LandFill

Leah Liddle

September 1, 2025

Contents

1 Introduction

1.1 Overview

Magic: The Gathering (MTG) is a Trading Card Game (TCG) designed by Wizards of the Coast (WOTC). Players take the role of a wizard, whose deck is a library of spells with which they battle one or more similarly equipped opponents. Pursuit of the hobby thus involves mastery of both gameplay and deck construction. While tournament-level participation relies on "netdecking" - copying a decklist with a history of competitive success - the hobby is intended to have a significant creative component, with the unranked Commander format gaining popularity in recent years through its focus on unique and personal decks [8].

In addition to spell cards, decks also contain "land" cards, which generate "mana", a resource expended to cast spells. A deck's lands collectively form its "manabase". Mana comes in five colours: White, Blue, Black, Red and Green (abbreviated respectively to W, U, B, R and G, or WUBRG collectively). Spells require specific colours, and lands produce one or more colours. A deck including spells of many colours has access to more spells, but runs an increased risk of a "colour screw", in which the player does not draw lands of the colours they need. Whereas selecting a list of spells is a stimulating creative pursuit, choosing a deck's manabase is less so: within a set budget, and notwithstanding the minority of lands which come with effects outside mana provision,

there are objectively lists of lands that will maximize a player's chances of being able to play their spells.

LandFill is a webapp that automates this aspect of deck building. Users input a list of spell cards, select some broad preferences for their manabase, and are provided with a manabase that has been optimized within those preferences to facilitate reliable casting of their spells. This optimization is necessarily approximate: Churchill *et al* have demonstrated that, as it is possible to construct within an MTG game a Turing Machine whose halting is the necessary condition for a player's victory, a deck's winning strategy is undecidable [3], meaning that it is beyond LandFill's capacity to determine what lands support the most decisive plays. Nevertheless, the heuristic that the winning MTG player is typically the one who spends the most mana over the course of the game [12] provides an opening for approximating the optimum via Monte Carlo methods. Using a stripped-down MTG simulator, in which the simulated player aims only to spend as much mana as possible each turn, LandFill estimates over iterated games how effective a given manabase is. It then produces successively optimized manabases via a Hill Climbing Algorithm. While LandFill may never rival the experienced eye of a seasoned competitive deckbuilder, it is nevertheless my contention that an app that provides a list of lands of demonstrated efficiency, via a click of a button, would pay dividends for casual players in ease of deckbuilding and satisfaction of games.

1.2 Thesis Layout

My writeup here documents the development and testing of LandFill. I will begin by outlining relevant MTG rules and design trends, and the difficulties these introduce for optimization, followed by an overview of how LandFill will approach these. I will then outline my initial consultations with MTG players, and the features they require from a manabase optimization app. Dividing LandFill into four components - a MTG card database, an MTG game simulator, an optimization algorithm implementer, and a user interface - I will then outline its structure, and justify the decisions made during development. I will then summarize a second round of user-testing, and outline the value added by LandFill. I will conclude with an evaluation of the strengths and limitations of the app, and areas for future development.

2 Problems and Proposed Solutions to MTG Manabase Optimization

This section will provide an overview of MTG's rules and design practices insofar as they relate to manabase optimization, and the challenges which emerge form this. It then outlines the algorithmic responses to these challenges that will be used by LandFill. It will then touch on how incorporation of this into a deployable consumer product will be tested and developed. Finally, it will situate LandFill within existing scholarship on MTG automation and optimization.

2.1 Gameplay Concepts and Terminology

2.1.1 Overview

MTG can be played in several "Formats", with different deckbulding stipulations but largely identical rules, an overview of which is provided here.

Figure 1: A spell card

At the start of an MTG game, each player shuffles their deck (sometimes referred to as a "library") and draws a "hand" of seven cards. A player may "mulligan" a poor hand, shuffling it back into the deck and drawing a fresh one. While mulligan rules vary, players typically incur an increasing penalty for each mulligan performed. One additional card is drawn at the start of each turn. Cards in hand may be played onto the "battlefield". Each turn, a player may play one land card, which they may use once per turn by "tapping" it (turning it 90°). All lands untap at the start of each turn. A player, if they draw sufficient land cards, should therefore have access to one mana on their first turn, two mana on their second turn, and so forth. Spell cards which increase the amount of mana available per turn at a higher rate than this are called "ramp" spells.

The "mana cost" of most spells includes a generic cost, payable by any mana, and any number of "pips", each representing a single required colour. Figure 1 shows a card requiring two black mana, one red mana, one blue mana, and four generic mana. It would thus be said to have a "cost" of UBBR4, and a "converted mana cost" (CMC) of eight. The spread of CMCs across a deck's spell cards is called its "curve".

Each colour is produced by a "basic land": Plains (W), Island (U), Swamp (B), Mountain (R) and Forest (G). Whereas no deck may contain more than four copies of the same card (sometimes one copy, in "singleton" formats), any deck may contain any number of basic lands. In addition to the type "land", a land card may have subtypes, providing opportunities for synergy. Confusingly, the five basic lands, in addition to being named cards, are also card subtypes, collectively called the five "basic landtypes". Tropical Island, for example, is a non-basic land which has the subtypes Island and Forest. Any card, therefore, which references "an island" or "a forest" could have that criteria met by Tropical Island, a Basic Forest or a Basic Island. Within MTG player parlance, which will be used in this text, saying "a Forest" may refer to any card with the Forest subtype, while saying "a Basic Forest" refers to the specific card named Forest (Basic Forest, helpfully, also has the Forest subtype).

Any card with a basic landtype taps for the corresponding colour of mana by default. However, not every card that taps for that colour of mana has that subtype. Tropical Island, for example, taps for both G and U, as does Hinterland Harbour, which is neither a Forest nor an Island.

