Exploring Different Deck Building Algorithms in Slay the Spire

Justas Zelnia

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2023

Abstract

Slay the Spire is a digital roguelike card game that involves building a deck of cards while progressing through different levels. In this project, we try to replicate some of the strategies that professional players use for drafting cards. The first algorithm we propose will try to pick cards in a way that creates synergies which are often present in good decks. The second algorithm will try to dynamically evaluate card rewards based on their predicted performance in the future. The effectiveness of the algorithms will be evaluated, and possible applications to other games and decision-making scenarios will be discussed.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Justas Zelnia*)

Acknowledgements

I would like to thank my supervisor David Symons for giving me the opportunity to work on this project as well as his guidance, valuable suggestions and feedback.

Table of Contents

Chapter 1

Introduction

Slay the Spire [20] is a popular roguelike deck-building video game developed by American studio MegaCrit. Roguelike is a subgenre of video games, which typically involve characteristics such as producedurally generated levels, turn-based gameplay, grid-based movement, and permanent death of the player character. This means that every playthrough of a roguelike is independent. However the player can unlock items, characters, additional levels, which become available in future playthroughs.

1.0.1 Motivation

Building a good deck is one of the most important parts in playing Slay the Spire. It is a difficult challenge for a human player and an even greater one for a computer. In a game like chess, it is possible to evaluate a set of neighboring states to decide which move to make, however in a highly complex game like Slay the Spire, this would not even scratch the surface of possibilities. Nevertheless, some human players are able to play the game at an incredible proficiency, reaching high win rates on even the most difficult settings. Successfully mimicking the decision processs of a human player in Slay the Spire would give us a framework for applying similar methods to other card games and decision problems.

1.0.2 Goals

We will only be focusing on the first class - the Ironclad. The goal of the project is to develop intelligent strategies for one of many parts of the game - picking the best cards to add to a deck. Consequently, we may not arrive at a solution which is able to consistently win the game, however, we wish to be able to find similarities between the algorithm and human reasoning. We would like to be able to compare different approaches in terms of strengths and weaknesses. Finally, we wish to obtain an agent which works not only on paper, but one that can be replicated and tested on the real game.

The main hypotheses which we are putting to the test are:

- It is important to consider synergies between cards in order to build a good deck.

- Short-term decisions are more important than long-term decisions on a high difficulty setting.
- By making good short-term decisions, we will be able to construct a deck which scales well into the more difficult fights of the game.
- We can focus solely on preparing for the most difficult enemies in the game, while ignoring the easier ones.

1.0.3 Contributions

We will propose two algorithms, based on a bottom-up and top-down approach respectively.

The bottom-up approach will give us an algorithm which is able to build decks with strong synergies and win conditions. This proof of concept can be extended to consider more uncommon types of synergies, as well as the interaction between the deck and other parts of the game, such as relics.

The top-down approach will employ an original framework for evaluating cards. Some of the traits of the approach are shared by algorithms in Hearthstone research, however, there are no papers discussing the use of the specific algorithm that we propose. The card drafting agent will show promise on the highest difficulty setting in the game by consistently being able to survive act 1 and will be able to occasionally win games on easier difficulties. The structure of the proposed algorithm is quite general, which means that it could be applied for other decisions in the game, such as card upgrades and removals. Similarly, it is possible to apply the same idea to create a deck building policy for other strategy card games like Monster Train [21] and Griftlands [19].

1.0.4 Structure of the Report

In the remainder of the introduction, we will introduce the reader to the rules and most important aspects of the game, which we will be referring to throughout the report. In the background chapter we will explore previous research on Slay the Spire and similar games. We will examine guides for the game, which will give us an insight into how good players make decisions. Then we will look at the the tools and mods that have been developed for the game, many of which will be utilized in our algorithms. Chapter 3 will provide an in-depth explanation the bottom-up and top down algorithms, the reasoning behind them, and the various decisions made throughout their development. Furthermore, we will outline the technical setup that was built in order to test them. In Chapter 4, we will evaluate the performance of the algorithms on different difficulty settings and analyze the patterns displayed. Finally, in Chapter 5 we will reflect on the algorithms' performance, trade-offs, potential applications and future work.

1.1 The Game

1.1.1 Overview

In Slay the Spire, each playthrough or ”run” begins with the player being given a base deck of 10-12 cards. The goal is to complete 3 acts, in each act navigating through 16 floors of a procedurally generated map, while modifying the deck by adding new cards and removing existing ones. There also exists an optional 4th act, which acts as an extra challenge, but it requires specific conditions to enter and is not necessary to beat the game. Throughout each act, the player needs to defeat various monsters while keeping their health above 0.

1.1.2 The Map

As the player navigates through the map (Figure 1.1), they may enter different types of rooms, which can can be categorized into a few types:

- Monster encounter - the player needs to use their existing deck to defeat some monsters. Winning the encounter awards the player with some gold, a selection of up to 1 of 3 random cards and sometimes a potion. Losing the encounter ends the run. Monsters are chosen from a pool of possible ones, based on the current act and the number of encounters completed in the act so far.
- Elite Monster encounter - the player needs to defeat a stronger monster, chosen randomly from 3 elite monsters specific to the current act. Winning the encounter awards the player with some gold, a relic, which provides a passive bonus for the rest of the run, and a selection of up to 1 of 3 random cards. Losing the encounter ends the run.
- Rest Site - allows the player to either heal for 30% health or permanently upgrade a card of their choice.
- Treasure Room - located at the halfway point of each act, rewards the player with 1 random relic.
- Merchant - allows the player to spend gold to buy some of the available cards, potions, relics, and/or remove a card from their deck.
- Event - randomly chosen from a pool of events based on the current act, they usually require the player to choose from a few options, which give the player some bonus, penalty, or both.
- Boss encounter - located at the penultimate floor of each act. The player needs to defeat an incredibly difficult monster chosen randomly from 3 bosses specific to the current act. Winning the encounter rewards the player with some gold, a choice between 3 rare cards, a choice between 3 boss relics and sometimes a potion. The floor before a boss encounter is always a rest site.

Some of the locations on the map are unknown. When a player visits one of these rooms, they may encounter an Event, a Monster, a Shop or a Treasure Room.

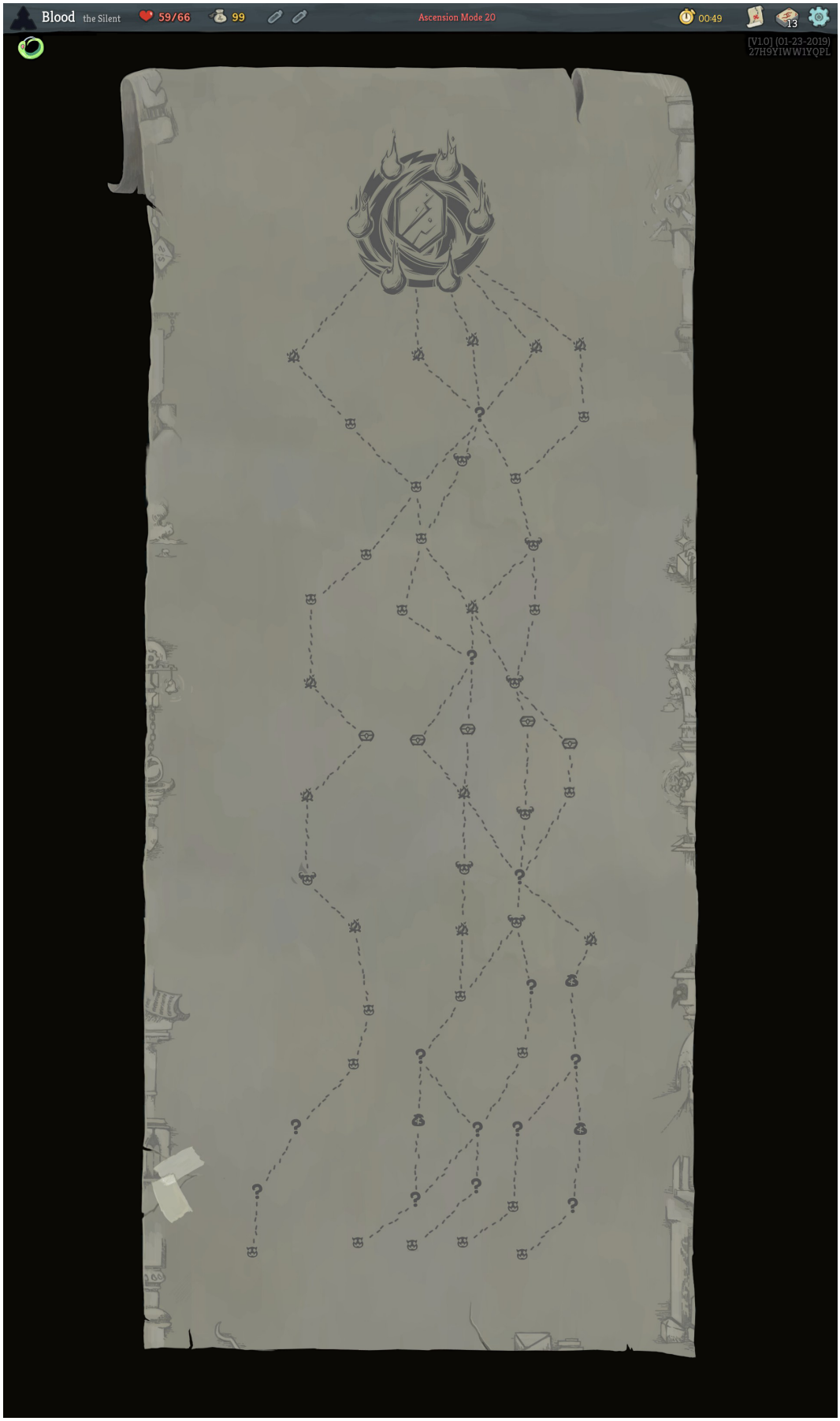


Figure 1.1: Procedurally generated map for act 1. The player chooses a starting room from one of the 5 rooms at the bottom and works their way up. Various types of rooms are marked using different symbols.

At the start of act 1, before proceeding to the 1st floor, the player must choose a blessing from 2 - 4 possible choices as part of a special event.

1.1.3 Battles Against Enemies

Figure 1.2: Monster Encounter

In encounters, the player uses their current deck to face off against a set of enemies, trying to defeat them, while minimizing damage taken. Each turn, the player gains a set amount of energy (3) and draws a hand of cards from their deck. At the end of the turn, any cards remaining in the player's hand are placed on the discard pile. When the deck becomes empty and the player needs to draw cards, the discard pile is shuffled and becomes the new draw pile.

