Pázmány Péter Catholic University
Faculty of Information Technology and Bionics

Computer Science Engineering

B.Sc. Diploma

Estimating board game balance and replayability with machine learning

Author
Zsófia Balogh

Supervisor
Gergely Feldhoffer, PhD

Budapest, 2020

Declaration of Authorship

I, Zsófia Balogh, student of the Faculty of Information Technology and Bionics at Pázmány Péter Catholic University, hereby declare that this B.Sc. diploma is my own, unaided work and that I have not used any sources other than those listed in the bibliography and identified as references. I further declare that I have not submitted this document to any other institution in order to obtain a degree.

signature

Pázmány Péter Katolikus Egyetem
Információs Technológiai és Bionikai Kar

1083 Budapest, Práter utca 50/a www.itk.ppke.hu

Szakdolgozat témabejelentő nyilatkozat

Ikt.: I/894/2020

Név: **Balogh Zsófia**	Neptun kód: **G728XZ**
Képzés: **Mérnökinformatikus BSc (IANI-MI)**	

Témavezető

Név: **Dr. Feldhoffer Gergely**

Dolgozat

Dolgozat címe: Estimating board game balance and replayability with machine learning

A dolgozat témája:
Provide an abstraction for describing the rules of turn-based, competitive board games. Research and implement machine learning algorithms (minimax, genetic algorithms, reinforcement learning) to form winning strategies for the given ruleset. Evaluate the results in terms of game balance and replayability.

A hallgató feladatai:
The student must:
1. Provide a universal API that allows the user to describe the mechanics of a board game with the following restrictions:
 a. the game is competitive, there is a well defined winner (or loser) at the end
 b. a fixed number of players take turns after one another
 c. the actions players can take in their turn are predefined by the rules
 d. playing the game requires only the knowledge of the rules, there is no dependence on outside information

Other restrictions may be added later to make sure the project can be implemented on time.

2. Develop competing agents that can play the game and come up with winning strategies.

3. Evaluate the game's mechanics based on these strategies by the following criteria:
 a. Game balance: identify if there are any over- or underpowered elements in the game (turn order, specific resources, actions or other game elements).
 b. Replayability: analyze the diversity and qualities of the winning strategies. Having an easy, surefire winning strategy can make the game boring quite quickly which won't lead to many replays. Having no absolute winning strategy, or having multiple, comparable alternative strategies will lead to more willingness to play the game again.

4. Describe and evaluate at least two different contemporary board games.

Kelt: Budapest, 2020. 04. 23.

Kérem a szakdolgozat témájának jóváhagyását.

Aláírva a Space rendszerben
2020.04.23. 02:00:00
Balogh Zsófia
hallgató

A témavezetést vállalom.

Aláírva a Space rendszerben
2020.04.23. 02:00:00
Dr. Feldhoffer Gergely
témavezető

A szakdolgozat témáját az Információs Technológiai és Bionikai Kar jóváhagyta.

Aláírva a Space rendszerben
2020.11.04. 12:43:01
Iván Kristóf
dékán

A hallgató a kiírásban foglalt feladatokat teljesítette. A dolgozat a TVSZ-ben foglalt tartalmi és formai követelményeknek megfelel.

Aláírva a Space rendszerben
2020.12.16. 07:34:46
Dr. Feldhoffer Gergely
témavezető

ABSTRACT

As board game popularity continues to grow with the help of social media and Kickstarter, so does the number of games released each year. A huge part of the game design process is centered around trying to guarantee a pleasant experience for the players by avoiding unwanted scenarios such as unbalanced mechanics or dominating strategies.

Tailoring the game to accommodate a varying number of players or have different degrees of strategic depth is no easy task. As the complexity of the game increases, so does the difficulty of maintaining balance and integrity.

It was already shown by multiple studies that AI can be a great tool to help with playtesting.

The contribution of this work is `sfinks`, a proof-of-concept library that can be used to describe the rules of an arbitrary board game. Based on these rules it can train AI agents with reinforcement learning to be able to play the game and generate gameplay data for analysis, thereby accelerating playtesting.

The capabilities of the library are demonstrated through implementation and gameplay analysis of three different board games: *Tic-tac-toe*, *Colors*, and *Bird Lady*.

ACKNOWLEDGMENTS

I would like to thank my teachers

- Magyar Eszter
- Dr. Nagy Piroska Mária
- Szoldatics József
- Dr. Feldhoffer Gergely
- Nyékyné Dr. Gaizler Judit

who introduced me to mathematics, programming and even board games. They had a profound effect on my life and without them this document would not have come to life.

CONTENTS

FIGURES

LISTINGS

TABLES

INTRODUCTION

1

As board game popularity continues to grow with the help of social media and Kickstarter [1][2], so does the number of games released each year. A huge part of the game design process is centered around trying to guarantee a pleasant experience for the players by avoiding unwanted scenarios such as unbalanced mechanics or dominating strategies. Tailoring the game to accommodate a varying number of players or have different degrees of strategic depth is no easy task. As the complexity of the game increases, so does the difficulty of maintaining balance and integrity. Even a small change can have a big—and maybe at first untaught or unseen—effect on the gameplay as a whole, so the design process requires a great deal of care with lots of testing over many iterations.

Using AI to help with playtesting is not a new idea, Silva[3][4][5] and Mugrai[6] for example both explored it in the context of specific games. Building on the concept of AI-based playtesting, this work takes a different approach and explores the possibility of creating a "universal" engine that is able to learn how to play different kinds of games and then provide gameplay data for analysis for the game designers.

1.1 DOCUMENT STRUCTURE

Chapter 2 contains relevant background information:

- section 2.1 explores factors of enjoyment when playing board games,
- section 2.2 lists the machine learning concepts used throughout this work.

Chapter 3 details the design considerations and implementation of the `sfinks` engine, a framework designed to be able to learn to play different turn-based, competitive board games.

Chapter 4 introduces two board games, Colors and Bird Lady and explores their gameplay characteristics using the AI agents provided by `sfinks`.

Chapter 5 summarizes the completed work and provides a list of possible improvements.

BACKGROUND

2.1 BOARD GAMES

Games can be defined as “physical or mental competitions conducted according to rules with the participants in direct opposition to each other”[7].

Board games are mental games that usually involve counters or pieces placed or moved around—in accordance with the game rules—on a predefined surface called the board.

Board games have been around for a really long time. The oldest board game known to have existed, called Senet, is dated around 3500–3100 BC[8]. The oldest board game still played in its original form today, the game of Go, is thought to have originated at least 2500–4000 years ago[9].

Board games come in many shapes and forms, rules can range from very simple to deeply complex. The number or complexity of the rules does not necessarily correlate with the time and effort needed to master the game. Games like Go or Chess for example have great strategic depth while only having a few simple rules.

Theme versus mechanics

Among board game hobbyists, modern board games are often categorized either as *amerigame*s (also called as American-style, or ameritrash[10]) or as *eurogame*s (also called as Euro-style or German-style).

American-style games can be thought of as theme-forward, often having exciting, dramatic themes, backed up by lots of rules. Players usually interact through direct conflict, player elimination is also common. Success, failure, or effectiveness of an action is likely to be based on chance (decided by roll of the dice for example). The end of the game is usually tied to an in-game event of some sorts, which can lead to very long play sessions.

Betrayal at house on the hill[1] is a typical amerigame. Players, all friendly at first, explore a haunted house filled with all kinds of danger, items and omens. The layout of the house is built randomly during gameplay, so each session is completely different. Eventually something called the “haunt” begins (based

[1] https://boardgamegeek.com/boardgame/10547/betrayal-house-hill

on a series of dice rolls called “haunt checks” each time a player picks up an omen), where the plot is revealed: one player usually “betrays” the others. From this point on the traitor is playing against the rest of the players. Players are eliminated from the game if their character dies. Each haunt comes with its own set of rules and win conditions for both sides.

Eurogames can be thought of as mechanics-forward, often having very general themes that do not contribute much to the gameplay experience. Player interaction is usually indirect and realized through shared actions or resources. Random elements are rarely used during gameplay, actions do not fail, the consequences of an action are fully predictable. The games usually have an “internal timer” that keeps the game going on for a certain amount of time, after which the winner is determined.

Terra Mystica[2] is a typical eurogame. It starts with each player picking one of fourteen factions, which have different special abilities, but other than that operate by the same set of rules. While these factions all have fantasy names likes “Chaos Magicians” or “Mermaids”, it does not really add anything to the gameplay. The only important thing about them are their color and their special abilities. During the game, players expand their territories by terraforming land and constructing buildings on the game map. Each game lasts through six rounds, during which the players take actions after one another until all of them decide to pass. Actions include digging, upgrading buildings, improving infrastructure and so on. Nothing is decided at random, everything is a direct consequence of the players’ decisions. At the end of the sixth round the player with the most points is declared the winner.

As these two categories of board games play very differently, players’ expectations of them are also different. American-style games are more focused on storytelling and immersing the players in the given theme, which in itself can provide significant replayability. *Betrayal at house on the hill* for example comes with a booklet containing 50 totally different haunts.

Eurogames with weak themes and low randomness are much more reliant on the promise of guaranteeing a balanced, strategically deep experience in order to engage players in the long run. While no one expects *Betrayal at house on the hill* to be balanced or strategic, *Terra Mystica* is loved and played

[2] https://boardgamegeek.com/boardgame/120677/terra-mystica

because players are looking for the challenge of mastering the underlying strategy in a "fair" environment, where skilled players are more likely to win.

Factors in the enjoyment of (euro)games

In a survey conducted by Stuart Woods[11] among game hobbyists, for the question

> "To what degree (very important / quite important / not very important) is the presence of the following elements important to your enjoyment of board and table games?"

the top three elements chosen by the respondents as very important were *potential for replayability*, *strategic depth* (longer-term plans) and *tactical play* (momentary decisions). Less than 10% stated that these elements are not very important for their enjoyment.

Woods and Juul[12] draw the following connections between replayability, strategy and tactics:

- strategic and tactical choices only feel significant, if the decision to make them is a difficult one,
- when there exists a strategy that is obviously more successful than others, the game becomes predictable and uninteresting,
- if a given tactical play is clearly more beneficial over the alternatives, then game can feel "scripted", as if it does not require actual decision-making from the player.

2.2 MACHINE LEARNING

Tom M. Mitchell[3], a pioneer in the field defined machine learning[13] as:

> "Machine learning is the study of computer algorithms that allow computer programs to automatically improve through experience. A computer program is said to learn from experience **E** with respect to some task **T** and some performance measure **P**, if its performance on **T**, as measured by **P**, improves with experience **E**."

[3] https://www.nae.edu/30398.aspx

Learning to play chess for example can be formalized as:

- **E**: the experience of having played many games of chess,
- **T**: the task of playing a game of chess,
- **P**: the probability that the program will win the next game.

Based on the kind of experience, machine learning can be divided into three main categories:

Supervised learning: the program is presented by data—prepared by a "supervisor"—consisting of input–output pairs. The goal of the program is to learn the general rule that maps these inputs to their desired outputs and be able to label data it has not seen before. Supervised learning problems can be divided into two categories: classification that involves predicting a class label, and regression that involves predicting a numerical value. An example of a supervised learning problem would be classifying emails as spam or not spam.

Unsupervised learning: the program is presented with data consisting of only inputs, without outputs or target variables. The goal of the program is to describe or extract relationships in the given data set. An example of an unsupervised learning problem would be to group similar music tracks into categories for a recommendation engine.

Reinforcement learning: the program must perform a certain goal while interacting with a dynamic environment. The interactions yield reinforcements, which can be positive (rewards) or negative (punishments) in relation to the system goal. The goal of the program is to maximize the sum of reinforcements. An example of a reinforcement learning problem would be for a robot to learn to navigate from point A to B.

Out of the above categories, learning to play a board game from scratch clearly falls into reinforcement learning. The dynamic environment is the game itself, with the interactions being the player making moves or taking turns. Reinforcements usually come at the end of the game when the winner is decided. The rest of this chapter introduces the reinforcement learning concepts used in the `sfinks` engine that was created as part of this work.

2.2.1 *Reinforcement learning*

The goal of reinforcement learning in short is *learning to make a good sequence of decisions*[14]. There are four key concepts that come into play:

Optimization: the goal is to find an optimal (or at least a very good) way of making decisions yielding best outcomes.