Although a small minority of lands produce more than one mana per tap, this is exceptionally rare. Through this writeup, a land which land which "produces BUG", taps for Black, Blue *or* Green. Reflecting this, I will use the below standard to represent a player's hand or "battlefield" (zone to which cards are played from the hand):

$$\text{Hand} = \big[\text{Spell}(RUG) \quad \text{Land}(RU) \quad \text{Land(Basic Mountain)} \quad \text{Land(Basic Island)}\big]$$

If a card is identified by a set of pips instead of a name, note that if it is a spell this refers to the colours it *requires*, while if it is a land it refers to colours it *can produce*. In the above hand, the Spell requires Red *and* Blue *and* Green mana, while the first land produces Red *or* Green.

Lands that produce two colours are called Dual Lands, while lands that produce three are called Tri Lands; the small subsect of lands that can produce all five colours are called WUBRG lands. Some lands can produce colourless mana (C), which is only useful in generic costs. Since 2015, WOTC have printed some spells which require specifically colourless mana, but this is rare.

Lands which provide an ability outside mana production are called "utility lands," and are irrelevant to LandFill's calculations.

2.1.2 Land Balancing and Cycles

Within WOTC design principles, ff card A is better than card B in at least one way, and worse in no ways, A is considered "strictly better" than B [7]. Tropical Island, tapping for UG, is strictly better than both Basic Island and Basic Forest. Since early sets, however, WOTC have generally avoided printing land cards which are strictly better than basic land cards [21]. Virtually all land cards which produce more than one colour of mana are either "balanced" (given a downside), or produce mana via a more complex mechanism. Breeding Pool, for example, is identical to Tropical Island save that it enters already tapped (and thus unusable on the turn it is played) unless the player pays 2 life when playing it. Lands are typically printed in cycles, which share a common balancing mechanism but produce different colours. Stomping Ground, for example, has the same stipulation as Breeding Pool, but produces RG instead of UG. Cycles usually carry informal names within the community: Breeding Pool and Stomping Ground are both "Shock Lands." The prevalence of cycles such as Check and Fetch lands (see Figure 2), additionally means that a cycle with basic landtypes may be considered strictly better than an identical cycle without.

To illustrate the complexity this introduces to optimization, consider as an example the lands Prairie Stream and Deserted Beach, both referenced in Figure 2. Since any situation in which Prairie Stream would enter untapped would also allow Deserted Beach to enter untapped, but the same is not true vice-versa, Deserted Beach is, taken in isolation, a stronger land. Consider, however, the following situation (in which all land cards are depicted in Fig 2). Two players play two identical UW decks with identical UW manabases consting of Flooded Strand, Hallowed Fountain, and multiple Basic Plains and Basic Island cards. The only difference is that one includes a Prairie Stream, and the other includes a Deserted Beach.

Both decks draw the below opening hand:

$$\text{Hand} = \big[\text{Spell}(UWW) \quad \text{Spell}(UUW) \quad \text{Spell}(UU) \quad \text{Land(Basic Plains)} \quad \text{Land(Flooded Strand)}\big]$$

Since the player needs copious amounts of both U and W mana, it behoves them to use the Flooded Strand to search their library for a non-basic Plains or Island capable of producing UW. Since this hand contains no spells of CMC=1, there is no disadvantage to playing a tapped land on

Figure 2: UW and RG lands of different cycles. Left to right: "Battle", "Shock", "Check", "Fetch", "Filter" and "Slow" Lands. Notice the "hybrid" mana cost of the filter land's second ability, meaning that requires a mana of either of its colours to activate, and the diamond marker in the output of its first ability, meaning that said ability only produces colourless mana

their first turn. The Prairie Stream player may then go and fetch the Prairie Stream at no downside. However, the Deserted Beach player has to fetch the Hallowed Fountain, leaving Deserted Beach in their library. Consider, then, that when each player shuffles for the Flooded Strand's effect, the UW land remaining in their respective library (Deserted Beach for the Deserted Beach player, and Hallowed Fountain for the Prairie Stream player) is placed on top. Since the Hallowed Fountain can come in untapped for a trivial life point investment, the Prarie Stream player therefore has the option of playing the spell that costs UU, whereas the Deserted Beach player - able to play only a tapped Deserted Beach or a Basic Plains, which produces only W - cannot do so.

Therefore, the appropriateness of playing a Prairie Stream vs a Deserted Beach in a given manabase depends on, in addition to other considerations such as the presence of Check Lands:

- The probability of a player beginning a turn with N lands on the battlefield in which N is greater than 1 and at least two lands are basic.
- The probability of a player beginning a turn with N lands on the battlefield, a fetch land in hand, and no possible set of spells to play with combined CMC of $N + 1$

Optimization, therefore, may be thought of as a simultaneous two-part process:

- Replacing Basic Land cards with multicolour lands that improve P, until a point is reached at which the accumulated downsides of those lands start to outweigh the diminishing returns from that improved access...
- ...while choosing the multicolour lands whose downsides are the most significantly ameliorated by the specific spread of CMC and cost values of the spells in the deck, and by the interactions between those lands and other lands in the manabase.

This makes manabase generation a Combinatorial Optimization problem. If a deck requires a manabase of M land cards, and operates with a performance of P for any given manabase (assigning a single performance metric for a deck is complicated; see 6.3), the question is what combination of M cards from the set of all lands that produce one or more of the deck's colours maximizes the value of P.

There are therefore two problems to solve. The first is to find a method for determining P from a particular deck. The second is inherent to most Combinatorial Optimization problems: given that these typically deal with more candidate solutions than can be feasibly investigated seperately, it is necessary to find a method of optimization that will traverse only a relevant subsection of the total search space.