It is worth noting that there exists another pile of cards called the exhaust pile, containing cards that are no longer usable for the duration of the encounter. Some cards are immediately exhausted after playing, others can be exhausted through other card or relic effects. Ethereal cards are exhausted if they remain in the player's hand at the end of their turn. Cards can sometimes be retrieved from the exhaust pile through the use of specific cards.

During their turn, the player can play cards while they have some in their hand and have enough energy. The player can always see what actions the enemies intend to take on their turn, in order to prepare accordingly. After the player's turn, the enemies perform actions according to their intents. Intents can be actions such as attacking, creating block, casting positive status effects on themselves or negative status effects on the player. The player and the enemies alternate turns until either one of the sides is defeated.

1.1.4 Cards

Note: In this section and for the rest of the report, the word buff refers to some kind of positive effect, while the word debuff refers to a negative effect.

There are 5 main types of cards:

- Attack - deals damage to an enemy and may have a secondary effect.
- Skill - provides an effect, such as block, utility, buff or debuff. Unlike an attack, it cannot deal direct damage.
- Power - a permanent upgrade that lasts for the entire encounter.
- Status - these cards can only be added to the deck during an encounter and are removed from the deck at the end. They are designed to bloat the deck and prevent the player from drawing beneficial cards, with some of them having additional negative effects.
- Curse - an unplayable card that is added through an event. Most curses have negative effects and are designed to bloat the deck, similarly to status cards. They remain in the deck permanently until removed.

Cards also have different rarities - common, uncommon and rare - which have different probabilities of appearing in card rewards. The player starts each run with a set of basic (starter) cards, which cannot be obtained through card rewards.

In this report we will often be referring to cards by names. A list of Ironclad cards, along with their descriptions is provided in Appendix A, if necessary.

1.1.5 Relics

Relics can be obtained from Elite and Boss combat rewards, Shops, Treasure Chests, Events. They usually provide a passive bonus for the rest of the run, such as giving the player one extra energy in combat, increasing the damage of the first attack each combat, making shops cheaper, etc. Some of the relics come with a negative effect, which forces the player to take risks and adapt their playstyle.

1.1.6 Status Effects

There are many different status effects in Slay the Spire, which can either be applied on/by both the enemies and the player. It is useful to know the main types of status effects.

- Dexterity - can be either positive or negative. Increases/decreases block gained from cards by X.
- Strength - can be either positive or negative. Increases/decreases attack damage by X.
- Vulnerable - target takes 50% more damage from attacks. Lasts for X turns.
- Weak - target deals 25% less damage. Lasts for X turns.

Figure 1.3: Card Reward

- Frail - can only be applied by enemies. Block gained from cards is reduced by 25%. Lasts for X turns.

1.1.7 Potions

Potions are single-use items, that can only be used in combat, with a few exceptions. Once consumed, a potion grants an immediate effect, such as attack, block, dexterity, random card.

1.1.8 Difficulty

Ascension mode provides extra difficulty to the game by adding 20 different Ascension levels. Each Ascension level introduces a modifier on top of the previous ones, which makes certain aspects of the game more difficult. We will refer to the default game difficulty setting as Ascension 0 throughout the report and Ascension levels by their numbers. The modifiers for each Ascension level can be found in Appendix B.

Chapter 2

Background

2.1 Existing Research

There is little existing research relating particularly to Slay the Spire. A paper by Mikolaj Trojanowski and Johan Andresson [24] analyses the effects of randomness in the game and how much player skill impacts success at the game. In order to evaluate a deck, the authors calculated a deck score, based on a card tier list created by the community. They found that the final deck score in winning runs was twice that of the non-winning groups. The analysis also showed that even though winning runs had a higher card count than losing ones, these runs had twice less cards per floor reached. This means that winners did not prefer having very large decks, because that would give them less control over the gameplay and which cards they were going to draw.

There has been some more research around deck building in collectible card games (CCGs), such as Hearthstone [2] and Magic the Gathering [25]. An interesting approach is presented in [15]. The authors suggest an algorithm that starts with a set of random decks. At each step, the worst decks are discarded and the best ones are mutated to create a new set of decks. In order to evaluate decks, the authors used an automatic Hearthstone battle AI, which is similar in spirit to the tool we will use to run battle simulations for our top-down algorithm.

In a paper about a learning deck-building algorithm for Hearthstone [7], the authors explain that deck building algorithms can be divided into two categories - heuristics and metaheuristic searches. They claim that the former requires intensive human knowledge, whereas for the latter only Genetic Algorithms (GAs) are used. The algorithm described in the paper similarly requires to be trained beforehand, but is then able to make fast deck-building decisions.

Another interesting approach that has been taken in order to analyse deck types in CCGs is learning contextual similarity between different cards [17]. This allows the construction of decks that are alike in quality by replacing some cards in one deck with other cards having similar benefits.

Comparably, in the paper ”Hearthstone deck-construction with a utility system” Stiegler et al. [23] propose an algorithm which is able to complete decks with suitable cards

using heuristics such as Cost Effectiveness, Synergies, Mana Curve and Strategic Deck Parameters. Some of these heuristics are specific to the game of Hearthstone. According to the results, the algorithm was able to find sufficient replacements for cards more than 97% of the time.

Although there are some similarities, it is important to understand some fundamental differences between Slay the Spire and CCGs. CCGs are games between two players, who use their constructed decks, while in Slay the Spre the player has to battle enemies, who use relatively predictable sets of moves, instead of card decks. Moreover, in CCGs, the deck is finalized before the game even starts, while in Slay the Spire it evolves throughout the run as the challenges evolve simultaneously.

2.2 How Good Players Approach Deck Building

There exist other types of the game's analysis, other than research. For instance, an article by Ethan Gach [9], among other things, provides suggestions for some deckbuilding related aspects of the game. Firstly, it suggests adding some power cards to the deck early on and picking cards that synergize well with the powers as early as possible. Additionally, it suggests that based on the options in act 1, the player must decide to go for one of two deckbuilding approaches. The first one being trying to create a deck with a powerful core of around 15 cards. It is mentioned that in this approach, it is important to remove weak cards, such as the basic cards that the Ironclad starts with (Strike, Defend in particular). The second approach is to create a thick deck which has less powerful cards on average but is easier/cheaper to achieve and provides some more flexibility. The article also mentions an important fact – it is sometimes best to skip a card reward. Forcibly adding weak cards to the deck can decrease its overall power and dilute the synergy between existing cards. However, it is often not a good idea to skip card rewards early into act 1, because they will most likely be better than the starter cards. Finally, the article suggests that the Ironclad cards Armaments, Battle Trance and Inflame should be picked up almost any time they are offered.

There is also a highly rated guide [13] by over 950 users on the steam game page by the user Forgotten Arbiter. It goes into detail about the strategy that the author uses to to get series of consecutive wins (win streaks) with the Ironclad class. The main deck building aspects covered by the guide are:

- Decks should be defense-oriented, only containing a minimum amount of offensive cards required to win fights.
- Players should keep the deck small, only taking as many cards as they can upgrade. In order to achieve this, card removals should be utilised, removing Strike cards first, followed by Defends.
- Upgrading Whirlwind, True Grit and Body Slam first, generally followed by powers, utility skills, defensive skills and finally attacks.
- A spreadsheet [12] is included with the guide, listing the priority of different cards in each act. The spreadsheet outlines 3 cards as win conditions, around which a deck must be focused around. These cards are Limit Break, Barricade,

and Demon Form. Importantly, the spreadsheet mentions that these cards should only be picked in act 1 if the player can afford to have a dead card in their deck, as they don't provide much value in the beginning of the game, however these cards become the top priority in the second and third acts.

It is not mentioned explicitly, which ascension level this guide targets. It is quite unlikely (but not impossible), that the win streaks were obtained on ascension 20, as a win streak of 16 runs is mentioned, which is extremely difficult to achieve even for the most skilled Slay the Spire players.

Concerning Ascension 20 in particular, an article by Harry Alston [16] highlights the various aspects that are crucial to master in order to win on the most difficult setting. The author recommends to "build a deck that deals with your short-term goals and scale it later". These short term goals are most often either an elite or boss battle in the current act, which can end the run if the player is not prepared. However, it is important to try to find opportunities to add cards that provide scaling, which means improving the ability to generate block, gain attack, draw cards or similar.

2.3 Data-Based Ironclad Card Tier List

An online tier list [1] has been made for Ironclad (and other character) cards, based on data collected from thousands of Ascension 17+ runs. It provides a score for each card based on the act it is offered in. The scores for cards are calculated based on their win rate. What is noticeable, is that all cards have relatively similar scores for act 1, but the score disparity is much more clear for act 3.

2.4 Mods

Most of the existing tools for Slay the Spire are in the form of mods. In general, the term *mod*, which is short for *modification*, refers to user created in-game content which may change or improve various aspects of the game. Mods for Slay the Spire can be loaded using the ModTheSpire [18] mod loader. Steam - one of the most popular video game digital distribution services allows users to install ModTheSpire and other mods easily from the game's Steam workshop page. However, it is also easy to build mods into JAR files manually from source code, which is particularly useful for making custom changes to the mods. Additionally, ModTheSpire provides the @SpirePatch annotation, which can be used to make changes to methods inside the mod's or even the game's source code.

For the purpose of building a Slay the Spire AI, the following mods are useful:

- **BaseMod** [8] provides an extra layer on top of ModTheSpire's low-level API to make the creation of mods easier. Most of the other mods require this mod as a dependency.
- **CommunicationMod** [10] provides a protocol for allowing another process to control the game. The command for launching the process must be specified in

a configuration file. The process sends commands to the game using standard output and receives a JSON representation of the current game state whenever it is determined to be stable (no longer changing without external input). The author of this mod has also created a simple AI in python called spirecomm [11]. Spirecomm works by listening for game state notifications from CommunicationMod. After each received message it calculates and sends the action to be played back to the game. It makes decisions based on simple heuristics and can rarely win Ascension 0 runs.