Delayed consequences: as opposed to supervised or unsupervised learning, where there is a single output for a given input, in reinforcement learning an agent is interacting with an environment through a sequence of actions. The impact of the decisions might not be apparent in the moment making them, which introduces two challenges:

- the decisions' long term ramifications have to be taken into account when planning, not just the immediate rewards,
- it is hard to attribute credit to decisions while learning, because the rewards can come much later in time.

Exploration: information has to be acquired through experience. The agent does not know in advance how its decisions are going to effect the environment or what decisions are associated with good outcomes. This means the agent will only gather information about the actions it is choosing to take.

Generalization: the agent needs to be able to react to situations it has not seen before.

Sequential decision making process

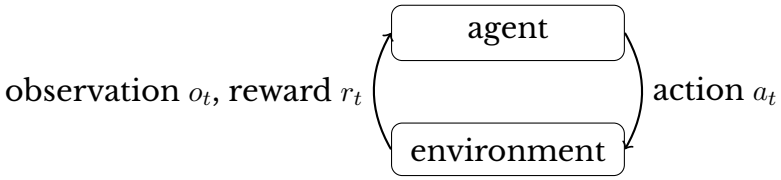


Figure 2.1: Agent–environment interaction in reinforcement learning.

Sequential decision making can be formalized as follows: the decision maker is called the *agent*, who is interacting with the *environment*. At each time step *t* the agent takes an action a_t. The environment is updated based

on a_t and emits observation o_t and reward r_t. The agent receives o_t and r_t and makes the next decision to take action a_{t+1}. This interaction is visualized in figure 2.1.

The sequence of these interactions is called a trajectory

$$T_t = (a_1, o_1, r_1, \ldots, a_t, o_t, r_t).$$

The way the environment changes can be deterministic, yielding a single observation and reward for a given state and action, or stochastic, yielding many possible observation and reward pairs for a given state and action.

The agent is making a decision based on its internal state, which is a function of the trajectory $s_t = f(T_t)$.

Common components of a reinforcement learning algorithm

A *model* is the agent's representation of how the environment changes in response to the actions it takes. Transition models predict the state, while reward models predict the immediate reward.

A *policy* π determines how the agent is choosing its actions, it is a mapping from states to actions $\pi : S \to A$. This too can be either deterministic, yielding a single action for a given state, or stochastic, yielding a distribution of actions for a given state.

The *value function*, V^π—which can be used to quantify the goodness/badness of a given state—is the expected discounted sum of future rewards (G_t) when the agent is acting based on policy π.

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \ldots \ \gamma \in (0, 1)$$

$$V^\pi(s_t = s) = \mathbb{E}[G_t | s_t = s],$$

where γ is the discount factor for weighting immediate versus future rewards.

The state–action value of a policy, $Q^\pi(s, a)$—which can be used for policy iteration—is the expected reward when taking action a in state s and then continue taking actions following policy π.

$$Q^\pi(s, a) = R(s, a) + \gamma \sum_{s' \epsilon S} \mathbb{P}(s'|s, a) V^\pi(s'),$$

where $\mathbb{P}(s'|s, a)$ is the probability of getting from state s to s' when taking action a.

Reinforcement learning agents can be divided into two categories based on the above components:

Model-based agents have an explicit model, and may or may not have a policy or a value function.

Model-free agents have an explicit value function and/or policy, but do not have a model.

Exploration and exploitation

The agent can only gather experiences from the actions it chooses to take. This can lead to a dilemma, because the agent has to decide between taking actions it knows to be good based on past experience (exploitation) and trying out new things that might enable it to make better decisions in the future (exploration).

Fundamental reinforcement learning problems

Evaluation: estimate the expected rewards given a policy.

Control: find the best policy.

Monte Carlo Policy Evaluation

The idea behind Monte Carlo Policy Evaluation is simple; if all the trajectories are finite, they can be sampled based on a policy π, and the expected return can be calculated as the average of the returns of these generated trajectories. The great thing about this method is that it only requires the trajectories to be finite, it does not require the presence of a model of the environment.

THE SFINKS FRAMEWORK

3

3.1 SPECIFICATION

`sfinks` is a framework for describing the rules of an arbitrary board game. It provides reinforcement learning agents that can learn to play the game. It can also output gameplay data generated by these agents.

3.1.1 *Restrictions*

In order to limit the scope of the project, the following restrictions apply to the board games the framework can handle at the moment:

- the game is played by a fixed number of players at a time,
- there is no player elimination, all players play through the entire game,
- the game ends in a finite number of actions,
- the players take their actions after one another in succession,
- the actions players can take are predefined by the rules,
- the actions players can take are deterministic, having a single, well-defined effect on the game state,
- playing the game requires only the knowledge of the rules, there is no dependence on outside information,
- all players have the same perception of the game state, there is no hidden information.

3.2 IMPLEMENTATION

`sfinks` is a cross-platform library written in C++17 (compilation was tested with g++ 10.2.0, clang++ 10.0, and MSVC 2019). Its only dependency is `cereal`[1], a header-only serialization library.

3.2.1 *Main components*

`sfinks` is comprised of three main elements:

- `sfinks::Agent` is a reinforcement learning agent,
- `sfinks::Game`: is an interface formalizing the rules of a board game,
- `sfinks::Engine` is a class that ties the `sfinks::Game` and `sfinks::Agent` together and provides methods for the user to generate gameplay data.

sfinks::Agent

`sfinks::Agent` is a model-free reinforcement learning agent operating based on an epsilon-greedy policy to balance exploration and exploitation. This can be seen in listing 3.1.

```
1  template <typename ActionId>
2  auto Agent::choose_action(const std::vector<ActionResult<ActionId>> &results)
3      -> ActionId
4  {
5    std::random_device rng_device;
6    std::mt19937 rng{rng_device()};
7    std::uniform_real_distribution<> uniform(0.0, 1.0);
8
9    const auto &[action_id, state, score] =
10       uniform(rng) < _exploration_rate ? random_action(results) :
11                                          policy_action(results);
12   _states_seen.push_front(state);
13   return action_id;
14 }
```

Listing 3.1: The epsilon-greedy policy of `sfinks::Agent`.

The greedy part of the policy, shown on listing 3.2, is very simple; choose the action that leads to the most valuable state.

[1] https://uscilab.github.io/cereal/

```
template <typename ActionId>
auto Agent::policy_action(const std::vector<ActionResult<ActionId>> &results) const
    -> ActionResult<ActionId>
{
  return *std::max_element(results.begin(), results.end(),
    [this](const auto &lhs, const auto &rhs) {
      return state_value(lhs.state) < state_value(rhs.state);
    }
  );
}
```

Listing 3.2: The greedy part of the policy of `sfinks::Agent`.

The agent keeps track of the state values in a tabular fashion and uses incremental Monte Carlo value iteration to update them, this can be seen in listing 3.3.

```
void Agent::process_reward(double reward) {
  for (const auto &state : _states_seen) {
    const auto &[it, insertion_result] = _state_values.try_emplace(state);
    auto &[state_value, times_seen] = it->second;
    times_seen += 1;
    state_value += (1.0 / times_seen) * (reward - state_value);
    reward = state_value;
  }
}
```

Listing 3.3: Incremental Monte Carlo value iteration in `sfinks::Agent`.

sfinks::Game

The purpose of `sfinks::Game` is twofold; it provides the structure for describing the rules of a board game the engine can work with, and based on this structure it can assemble the game state that the agents evaluate. The interface to describe the rules can be seen in listing S.2. It requires three template parameters: `<PlayerId>`, `<ActionId>`, and `<ResourceId>`.

`<PlayerId>` and `<ActionId>` could be anything, they do not carry any game specific semantic meaning. They are only used for what their names suggest, identifying players and actions. While these could be specified by the framework as concrete types, leaving them to be specified by the user provides greater flexibility, especially when it comes to `<ActionId>`, which could carry

relevant information for performing or undoing an action instead of just being a simple identifier. Examples of this can be seen in listings 4.3 and C.1.

Based on these identifiers, the game to be described needs to implement functions that provide information about

- all the players in the game,
- the current player,
- the current score of a given player,
- a list of available actions the current player can take,

and provide means to perform and undo a given action by a given player.

`<ResourceId>` encapsulates the notion of a dynamic game component that players can own. This can include spaces on the game board, game tokens, etc. In this first version of `sfinks` there is only a single category of resources, which are thought of as something "countable". This does have expressive power, but it is not necessarily the natural way of thinking about all the different game components.

The game to be described needs to implement functions that provide information about

- all the resources in the game,
- the number of a specific resource the game comes with,
- the number of a specific resource owned by a given player.

Based on this, the game state from the point of view of a given player is assembled as follows: for every resource, take note of how many the player owns, and whether there are still more available. This is not necessarily something that would work universally across different types of games (it could be the case that this is not enough information), but plays well with the agents' tabular method of keeping track of state values by keeping the state space limited.

sfinks::Engine

`sfinks::Engine` facilitates the reinforcement learning process by generating gameplay sequences and rewarding the agents appropriately. This can be

seen on listing S.1. It also provides functions to generate gameplay data for analysis by pitting different kinds of agents against each other, an example of this can be seen on listing 3.4.

3.3 USAGE EXAMPLE: TIC-TAC-TOE

Tic-tac-toe is a game for two players, $\mathbb{X}$ and $\mathbb{O}$, who take turns marking spaces in a 3×3 grid. Whoever manages to place three of their marks in a horizontal, vertical, or diagonal row is the winner. If the board is filled without anyone achieving this goal, the game results in a draw.

Listing 3.4 shows the usage of `sfinks::Engine`, while listings T.1, T.2 and T.3 contain the game logic that implements `sfinks::Game`.

The "resources" the players share are the nine cells of the grid, each identified by an integer between 0 and 8. The actions in this case are analogous to these resources (since the only thing the players can do is to occupy the cells), so they are also identified by an integer between 0 and 8. Players themselves are identified by an enum, which also doubles as their mark they place in the cells.

The example highlights that throughout the game implementation there is no notion of reinforcement learning, game state, agents, or anything alike. These details are all hidden behind the abstraction the framework provides.

```cpp
#include <sfinks/engine.hpp>
#include <tic_tac_toe/game.hpp>

auto main() -> int {
  tic_tac_toe::Game game;
  sfinks::Engine engine(&game, "tic_tac_toe");

  constexpr int num_of_learn_games = 1'000'000;
  engine.learn(num_of_learn_games);

  constexpr int num_of_play_games = 10'000;
  for (const auto &player_id : game.player_ids())
    engine.play_against_random_opponents(player_id, num_of_play_games);

  engine.play_according_to_policy(num_of_play_games);
}
```

Listing 3.4: Playing Tic-tac-toe with `sfinks`.

GAMEPLAY ANALYSIS

4.1 OUTLINE

This chapter introduces two board games that are analyzed by the help of the `sfinks` engine. The general outline is as follows:

1. Implement the interface required by `sfinks::Game`.
2. Pit random agents against each other to see if there are any inherent imbalances in the game.
3. Train sfinks agents to be able to play the game to the point where they can easily win against random opponents.
4. Pit the sfinks agents against opponents playing greedily to see how well the greedy approach works for the game.
5. Pit the sfinks agents against each other and look for any emerging patterns, over- and underpowered elements in the generated gameplay data.

4.1.1 *How to read the diagrams in this chapter*

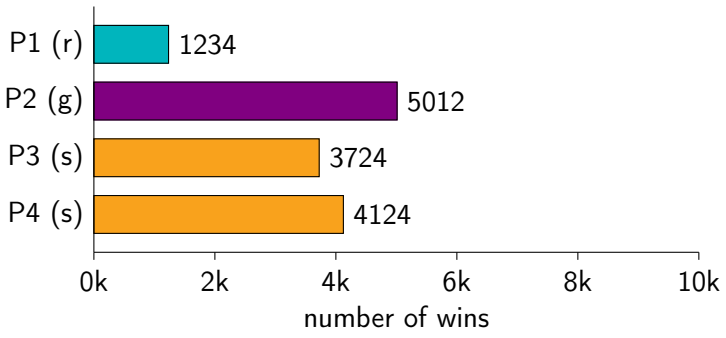


Figure 4.1: A four-player game, **r**andom vs **g**reedy vs **s**finks, 10000 games.

Figure 4.1 shows four players competing against each other in the same game:

- P1, a **random** agent: identified by the color blue and the letter (**r**), it selects its actions randomly,

- P2, a **greedy** agent: identified by the color purple and the letter (**g**), it selects the action that leads to the state with the highest score (disregarding any potential future rewards),
- P3, a **sfinks** agent: identified by the color orange and the letter (**s**), it selects the action that leads to the state with the highest state value based on prior experience,
- P4, another **sfinks** agent: it has the same decision-making mechanism as P3, but not necessarily the same experience.