2.2 Simulation and Optimization

LandFill approaches this problem via the implementation of two algorithms: an internal algorithm, which simulates MTG games and assesses the performance of a given manabase, and an external one, which provides the internal one with a series of increasingly optimized decks to test. These will be referred to as the "Simulator" and the "Optimizer", and the broad structure of both, and the relationship between them, will be introduced below.

2.2.1 LandFill Simulation

In MTG parlance, "Goldfishing" refers to testing out a deck by taking repeated turns against no opponent, and assessing the performance of the cards in the absence of opposing disruption [16]. Simulating all possible spell card interactions is an unfeasible undertaking here, and not necessarily a helpful one: Churchill *et al* have demonstrated that, as it is possible to construct within an MTG game a Turing Machine whose halting is the necessary condition for a player's victory, even a deck able to model all spell interactions would not be able to interpret and execute any deck strategy[3]. However, since our concern is only with the ability of a manabase to deploy spell cards, there is a useful heuristic to fall back on: in a game, the winning player is typically the one who spends the most mana [12]. The simulator, therefore, need only try to spend as much mana as possible each turn. Broken down into constituent steps, the algorithm for a simulated game is as follows:

1. Draw an initial hand of 7 cards.
2. Mulligan as necessary.
3. Draw an additional card at the beginning of each turn.
4. Identify which playable land will allow the expenditure of M mana, where M is the maximum that may be spent that turn.
5. Play that land.
6. Play a combination of spells with total CMC C.
7. Repeat steps 3-6 for each turn of the simulated game.

Within our heuristics, we may safely choose a combination of spells at cost C at random, since we have no way of telling which would be relevant in any given game, and can only estimate

their value as an efficient use of mana. However complexity is introduced by situations in which multiple lands allow for the spending of C mana, such as in the sample hand drawn in the previous section illustrating the appropriateness of Slow Lands vs Battle Lands: in the absence of any spells of CMC=1, playing either the Basic Plains or the Flooded Strand yelds $C = 0$. The simulator, therefore, must have some capability of assessing which lands maximize expenditure on future turns.

2.2.2 LandFill Optimization

In a Monte Carlo search, solution space is explored by taking the random outputs of stochastic processes, and sampling the resulting distribution to approximate the typical values of that process [15]. Assuming correct function of the aforementioned simulator, a Monte Carlo assessement of a manabase would be conducted by giving it a deck, running a large number of games with it, and recording the deck's average performance. The performance of a deck fed into a large number of games becomes P the objective function for the combinatorial optimization problem outlined at the end of 2.1.2.

LandFill's search space is L-dimensional, where L is the number of lands required by the deck, in which each dimension represents the quantity of a given land in the manabase (limited, in the Commander format, to values 0 or 1 for all non-basic lands). LandFill uses a variant of Hill Climbing optimization, also known as Neighbourhood Search. In Hill Climbing optimization, once an initial solution $\underline{x}_c$ is determined, all solutions in its neighbourhood $N(\underline{x}_c)$ - ie, all solutions which differ from by some simple transformation $\underline{x}_c$ - are examined. The first solution to return a higher value is adopted as the new $\underline{x}_c$ [23]. In the context of manabase optimization, the iterations are as follows:

1. Generate an initial manabase, $\underline{x}_c$, for the input deck.
2. Conduct many simulations and determine the sample average performance, $F(\underline{x}_c)$, of the deck. By the law of large numbers, this can be treated as an approximation for the true average performance.
3. Exchange a land in the deck for a different one, creating a new manabase, $\underline{x}_t$.
4. If $F(\underline{x}_t) > F(\underline{x}_c)$, adopt $\underline{x}_t$ as the new value of $\underline{x}_c$, and return to step 2.
5. If not, return the original land to the manabase and return to step 2, this time making a different, previously unexplored substitution.
6. If all lands in the deck have been systematically replaced with every candidate land that could replace them, and no value has exceeded $F(\underline{x}_c)$, return $\underline{x}_c$ as an optimized manabase.

Since Hill Climbing is a "greedy" optimization algorithm, it searches for *local* rather than *global* maxima. There exist viable optimization algorithms, such as Simulated Annealing, which are less susceptible to this[23]. However, given that the simulator itself can only loosely approximate the decision-making process of an actual MTG Game, LandFill falls into the category of optimization problem for which "the optimal solution", as described by Zanakis and Evans, "[is] 5" [28]. To be a valuable product, LandFill needs only to be able to create a more reliable manabase than a human player could in a comparable length of time.

2.3 Development and Testing Methodologies

In software development, "Verification" testing tests code functionality according to designer specifications, while "Validation" testing tests whether the code meets user needs [24]. The schedule for each of these is detailed below

2.3.1 Verification Testing

My schedule for verification testing was couched within an "Iterative" development process, whereby each feature was independently designed and tested before being added to the core product[2]. The Simulator and Optimizer are two of four component subsystems to LandFill, arranged like so:

1. The *Database* - stores information about MTG cards.
2. The *Simulator* - simulates games using information from the database.
3. The *Optimizer* - assessess decks using performance data from the simulator.
4. The *Interface* - allows for use of the optimizer by a lay customer.

Since each component utilizes the one before it,, LandFill naturally lends itself to an Iterative Development process. The classic alternative of "Waterfall Development", in which all testing is withheld until product completion [2], would in this case forbid me from factoring in shortcomings of a given component into the design of the component superceding it if those shortcomings were not discovered until after development. Moreover, iterative development puts LandFill in good stead for its anticipated lifecycle. Since WOTC regularly print new cards, and each new mechanically distinct land must be individually coded, LandFill will always need to be able to accept new additions to its codebase.

Since verification testing is to be conducted on each component seperately, testing approaches for each component will be outlined in the section of this thesis that deal with that component.