- **SuperFastMode** [22] adds the possibility to speed up the game by up to 1000% by accelerating bottleneck animations and actions.
- **LudicrousSpeed** [4] is a Slay the Spire mod that allows a calling library to execute execute commands very quickly for use in AI projects. It is crucial for time-efficiency reasons.
- **STSStateSaver** [6] allows the creation and loading of in-battle save states. It extends the base game saving functionality, which only allows to save and resume the game from the start of a battle.
- **scumthespire** [5] is a battle AI mod, which uses a search-based algorithm to simulate possible sequences of moves and find one that ends the combat with the most remaining health. The mod utilises a server, which also runs a separate instance of the game and must be launched by the user either via command line or UI. The client (game) sends a request to the server to process a battle. The server then calculates a good sequence of moves and sends the sequence of commands back to the client. The commands are then performed in the game. The mod makes use of the State Saver mod to revert to previous states during the search process, while the Ludicrous Speed mod enhances the processing speed. Battles take up to a few seconds to process. Notably, the sequence of moves generated by the algorithm is not always optimal, especially in later parts of the game, due to the very large search space.
- **CommunicationModExtension** [3] extends the list of commands supported by the CommunicationMod, by adding savestate and loadstate commands from STSStateSaver mod.

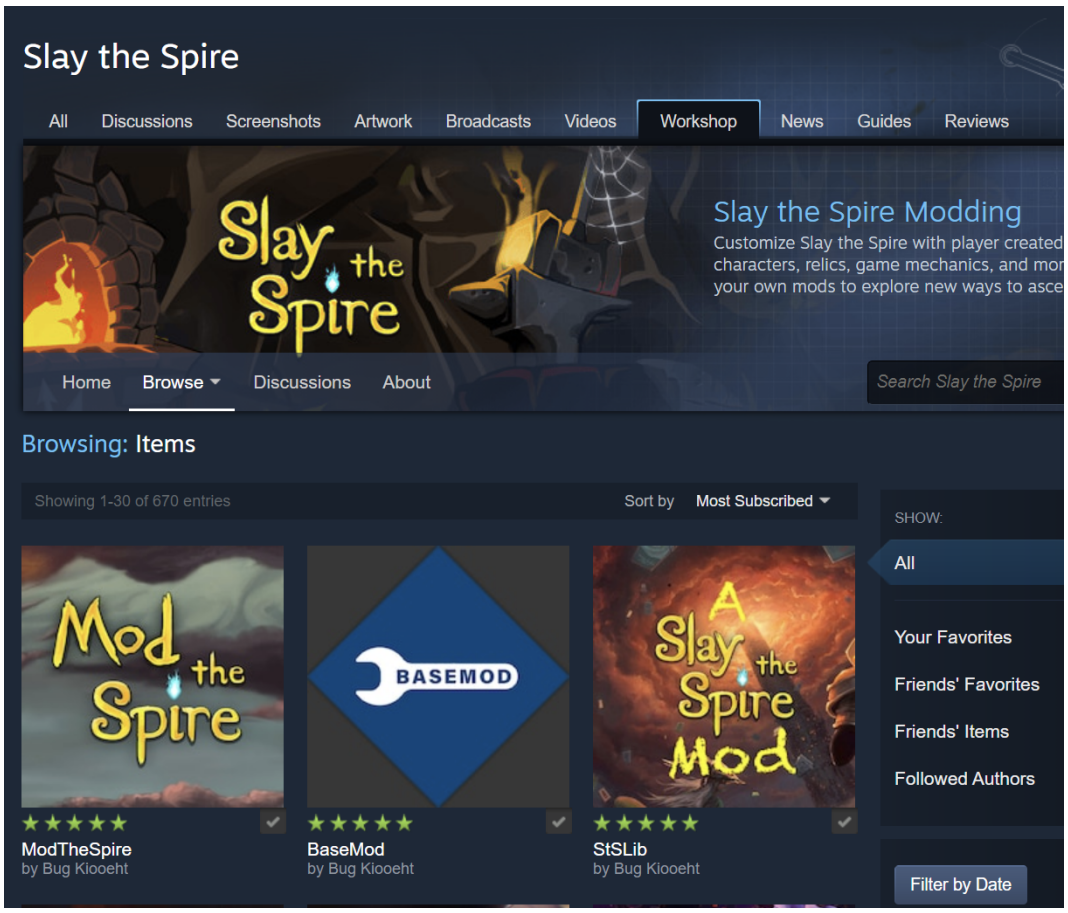


Figure 2.1: Steam workshop page for Slay the Spire, from where users can install mods for the game

Figure 2.2: The in-game mod menu, where mods can be configured and the external process can be launched to interact with the communication mod.

2.5 Decompiling the Game

In order to develop mods or patches for the game, it is important to understand how the different classes in the game's source code work. The source code can be obtained by decompiling the game's JAR using a Java decompiler such as JD-GUI.

2.6 Headless version of the game

Alternatively, there exists a headless version of Slay the Spire, called sts_lightspeed [14] written in C++ 17, which can be compiled and played in the command line. The project supports everything necessary for playing the Ironclad class. According to the description, it 100% accurately mimics the randomization of the real game and it can perform 1 million playouts in 5 seconds using 16 threads. Additionally, it contains an implementation of a Monte Carlo tree search algorithm for the battles. There is work in progress for integrating this project with CommunicationMod, however it is not currently usable.

Chapter 3

Methods

We consider two main approaches to constructing a deck, which differ in the way we evaluate each card reward. In a *bottom-up* approach we consider how useful a card would be, given the current state of the character. A *top-down* approach considers possible future states of the deck and evaluates card choices based on how likely they are to lead to one of the strong decks.

Why do we explore these approaches in particular? We could instead use machine learning to train a model on data collected from runs of human players. However, one of our goals was to create an algorithm which draws inspiration from the human decision process. Moreover, there is no one correct way of playing the game. Different players use different strategies and favour certain cards, which means that it would be difficult for a model to converge to a consistent strategy.

3.1 Bottom-up approach

The primary challenge in developing an effective deck building policy via a bottom-up approach is the need to address a multitude of diverse factors. These could be, but are not limited to, the current deck state, relics, player health, map options, gold, act boss. In this approach, we will focus only on the current deck and the current act, because these are likely the most important things to consider. Additionally, it will allow us to more easily see the logic behind decisions and keep the algorithm from being overloaded with various conditions/heuristics.

We build on top of the existing Spirelogs Ironclad card tier list. Note that the Spirelogs tier list essentially consists of three different tier lists, for each of the first 3 acts.

Why is it not a good idea to just pick cards based on the tier list? The reason is that a card's value can greatly differ depending on what other cards are present in a deck. Consider the following scenario:

We are on act 2, our deck consists of the starter cards, as well as Evolve, 2 copies of Power Through, 2 copies of Wild Strike and 2 copies of Reckless Charge.

Now we are offered a choice of either adding Reaper or Fire Breathing to our deck.

Card	Act 1	Act 2	Act 3	Overall Score	Upgrade Bonus
Corruption	81	149	130	366	+158
Entrench	50	109	316	355	+72
Reaper	102	84	130	336	+33
Exhume	82	98	171	320	+74
Havoc	52	91	238	301	+15
Sentinel	59	70	262	295	+80
Burning Pact	29	142	154	287	+59
Seeing Red	70	95	152	276	+61
Bludgeon	101	65	91	263	+88
Feel No Pain	74	79	99	257	+121
Uppercut	69	103	75	257	+92
Shockwave	104	73	15	257	+121

Figure 3.1: The in-game mod menu, where mods can be configured and the external process can be launched to interact with the communication mod.

If we just look at the Spirelogs tier list, we may think that Reaper (score 84) is clearly a better card than Fire Breathing (score 29). However, considering that the cards in our deck generate a lot of status cards, Fire Breathing would be an extremely valuable addition. On the other hand, Reaper is a very weak card on its own, and we do not have ways to generate strength, which would make the card more powerful.

3.1.1 Synergies

This motivates an approach that tries to add cards in a manner that creates synergies in our deck. However, this is not a very easy task. Cards have various different effects, which makes for a lot of cases that we would need to consider. In order to simplify the problem, we will define 4 groups of synergistic cards (synergy classes) and try to encourage our agent to achieve these synergies. Experienced Ironclad players will likely recognize these synergy classes, as they commonly make up the core of winning Ironclad decks. The synergy classes we define are:

1. **High Strength Synergy** This class is made up of cards that either produce strength, or greatly benefit from it.

   Strength generating cards include Demon Form, Flex, Inflame, Spot Weakness and Limit Break.

   Cards that uniquely benefit from strength include Heavy Blade, which becomes a reliable high-damage attack, and Reaper, which provides damage and sustain by healing the Ironclad based on damage dealt.

   Other cards that synergize well are multi-hit attacks like Sword Boomerang, Twin Strike, Pummel and Whirlwind.

2. **High Block Synergy** This class of cards utilizes the Barricade power to retain block between turns.

   Cards that generate a lot of block, like Impervious and Power Through synergize very well with Barricade and combined with Entrench, the Ironclad can build up enough block to negate all enemy attacks.

Body Slam allows the player to utilize block to cheaply deal high damage, whereas Juggernaut provides a way to deal consistent damage to enemies.

3. **Exhaust Synergy** Corruption is a centerpiece of this synergy class, because it is a very powerful card, which allows the Ironclad to cast skills for free, at the expense of them being exhausted.

   Additionally, cards that add benefits to exhausting, such as Dark Embrace, Feel No Pain and Sentinel make cards with this effect more useful.

   Exhume also works well here, since it allows to pick and play an exhausted card again.

4. **Status Synergy** Cards like Wild Strike, Power Through, Reckless charge and Immolate provide a stronger effect than other cards of the same cost, at the expense of filling up the deck with unusable status cards.

   The Evolve and Fire Breathing powers mitigate the negative effects of these cards by improving card draw and damage.

   Second Wind and Fiend Fire allow the player to remove unusable status cards for lots of block/attack.

The full list of cards for each Synergy class can be seen in the table.

Synergy Classes			
High Strength	High Block	Exhaust	Status
Flex	Barricade	Corruption	Evolve
Demon Form	Body Slam	Dark Embrace	Wild Strike
Limit Break	Entrench	Feel No Pain	Fire Breathing
Spot Weakness	Impervious	Sentinel	Power Through
Inflame	Power Through	Exhume	Second Wind
Pummel	Juggernaut		Fiend Fire
Twin Strike			Immolate
Whirlwind			Reckless Charge
Sword Boomerang			
Reaper			
Heavy Blade			

Obtaining enough cards from either of these synergy classes can be enough to achieve a winning Ironclad deck.

3.1.2 Algorithm

When considering a card reward, we will use the following evaluation scheme for new cards to encourage the formation of synergies:

1. Suppose the card we are evaluating is C.
2. The score for C is initialized to a base value equal to its Spirelogs score for the current act, lets call it $BASE(C)$.