Figure 4.1 shows that out of 10000 games played

- Player 1, who played randomly, won 1234 games,
- Player 2, who played greedily, won 5012 games,
- Player 3, who played according to some pre-learned policy, won 3724 games,
- Player 4, who played according to some pre-learned policy, won 4124 games.

Players took turns in the order their identifiers suggest:

$$\text{Player 1} \to \text{Player 2} \to \text{Player 3} \to \text{Player 4} \to \text{Player 1} \to \dots$$

The number of wins includes ties for first place, so the sum of the individual win values might be bigger than the number of games played.

4.2 TIC-TAC-TOE

The rules and implementation of Tic-tac-toe can be found in section 3.3. Tic-tac-toe is a solved game[15], it is known that

- the first player can always achieve at least a draw when playing optimally,
- the second player can always achieve at least a draw when playing optimally, but can never win against a first player who is playing optimally.

Since Tic-tac-toe is a very simple game with a small state space[16], it is a great fit for a "sanity check" to see how well the sfinks agents perform.

4.2.1 *Analysis*

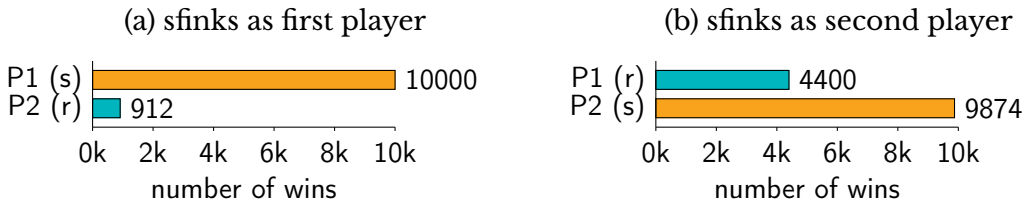


Figure 4.2: Tic-tac-toe, **s**finks vs **r**andom, 10000 games.

Figure 4.2 shows a sfinks agent competing against a random agent.

When playing as the first player, the sfinks agent never lost, it won 9088 games, and tied in the remaining 912. It learned to start every game with marking the center cell of the grid which leads to the highest chance of winning[15].

When playing as the second player, the sfinks agent won 5600 games, tied in 4274 games, and lost only 126 games.

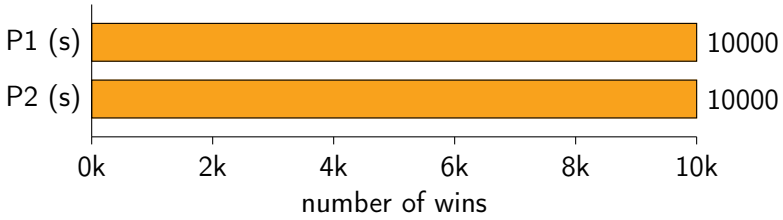


Figure 4.3: Tic-tac-toe, **s**finks, 10000 games.

Figure 4.3 shows 2 sfinks agents playing the game. All 10000 games ended in a draw—the best result both players can achieve when playing optimally—as expected.

4.3 COLORS

Colors is a game for 2 or 3 players, who take turns collecting colored marbles of different values. The player with the marble collection of the highest value wins the game.

4.3.1 *Rules*

Setup

There are 5 types of marbles in the game: red, green, yellow, blue and black. Black marbles are only used for penalties, so they are always available and cannot be taken voluntarily. The others have the following quantities:

- red: 3,
- green: 5,
- yellow: 4,
- blue: 2.

These are put on the table and the starting player is chosen randomly.

Playing the game

Players take turns after one another collecting a single marble with the following restrictions:

- the players are forbidden to take the same colored marble as in their previous turn,
- if a player took a red marble in their previous turn, now they must take a blue one,
- if a player took a green marble in their previous turn, now they must take a yellow one,
- if a player must take a specific marble that is not available, they must take a black marble instead.

End of the game

The game ends when there are no more available marbles (aside from black).

Scoring

Players now add up the values of their marbles (including black ones). The player with the most points wins the game. In case of a tie, all tied players are considered winners. The values of the marbles are as follows:

- red: 7,
- green: 4,
- yellow: 0,
- blue: -4,
- black: -8.

Example game

P1	P2	P3
yellow	red	blue
red	blue	green
black	green	yellow
red	yellow	green
black	green	yellow
green		

Table 4.1: A game of Colors with 3 players.

Table 4.1 depicts a game of Colors played by 3 players. The scores are:

Player 1: yellow (0) + red (7) + black (-8) + red (7) + black (-8) + green (4) = **2**.

Player 2: red (7) + blue (-4) + green (4) + yellow (0) + green (4) = **11**.

Player 3: blue (-4) + green (4) + yellow (0) + green (4) + yellow (0) = **4**.

Player 2 wins the game.

4.3.2 *Implementation*

Listing C.1 shows the general structure of the game implementation. The template parameters required by `sfinks::Game` are:

- `PlayerId`: a non-negative integer,
- `ResourceId`: the resources here are the marbles which can be identified by their color, so `ResourceId` is an enum of color names,
- `ActionId`: a struct encapsulating the information needed to perform a player turn:
  - the id of the acting player,
  - the color to take this turn,
  - and the color taken in the previous turn.

`colors::Player` keeps track of the marbles owned by the player and the color of the marble taken last turn. `colors::Game` keeps track of all the players and the available marbles while implementing the interface required by `sfinks::Game`. The implementation of the game logic is straightforward, listing C.2 shows `colors::Game::available_actions()` as an example.

4.3.3 *Analysis*

Colors for 2 players

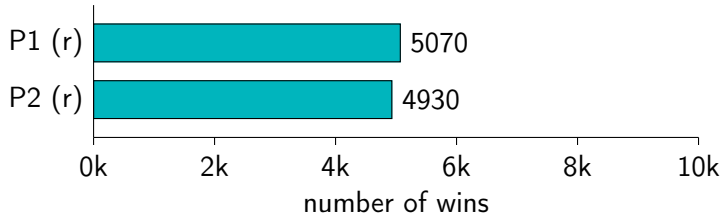


Figure 4.4: Colors with 2 players, **r**andom, 10000 games.

Figure 4.4 shows 2 random agents playing Colors against each other. The win ratio is roughly the same, which does not suggest any inherent imbalance between the two player positions.

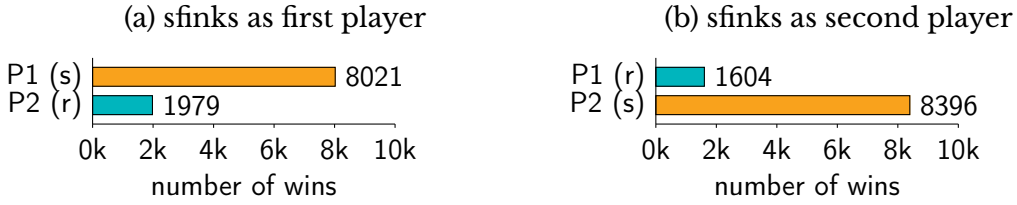


Figure 4.5: Colors with 2 players, **s**finks vs **r**andom, 10000 games.

Figure 4.5 shows a sfinks agent competing against a random agent. The sfinks agent wins by a huge margin in both positions, which suggests that it managed to learn the game adequately.

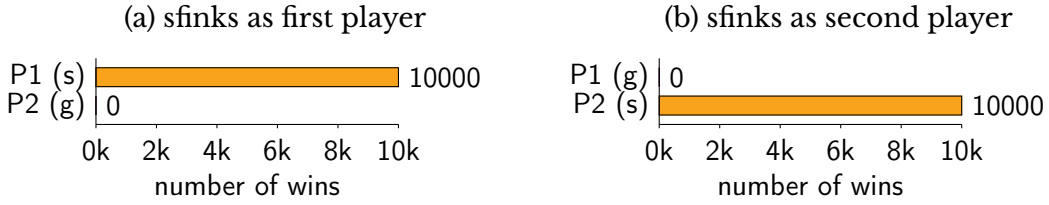


Figure 4.6: Colors with 2 players, **s**finks vs **g**reedy, 10000 games.

Figure 4.6 shows a sfinks agent competing against a greedy agent. The sfinks agent wins all the games in both positions showing that the greedy approach simply does not work. This suggests that the winning strategy is not obvious, which is good for replayability.

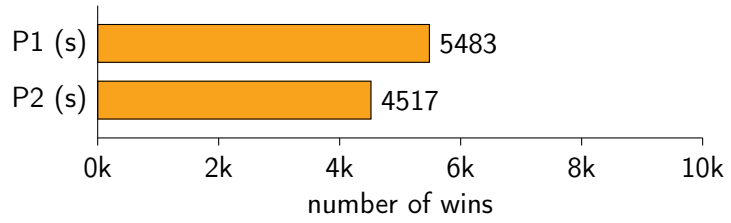


Figure 4.7: Colors with 2 players, sfinks, 10000 games.

Figure 4.7 shows 2 sfinks agents playing the game. The win ratio is roughly the same and there are no ties.
The most commonly taken set of winning actions for Player 1 were

$$yellow \rightarrow red \rightarrow blue \rightarrow red \rightarrow black \rightarrow green \rightarrow yellow \rightarrow green$$

308 times (out of 5483).
The first color picked up by Player 1 in the games it has won was:

- yellow: 3187 times,
- green: 1548 times,
- red: 651 times,
- blue: 97 times.

The most commonly taken set of winning actions for Player 2 were

$$red \rightarrow blue \rightarrow yellow \rightarrow red \rightarrow black \rightarrow yellow \rightarrow green \rightarrow black \rightarrow green$$

379 times (out of 4517).
The first color picked up by Player 2 in the games it has won was:

- red: 2445 times,
- yellow: 880 times,
- green: 664 times,
- blue: 528 times.

This suggests that there are many different ways to win, which is great for replayability.

Colors for 3 players

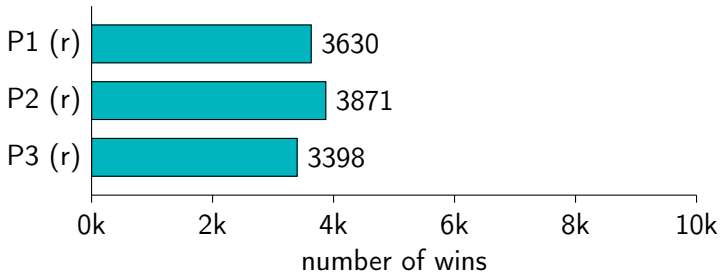


Figure 4.8: Colors with 3 players, **r**andom, 10000 games.

Figure 4.8 shows 3 random agents playing Colors against each other. The win ratio is roughly the same, which does not suggest any inherent imbalance between the three player positions.

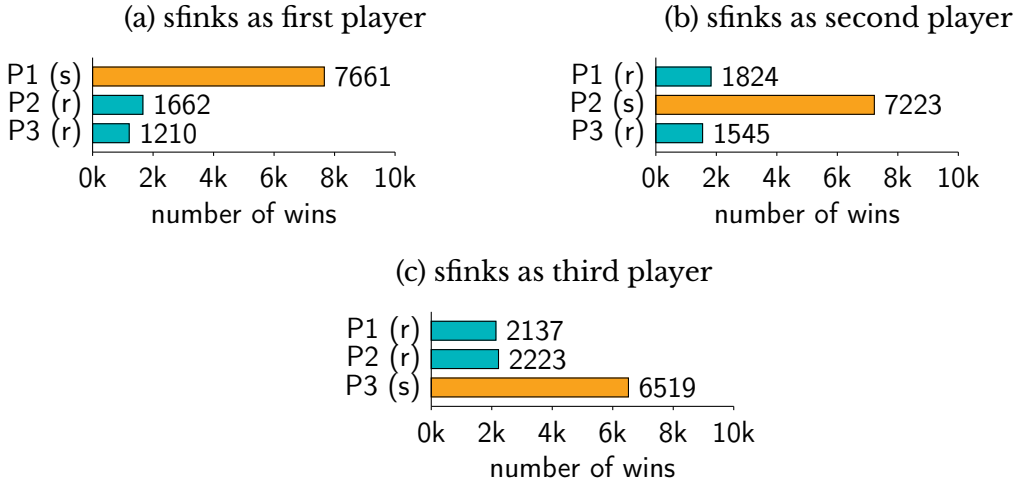


Figure 4.9: Colors with 3 players, **s**finks vs **r**andom, 10000 games.