2.3.2 Validation Testing

Since LandFill's viability ultimately rests on its comparison to a human deckbuilder and not in its ability to find a consistent global optimum, it must:

- Be flexible enough to fit easily into a range of different deck construction strategies.
- Accommodate, via user input, game-extrinsic manabase restrictions such as price, availabilty and personal preference.

To ensure this, in addition to validation testing conducted after development of the initial product, I conducted user research prior to development, to ensure that user needs were accounted for throughout the design process. User research consisted of the following:

- A series of Semi Structured Interviews, which gauge the way prospective users create MTG decks and the manabases thereof.
- A series of Think-Aloud evaluations of a mockup, in order to observe the patterns a user falls into when using an app to this purpose.

- A Kano Questionnaire, the questions of which are based on results of the previous evaluations, in order to guide development priorities.

Validation tests, conducted after the development, consisted of the following steps.

- Two seperate deckbuilding exercises, one of which allowed use of LandFill, and one of which did not.
- Two Task-Load Index (TLX) questionnaires, to determine the relative difficulty of each.
- A semi-structured interview to assess the testee's opinion of the software, and their suggestions for future developments.

2.4 Relevance to Existing Material

LandFill engages with three areas of prior research:

- Academic interest in MTG automation.
- Use of computer models in manabase analysis within the MTG player community.
- Existing deckbuilding apps.

I will outline its engagement with these areas below.

2.4.1 Academic Interest in MTG Automation

Much modern academic interest in MTG, from a computer science perspective, rests on Ward and Cowling's landmark 2009 paper, "Monte Carlo Search Applied to Card Selection in Magic: The Gathering". Ward and Cowling hypothesize argue that methods used to automate other imperfect information games, such as Bridge and Poker, may be applied to MTG [25][6][1]. A decade hence, interest in this problem has resurged thanks to the rollout of WOTC's virtual MTG venue, MTG:Arena, which, although primarily a player-vs-player (PvP) engine, features an AI opponent, nicknamed Sparky, who, although useful in gameplay tutorials, presents no challenge to experienced players[1].

Ward and Cowling point to the inherent difficulty of automating games of imperfect information: strategic thinking in MTG is confounded by the unknowability of both the opponent's hand and the next card to draw. Since LandFill's goal is simply to spend as much mana as possible each turn, these concerns are both rendered irrelevant: the hand is simply a set of resources to be allocated as efficiently as possible. This puts it somewhat outside the realm of scholarship inaugurated by Ward and Cowling, which is chiefely concerned with how to encode strategy given the complexity of gameplay and the limited information. Drawing on methods used in other imperfect information games, such as Go and Poker, Ward and Cowling's automated players use Monte Carlo analysis to examine possible outcomes of combat between creature spells once cast [25], while Alvin *et al* explore graphical representation of card synergies [1]; in both cases, effective use of lands to play spells is trivial, as the deciding factor in which spell to cast is based on the exigencies of the game state. Indeed, Esche's 2018 research into optimal MTG strategy eschews casting any multicolour spells, testing his virtual player only on a mono-red deck [6]. LandFill's simulator represents a

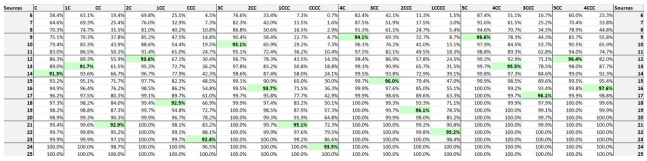

Figure 3: Frank Karsten's recommendations for how many lands a deck should include capable of producing colour C; he recommends adopting the highest Y-axis score corresponding to a card whose mana cost appears in your deck [12]

small engagement with this line of research, as it offers a fast method of determining optimal mana usage.

Automation of deckbuilding, rather than of play, is more relevant to LandFill but less well studied. Sverre Johann Bjørke and Knut Aron Fludel explore the use of a genetic algorithm to generate decks out of a set card pool (as is done by human players in the "Limited" format); however, their results are inconclusive. As their work relied on a full automated AI player - which, as established, still underperform compared to human players - their generated decks only performed well against other decks also played by the same automated player, and faltered against human opponents. LandFill represents a narrowing of the ambitions of Bjørke and Fludel's work. From a purer mathematics perspective, Riccardo Fazio and Salvatore Iacono have conducted some research into how to quantify the mana requirements of a deck; however, their work is limited only to assessing the quantity of each colour required, and not how these quantities are spread across mechanically distinct lands.

2.4.2 Manabase Analysis within the Player Community

Guides exist on how to write basic scripts in order to use Monte Carlo techniques to analyse a given deck [27]. A central figure in this practice is Frank Karsten, who has written extensively on how to determine a manabase for a multicoloured deck. His seminal series of articles, *How Many Sources Do You Need to Consistently Cast Your Spells?*, use Monte Carlo search to produce a grid (see Figure 3) outlining how many lands of colour C you need depending on the most demanding spell requiring colour C in your deck. Rather than simulate gameplay, Karsten draws random hands for each possible spell (abstracted to just its mana cost), to determine the number of lands of the spell's colour that would need to be included in the deck in order to reliably cast it on the turn when the $M'th$ land is played, where M is the spell's CMC.

While this article engages broadly with the issue of how many multicolour lands are necessary in a deck before they begin to incur diminishing returns, it cannot assess when these diminishing returns become a greater handicap than expanded colour access. To understand whether that land will be able to use its full productive capacities when played as the M'th land requires an understanding not just of the coloured lands drawn previously, but the behaviour of these lands in a state of play. This is where I believe the implementation of a stripped down play simulator will pay dividends.

