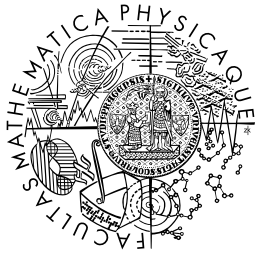

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

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: IOI

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 artificial intelligences 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 effecive 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 (Aplication Programming Interface) provided by the game is sufficient for implementation of such AI players, and that the API is reasonably easy to use.

1. Related Work

Before we discuss the design of *Colonizers* and its implmentation, 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 implementaion.

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 does not provide support for any other machine learning methods.
- 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. 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.

Since Minimax is only useful in the aforementioned types of games, we will look to the MaxN algorithm [4].

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.

The procedure MaxN is defined recursively (as presented by Luckhardt and Irani [4]) 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.

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.

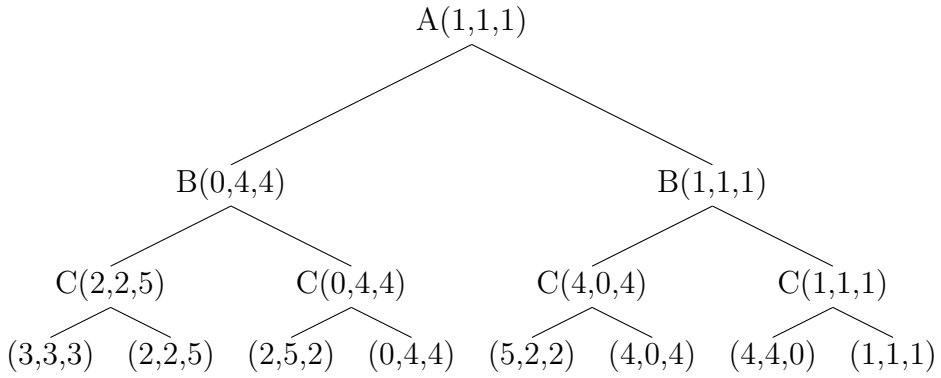

Figure 1.4: MaxN tree example.

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 [5] is an algorithm for searching trees. 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.

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) [6].

While determinization is a viable strategy, it is not without its pitfalls. Russel and Norvig speculate that since all information is revealed after determinization, the resulting AI will never make information-gathering plays. [7]

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 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." [8]

Other potential problems include two mentioned by Frank and Basin [9]. 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: "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) [6], 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. In order to balance exploration and exploitation of actions in the tree, a multi-armed bandit algorithm is used during tree traversal.

Figure 1.5 shows high-level pseudocode for the SO-ISMCTS algorithm, as presented by Cowling, Powley and Whitehouse [6].

```
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/acitons compatible
            with d) using the chosen bandit algorithm
        until a node v is reached such taht 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.

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 conspiration, 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 comprise of turns. Turns then comprise 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 buildings 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. 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 subsection 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.

The colonist pick phase creates a situation where players have asymmetrical information. For example, the first player knows which colonist was removed at

[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.

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 is comprised 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). 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

The deck starts with 52 modules. The following is a table of all modules available:

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.

3. AI Framework

3.1 Design

Firstly, the choice of technologies used for creating *Colonizers* warrants some explanation. The architecture of the application comprises 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. There are multiple such frameworks, notably React and Vue, which would have also made 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

TODO: how to add new intelligence - add to folder or file picker?

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.

4. Used Algorithms

4.1 Random Decisions

4.2 Heuristics

4.3 MaxN

4.4 Information Set Monte Carlo Tree Search

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 [10]. 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 subsection 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.

- `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.

5.1.2 Findings

Experiment 1

First off, let us focus on the experiment runs with `RandomIntelligence`. Results of the 1000 runs can seen in table 5.1.

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. If we assume that the rank at the end of the game follows a normal distribution, we can compute a confidence interval. Let $\bar{x}$ be the sample mean, let s be the sample standard deviation and let n be the sample size. We are looking for a confidence interval for the unknown mean μ. The $100(1-\alpha)\%$ confidence interval for μ is

$$\hat{\mu}_L = \bar{x} - z_{\alpha/2} \cdot s/\sqrt{n},\ \hat{\mu}_U = \bar{x} + z_{\alpha/2} \cdot s/\sqrt{n},$$

The sample mean for rank among first position AIs is 2.3 as seen in table 5.1, and the sample standard deviation is 1.1069. If we want a 95% confidence interval, we will use $z_{\alpha/2} = 1.96$. This gives us the confidence interval of

$$\hat{\mu}_L = 2.2314,\ \hat{\mu}_U = 2.3686$$

We can also compute a 95% confidence interval for the mean rank of AIs in position 3:

$$\hat{\mu}_L = 2.4821,\ \hat{\mu}_U = 2.6239$$

These intervals do not overlap, therefore we can conclude that there is a statistically significant difference between the rank means among AIs at different positions. 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 are 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 seen in table 5.2.

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 computing confidence intervals for the first and fourth ranks. The 99% confidence interval for the first position is

$$\hat{\mu}_L = 2.7019,\ \hat{\mu}_U = 2.8981$$

while the 99% confidence interval for the fourth position is

$$\hat{\mu}_L = 2.1458,\ \hat{\mu}_U = 2.3102$$

The intervals do not overlap, therefore we can conclude that there is a statistically significant difference between the means of these positions' respective average ranks.

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 [4], which is itself based on Minimax [11]. 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, we 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

[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

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 subsection 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.

- `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.

5.2.2 Findings

WORK IN PROGRESS

Experiment 3

Experiment 4

Experiment 5

Conclusion

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

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

[6] 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.

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

[8] 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.

[9] 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.

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

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

List of Figures

List of Tables

A. Attachments

A.1 User Documentation

A.1.1 Installation

A.1.2 User Interface

A.2 Developer Documentation

A.2.1 Prerequisites

A.2.2 Project Structure

A.2.3 Game Engine

A.2.4 Artificial Intelligence

A.2.5 User Interface

A.2.6 Experiments