Figure 4.9 shows a sfinks agent competing against two random agents. The sfinks agent wins by a huge margin in all three positions, which suggests that it managed to learn the game adequately.

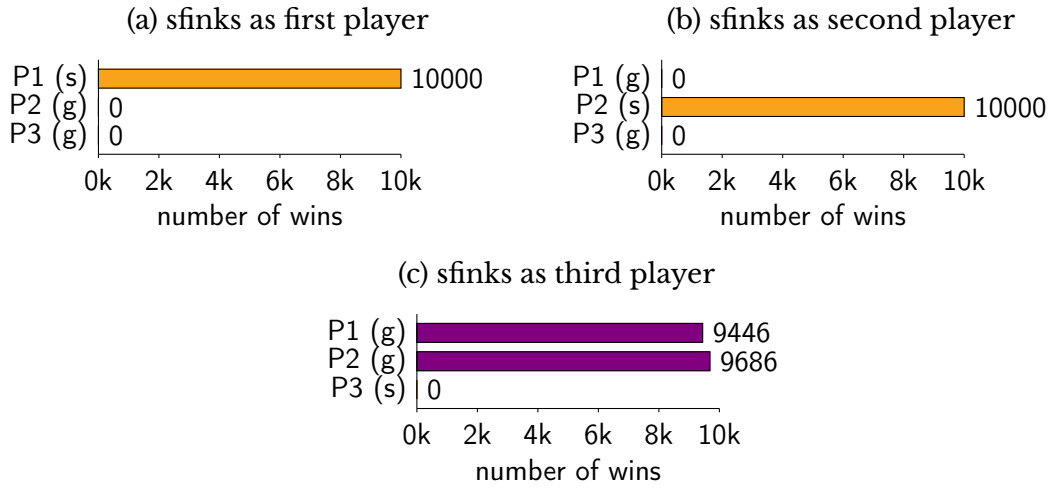


Figure 4.10: Colors with 3 players, **s**finks vs **g**reedy, 10000 games.

Figure 4.10 shows a sfinks agent competing against two greedy agents. The sfinks agent wins all games if playing as Player 1 or Player 2, but loses all games as Player 3 (while the two greedy agents win with a tie in almost all cases). This leads to two separate conclusions:

For one, the greedy approach is beatable in many scenarios, which would mean the winning strategy is not obvious, which is good for replayability.

As for the other, it seems like if the first two players both play greedy, the third player has no way to win. This suggests an imbalance between player positions, depending on how the first two players play, the third might not have any chance of winning regardless of the actions it takes.

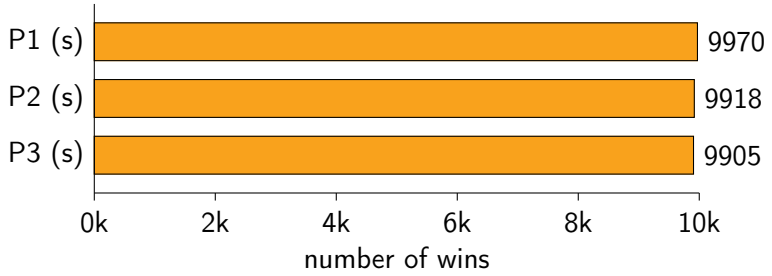


Figure 4.11: Colors with 3 players, **s**finks, 10000 games.

Figure 4.11 shows 3 sfinks agents playing the game. Almost all games led to a three-way tie with the same moves taken (shown in table 4.2), and since everyone wins, no one has an incentive to change their strategy, which renders the game not worth playing.

P1	P2	P3
green	green	yellow
yellow	yellow	green
red	red	yellow
blue	blue	red
green	green	

Table 4.2: A three-way tie in Colors.

Summary

For two players: there are no obvious imbalances. The greedy approach is easily beatable. There are multiple ways to win. These factors offer great replayability, and are essential, since the game contains no random elements.

For three players: here the game has multiple problems. The third player is not fully in control; depending on how the the first two players play, he could be left with no ways to win. Players can also achieve a three-way tie where everyone wins, which when discovered could make it pointless to play the game any further.

4.4 BIRD LADY

4.4.1 *Overview*

Bird Lady is card based game for 2 or 3 players. It is a simplified version of *Cat Lady*[17]. Players take on the role of bird ladies, drafting two cards at a time, collecting aviaries, food, and of course birds. They have to make sure they gather enough food for all of their birds, because hungry ones will subtract points from their final score. The player with the highest total victory points (VPs) wins the game.

4.4.2 *Setup*

The *mystery bird* cards must be separated out and put on the table face up (4.12(a)). The rest of the cards are shuffled into a deck that is placed face down on the table (4.12(b)). Four cards are drawn and arranged in a 2×2 grid (4.12(c)). The first player is selected randomly.

4.4.3 *Playing the game*

Players take turns after one another in a clockwise manner until the end of the game.

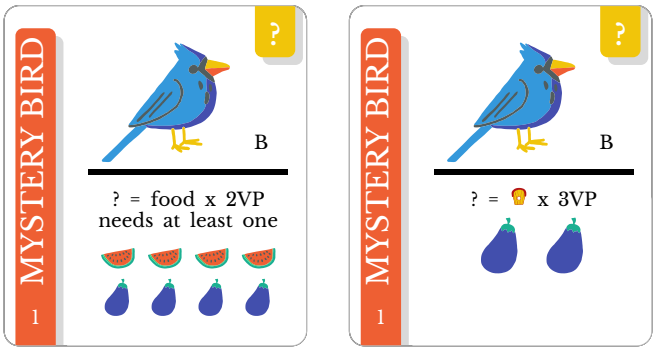
Taking a turn

During a player's turn, one must take an entire row or column of cards from the face up game cards on the table. When players take a card, they place it face up in front of themselves. In the case of 4.12(c) the options are:

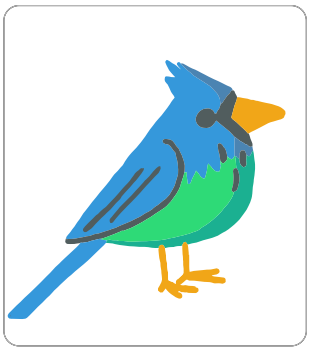
- **left column**: food (vegetable), bird,
- **right column**: aviary, food (tomato),
- **top row**: food (vegetable), aviary,
- **bottom row**: bird, food (tomato).

Optionally at the beginning of a turn (before taking cards) the player can opt to discard two *egg* cards to acquire his/her choice of an available *mystery bird* (4.12(a)). This can only be performed once per turn.

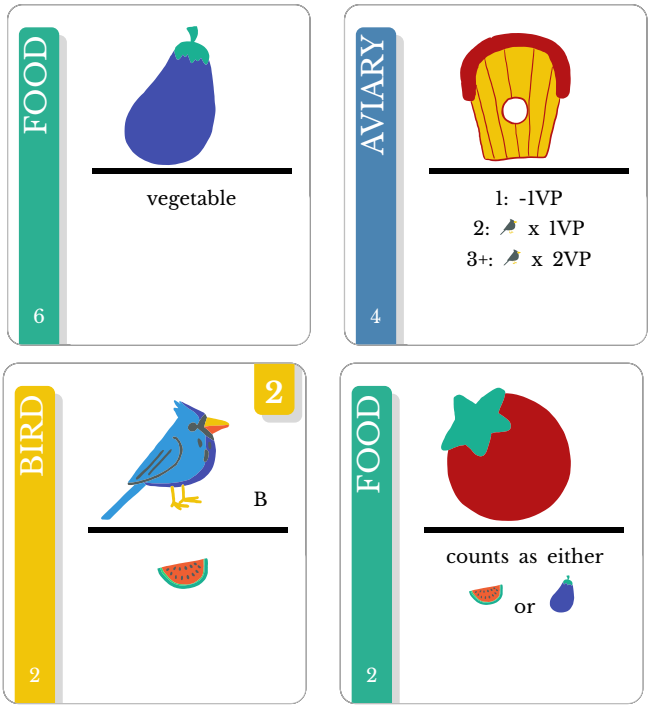
The row or column is re-filled with two face up cards drawn from the deck, then it is the next player's turn.



(a) mystery birds



(b) deck



(c) available cards

Figure 4.12: Bird Lady setup.

End of the game

The game ends when an empty row or column needs to be filled and there are not enough cards in the deck to do so.

Scoring

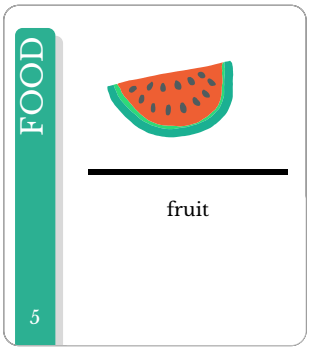
At the end of the game players must assign their food cards to their birds or declare them leftover. Then they add up all their VPs from *birds*, *mystery birds*, *eggs*, and *aviaries* including any penalties. The player with the most VPs wins the game. In case of a tie, all tied players are considered winners.

4.4.4 *Card types*

The full list of cards is shown in B.2. Each card has a number in its lower left corner showing how many of it is included in the game.

Food (4.13(a))

Food cards are needed to feed birds at the end of the game. There are two types of food: **fruit** (pictured by a watermelon) and **vegetable** (pictured by an eggplant). A **tomato** can be used as any type of food. At the end of the game for each pair of leftover/unused food the player loses 1 VP.



(a)

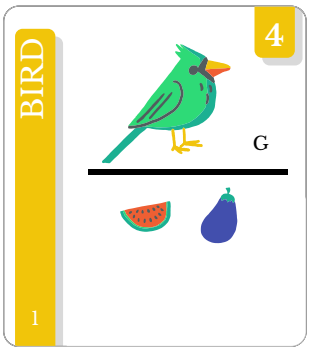
Birds (4.13(b), 4.13(c), 4.13(d))

At the end of the game all birds need to be fed, their needs are shown in the bottom of the card. If the bird is fully fed, the player gains VPs equal to the number shown on the top right corner of the card. If the bird is not fed, the player loses 1 VP.

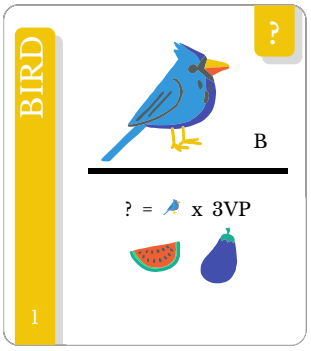
Birds cannot be partially fed, meaning they either take all their required food or none. For example if bird 4.13(b) is fed one *fruit* and one *vegetable*, the player is awarded 4 VPs. If it is not fully fed, the player will lose 1 VP.

Birds also have a color indicated by their picture and a capital letter next to their picture (**B** = blue, **G** = green). Bird 4.13(c) awards 2 VPs for each fully fed blue bird (including itself) when fully fed.

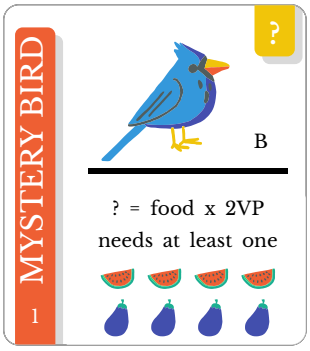
Mystery birds work the same way as normal birds, but they can only be acquired through eggs. Mystery bird 4.13(d) can be fed a maximum of 4 fruits and 4 vegetables, but only needs one food to count as fully fed. It grants the player 2 VPs for each assigned food.



(b)

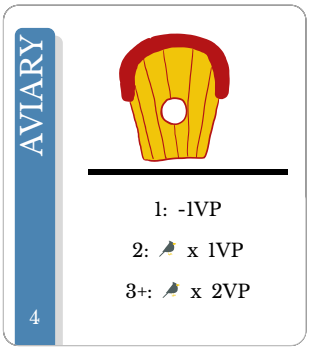


(c)

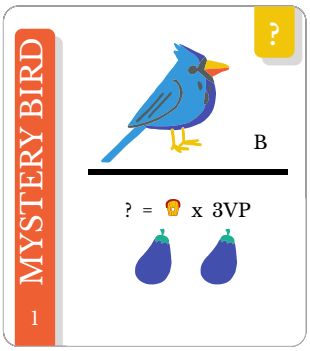


(d)

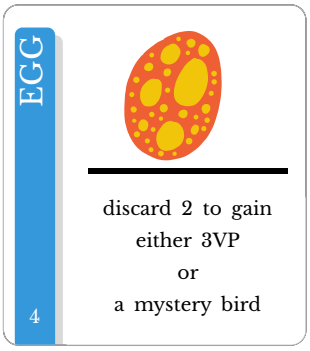
Figure 4.13



(a)



(b)



(c)

Figure 4.14

Aviaries (4.14(a))

At the end of the game the players gain VPs based on the number of aviary cards they had collected. If they have only one, they lose 1 VP. If they have two, they gain 1 VP for each of their fully fed birds. If they have three or more, they gain 2 VPs for each of their fully fed birds.