This approach has been used by Karsten elsewhere, to analyse manabase choices in the context of the Limited format (which features small decks and a heavily restricted set of lands to choose from)[11] and the Standard format around the release of the Ixalan Set[10]. In the latter article, Karsten's engagement is limited to one nonbasic land cycle - Check Lands - and only to their

Figure 4: A screenshot of a deck under partial construction in Arena, with the decklist on the right-hand side column. Basic Plains and Basic Islands are automatically added as white and blue spells are.

probability of entering tapped, not the probability that their entering tapped incurs a gameplay downside in the context of a given deck. In the former article, Karsten engages explicitly with the question of at what point the disadvantages of dual lands outweigh the advantages of greater colour access, making this work a natural precursor to LandFill. Karsten's simulator is likely to converge on a more reliable performance estimate for a given manabase, running orders of magnitude more times than LandFill does. However, in addition to lacking deck construction functionality, its simplified game simulation model only uses one type of nonbasic land, and utilizes a much more straightforward decision making process. Ultimately, this restricts this analysis into simply a matter of deck-building best-practices. LandFill, therefore, may be seen as an attempt to adapt Karsten's methodology for this area of his research into into a widely applicable deckbuilding tool in the vein of his grid.

2.4.3 Existing Deckbuilding Apps

I have identified three pieces of software which support the same phase of deck-building as LandFill, and will examine their functionality below.

The first is MTG: Arena (screenshot in Figure 4), which is capable of automatically filling a deck with an appropriate proportion of each basic land in accordance with the deck's colours. While this is relevant for the Standard and Limited formats largely played on Arena, the deckbuilding restrictions of which only offer few nonbasic lands, it is less appropriate for other formats, in which a sizeable proportion of a manabase will be nonbasic; it is also only accessable to decks constructable via the limited set of cards available on MTG Arena.

The second is the website ManaGathering (screenshot in Figure 5), a database of nonbasic lands sorted by colour. Players input the colours of their deck and are given a list of nonbasics within those colours, sorted by cycle. Although ManaGathering has no optimization or manabase generation facility, it fulfills a similar role in the deck building process as LandFill, facilitating the choice of generically strong mana-producing lands once utility lands or lands fulfilling a niche strategic concern of the deck have been chosen. A broad success metric for LandFill is that it should return a better result than a player using ManaGathering could in a comparable time.

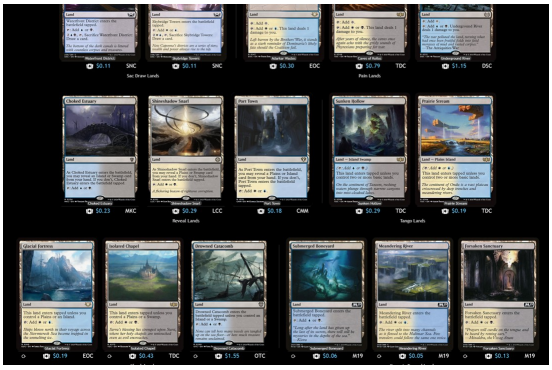

Figure 5: A zoomed-out screenshot of an excerpt from the ManaGathering page for a WUB deck. Note the WU, UB and BW lands arranged by cycle.

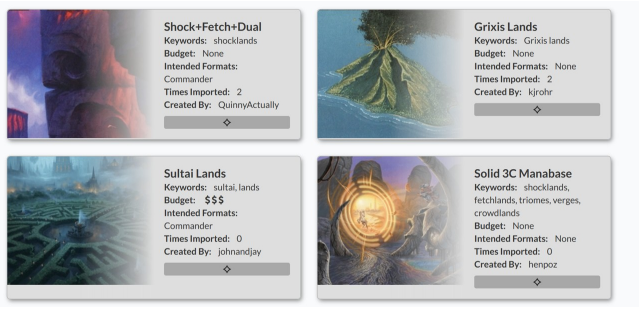

Figure 6: A screenshot of community created land packages on Archidekt, categorized by, among other things, price and color. These can be imported into a decklist.

Finally, the website Archidekt (screenshot in Figure 6) combines a basic-land allocator *a la* MTG: Arena with a communal "package" system. Players create "packages" of lands which are saved publically on the site, and may be imported by other players. For example, a player may get a list of colour-appropriate lands of a reasonable price by importing a given manabase package and then automatically generating a list of basic lands. The question of whether LandFill or ArchiDekt are capable of producing more reliable manabases is unlikely to be answered in the timeframe of this investigation, but LandFill represents an alternative approach; morever, as Archidekt's functionality is limited to decks that are stored within its database, it offers a more flexible service to deckbuilders who may prefer other collection-tracking databases such as TappedOut or Moxfield.

It is worth noting that while Archidekt is the only card database to offer this feature, it is not the only card database. Others include Moxfield, TappedOut and DeckBox, all of which allow the player to upload a decklist to be stored and visually displayed. Although providing very different value to LandFill, they will be referenced throughout this writeup, as, as will be covered in 3, decklists both entered into and extracted from LandFill will often likely be moving to and from such databases.

2.5 Library/Language Choices

My choices of language and libraries were informed by two main priorities. Due to my short turnaround time, it was important that I use libraries and langauges with substantial community support for web development. Meanwhile, as a usable app, LandFill benefits from high performance so as to maximize the number of simulations it can run, but does not need to offer a complex user interface nor store user data, prompting me to favour high-performance tools over complex and scalable ones.

In places where these requirements are at odds, I prioritized the former: my choice of Python as a backend and Javascript as a frontend was driven largely by the popularity of these languages in web design. However, in other decisions, the two requirements informed each other constructively. I chose Flask as a backend web framework as its simplicity made it both easy to learn and reputably faster; contenders like Django are made both slower and more complex due to their abundance of features (FastAPI, potentially lighter and faster than Flask, was discarded due to its smaller userbase and thus relative paucity of learning resources). React, which I chose as my frontend framework, similarly sports a wealth of support resources, and features a Virtual DOM that lowers performance overheads when users make small input changes - a relevant concept here, as users will likely order several simulations with small preferential changes on each one.