3. We define a synergy multiplier $MULT$, which is a real value in the range $(1, \infty)$. We will use $MULT = 1.5$. Picking a low value of $MULT$ would not make our algorithm very different from one that simply follows the Spirelogs tier list, due to the high score variance within the list. On the other hand, selecting a value of $MULT$ greater than 2 would be impractical since it would imply that a card's quality is more than twice as good due to a single synergy.
4. The score is multiplied by $MULT^k$, where k is the sum of the number of cards in each of the synergy classes that C is in.
5. Hence the adjusted score for C is $BASE(C) \times MULT^k$.

As a toy example, lets assume that we are on Act 2 and our deck currently contains only the cards Limit Break, Reaper, Pummel and Wild Strike. The card reward consists of Demon Form, Power Through and Disarm.

1. We initialize the base scores from Spirelogs scores. $BASE(Demon\ Form) = 24$, $BASE(Power\ Through) = 43$, $BASE(Disarm) = 60$.
2. Count the number of existing synergies for each card. $k(Demon\ Form) = 3$, because Demon Form belongs to the High Strength synergy class, just like Limit Break, Reaper and Pummel. $k(Power\ Through) = 1$, because Power Through shares the Status Synergy class with Wild Strike. $k(Disarm) = 0$, because it doesn't synergize with anything in the deck.
3. We calculate the adjusted scores:

   $Score(Demon\ Form) = 24 \times 1.5^3 = 81.$

   $Score(Power\ Through) = 43 \times 1.5^1 = 64.5.$

   $Score(Disarm) = 60 \times 1.5^0 = 60.$
4. We pick Demon Form, as it has the highest adjusted score $(81 > 64.5 > 60)$.

As we can see, our evaluation of a card scales exponentially with the number of synergies that are already in the deck. Logically, this makes sense. Having multiple cards that fit well together usually allows for more complex interactions. Moreover, it becomes more probable that we will be able to draw and play cards in a way that exploits the benefits of their synergy. Since the game has random elements, unlucky card draws may be the difference between a win and a loss.

3.1.3 Early Game Insurance

There is one more important thing that we need to do. While it is important to consider synergies, we also need to survive the first act. In order to accomplish this, we need to give some priority to cards that can increase our blocking capabilities and damage output in the short term, even if they are not very valuable in the long term. So we will create an extra 5^{th} class of **Early Priority** cards.

Early Priority
Pummel Strike
Rampage
Bludgeon
Hemokinesis
Shrug It Off
Cleave
Wild Strike
Carnage

As we can see, these cards are an improvement over the starter cards, but will likely not provide much scaling to our deck.

How should we adjust our evaluation formula? If a card belongs to early priority cards, we will simply adjust its evaluation using an extra multiplier EP, which will be a value in the range $(1,\infty)$. We will use $EP = 1.75$. The reasoning for this is again related to the high variance in Spirelogs tier list. We want to very strongly encourage early priority cards to be picked in act 1, unless we are offered cards which are vastly superior. This approach is by no means perfect, but it will hopefully increase our algorithm's survivability in the first act.

Hence our final formula for evaluating cards is:

$$Score(C) = \begin{cases} EP \times BASE(C) \times MULT^k, & \text{if } C \text{ is an Early Priority card} \\ BASE(C) \times MULT^k, & \text{otherwise} \end{cases}$$

3.2 Top-down approach

If we consider how a skilled player plays the game, a bottom-up approach only accounts for a part of the decisions that go into making a card choice, mainly the current state of the character. We propose an innovative idea for a top-down algorithm, which will be able to implicitly account for the considerations that a human player would take. Neither this idea, nor similar ideas have been explored yet in Slay the Spire research, mods, or tools.

If we think very abstractly at what we need to do to win a Slay the Spire run, it simply boils down to winning all of the enemy combats. Hence all of our decisions in game should in some way lead to minimizing our health loss in enemy battles. We will base our algorithm around this observation.

3.2.1 Algorithm

Assume that we are currently considering a card reward. The algorithm will use the following subroutine:

1. Denote c as the number of cards in the current card reward. Lets pick a number N - the number of card rewards we will look at into the future, including the current card draw. We will also call this the lookahead.

2. Since we don't know what card rewards we will receive in the future, we will mimic the game's random number generation algorithm to produce $N-1$ imaginary ones.
3. We then select an enemy encounter, which would be likely after this many card draws.
4. We will consider all ways of selecting an option (picking one of the cards or skipping) at each of the card rewards.
5. For each combination of options, we will simulate a battle against the enemy encounter pretending that we start with full health, and we will write down the amount of health remaining after.
6. As a result, we will have produced $(c+1)^N$ battle results. We will represent each result as a key-value pair, where the key is the option we picked at the first card reward (the one we are actually offered in game), and the value is the amount of health remaining after simulating the encounter.

In step 2 we need to decide what type of card rewards we are going to generate. Card rewards in the game are rewarded based on the type of encounter - normal, elite or boss. For our subroutine, we will only restrict ourselves to generating normal combat rewards. The reasons for this are:

1. Normal combat rewards are the most common.
2. All of the card rarities have a chance of appearing in a normal card reward.
3. The base probability of each card type appearing in a normal/elite combat reward are very similar. They don't differ by more than 10%.
4. Even though boss combats are unique because they contain 3 rare cards, they are only awarded once per act. Moreover, they only impact our deck in the next act, whereas we only focus on the current act.
5. It allows us to isolate deck building from other aspects of the game. In particular, we may not know for sure what the future card reward types will be. This could happen if the map traversal logic is decided step-by-step or is isolated from the deck building logic. We also cannot predict whether future unknown locations will contain monster encounters (and hence card rewards).

How can we effectively use the results from this subroutine? Lets use the average of the *cnt* best results for each starting card as a representation of how good the card pick is with respect to the set of card rewards that we were given. We should only consider small values for *cnt*, as an optimal agent would be able to draft in a way that only leads us to (one of) the best deck states. This gives us $c+1$ different values $avg_0, avg_1, ..., avg_{c+1}$.

We can also extend the subroutine to test the current combination of options on not just one, but many different enemy encounters. We first calculate the amount of remaining health after each battle. Then we write down the minimum remaining health after any

encounter as he score for this combination. To get the final score for one of the initial options, we use the same averaging as before.

The pseudocode (similar, but not 1:1 with python) for the modified subroutine looks as follows:

```
extra_cards = []
card_results = dict()

# all_card_rewards contains all card rewards
# the real card reward is at index 0
# other card rewards are generated ones

def rec_search(cur_pick, monsters):
    if cur_pick == len(all_card_rewards):
        # get cards that were added (ignore skips)
        added_cards = []
        for card_name in cur_extra_cards:
            if card_name != "":
                added_cards.append(card_name)

        # evaluate selection
        min_health = 1000000000
        for monster in monsters:
            end_health = simulate_battle(
            monster_name=monster,
            extra_cards=added_cards)

            min_health = min(min_health, end_health)

        card_results[extra_cards[0]].append(min_health)
    else:
        for card_name in all_card_rewards[cur_pick]:
            extra_cards.append(card_name)
            rec_search(cur_pick + 1, monsters)
            extra_cards.pop()

        # skip
        extra_cards.append("")
        rec_search(cur_pick + 1, monsters)
        extra_cards.pop()
```

Looking at the average values obtained from a single subroutine might already give us some idea of which options are the best, however, to improve accuracy, we can simply run the subroutine multiple times and average the results to receive our final scores for each option. This gives us a complete algorithm for evaluating rewards.

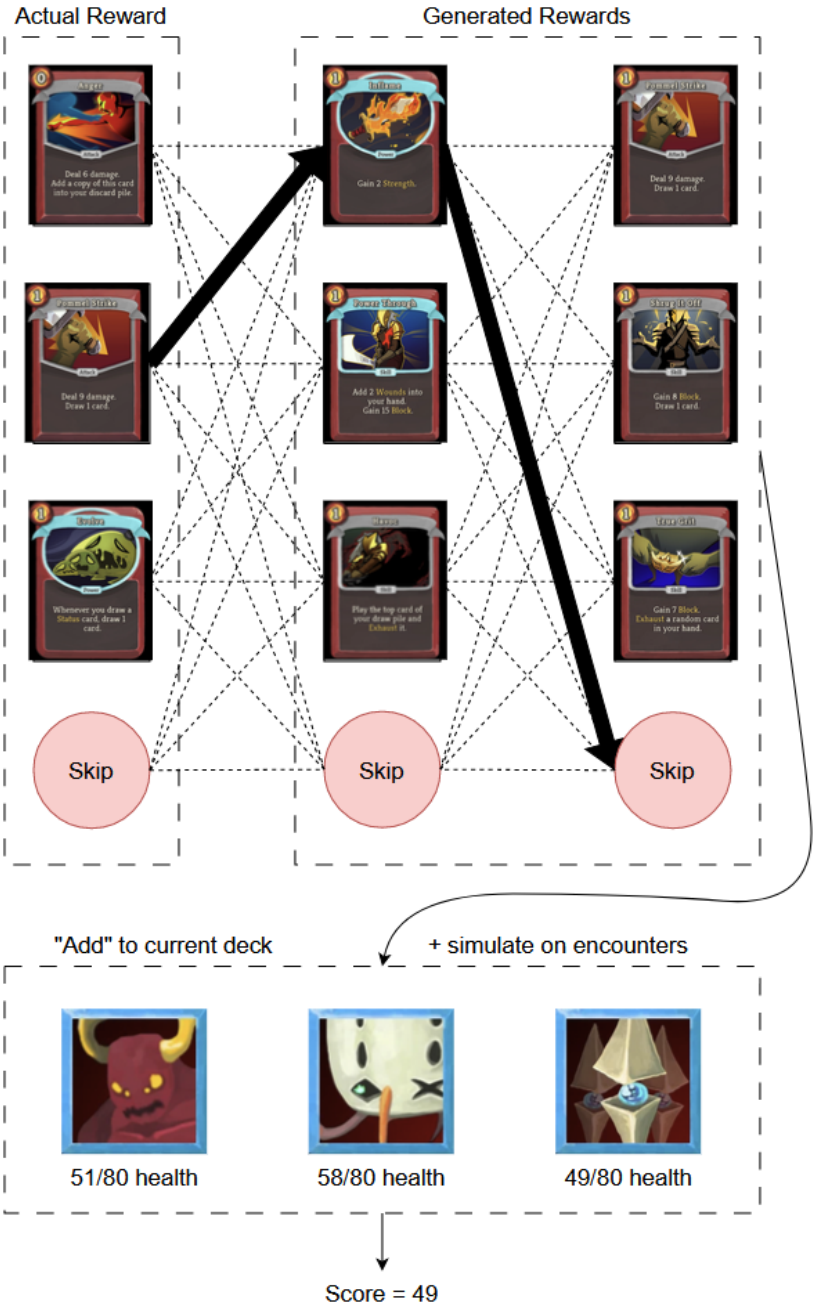


Figure 3.2: Illustration of an iteration of the top down algorithm for lookahead 3. For each combination of options, we simulate it against a set of enemies and find the minimum health remaining out of all of the encounters. For the chosen options in the example (Pommel Strike + Inflame + Skip), the minimum remaining health is 49 after the fight against 3 Sentries. Note that if we picked the hardest encounter for our deck beforehand, we would only test on that encounter.