Mystery bird 4.14(b), when fully fed, awards 3 VPs for each aviary the player has.

Eggs (4.14(c))

Eggs can be used to acquire mystery birds. At the end of the game each pair of egg cards are worth 3 VPs.

4.4.5 *Implementation*

The implementation consist of the following classes:

bird_lady::CardHandle

This is an enum that encapsulates the different types of cards in the game, and is used as the `ResourceId` in the context of `sfinks::Game`.

```
enum class CardHandle {
  none,
  fruit, vegetable, tomato,
  aviary, egg,
  bird_01, bird_02, bird_03, bird_04, bird_05,
  mystery_bird_01, mystery_bird_02
};
```

Listing 4.1: `bird_lady::CardHandle`

bird_lady::Board and bird_lady::Deck

`bird_lady::Board` is responsible for handling the 2×2 grid of cards the players are drawing cards from. It stores 4 `bird_lady::CardHandle`s internally and provides an interface to access them by row or column, returning `bird_lady::BoardSlice`s.

`bird_lady::Deck` encapsulates the deck of cards on the table. It can provide new cards for the board in the form of a `bird_lady::BoardSlice`.

```
struct BoardSlice {
  CardHandle first;
  CardHandle second;
};
```

Listing 4.2: `bird_lady::BoardSlice`

bird_lady::Player

This class keeps track of the cards a player acquires during gameplay.

bird_lady::BirdScorer and bird_lady::Bird

`bird_lady::Bird` encapsulates the properties of a given bird relevant for scoring, its color, food requirements, and the way how the awarded VPs are calculated if the bird is fully fed.

`bird_lady::BirdScorer` is responsible for calculating the score for the bird and food cards of a player. It determines the highest possible score using the *Z3 Theorem Prover*[1]. The problem is formalized by the following statements:

- each food a bird can take is registered as variable in Z3 with the constraint that it can either be the required food type, a tomato, or `no food`,
- a bird is considered fully fed if none of its food variables have the value `no food`,
- the number of variables taking a specific food value is limited by the number of that type of food the player gathered during gameplay,
- the total to maximize consists of the sum of the values of the birds registered (some reward if fully fed, penalty otherwise) plus the penalty for leftover food.

Parts of the implementation can be seen in listings B.1, B.2, and B.3.

bird_lady::Scorer

This class is responsible for calculating the total score for a given player.

bird_lady::Game

This is the class that ties everything together and implements the interface required by `sfinks::Game`. It manages the players, the board, the deck, and the mystery bird cards.

The template parameters required by `sfinks::Game` are:

- `PlayerId`: a non-negative integer,
- `ResourceId`: cards are the shared resources in this game, so `bird_lady::CardHandle` is used,

[1] https://github.com/Z3Prover/z3

- `ActionId`: a struct encapsulating the information needed to perform a player turn:
  - which column or row was selected by the player,
  - the id of the acting player,
  - the cards in the selected column or row,
  - a mystery bird (with the value of `CardHandle::none` if the player does not take a mystery bird as part of the action).

```
using PlayerId = size_t;
using ResourceId = CardHandle;

struct ActionId {
  size_t slice_id;
  PlayerId player_id;
  BoardSlice slice_contents;
  CardHandle mystery_bird;
};

class Game : public sfinks::Game<PlayerId, ActionId, ResourceId> {
public:
  Game(int number_of_players);
  // functions required by sfinks::Game
  // ...
private:
  size_t _current_player_id = 0;
  std::vector<Player> _players;
  std::list<CardHandle> _mystery_birds;
  Board _board;
  Deck _deck;
};
```

Listing 4.3: `bird_lady::Game`

4.4.6 *Analysis*

Bird Lady for 2 players

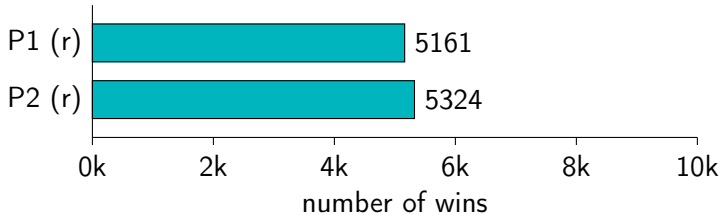


Figure 4.15: Bird Lady with 2 players, **r**andom, 10000 games.

Figure 4.15 shows 2 random agents playing Bird Lady against each other. The win ratio is roughly the same, which does not suggest any inherent imbalance between the two player positions.

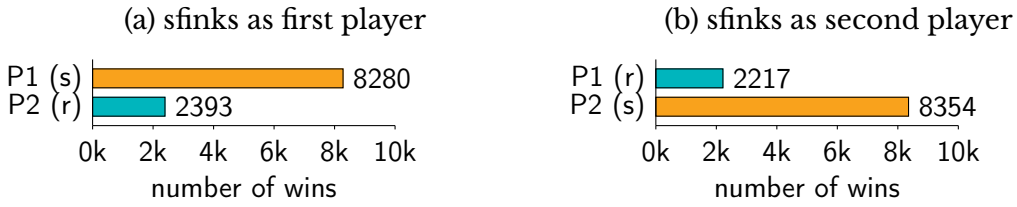


Figure 4.16: Bird Lady with 2 players, **s**finks vs **r**andom, 10000 games.

Figure 4.16 shows a sfinks agent competing against a random agent. The sfinks agent wins by a huge margin in both positions, which suggests that it managed to learn the game adequately.

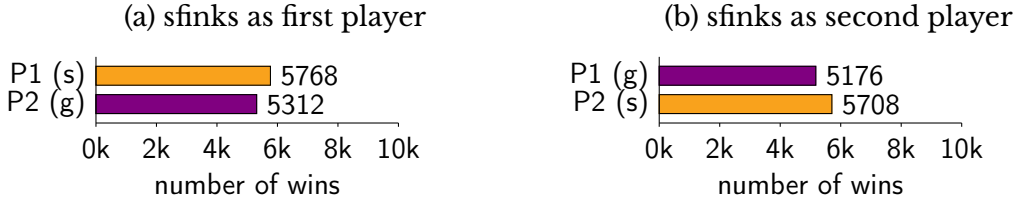


Figure 4.17: Bird Lady with 2 players, **s**finks vs **g**reedy, 10000 games.

Figure 4.17 shows a sfinks agent competing against a greedy agent. The win ratio is roughly the same, with the sfinks agent coming up slightly ahead in both positions. Based on this, the greedy strategy seems viable, but also beatable. It could also be the case that with more training, the sfinks agent could achieve a better a win ratio.

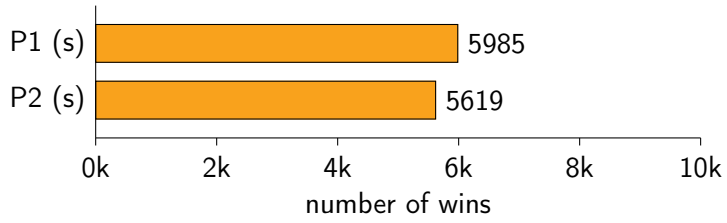


Figure 4.18: Bird Lady with 2 players, sfinks, 10000 games.

Figure 4.18 shows 2 sfinks agents playing the game. The win ratio is roughly the same, 1604 games ended in a tie.

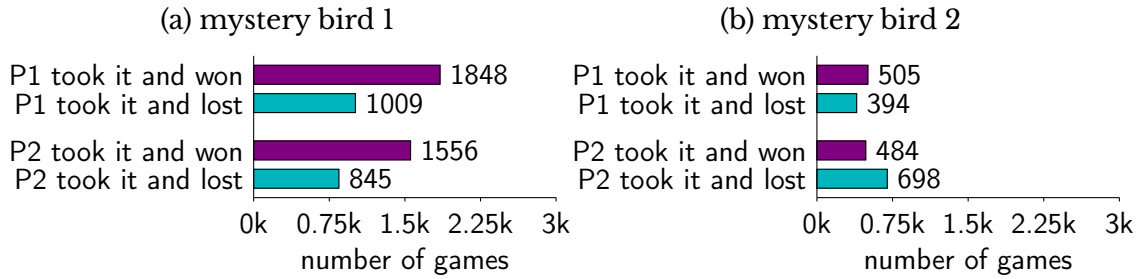


Figure 4.19: Bird Lady with 2 players, mystery bird preferences, 10000 games.

Figure 4.19 shows the players' preferences when choosing mystery birds. Mystery bird 1 is highly preferred over mystery bird 2 by both players, indicating that mystery bird 2 is underpowered. As for mystery bird 1, players won more than 64% of their games when they managed to take it, this signals that this card is overpowered when playing with two players (figure 4.25 shows that this does not occur when playing with three players).

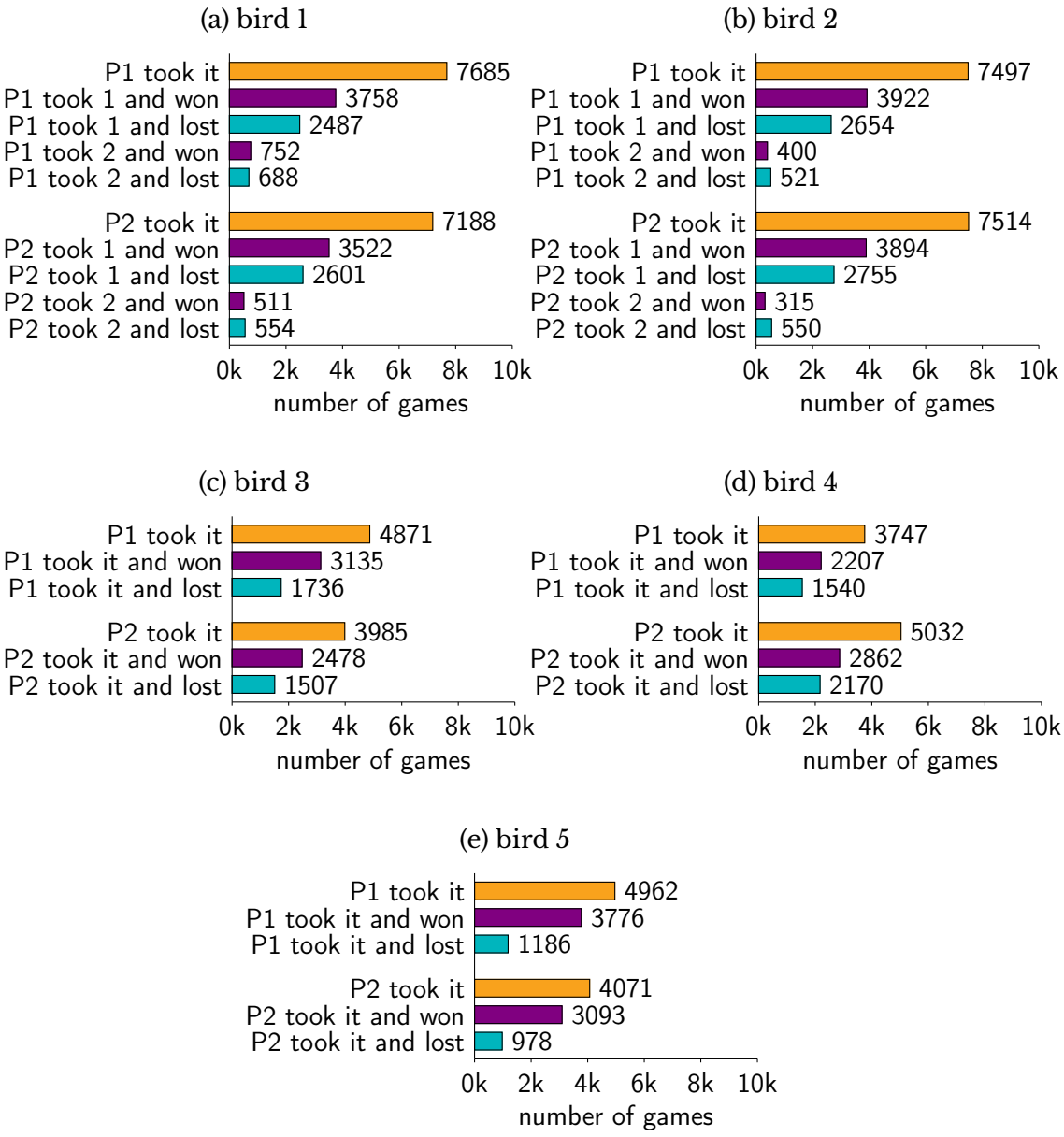


Figure 4.20: Bird Lady with 2 players, bird preferences, 10000 games.

