the games as well.

```
{
    "Players": [
        "Name": "ISMCTS",
        "PlayerEndInfo": {
            "Ranking": 3,
            "VictoryPoints": 24,
            "Player": {
                "ID": 1,
                ...
            }
        }
    ],
    "Duration": "00:12:28.1879334"
}
```

Figure A.26: Experiment result JSON file structure (simplified).

The class `Scenarios` contains the configuration and setup for the experiments, and each scenario then calls `ExperimentRunner` to performed the experiment itself. There is no need to configure anything more than passing the program the required parameters, the experiment scenarios are set up exactly as they were performed by the author.

A noteworthy point is the implementation of the shuffling of players between games. Since the game engine already possessed an implementation of the Fisher-Yates Shuffle [16] for shuffling lists, this implementation was reused for shuffling the players themselves. This means that if we were to run the experiments without

shuffling, and instead assigning the players the same positions they would have been assigned by the shuffle, the results of this experiment would be different. This is because running the shuffle manipulates the game engine's random number generator.