3.2.2 Optimizing Execution Speed

Unfortunately, in practice with our setup (see Setup section), an algorithm like this is very slow. We will propose some efficiency improvements, which will allow us to

test a single Ironclad run in around 3 hours on average. So how can we optimize this algorithm while still retaining the essence of it?

Let's take a closer look at monster encounters. Both elite monster and boss encounters are a significantly more difficult challenge than normal monster encounters. While boss encounters are unavoidable, good players will also try to fight many elites during their run, since they provide relics, which may significantly strengthen the character. Hence we will not lose much accuracy in our evaluation if we only simulate battles on elites and bosses. Now, a natural way to pick an encounter is to pick elite battle when we are in the first part of the act and boss battle when we are in the second half. This makes sense, because a boss battle will likely be too difficult at the start of the act, whereas elites might not be a big enough challenge in the second half, assuming that our deck is improving smoothly.

Unfortunately, this might still be slow. We can get a threefold improvement by first simulating an encounter against all relevant elites/bosses depending on whether we are in the first or second half of the act and then picking the hardest of these encounters to run our actual simulations on. As a result of this, we will lose some accuracy, but the essence of our algorithm will remain the same.

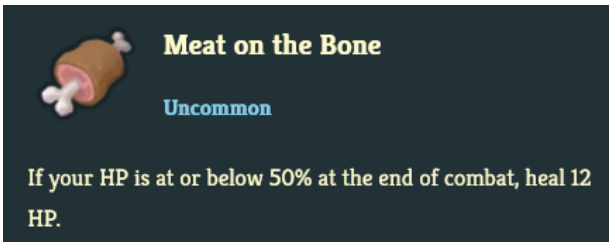
As far as parameters go, we will be using lookahead 3 in our implementation. The number of combinations we need to evaluate is exponential in terms of the lookahead and this will already give us 4^3 combinations (16 for each option), assuming that we don't have any relics that affect the number of cards in a reward. To obtain the score for an option with respect to a set of rewards, we will average the 4 best out of 16 scores from our subroutine. In total, we will only be generating 5 imaginary scenarios, i.e. doing 5 runs of the subroutine per card reward. We do this in order to be able to evaluate a reward in reasonable time (which still takes a few hours for a whole run). With better computational resources or evaluation framework we would consider more scenarios, as this would give us a more accurate average, but in order to evaluate the algorithm as a proof of concept, 5 will be enough.

To summarize the whole algorithm (including optimizations):

1. Consider either current act elites (if we are in first half of the act) or current act bosses.
2. Find the hardest encounter by simulating a battle against all relevant elites/bosses. We will use only this encounter for simulations in the subroutine.
3. Perform the subroutine 5 times, each time generating 2 extra imaginary card rewards.
4. For each subroutine, average the scores of the 4 best combinations (out of 16) to obtain a score for each option in the card reward.
5. Average the scores from all 5 subroutine runs for each option to obtain its final score.

3.2.3 Special Cases and Considerations

- Because boss battles are the last encounter of each act, we will only be able to use the reward in the next act. Because of this, it makes sense to simulate battles against the elites of the next act. However, we need to account for the fact that we also receive a powerful relic reward after the boss card reward. In my exploration I found that without this relic, the simulations produced poor results and often even resulted in skipping the card reward. I fixed this by increasing maximum energy by 1 for the simulations, which accounts for the relic reward, which often gives a similar in-battle effect. This makes our simulations more accurate, especially considering that rare cards are often slower or consume more energy.
- Potion usage might affect the end health of a battle. A good deck that uses no potions may end the simulation with less health than a deck that uses all potions. In order to avoid this, we will always perform simulations without potions. Note that an alternative would be to modify the Battle AI, but we treat the Battle AI as a black box algorithm, because it is out of the project scope.
- Some relics might also lead us to getting inaccurate card scores. For instance a deck that consumed Lizard Tail during simulation might get a better end health score than a deck which did not. With the relic Meat on the Bone, the best strategy might be to end the battle just below 50% health in order to heal. However, this can cause two decks of different strengths to get a similar score. We will run simulations without these relics whenever they are present in our inventory.



- In each act we know for sure which boss we are going to face, so does it make sense to test our future decks on other bosses as well? We will consider all bosses in acts 1 and 2, because this will prepare our deck for a more diverse range of enemy combats and make it stronger overall. However, in act 3, there is no need to simulate against the other bosses, because the boss will be the last encounter

of the game (unless the player somehow acquired the key to act 4, which is out of our control).

- We will not consider the card 'Clash' when evaluating card rewards. This card is an attack, which can only be played if all other cards in hand are attacks. It is considered by many to be the worst Ironclad card and has a score of 0 on the Spirelogs tier list. But the real reason is that it works poorly with our algorithm. When the card is being evaluated and our current deck is small, it is quite easy to find ways to play this card, so the algorithm thinks that it is good. However, as the deck gets bigger, there are almost no opportunities to play the card, so it basically becomes an extra curse in the deck.

3.3 Setup

So far, we've only described the theoretical side of the algorithms. In reality, we need to build a framework which will allow us to implement and test the described deck building algorithms in real Slay the Spire runs. The difficult part was building the required setup for the top-down algorithm, due to the fact that similar deck building algorithms have not been implemented by others in the past. Some parts of the setup were relevant to the bottom-up approach as well, which was otherwise not too difficult to implement.

We will make use of the CommunicationMod, which allows an external program to communicate with the game by sending commands and receiving game state changes. Additionally, spirecomm is a good starting point for us, because it is a functional Slay the Spire agent, and it takes care of other decisions in the game, which are out of our project scope. However, we are still missing various components.

1. We need a strong Battle AI. For the top-down algorithm, this is a crucial part, because otherwise we will not be able to run accurate simulations.
2. We need to be able to perform simulations on a custom encounter with extra cards.
3. We need to be able to invoke these commands from spirecomm.

We will achieve this by making use of and modifying some existing mods.

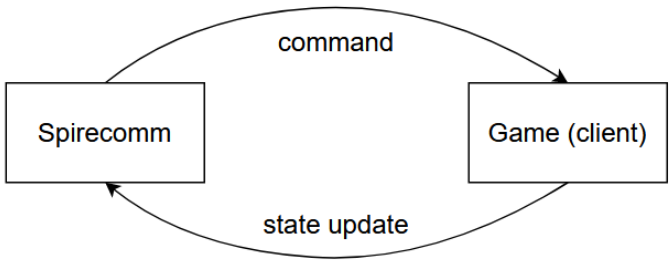


Figure 3.3: Interaction between spirecomm and the game using CommunicationMod

3.3.1 Custom States

We will make use of the existing STSStateSaver mod. As explained in the background chapter, this mod allows us to create and load in-game states. We will extend the mod by creating a method with the signature

```
public void loadCustomBattle(
    String monsterName,
    ArrayList<AbstractCard> extraCards,
    int playerHealth,
    int extraEnergy)
```

which mimics most of the logic from the method

```
public void loadState()
```

which is used for loading saved states. In our custom method, we will apply the changes indicated by the parameters. We will distribute the extra cards randomly throughout the deck and refresh our hand with a new card draw. We also need to carefully make sure that all relic effects are consistent with the newly loaded encounter. For instance, some relic effects only apply in elite battles - Sling of Courage and Preserved Insect, and some only apply in elite and boss battles - Slaver's collar. This is important to consider in cases such as when we want to load an elite enemy encounter onto a saved normal enemy battle state.

3.3.2 Battle AI

As mentioned in the background chapter, there already exists a strong battle AI mod called scumthespire, which uses STSStateSaver. Normally, it requires a separate instance of the game running in server mode, which does all of the calculations and sends them back to the client to be executed. The autobattler is invoked by clicking a designated button during an encounter. We will make use of scumthespire for both regular battles and battle simulations.

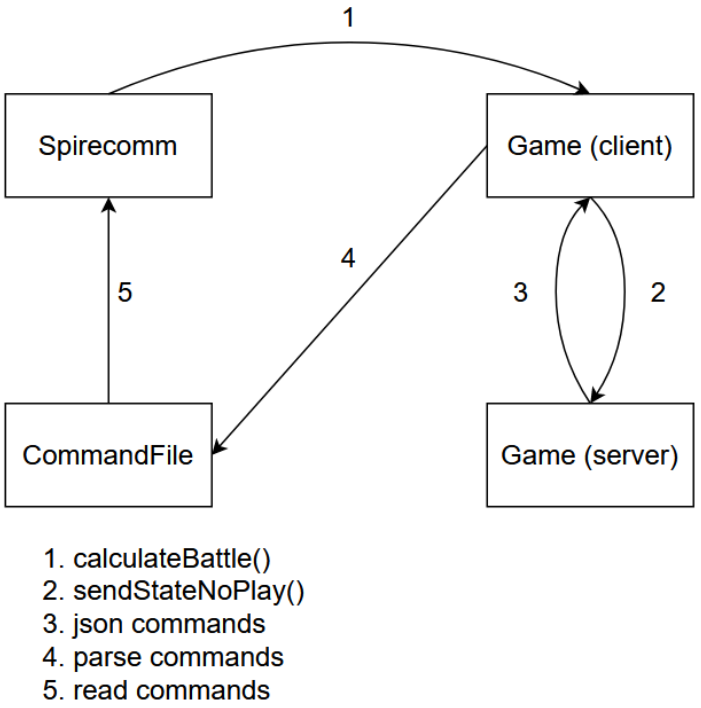
For regular battles, we introduce some custom methods. The most important ones are:

- `public void calculateBattle()`

  This is the method to be called by CommunicationMod when we want to calculate a battle.

- `public void sendStateNoPlay()`

  This method is called from *calculateBattle* after some boilerplate code and initializations. It sends the current in-battle state to the server which returns a JsonArray containing the command list back to the client. We then parse and write the commands to a file which can be read by spirecomm. We cannot simply execute the commands, because it would interfere with the execution flow of spirecomm, which would still receive game state updates and try to find an appropriate action.