Figure 4.20 shows how many times each bird card was taken by the players. For bird 5, players won more than 75% of their games when they managed to take it, this signals that this card is overpowered.

Bird Lady for 3 players

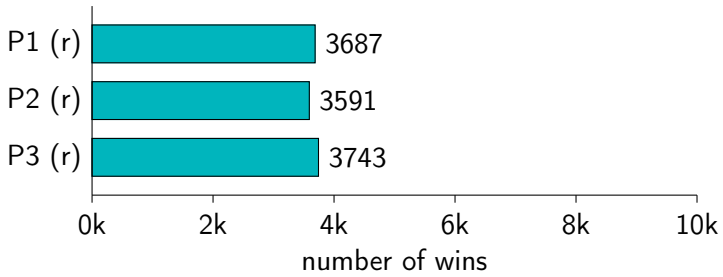


Figure 4.21: Bird Lady with 3 players, **r**andom, 10000 games.

Figure 4.21 shows 3 random agents playing Bird Lady against each other. The win ratio is roughly the same, which does not suggest any inherent imbalance between the three player positions.

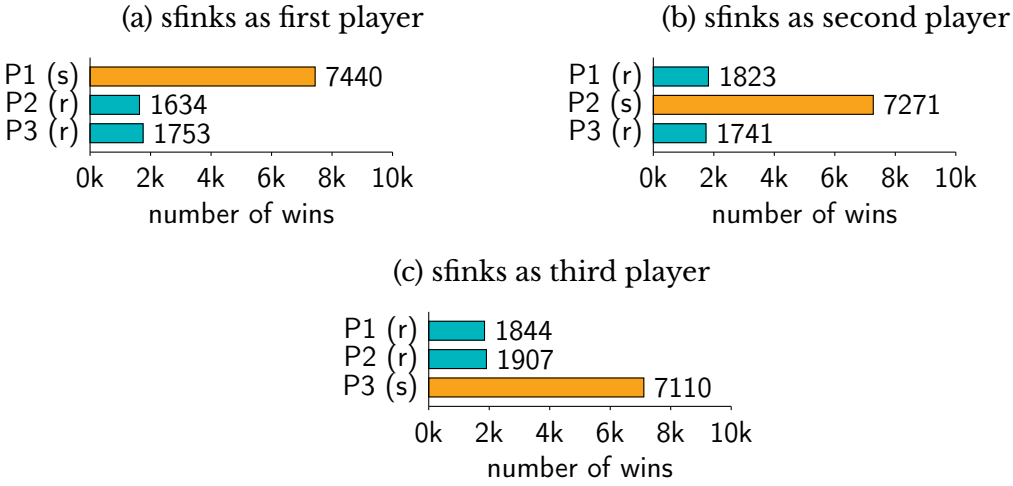


Figure 4.22: Bird Lady with 3 players, **s**finks vs **r**andom, 10000 games.

Figure 4.22 shows a sfinks agent competing against two random agents. The sfinks agent wins by a huge margin in all three positions, which suggests that it managed to learn the game adequately.

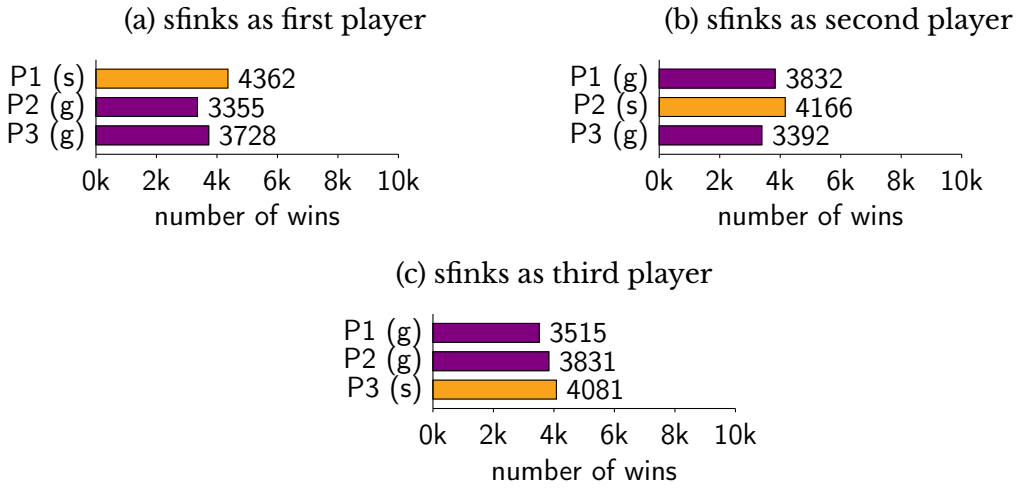


Figure 4.23: Bird Lady with 3 players, **s**finks vs **g**reedy, 10000 games.

Figure 4.23 shows a sfinks agent competing against two greedy agents. The win ratio is roughly the same, with the sfinks agent coming up slightly ahead in all three positions. Based on this, the greedy strategy seems viable, but also beatable. It could also be the case that with more training, the sfinks agent could achieve a better a win ratio.

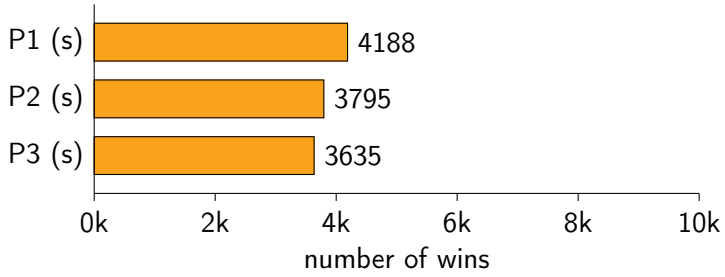


Figure 4.24: Bird Lady with 3 players, **s**finks, 10000 games.

Figure 4.24 shows 3 sfinks agents playing the game. The win ratio is roughly the same, 1618 games ended in a tie.

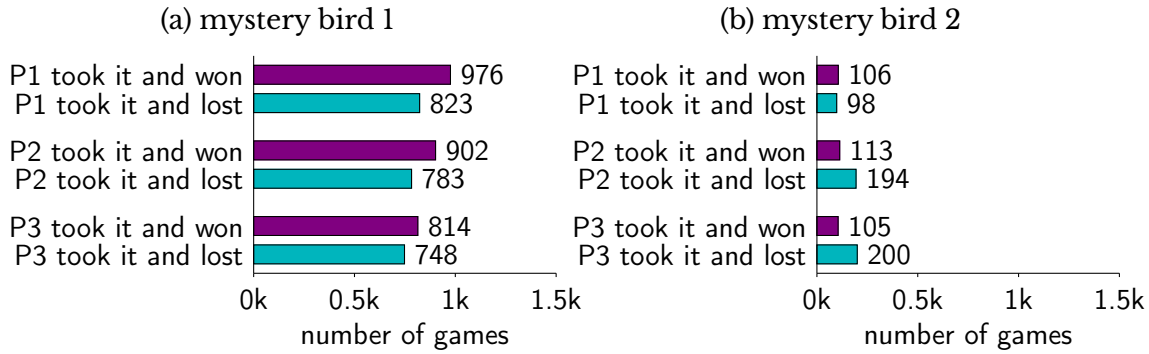


Figure 4.25: Bird Lady with 3 players, mystery bird preferences, 10000 games.

Figure 4.25 show the players' preferences when choosing mystery birds. Mystery bird 1 is highly preferred over mystery bird 2 by both players, indicating that mystery bird 2 is underpowered.

Summary

For both two and three players: the fact that the greedy approach is beatable and that the game is inherently random makes for good replayability.

For two players: Mystery bird 1 and bird 5 are overpowered. Mystery bird 2 is underpowered.

For three players: Mystery bird 2 is underpowered.

SUMMARY

5.1 EVALUATION OF THE COMPLETED WORK

5

The end result of this project is the `sfinks` library. The source code can be found on github: https://github.com/zosiu/sfinks.

It was shown through multiple examples that

- it is possible to describe the rules of different board games with the interface that `sfinks::Game` provides,
- reinforcement learning is properly abstracted away from the end user,
- the `sfinks::Agent`, even though it is rather rudimentary, was able to learn to play different board games to an extent where the generated gameplay data was useful for uncovering undesirable characteristics.

It can be concluded that the idea shows promise and would worth further investigation.

5.2 FUTURE WORK

`sfinks` could be improved in a number of ways. The most pressing issue is the tabular state handling of the agents, which should be swapped out for some kind of approximation method. This would yield two huge benefits: it would significantly reduce the memory needed to store and handle the agents' internal state, and it would also enable the agents to have some ideas about states they have not encountered before. The resource abstraction that is currently used to build the game state could also be used to generate features for the approximation.

The framework would also need to support actions with stochastic outcomes in order to be able to describe a "universal" board game. This would require modification in two areas: the policy of the agent, and the rule description interface.

APPENDIX

BIRD LADY

```
void BirdsScorer::register_bird(CardHandle card_handle, const Bird &bird) {
  Z3BirdData bird_data{card_handle};
  const size_t bird_id = _birds.size();

  for (const auto color : {BirdColor::blue, BirdColor::green})
    bird_data.z3_color_consts.emplace(
        color, _context.int_val(bird.color == color ? 0 : 1));

  for (const auto &[food, quantity] : bird.feeding.food_requirements) {
    for (size_t i = 0; i < quantity; i++) {
      std::string food_var_name = "bird_" + std::to_string(bird_id) + "_food_" +
                                  std::to_string(bird_data.z3_food_vars.size());
      auto it = bird_data.z3_food_vars.emplace(
          bird_data.z3_food_vars.end(),
          _context.int_const(food_var_name.c_str()));
      _optimizer.add((*it == z3_food_type(food)) ||
                     (*it == z3_food_type(BirdFood::tomato)) ||
                     (*it == z3_food_type(BirdFood::none)));
    }
  }

  _birds.emplace_back(bird_data);
}
```

Listing B.1: Registering birds in Z3 in Bird Lady.

```
auto BirdsScorer::z3_bird_unhappiness(const Z3BirdData &bird_data) const
    -> z3::expr {
  const Bird &bird = consts::birds.at(bird_data.bird_id);
  switch (bird.feeding.happy_when) {
  case BirdHappiness::fully_fed:
    return std::accumulate(
        bird_data.z3_food_vars.begin(), bird_data.z3_food_vars.end(),
        _z3_consts.at(zero), [this](const auto &exp, const auto &z3_food_var) {
          return z3::ite(z3_food_var == z3_food_type(BirdFood::none),
                         _z3_consts.at(one), _z3_consts.at(zero)) +
                 exp;
        });
  case BirdHappiness::fed_at_least_one_food:
    return z3::ite(z3_bird_proper_food_count(bird_data) > _z3_consts.at(zero),
                   _z3_consts.at(zero), _z3_consts.at(one));
  default:
    std::throw_with_nested(std::invalid_argument("unknown happiness criteria"));
  }
}

auto BirdsScorer::z3_bird_score(const Z3BirdData &bird_data) -> z3::expr {
  const Bird &bird = consts::birds.at(bird_data.bird_id);
  z3::expr scoring_expr(_context);
  switch (bird.scoring.type) {
  case BirdScoringType::simple:
    scoring_expr = _context.int_val(1);
    break;
  case BirdScoringType::for_every_blue:
    scoring_expr = z3_number_of_happy_birds(BirdColor::blue);
    break;
  case BirdScoringType::for_every_aviary:
    scoring_expr = _z3_consts.at(number_of_aviaries);
    break;
  case BirdScoringType::for_every_food_fed:
    scoring_expr = z3_bird_proper_food_count(bird_data);
    break;
  default:
    std::throw_with_nested(std::invalid_argument("unknown scoring type"));
  }

  return z3::ite(z3_bird_unhappiness(bird_data) == _z3_consts.at(zero),
                 scoring_expr * _context.int_val(bird.scoring.multiplier),
                 _z3_consts.at(unhappy_bird_penalty));
}
```