The use of Object Relational Mappers (ORMs) is common in web design, usually as a component of a CRUD (Create, Read, Delete, Update) interface taking user data. Although LandFill makes heavy use of Object Relational Mapping to store a database of MTG cards, the user needs to only read the database. For this reason, SQLAlchemy, an ORM esteemed for rapid performance at the cost of easy data amendment**??**, was the obvious choice.

2.6 LandFill Structure

With both the inherent challenges of manabase optimization and the set of potential user needs thus established, LandFill will be built as four interacting components, outlined below.

- The Database - LandFill's database of MTG card objects, stored in the backend as mtg.db.
- The Simulator - see 2.2.1
- The Optimizer - see 2.2.2
- The Web Interface - a user-friendly web interface designed according to the priorities set out in 3

3 Pre-Development User Research

3.1 Overview

I began development by holding one-on-one user research sessions with eight MTG players. The sessions were divided into two parts, a Semi-Structured Interview and a Think-Aloud Mockup Test, with an additional Kano Questionnaire circulated to users afterwards.

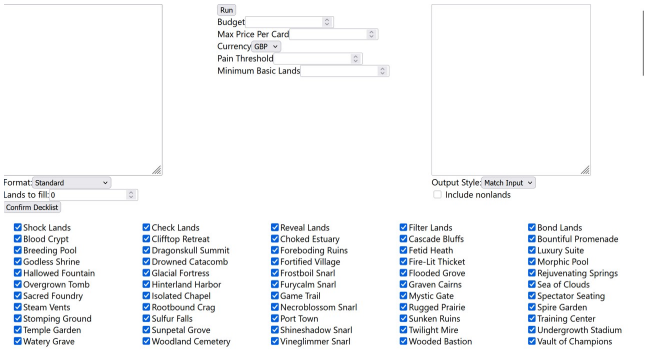

Figure 7: Draft front-end used in the mockup testing.

3.1.1 Semi-Structured Interview

In addition to specific feedback on possible features, I was interested in understanding more about how players typically create manabases. I therefore chose to lead with a Semi-Structured Interview. This is a data-gathering method in which the interviewer stays flexible on how and in what sequence questions are asked, to allow unexpected themes and topics to emerge **??**. My questions were as follows:

- How do you approach selecting lands for a deck, and how does this vary across formats you play?
- How do you approach acquiring lands for a deck (eg, do you assemble a list of cards to purchase, do you assemble a list of cards you already have - and if so, do you have a good knowledge of what lands you own)?
- What role do existing deckbuilding support apps, such as Moxfield and TappedOut, play in your process?
- Do you factor the strategy of your deck into land choices in terms of pure mana production (ie, not including utility lands).
- How do you mulligan? How does this vary across formats that you play?

3.1.2 Think-Aloud Mockup Testing

In a "Think-Aloud Evaluation", users are asked to narrate aloud their thoughts and opinions while attempting to use a system [26]. This is an appropriate method for early development since it can be conducted on a "mockup", an aesthetically versimillitudinous but non-functional representation of the planned interface. Users are occasionally prompted for input, and may be given solutions to problems if necessary, but are largely expected to use the product unassisted. The LandFill mockup presented to users is displayed in Figure 7.

3.1.3 Kano Questionnaire

Kano Analysis is an approach to user evaluation that examines the emotional response of a prospective user to the presence or absence of a given feature. I developed a Kano Questionnaire after analysis of the Interview and Think-Aloud data, so as to prioritize which features, suggested by individual testers, were reflective of more widespread demand. A generic Kano template is displayed in Figure 8. My questionnaire listed the following proposed features:

- The option to exclude from consideration all lands which always enter tapped.
- The option to exclude from consideration all lands above a certain price.
- The option to exclude any individual land or cycle from consideration via the player's own preference.
- The option to mark some lands as mandatory for LandFill to include.
- The option to "weight" lands, so that LandFill prioritizes a player's preferred cycles in its evaluation.
- The option to input a list of lands as well as a list of nonlands and have LandFill choose the best of these, rather than explore all possible lands.
- The option to tell LandFill not to recommend "Off-colour Fetches". For example, a Fetch Land that can search for an Island or Plains can still fetch Islands in a UR deck. This was considered distasteful by one test subject.
- The abiilty to see an image and description of any suggested land/cycle.
- The ability to view performance metrics for the deck after manabase generation.
- A FAQ explaining how LandFill determines lands.
- The ability to specify how much life a player is comfortable to lose to land cycles that require a life point investment, ie, Shock Lands.
- The ability to view low-performing lands in a given deckilst, to inform a player's choice of supplemental mana generation spells, or what to replace on the release of new cycles.
- The ability to generate manabases for, respectively, the Commander, Modern, Legacy, Pauper and Limited formats.
- The ability to copy a decklist into and out of LandFill with minimal reformatting from, respectively, the following database apps: TappedOut, Archidekt, Moxfield, DeckBox.

3.2 Emerging Themes

While initial user-testing yielded multiple small design considerations, and will be cited as various development decisions are outlined throughout this writeup, key themes are outlined below.

Figure 8: A question in a generic Kano questionnaire

3.2.1 Commander/Casual Preference

Commander, Limited and Pauper were the three most popular formats among interviewees. Of these three, a plurality of interviewees acknowledged that Commander would be the only format for which they would consider using LandFill. The other two were disqualified by users in general on the following grounds:

- Limited - in this format, cards are assigned to a player at random immediately before a game. Decklists are therefore not generally uploaded to any databases, and inputting all cards into LandFill, especially on a mobile device, would be impractical.
- Pauper - this format restricts decks to only common and cheap cards. In Pauper, use of utility lands, or lands with specific synergistic qualities, is so normalized that manabase strategies prioritize this over consistent mana generation.