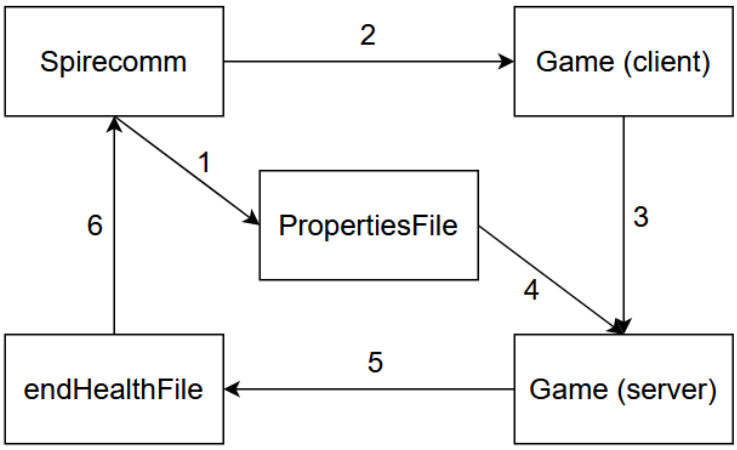
We also introduce some new methods for custom battle simulation. The most important ones are:

- `public void simulateCustomBattle()`

  This is the method to be called by CommunicationMod when we want to simulate a custom battle. It accepts no parameters, because all of the properties will be written to a file by spirecomm.

- `public void sendCustomState()`

  This method works similarly to the general one. It sends a game state to the server, which applies the custom changes by using the *loadCustomBattle* method from STSStateSaver and performs calculations. In this case we are not concerned about the command list, just the remaining health after the battle, which is written to a file that can can be read by spirecomm.



1. write simulation properties to file
2. simulateCustomBattle()
3. sendCustomState()
4. read and apply battle properties
5. write remaining health to file
6. read remaining health from file

It is important to note that if we try to apply custom battle changes to the card reward screen, it will not work. We need to instead save an in-battle state at the start of the encounter and apply our changes to that state.

3.3.3 CommunicationMod

The changes related to CommunicationMod are relatively simple. We just add custom commands for starting the server, calling *calculateBattle* and *simulateCustomBattle* methods. We name the commands ”startserver”, ”calcbattle” and ”simulatebattle”. CommunicationMod can be extended by either modifying the source code directly or by patching the relevant CommunicationMod code using *@SpirePatch* annotation, similarly to how it is done for save state commands in CommunicationModExtension. To make our life easier, we also implement a fourth command which allows us to resume a game from save file when starting spirecomm, however we will not describe it here as it is not in the project scope.

3.3.4 Spirecomm

Finally, we need to make use of the new functionality in spirecomm. We will start the server once before executing any other commands. We wait for the server to be fully initialized by listening to a flag which will be set to true inside a predefined file. Inside an encounter, we will use the ”calcbattle” commmand, which will write the command sequence into a file. Then we will simply send these commands to the game one by one. When offered a card reward, the top-down algorithm will be able to utilise the ”simulatebattle” command to test custom encounters and read the remaining health from a file.

3.3.5 SuperFastMode

We utilised the SuperFastMode mod to speed up animations and actions in Slay the Spire runs. It did not make as much of a difference on the top-down algorithm, as simulating battles was a bigger bottleneck, but it made playing out runs slightly faster.

3.3.6 Notes

Unfortunately, due to some animations that happen during Hexaghost battle, it is not possible to load a custom state into server and simulate battles against this act 1 boss. However, it is still possible to use the battle AI to fight Hexaghost in a real encounter. Luckily, no other bosses or elite enemies caused issues. Our top-down algorithm will be slightly worsened by this fact, but it is still very usable and there should be no problems past the first act.

Additionally, it's important to note that the scumthespire battle AI does not perfectly mimic how a human player would play. Due to the fact that it utilises loading and saving of states, it will implicitly get information about some random factors in the game. Effectively, the battle AI will more or less 'know' how cards are ordered in a player's deck. This does not make the AI unrealistically good. In fact, it still does not perform as well as a human when given a complex deck and difficult encounter. When evaluating the top-down algorithm we will need to keep in mind that it may be slightly biased towards cards with random effects.

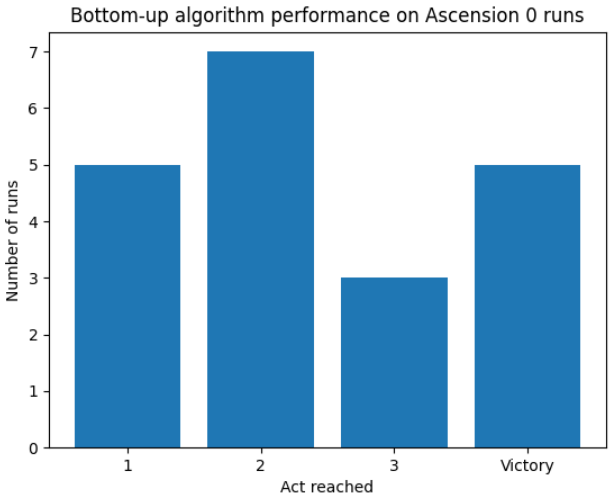
Chapter 4

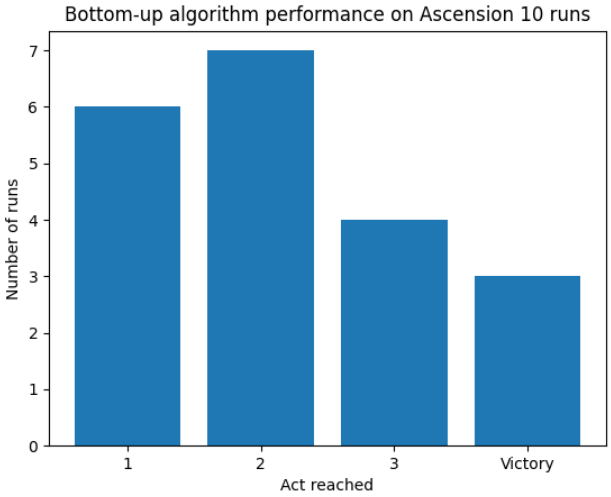
Evaluation

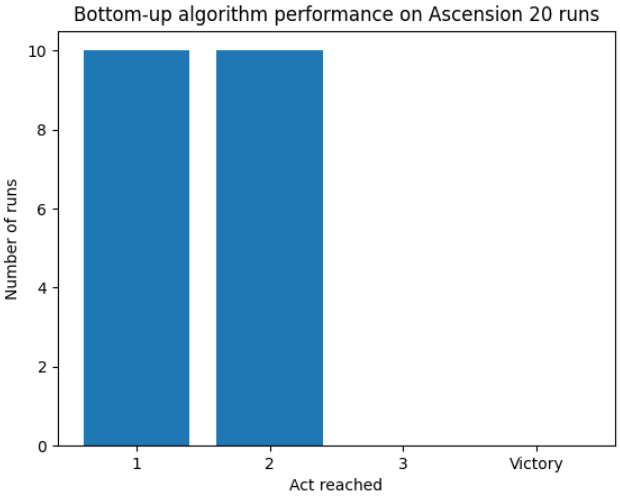
In this chapter we will evaluate the performance and decision process of the two algorithms. Playing out Slay the Spire runs with the algorithms was a time-consuming process. The bottom up approach took about 10 minutes per a victorious run, whereas the top-down algorithm was much slower, taking as much as 3 hours. Because of this, the amount of statistical analysis we will be able to do is very limited, especially for the top-down approach. The algorithms should mostly be viewed as a proof of concept and we will not be evaluating the execution speed, as it is something that can be improved with a different testing setup. Nevertheless, we will be able to do some meaningful analysis by taking a look at the decision process and other specifics, which will give us an insight into how good the algorithms are.

4.1 Performance on Different Ascensions

In this section we examine how well the algorithms perform on different Ascension levels. Firstly, let's look at the performance of the bottom-up algorithm, which we were able to get a moderate amount of results for. We simulated runs on Ascensions 0, 10 and 20, by using Spirecomm with the bottom-up deck-building algorithm. Note that by Ascension 0, we refer to the base game difficulty without any Ascension modifiers. The results can be seen below. The bar plots depict the number of runs that ended on act 1/2/3/won for each of the 3 different Ascension settings.


Bottom-up algorithm performance on Ascension 0 runs
Number of runs
0
1
2
3
4
5
6
7
1
2
3
Victory
Act reached


Bottom-up algorithm performance on Ascension 10 runs
Number of runs
0
1
2
3
4
5
6
7
1
2
3
Victory
Act reached

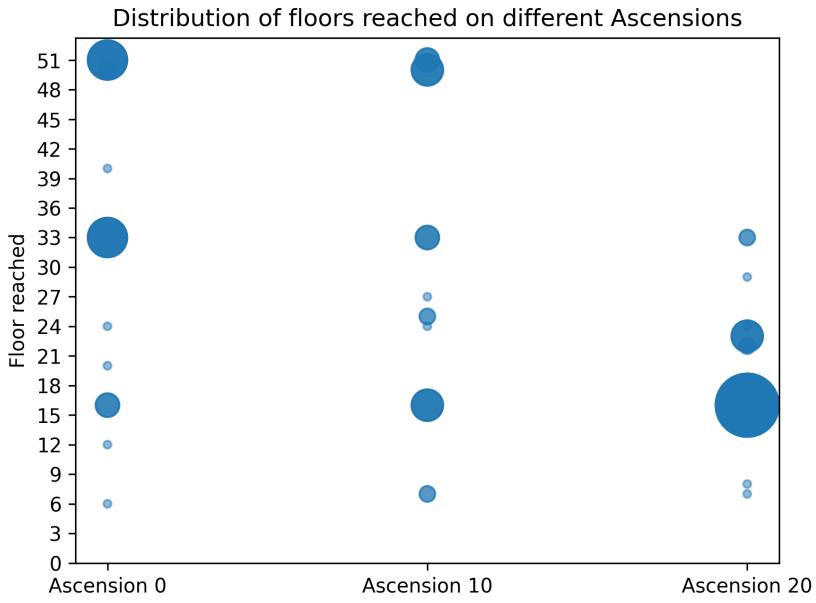

Bottom-up algorithm performance on Ascension 20 runs
Number of runs
0
2
4
6
8
10
1
2
3
Victory
Act reached

As expected, the best results were achieved on Ascension 0, whereas Ascension 20 was by far the most challenging setting. The results for Ascension 0 and 10 are surprisingly not very different, with the most notable distinction being the 2 extra wins achieved on the easiest setting.