Listing B.2: Bird scoring with Z3 in Bird Lady.

```
auto BirdsScorer::score() -> BirdsScoreDetails {
  for (const auto food :
       {BirdFood::fruit, BirdFood::vegetable, BirdFood::tomato})
    _optimizer.add(z3_food_availability_requirement(food));

  const auto total =
      z3_birds_score() + z3_leftover_food() / _context.int_val(2) *
                             _z3_consts.at(leftover_food_penalty_per_pair);
  _optimizer.maximize(total);
  _optimizer.check();

  z3::model m = _optimizer.get_model();

  int score = m.eval(total).get_numeral_int();
  int happy_birds = m.eval(z3_number_of_happy_birds()).get_numeral_int();
  int leftover_food = m.eval(z3_leftover_food()).get_numeral_int();

  return {score, happy_birds, leftover_food};
}

auto BirdsScorer::z3_food_availability_requirement(BirdFood food) const
    -> z3::expr {
  auto food_used = std::accumulate(
      _birds.begin(), _birds.end(), _z3_consts.at(zero),
      [food, this](const auto &exp, const auto &bird_data) {
        return std::accumulate(
                    bird_data.z3_food_vars.begin(), bird_data.z3_food_vars.end(),
                    _z3_consts.at(zero),
                    [food, this](const auto &exp, const auto &food_var) {
                      return z3::ite(food_var == z3_food_type(food),
                                     _z3_consts.at(one), _z3_consts.at(zero)) +
                             exp;
                    }) +
               exp;
      });

  return food_used <= z3_food_available(food);
}

auto BirdsScorer::z3_birds_score() -> z3::expr {
  return std::accumulate(_birds.begin(), _birds.end(), _z3_consts.at(zero),
                         [this](const auto &exp, const auto &bird_data) {
                           return z3_bird_score(bird_data) + exp;
                         });
}
```

Listing B.3: Using Z3 to maximize bird scores in Bird Lady.

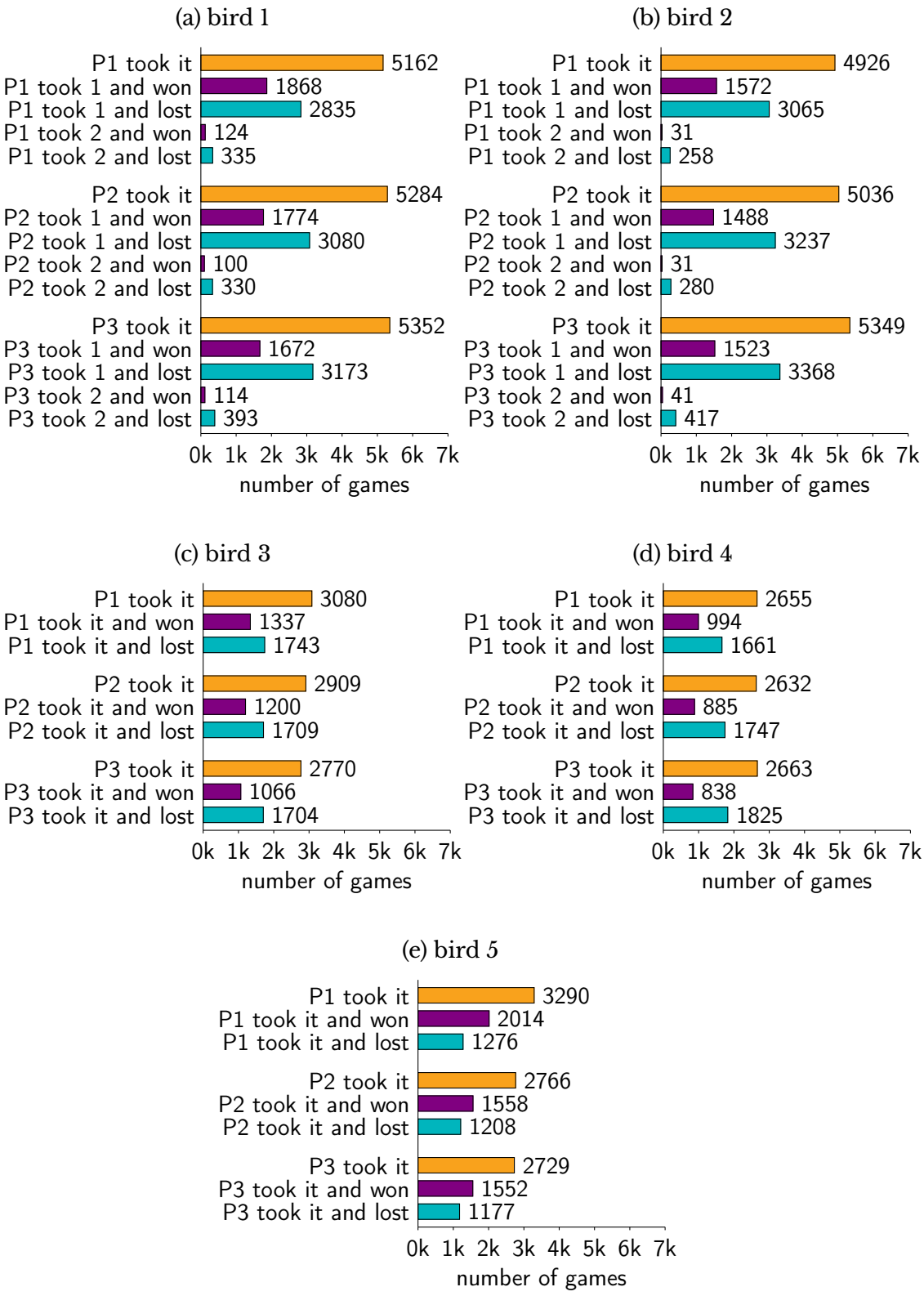


Figure B.1: Bird Lady with 3 players, bird preferences, 10000 games.

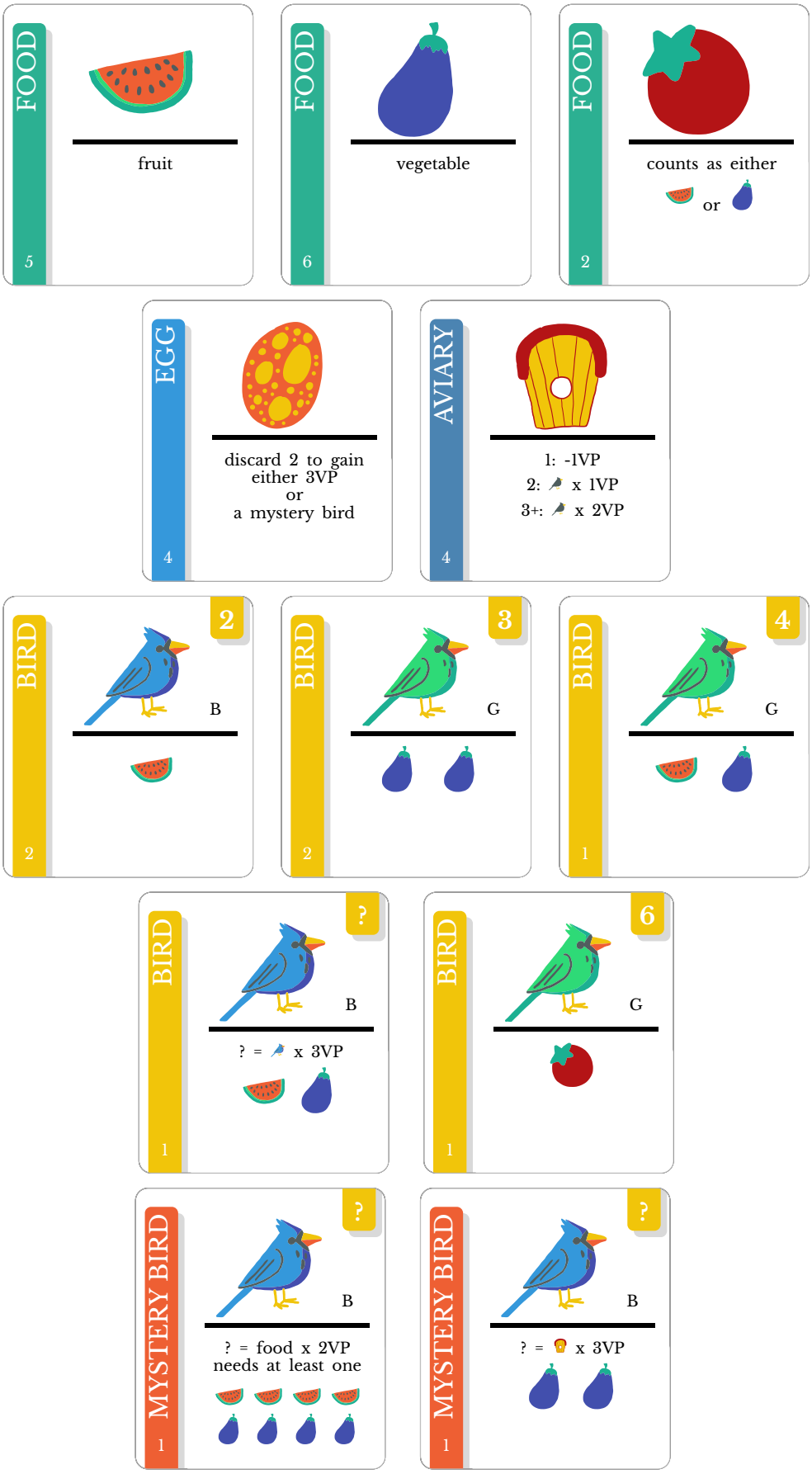


Figure B.2: Cards in Bird Lady.

COLORS

```
enum class Color { none, red, green, blue, yellow };

using PlayerId = size_t;

using ResourceId = Color;

struct ActionId {
  PlayerId player;
  Color color_to_take;
  Color color_taken_before;
};

struct Player {
  std::unordered_map<Color, size_t> colors = {
      {Color::none, 0},
      {Color::red, 0},
      {Color::green, 0},
      {Color::blue, 0},
      {Color::yellow, 0}
  };
  Color last_color_taken = Color::none;
};

class Game : public sfinks::Game<PlayerId, ActionId, ResourceId> {
  public:
  Game(size_t number_of_players);
  // functions required by sfinks::Game
  // ...
  private:
  std::vector<Player> _players;
  size_t _current_player_id = 0;
  std::unordered_map<Color, size_t> _available_colors;
};
```

Listing C.1: The class structure of Colors.

```
auto Game::available_actions() const -> std::vector<ActionId> {
  const Player &current_player = player_for_id(_current_player_id);
  const auto it = color_pairs.find(current_player.last_color_taken);
  // check if the color taken in the previous turn
  // requires a specific color to be taken this turn
  if (it != color_pairs.end()) {
    return _available_colors.at(it->second) > 0
               ? std::vector{ActionId{_current_player_id,
                                      it->second,
                                      current_player.last_color_taken}}
               : std::vector{ActionId{_current_player_id,
                                      Color::none,
                                      current_player.last_color_taken}};
  } else {
    std::vector<ActionId> actions;
    for (const auto color : {Color::red, Color::green,
                             Color::blue, Color::yellow})
      if (_available_colors.at(color) > 0 && // listing all available colors
          current_player.last_color_taken != color) // except the one taken last turn
        actions.push_back({_current_player_id,
                             color,
                             current_player.last_color_taken});

    if (actions.empty()) // if nothing is available, the player must take black
      actions.push_back({_current_player_id,
                          Color::none,
                          current_player.last_color_taken});
    return actions;
  }
}
```