While fewer interviewees had much experience with more competitive formats such as Standard, Legacy and Modern, several interviewees raised doubts as to the ability of any spontaneous manabase generator to gain traction within those scenes, as players in those formats habitually use existing deck archetypes, and attach to them manabases with proven competitive credentials. In responses to the Kano Questionnaire, most respondents expressed that they would like/be neutral on support for non-Commander formats, but only one respondent identified any format other than Commander as a minimum expectation.

I therefore decided that I would restrict LandFill to being solely a Commander product for initial development. Commander's deckbuilding restrictions are markedly different from other formats, and will be discussed at length in my outline of the Simulator (see **??**). I therefore deemed it more important to make a product that worked seamlessly for Commander decks, rather than one that attempted to accommodate potential decks across a wider range of formats.

3.2.2 Lack of Strategic Thinking but Strong Preferences

When asked about how the strategy of a given deck informed their choice of lands, nearly all test subjects said that it did so, but almost solely in regards to the ancillary effects of certain cycles – ie, Gain Lands (tapped dual lands that increase the player's life point total by 1 on entry) being popular in decks that trigger from gaining life. Only one said that they would actively choose slower but more colour-diverse lands for slower decks.

However, when presented with the Mockup outlined in 7, all users swiftly devoted themselves to using the tickboxes to remove lands or cycles that they did not want to have included in the deck, despite having been told to imagine this mockup returning a fully optimized list of lands. Several

users also expressed a desire for a weighting functionality, so that they may specify lands which they prefer.

Some testees also expressed a dislike of certain categories of lands or cycles, ie, any cycle of dual land that only ever enters the battlefield tapped, or any off-colour Fetch Land (see the relevant listed feature in **??**).

The lack of strategic decision-making in manabase assembly validates LandFill's use-case, as it suggests that, by testing lands against the needs of the deck, LandFill is putting more consideration into land selection than a typical player. However, it is clear that player input needs to include both:

- A positive component, in which players can force inclusion of potentially suboptimal lands such as Gain Lands due to deck strategy.
- A negative component, in which players are given extensive leeway to remove lands from consideration.

3.2.3 Deckbuilder Personas

Testees broadly designed manabases in one of two ways, embodied in the below personas:

- Persona A, who possesses a large collection of land cards and, on creation of a new deck, selects lands in the approrpiate colours from this collection.
- Persona B, who develops a new deck and chooses lands based on abstract preference, and then orders those lands.

This prompted me to ask testees which of the following two models of LandFill they would prefer:

- Model 1 - LandFill, in addition to taking a decklist of nonland cards, also takes a list of land cards that the player might have found in their collection, and returns the optimum subsection of these.
- Model 2 - LandFill takes a decklist of nonland cards and asks the user only to specify what lands they do not want, generating an optimized list from the remaining options.

I expected a preference towards Model 1 to be strongly associated with Persona A deckbuilders and vice versa, but to my surprise, a majority of deckbuilders across the personae preferred Model 2. Several Persona A deckbuiders preferred it because they were enthused by the prospect of being recommended lands they had not heard of before.

While this was a majority consensus, and all Kano respondents said they would be able tolerate the absence of Model 1 functionality, some users did express a desire for Model 1 functionality, making it a viable route for future development of LandFill. However, the initial design covered in this writeup will adhere to Model 2.

3.2.4 Flexible Input Parsing

All users polled made use of at least one online database (a plurality used TappedOut, with Moxfield and DeckBox also being popular). In the Think-Aloud evaluation, they copy pasted decklists from these sources into LandFill, and said they would return the list of outputted lands there. Interestingly, many copied not from the inbuilt export feature of these sites, but instead by simply copying the decklist as it is displayed on the page.

This suggests LandFill would benefit from a flexible input parsing device. LandFill should be able to accept a decklist pasted in from both the export features and front pages of all these databases, and format its outputs so as to be input into these services. Moreover, TappedOut and Archidekt allows for categorization of cards into custom types, which are included in the decklist when copied. The parser, therefore, must be able to recognise these custom types and parse them not as cards but as keys to a dict object which contains the list of cards corresponding to that category, and then display accordingly on output. This avoids players losing their custom categories when they utilize LandFill.

3.2.5 Mulligans

No testee was able to describe any consistent principles on which they based their decision to mulligan. All testees said the decision would be based not just on the castability of spells in the hand, but also the strategy enactable via those cards. It is not, therefore, my priority to provide a means to replicate a given LandFill user's mulligan preferences. For the initial design, the mulligan heuristic will be built into the simulator.

3.2.6 Life Loss, Cost, and the Knapsack Problem

Because user testing was conducted before the development of the Optimizer, I suggested two proposed features which provied unimplementable:

- Total manabase cost.
- The maximum amount of life a player was happy to lose to lands, such as Pain Lands and Shock Lands, which require a life point investment (this value is set via the "Pain Threshold" input on 7; that testees unanimously found this wording confusing is ultimately irrelevant, as the feature was not included).

These are theoretically easy to determine for any decklist. Manabase cost can be determined from the sum price of each card, data which can be easily included in the Database. While tracking life expenditure would be more difficult, as it would require the Simulator to try and minimize life point expenditure in a situation where there are multiple ways to spend the same amount of mana, it would also be possible to return the average life point penalty incurred over a Monte Carlo search along with the general performance metric. However, these are both examples of the 0-1 Knapsack Problem, itself a NP-hard combinatorial optimization problem, in which a subsection of weighted items must be chosen from a larger set that minimize the total weight (with weight, in this case, representing respective average life damage and card cost)[14]. Since this would involve simultaneously optimizing multiple areas of deck design, it would represent a sizeable increase in computational complexity for LandFill.