Why is there such a small difference between Ascension 0 and Ascension 10? Firstly, it might be due to the fact that we were only able to test on a moderately large sample size of 20 runs, which does not give us a perfectly accurate representation of the difference in difficulty. Secondly, the first Ascension modifier, which causes approximately 60% more elite enemies to spawn on the map, may be a blessing in disguise. By fighting more elites, we are able to get more relics, which makes us significantly stronger and able to survive longer.

We can see how harsh the difficulty is on Ascension 20. None of the runs were able to reach act 3 and only half of the runs reached act 2. It is expected that the algorithm would be unable to win any runs on Ascension 20, because it requires making very high level decisions for all aspects of the game. However, the fact that the agent was often defeated so early could mean that our algorithm was not picking cards well enough to deal with short-term problems.

Let's take another look at the results, but this time with a bit more granularity.



Note that floors 16, 33 and 50 are boss battles, which explains why so many runs ended there.

In the plot, we can see more clearly that a larger amount of runs successfully reached act 2 and 3 bosses on Ascension 0 than Ascension 10.

We can also see that a lot of the Ascension 20 runs actually reached the act 1 boss, but the deck was not good enough to beat it. In theory, our top-down algorithm should

usually be able to get further on Ascension 20, since it evaluates cards based on their usefulness in the short term. Playing out all 20 runs would take a very long time, but we were able to get results for 10 of the runs.

Run number	Floor reached (bottom-up)	Floor reached (top-down)
1	22	27
2	16	18
3	16	22
4	16	21
5	16	33
6	16	16
7	7	16
8	16	24
9	23	33
10	33	33

As we can see, the top down approach performs consistently better on Ascension 20, which means that the algorithm utilises the evaluations well for short term decisions. The most drastic difference is the increase from floor 16 to 33 (act 1 to act 2 boss) in run 5. The bottom-up algorithm entered the Slime Boss fight with 75/75 health and lost. The top-down algorithm entered the fight with 74/75 health and ended with 45. Let's see which cards the two algorithms picked on act 1 to determine why the top-down algorithm performed so much better.

Firstly, Neow granted Reaper as a random card reward at the start of the game. The card rewards offered in act 1 were:

1	Bloodletting	Perfected Strike	Shrug It Off
2	Shrug It Off	Wild Strike	Whirlwind
3	Heavy Blade	Shrug It Off	Entrench
4	Cleave	Flex	Bludgeon
5	Havoc	Flex	Thunderclap
6	Cleave	Carnage	Flame Barrier
7	Rupture	Clash	Reckless Charge

The bottom-up algorithm chose Shrug It Off, Shrug It Off, Shrug It Off, Bludgeon, Havoc, Carnage, Reckless Charge.

The top-down algorithm chose Perfected Strike, Wild Strike, skip, Flex, Havoc, Flame Barrier, skip.

As we can see, the top-down algorithm added a nice variety of cards. Wild Strike is a source of cheap damage and it synergizes with the effect of Perfected Strike, which is a good damage option for a higher cost. Flex amplifies the damage output with the only drawback being that it takes up a space in the hand. Flame Barrier is a good option for shielding strong attacks and dealing some extra damage, whereas Havoc is a situational card that can exhaust some of the worse cards. Skipping card rewards 3 and 7 makes sense, except arguably picking Shrug It Off could have been a good idea, as none of the other cards provide much additional value in the short term.

The problem with the bottom-up deck is that it does not have a very reliable damage source. The amount of Shrug It Off cards is excessive and does not help in fights that require high damage output, like Slime Boss. On the other hand, a card like Bludgeon is luck dependent and is only good on certain turns, since it uses all of the player's energy, making them unable to block on the same turn. Bludgeon is strong when played on a vulnerable enemy, but the only way to add vulnerable is through Bash, which is not a very reliable setup. Reckless Charge is a pretty good card, but it can bloat the deck with generated status effects. Carnage is probably the best attack card, but it is not cheap and must be played when drawn, due to the Ethereal effect (exhaust if unplayed).

As we can see, the top-down algorithm added cards that provided more versatile and reliable damage options, which is why it was much more successful on the Boss fight.

We also provide results of some runs by the two algorithms on some lower Ascensions. Note again, that the sample size is not very large due to the long running time of the top-down algorithm.

Ascension	Floor reached (bottom-up)	Floor reached (top-down)
0	31	Victory
5	Victory	Victory
5	16	16
5	Victory	Victory
5	Victory	Victory
10	42	33
10	50	50
10	33	33
10	33	22

As we can see, the performance of the two algorithms is very similar. The bottom-up approach outperforms the top-down one in two runs, whereas the opposite is true in one of the runs. The lower difficulty setting does not punish bad short-term decisions made by the bottom-up approach as well as Ascension 20, which is why the algorithm is able to create decks with stronger scaling.

4.2 Card Pick Rates

Below is a table detailing the pick rates of different cards when using the top-down algorithm. It is made of a small sample size of Ascension 0-10 runs. Nevertheless we might still be able to draw some conclusions from it.

Card Type	Card Name	Offered/Picked	Pick Rate
Skill	Battle Trance	3/3	100%
Attack	Feed	1/1	100%
Skill	Power Through	3/4	75%
Skill	Ghostly Armor	6/9	66.6667%
Attack	Double Tap	3/5	60%
Skill	Spot Weakness	4/7	57.1429%

Power	Inflame	3/6	50%
Attack	Fiend Fire	2/4	50%
Attack	Hemokinesis	1/2	50%
Attack	Carnage	4/8	50%
Skill	Burning Pact	2/4	50%
Attack	Offering	1/2	50%
Power	Berserk	2/4	50%
Attack	Twin Strike	4/8	50%
Attack	Pommel Strike	8/17	47.0588%
Attack	Blood for Blood	3/7	42.8571%
Skill	Armaments	6/15	40%
Attack	Exhume	2/5	40%
Power	Metallicize	1/3	33.3333%
Attack	Wild Strike	4/12	33.3333%
Skill	Shrug It Off	3/9	33.3333%
Attack	Pummel	1/3	33.3333%
Attack	Clothesline	5/15	33.3333%
Skill	Disarm	1/3	33.3333%
Power	Dark Embrace	1/3	33.3333%
Skill	Entrench	1/3	33.3333%
Attack	Thunderclap	5/16	31.25%
Attack	Iron Wave	4/13	30.7692%
Skill	Second Wind	3/10	30%
Attack	Perfected Strike	2/7	28.5714%
Skill	Shockwave	2/7	28.5714%
Skill	True Grit	3/11	27.2727%
Attack	Uppercut	1/4	25%
Attack	Impervious	1/4	25%
Skill	Infernal Blade	1/4	25%
Attack	Sever Soul	2/8	25%
Skill	Seeing Red	1/4	25%
Power	Juggernaut	1/4	25%
Skill	Havoc	3/14	21.4286%
Skill	Rage	1/5	20%
Attack	Anger	2/12	16.6667%
Attack	Immolate	1/6	16.6667%
Skill	Sentinel	1/6	16.6667%
Power	Rupture	1/6	16.6667%
Attack	Heavy Blade	2/13	15.3846%
Attack	Cleave	3/20	15%
Attack	Sword Boomerang	2/15	13.3333%
Skill	Warcry	2/17	11.7647%
Skill	Flex	1/9	11.1111%
Attack	Body Slam	1/13	7.69231%

Skill	Intimidate	0/5	0%
Power	Evolve	0/6	0%
Attack	Whirlwind	0/5	0%
Attack	Headbutt	0/8	0%
Attack	Clash	0/21	0%
Power	Corruption	0/2	0%
Power	Barricade	0/3	0%
Power	Combust	0/5	0%
Power	Feel No Pain	0/3	0%
Power	Brutality	0/2	0%
Skill	Flame Barrier	0/3	0%
Skill	Dual Wield	0/5	0%
Attack	Searing Blow	0/4	0%
Attack	Limit Break	0/2	0%
Attack	Reckless Charge	0/3	0%
Attack	Reaper	0/3	0%
Attack	Rampage	0/4	0%
Power	Demon Form	0/2	0%
Attack	Dropkick	0/2	0%
Skill	Bloodletting	0/11	0%
Attack	Bludgeon	0/4	0%
Power	Fire Breathing	0/4	0%

One thing that immediately stands out is the low pick rate of power cards. There is a good probability that part of this is due to the Battle AI algorithm we used, which might have not been able to utilize these cards effectively. However, it could also be due to the fact that we only looked at 2 future card rewards. This could have prevented the algorithm from seeing the usefulness of these cards in the bigger picture and consequently more consistent, low-risk alternatives were favoured. We can see that similarly cards like Reaper and Whirlwind were not picked, even though they can be win conditions with good supporting cards.

Interestingly, we can see that the cards which were mentioned as ”must pick” in the first article - Armaments, Battle Trance and Inflame, all had high pick rates. Without any external influence, the algorithm was able to achieve a similar conclusion to that of human analysis.

The low pick rate of cards like Bloodletting, Body Slam, Warcry is not very surprising, as they do not provide a lot of value by themselves. However, if were to analyze runs that were defeated less often in the early floors we would expect them to be more popular.

Strangely, cards with random effects were around/below average in popularity, even though the battle AI should have in theory been able to use them better than a human player. Perhaps this could be attributed to their unpredictable nature. Even if we

know exactly what these cards will do, there is no guarantee of them frequently being beneficial in encounters.

4.3 Example Run Analysis

We will take a closer look at a victorious Ascension 5 run by the top-down algorithm, highlighting some key card reward choices.

After an elite fight on floor 10, the agent was offered a card reward consisting of **Exhume+**, **Dual Wield**, **Wild Strike**.

The current deck, excluding basic cards, consisted of **Thunderclap+**, **Shockwave+**, **Uppercut**, **Ghostly Armor+**.

After running simulations on the Guardian boss encounter, the algorithm calculated the following scores.

Exhume+: 73.45, **Dual Wield**: 52.25, **Wild Strike**: 57.2, **Skip**: 56.5 .

The algorithm was able to spot how powerful **Exhume+** was in this situation. It could allow playing **Shockwave+** twice, which would be really powerful against the boss. It would also be able to retrieve **Ghostly Armor+** from the exhaust pile, if necessary.

At the beginning of act 3, the deck was doing poorly in simulations against elite encounters. At that point the deck consisted of the cards **2x Thunderclap+**, **Shockwave+**, **Uppercut+**, **Ghostly Armor+**, **Exhume+**, **Armaments+**, **Havoc+**, **Double Tap+**, **2x Flex+**, **Iron Wave+**, **Infernal Blade+**, **Seeing Red+**, **Immolate+**.