Listing C.2: `colors::Game::available_actions()`

SFINKS

```cpp
template <typename PlayerId, typename ActionId, typename ResourceId>
void Engine<PlayerId, ActionId, ResourceId>::play_a_single_game() {
  _game->reset();
  for (auto &[player_id, agent] : _agents)
    agent.reset();

  while (!_game->is_over()) { // make actions until the game is over
    PlayerId current_player = _game->current_player_id();
    Agent *current_agent = &_agents.at(current_player);

    auto available_actions =
      _game->available_actions_with_results(current_agent->greedy());
    ActionId choosen_action = current_agent->choose_action(available_actions);
    _game->perform_action(choosen_action, current_player);
  }

  // determine the top score
  std::unordered_map<PlayerId, double> scores;
  for (auto &[player_id, agent] : _agents)
    scores.emplace(player_id, _game->player_score(player_id));
  double winning_score = std::max_element(scores.begin(), scores.end(),
                                          [](const auto &p1, const auto &p2) {
                                            return p1.second < p2.second;
                                          })->second;

  // reinforce agents based on their score
  constexpr double winning_reward = 1.0;
  constexpr double losing_penalty = 1.0;
  for (auto &[player_id, agent] : _agents) {
    bool agent_won = std::fabs(scores[player_id] - winning_score) <=
                     std::numeric_limits<double>::epsilon();
    agent.process_reward(agent_won ? 1.0 : -1.0);
  }
}
```

Listing S.1: Playing through a game with `sfinks::Engine`.

```hpp
#pragma once

#include <string>
#include <tuple>
#include <unordered_map>
#include <vector>

#include <sfinks/action_result.hpp>

namespace sfinks {
class ResourceData;

template <typename PlayerId, typename ActionId, typename ResourceId>
class Game {
public:
  virtual void reset() = 0;
  [[nodiscard]] virtual auto is_over() const -> bool = 0;
  [[nodiscard]] virtual auto player_ids() const -> std::vector<PlayerId> = 0;
  [[nodiscard]] virtual auto current_player_id() const -> PlayerId = 0;
  [[nodiscard]] virtual auto player_score(const PlayerId &player_id) const -> double = 0;
  [[nodiscard]] virtual auto resource_ids() const -> std::vector<ResourceId> = 0;
  [[nodiscard]] virtual auto resource_count(const ResourceId &resource_id) const
      -> size_t = 0;
  [[nodiscard]] virtual auto resource_count_for_player(const ResourceId &resource_id,
                                                       const PlayerId &player_id) const
      -> size_t = 0;
  [[nodiscard]] virtual auto available_actions() const -> std::vector<ActionId> = 0;
  virtual void perform_action(const ActionId &action_id, const PlayerId &player_id) = 0;
  virtual void undo_action(const ActionId &action_id, const PlayerId &player_id) = 0;

  // ------------------------------------------------------------------------

  using State = std::unordered_map<ResourceId, ResourceData>;
  [[nodiscard]] auto state_from_the_point_of_view_of(const PlayerId &player_id) const
      -> State;
  [[nodiscard]] auto action_result(const ActionId &action_id,
                                   const PlayerId &player_id,
                                   bool include_score = false)
      -> ActionResult<ActionId>;
  [[nodiscard]] auto available_actions_with_results(bool include_score = false)
      -> std::vector<ActionResult<ActionId>>;
};
} // namespace sfinks
```

Listing S.2: `sfinks::Game`

TIC-TAC-TOE

```cpp
#pragma once

#include <array>
#include <sfinks/game.hpp>

namespace tic_tac_toe {

enum class Mark { none, x, o };
constexpr int board_size = 9;
using Board = std::array<Mark, board_size>;

using PlayerId = Mark;
using ActionId = size_t;
using ResourceId = size_t;

class Game : public sfinks::Game<PlayerId, ActionId, ResourceId> {
public:
  Game();

  [[nodiscard]] auto is_over() const -> bool override;
  [[nodiscard]] auto player_ids() const -> std::vector<PlayerId> override;
  [[nodiscard]] auto current_player_id() const -> PlayerId override;
  [[nodiscard]] auto player_score(const PlayerId &player_id) const -> double override;
  [[nodiscard]] auto available_actions() const -> std::vector<ActionId> override;
  [[nodiscard]] auto resource_ids() const -> std::vector<ResourceId> override;
  [[nodiscard]] auto resource_count(const ResourceId &resource_id) const
      -> size_t override;
  [[nodiscard]] auto resource_count_for_player(const ResourceId &resource_id,
                                               const PlayerId &player_id) const
      -> size_t override;
  void reset() override;
  void perform_action(const ActionId &action_id, const PlayerId &player_id) override;
  void undo_action(const ActionId &action_id, const PlayerId &player_id) override;

private:
  [[nodiscard]] inline auto board_is_full() const -> bool;
  [[nodiscard]] inline auto winner() const -> Mark;
  [[nodiscard]] inline auto is_winner(Mark mark) const -> bool;
  inline void switch_player();

  Mark _current_player;
  Board _board;
};

} // namespace tic_tac_toe
```

Listing T.1: tic_tac_toe.hpp

```cpp
#include <algorithm>
#include <numeric>
#include <stdexcept>

#include <tic_tac_toe/game.hpp>

namespace tic_tac_toe {

auto Game::is_over() const -> bool { return winner() != Mark::none || board_is_full(); }

auto Game::player_ids() const -> std::vector<PlayerId> { return {Mark::x, Mark::o}; }

auto Game::current_player_id() const -> PlayerId { return _current_player; }

auto Game::player_score(const PlayerId &player_id) const -> double {
  if (is_over() && winner() == Mark::none)
    return 1;
  return is_winner(player_id) ? 2 : 0;
}

auto Game::available_actions() const -> std::vector<ActionId> {
  std::vector<ActionId> action_ids;
  for (size_t i = 0; i < _board.size(); i++)
    if (_board.at(i) == Mark::none)
      action_ids.push_back(i);
  return action_ids;
}

auto Game::resource_ids() const -> std::vector<ResourceId> {
  std::vector<ResourceId> resource_ids(_board.size());
  std::iota(resource_ids.begin(), resource_ids.end(), 0);
  return resource_ids;
}

auto Game::resource_count(const ResourceId & /*resource_id*/) const -> size_t {
  return 1;
}

auto Game::resource_count_for_player(const ResourceId &resource_id,
                                     const PlayerId &player_id) const -> size_t {
  return _board.at(resource_id) == player_id ? 1 : 0;
}

void Game::reset() {
  _board.fill(Mark::none);
  _current_player = Mark::x;
}

```

Listing T.2: tic_tac_toe.cpp 1/2

```
49 void Game::perform_action(const ActionId &action_id, const PlayerId &player_id) {
50   if (_board.at(action_id) != Mark::none || is_over())
51     std::throw_with_nested(std::invalid_argument("action unavailable"));
52   _board.at(action_id) = player_id;
53   switch_player();
54 }
55
56 void Game::undo_action(const ActionId &action_id, const PlayerId &player_id) {
57   if (_board.at(action_id) != player_id)
58     std::throw_with_nested(std::invalid_argument("invalid undo"));
59   _board.at(action_id) = Mark::none;
60   switch_player();
61 }
62
63 Game::Game() : _current_player(Mark::x), _board({Mark::none}) {}
64
65 auto Game::board_is_full() const -> bool {
66   return std::all_of(_board.begin(), _board.end(),
67     [](const auto x) { return x != Mark::none; });
68 }
69
70 auto Game::winner() const -> Mark {
71   if (is_winner(Mark::o))
72     return Mark::o;
73   if (is_winner(Mark::x))
74     return Mark::x;
75   return Mark::none;
76 }
77
78 auto Game::is_winner(Mark mark) const -> bool {
79   const auto [b1, b2, b3, b4, b5, b6, b7, b8, b9] = _board;
80   return ((b1 == mark && b2 == mark && b3 == mark) || // row 1
81           (b4 == mark && b5 == mark && b6 == mark) || // row 2
82           (b7 == mark && b8 == mark && b9 == mark) || // row 3
83           (b1 == mark && b4 == mark && b7 == mark) || // column 1
84           (b2 == mark && b5 == mark && b8 == mark) || // column 2
85           (b3 == mark && b6 == mark && b9 == mark) || // column 3
86           (b1 == mark && b5 == mark && b9 == mark) || // diagonal 1
87           (b3 == mark && b5 == mark && b7 == mark));  // diagonal 2
88 }
89
90 void Game::switch_player() {
91   _current_player = _current_player == Mark::x ? Mark::o : Mark::x;
92 }
93
94 } // namespace tic_tac_toe
```

Listing T.3: tic_tac_toe.cpp 2/2

BIBLIOGRAPHY

[1] J. Stegmaier. (2019). "Top 10 reasons for the rise in popularity of tabletop games," [Online]. Available: https://web.archive.org/web/20201208220441/https://stonemaiergames.com/top-10-reasons-for-the-rise-in-popularity-of-tabletop-games/.

[2] C. Hall. (2019). "Tabletop games dominated kickstarter in 2018, while video games declined," [Online]. Available: https://web.archive.org/web/20201208215904/https://www.polygon.com/2019/1/15/18184108/kickstarter-2018-stats-tabletop-video-games.

[3] F. D. M. Silva, S. Lee, J. Togelius, and A. Nealen, "Ai as evaluator: Search driven playtesting of modern board games," 2017. [Online]. Available: https://aaai.org/ocs/index.php/WS/AAAIW17/paper/view/15171.

[4] F. de Mesentier Silva, S. Lee, J. Togelius, and A. Nealen, "AI-based playtesting of contemporary board games," en, in *Proceedings of the International Conference on the Foundations of Digital Games - FDG '17*, Hyannis, Massachusetts: ACM Press, 2017, pp. 1–10, isbn: 9781450353199. doi: 10.1145/3102071.3102105. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3102071.3102105 (visited on 08/24/2020).

[5] F. d. M. Silva, I. Borovikov, J. Kolen, N. Aghdaie, and K. Zaman, "Exploring Gameplay With AI Agents," *arXiv:1811.06962 [cs]*, Nov. 2018, arXiv: 1811.06962. [Online]. Available: http://arxiv.org/abs/1811.06962 (visited on 08/24/2020).

[6] L. Mugrai, F. Silva, C. Holmgard, and J. Togelius, "Automated playtesting of matching tile games," English (US), in *IEEE Conference on Games 2019, CoG 2019*, IEEE Computer Society, Aug. 2019, p. 8 848 057. doi: 10.1109/CIG.2019.8848057. [Online]. Available: https://nyuscholars.nyu.edu/en/publications/automated-playtesting-of-matching-tile-games (visited on 08/24/2020).

[7] *Game*, in *Merriam-Webster dictionary*. [Online]. Available: https://www.merriam-webster.com/dictionary/game (visited on 12/10/2020).

[8] P. A. Piccione, "In search of the meaning of senet," *Archaeology*, vol. 33, 1980, issn: 0003-8113. [Online]. Available: http://web.archive.org/web/20200727005445/http://www.piccionep.people.cofc.edu/piccione_senet.pdf.

[9] P. Shotwell. (2008). "The game of go: Speculations on its origins and symbolism in ancient china," [Online]. Available: https://web.archive.org/web/20190804010219/https://www.usgo.org/sites/default/files/bh_library/originsofgo.pdf.

[10] G. J. Schloesser, "Future of avalon hill," *The Games Journal*, Jul. 2000. [Online]. Available: https://web.archive.org/web/20201215194920/http://www.thegamesjournal.com/articles/FutureofAH.shtml.

[11] S. Woods, *Eurogames: The Design, Culture and Play of Modern European Board Games*, English, First. McFarland & Company, Inc. Publishers, 2012, isbn: 9780786467976.

[12] J. Juul, *Half-Real: Video Games between Real Rules and Fictional Worlds*. The MIT Press, 2005, isbn: 0262101106.

[13] T. M. Mitchell, *Machine Learning*. New York: McGraw-Hill, 1997, isbn: 978-0-07-042807-2.

[14] E. Brunskill. (2019). "Stanford cs234: Reinforcement learning | winter 2019," Stanford University, [Online]. Available: https://www.youtube.com/playlist?list=PLoROMvodv4rOSOPzutgyCTapiGlY2Nd8u.

[15] A. Singh, K. Deep, and A. Nagar, "A "never-loose" strategy to play the game of tic-tac-toe," *Proceedings - 2014 International Conference on Soft Computing and Machine Intelligence, ISCMI 2014*, pp. 1–5, Apr. 2015. doi: 10.1109/ISCMI.2014.13.

[16] S. Schaefer. (2002). "How many games of tic-tac-toe are there?" [Online]. Available: https://web.archive.org/web/20200224170433/http://www.mathrec.org:80/old/2002jan/solutions.html.

[17] J. Wood, *Cat lady rulebook*, 2017. [Online]. Available: https://web.archive.org/web/20201208220218/https://www.alderac.com/wp-content/uploads/2017/07/cl_rulebook-v2-6.pdf.

[18] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.