The agent was offered a card reward consisting of **Battle Trance+**, **Cleave**, **Rage+**.

The scores from simulations on Giant Head were:

Battle Trance+: 46.65, **Cleave**: 11.0, **Rage+**: 9.3, **Skip**: 8.3 .

As we can see, the algorithm correctly deduced that the deck was lacking card draw, making **Battle Trance+** and invaluable addition to the deck.

Finally, after adding a few more cards to the deck - **Power Through+**, **Pommel Strike**, **Second Wind+**, **Iron Wave** it was offered **Spot Weakness+**, **Ghostly Armor+**, **Sever Soul+**.

The scores from simulations on Donu and Deca were:

Spot Weakness+: 79.5, **Ghostly Armor+**: 76.75, **Sever Soul**: 76.45, **Skip**: 73.75 .

Here, even though the deck was already strong, the algorithm was able to infer that the deck lacked scaling the most and added **Spot Weakness+**, as a source of extra strength.

As we can see, the top-down algorithm is able to perform intelligent decisions at critical parts of the game, which we can find explanations for using reasoning.

Chapter 5

Conclusions

In this chapter, we summarize our findings from evaluating the two proposed algorithms. We will first justify whether the results achieved were good or did not meet expectations. Secondly, we will discuss the strengths and weaknesses of the two approaches. Thirdly, we will mention potential applications of the algorithms. Finally, we will consider areas of further work that could be done.

5.1 Performance

Judging by the results, the performance of the algorithms was even better than expected. The algorithms were able to get far and create winning decks on Ascension 10, which is a substantial achievement. Moreover, the top-down algorithm showed potential in Ascension 20 runs by being able to consistently beat the first act. It is important to keep in mind that the project only focused on one element of the game - card drafting. Hence it would be unreasonable to expect victories on Ascension 20, which requires harmoniously optimizing all parts of the gameplay and making as few mistakes as possible. The agent is still not intelligent enough to make choices that are good enough in various other aspects in the game. This includes knowing when we can smith at rest sites instead of resting, how to spend gold, what to do in events, what map route to take, what relics to choose.

We can conclude that short term decisions were more important than long term decisions on Ascension 20, as displayed by the difference in performance between the two algorithms. However, we also saw that simply taking short term decisions was not always enough, because it would result in the deck not having enough scaling. Consideration of synergies made our deck stronger when it was able to survive the early portion of the game.

We made the correct decision by deciding to only run simulations on elite and boss combats as almost all of the runs ended on these types of encounters.

5.2 Strengths and Weaknesses

As expected, the two algorithms showed different strengths and weaknesses. The top-down algorithm was able to make good short-term decisions. In the comparison of Ascension 20 runs, we saw that it outperformed the bottom-up algorithm quite consistently. However, the top down algorithm was prone to overly favouring cards with immediate value, and hence did not pick power cards as often as necessary. The opposite was true for the bottom-up approach. It excelled in runs where it could add good scaling cards without fully optimizing in the short-term, which is why we saw it mostly perform as well and sometimes even better than the top-down algorithm on the lower Ascensions.

5.3 Applicability

Apart from Slay the Spire card drafting, the exact bottom-up algorithm cannot be applied elsewhere. However, the fundamental idea of valuing cards higher depending on the number of synergies could be applied to other games. The only caveat is that it requires research and/or a deep understanding of the game in order to pick good synergy classes.

On the other hand, the idea of the top-down algorithm is mostly independent of the game's specifics. Hence the structure of the method for evaluation could be applied in other decision making problems. Below is a list of potential applications.

- Card drafting for other classes.
- Figuring out the best card to upgrade.
- Choosing which card to remove.
- Choosing between different relics (assuming they alter in-combat elements).
- Choosing between different potions.
- Creating a deck building algorithm for a similar roguelike card game, such as Monster Train or Griftlands.

The template of the algorithm would be used as follows:

- Consider the modified state after each choice.
- Generate future scenarios with imaginary changes to the state that could happen in the future.
- Evaluate these future states and calculate scores for choices based on their success in the future.
- Run for many times and average results in order to get more accurate scores.

5.4 Future Work

The following improvements could be done in the short-medium term:

- Using the headless version of the game, sts_lightspeed. This would require us to adapt and reimplement most of the code we are using currently as well as most of Spirecomm. As a result, we would be able to do millions of playouts in seconds. This would greatly increase the speed at which we can test the algorithms. Moreover, we would not need to resort to using optimizations for the top-down algorithm and we could tweak the parameters to increase the quality of decision making.
- Combining strengths of both approaches. For instance, we could draft cards using the top-down algorithm until we were strong enough against the currently relevant elites and bosses. Then we could switch to the bottom-up algorithm, which is better at picking long-term scaling cards.
- Dynamically increasing the difficulty of encounters we simulate on. We could monitor how well the deck performs against a current set of elites/bosses and we would switch to using harder enemies as the deck becomes better instead of this being tied to the current act.
- Evaluating cards with respect to the potential increase in value we get from upgrading them.

Bibliography

[1] Alleji. Spirelogs. https://spirelogs.com/.

[2] Blizzard Entertainment. Hearthstone. https://hearthstone.blizzard.com/en-gb.

[3] boardengineer. CommunicationModExtension. https://github.com/boardengineer/CommunicationModExtension.

[4] boardengineer. LudicrousSpeed. https://github.com/boardengineer/LudicrousSpeed.

[5] boardengineer. scumthespire. https://github.com/boardengineer/scumthespire.

[6] boardengineer. STSStateSaver. https://github.com/boardengineer/STSStateSaver.

[7] Zhengxing Chen, Christopher Amato, Truong-Huy D. Nguyen, Seth Cooper, Yizhou Sun, and Magy Seif El-Nasr. Q-deckrec: A fast deck recommendation system for collectible card games. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8, 2018.

[8] daviscook477. BaseMod. https://github.com/daviscook477/BaseMod/.

[9] Ethan Gach. Tips for Playing Slay the Spire. 2019.

[10] Forgotten Arbiter. CommunicationMod. https://github.com/ForgottenArbiter/CommunicationMod.

[11] Forgotten Arbiter. spirecomm. https://github.com/ForgottenArbiter/spirecomm.

[12] Forgotten Arbiter. Card priority spreadsheet. https://docs.google.com/spreadsheets/d/1QflNw6T9WANLqbEFI6KofA095iUTyvZW1dmw2ACK0IQ/edit#gid=0, 2018.

[13] Forgotten Arbiter. Ironclad Win Streak Guide. 2018.

[14] gamerpuppy. sts_lightspeed. https://github.com/gamerpuppy/sts_lightspeed.

[15] Pablo García-Sánchez, Alberto Tonda, Giovanni Squillero, Antonio Mora, and Juan J. Merelo. Evolutionary deckbuilding in hearthstone. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8, 2016.

[16] Harry Alston. Slay The Spire: Ascension 20 Complete Guide. 2021.

[17] Andrzej Janusz, Łukasz Grad, and Dominik Slezak. Utilizing hybrid information sources to learn representations of cards in collectible card video games. In *2018 IEEE International Conference on Data Mining Workshops (ICDMW)*, pages 422–429, 2018.

[18] Kiooeht. ModTheSpire. `https://github.com/kiooeht/ModTheSpire`.

[19] Klei Entertainment. Klei Entertainment. `https://www.klei.com/games/griftlands`.

[20] Mega Crit Games. Slay the Spire. `https://store.steampowered.com/app/646570/Slay_the_Spire/`.

[21] Shiny Shoe, Good Shepherd Entertainment. Monster Train. `https://www.themonstertrain.com/`.

[22] Skrelpoid. SuperFastMode. `https://github.com/Skrelpoid/SuperFastMode`.

[23] Andreas Stiegler, Claudius Messerschmidt, Johannes Maucher, and Keshav Dahal. Hearthstone deck-construction with a utility system. In *2016 10th International Conference on Software, Knowledge, Information Management Applications (SKIMA)*, pages 21–28, 2016.

[24] Mikolaj Trojanowski and Johan Andersson. Are you lucky or skilled in slay the spire? : An analysis of randomness., 2021.

[25] Wizards of the Coast. Magic: The Gathering. `https://magic.wizards.com/en`.

Appendix A

Ironclad card list

Figure A.1: Ironclad starter cards

Figure A.2: Ironclad common cards

Figure A.3: Ironclad common cards

Figure A.4: Ironclad uncommon cards

Figure A.5: Ironclad uncommon cards

Figure A.6: Ironclad uncommon cards

Figure A.7: Ironclad rare cards

Figure A.8: Ironclad rare cards

Appendix B

Ascension modifiers

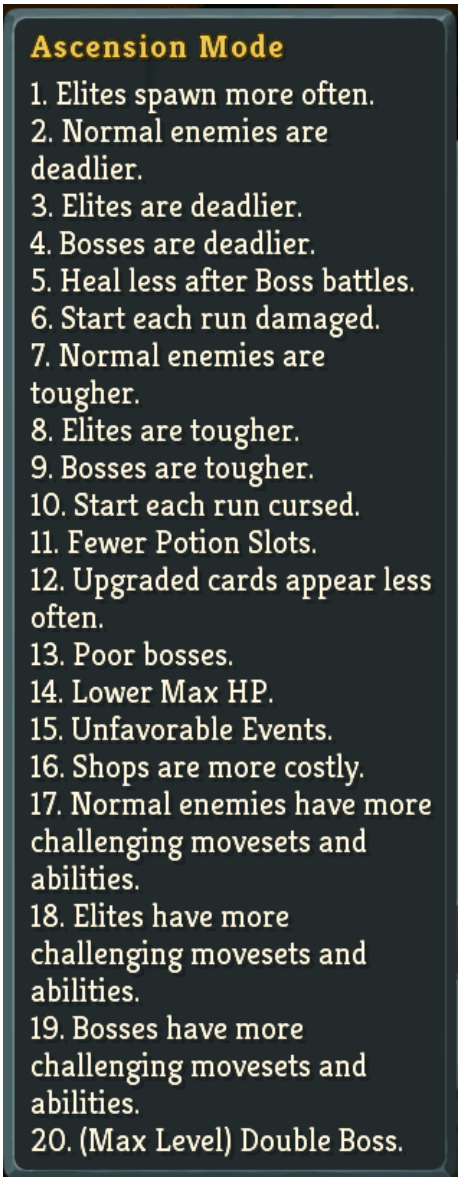


Figure B.1: Modifiers for Ascensions 1-20