Surprisingly, subjects were broadly ambivalent on both of these features. During the Think Aloud evaluation, no subjects attempted to make use of either. Kano data introduces some more nuance. The ability to specify an upper limit of life loss was something all but one respondant would consider desirable, although none considered it a minimum requirement, nor its absence to be something they would "dislike". The ability to specify price limits performed similarly, save that one respondent did indeed consider it a minimum requirement. This suggests that, while it is not worthwhile to simultaneously attempt to optimize these metrics while optimizing mana production, these features are not useless, and it is worth finding some approximate representation of them in design. My approach to the issue of life loss is covered in 6.1.2. My approach to price consideration is covered in 7.2.2.

4 The Database

4.1 Database Requirements

The database in which LandFill stores card objects will need regular updating by the site owner. This is for two reasons:

- WOTC regularly releaseses new sets of MTG cards.
- Since LandFill will factor price consideration into its choices, the database must reflect the up-to-date price of a given card.

In a given session, the database will be queried at two points:

- When a user's decklist is added, to determine the mana costs of nonland cards.
- To identify a list of candidate lands for the manabase.

This creates two constraints for the database: it must be up-to-date, and have reasonable response times over multiple queries for individual objects. An early prototype used the Scrython API (see 4.2) to fetch requested cards from an existing online MTG database. While this guaranteed up-to-date information, it more than ten minutes to retrieve a list of 50 nonland cards due to a combination of connection latency and the need to avoid overwhelming the server.

To accommodate both constraints, the database must store information locally, but be easily updatable.

4.2 Choice of Online Database - ScryFall/Scrython

One popular source of MTG card data for developers is mtg.json, a json reprsentation of all cards. Mtg.json is popular for the breadth of information it contains, including card reprints, making it useful for development of shopfronts or collection management software [18]. Since LandFill needs to know only the gameplay-relevant attributes of a card and its price, I favoured a more streamlined source. I ultimately chose to use the database "Scryfall", which is predomenantly used by players rather than programmers; I felt that this wider userbase would ensure up-to-date information. ScryFall offers an API for code integration, for which a python library, "Scrython" has already been developed. Scrython accepts card queries using the format of ScryFall's advanced search option, and returns a list of dict objects listing card attributes, referred to within LandFill's

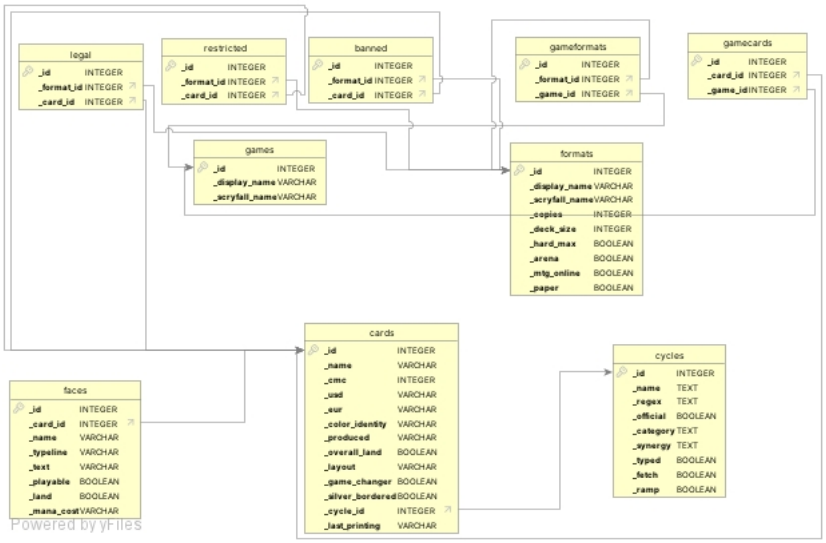

Figure 9: Layout of mtg.db

code and this writeup as "SCOs" (Scrython Card Objects). LandFill's database, thererefore, is a translation of these SCOs to SQLAlchemy's ORM format. Allowing for small pauses to prevent server overuse, downloading all 30,000 cards (at time of writing) takes around one hour, and could easily be run weekly, or in response to new sets.

In addition to ease of access ScryFall has the advantage of listing the colours produced by a given land card, eliminating the need to parse this from the card text. Its most significant shortcomings pertain to price data. First, it only offers a price in EUR and USD and not GBP, and it fails to list the price for a small minority of cards. As a simple fix for this, LandFill uses the CurrencyConverter library to determine the price in GBP from the price in EUR. It also fails to list the price for some cards. A future version of LandFill could potentially scrape an alternate database on download if a price is not found on ScryFall, but as this only impacts 10 land cards that have, at time of writing, been encoded into LandFill, for intial development I have simply entered the prices manually.

4.3 Database Layout

Since LandFill uses Flask, it is built not on SQLAlchemy but on the slightly more feature-heavy Flask-SQLAlchemy extension. This means that each mapped class extends Flask-SQLAlchemy's `model` ancestor class, rather than the `declarative-base` class used in classic SQLAlchemy. The relational mapping between models in mtg.db is shown in Fig. 9.

The two upper rows of tables in Fig. 9 map the many-to-many relationships which denote the availability of a card across formats and "games" (eg, physical MTG games, MTG: Arena); one card is legal in many formats, and one format has many legal cards. Since the initial deployment of LandFill focusses on the Commander format, this is largely irrelevant, but remains in the schema for use in future expansion. Discussion henceforth will focus solely on the models Card (table: cards), Cycle (table: cycles) and Face (table: faces).

Although not every attribute of a SCO is embodied in a model, in this initial development stage, LandFill errs in favour of storing too much data rather than too little, meaning that some card attributes, including `Card._silver_bordered` and `Card._last_printing`, are not relevant to this writeup. I will define attributes throughout this writeup as they become relevant.

4.4 Database Management

I developed a class, DatabaseManager, which contains simple methods that bulk-transfer card information from Scryfall to the Database via Scrython. Within my codebase, a single instance of DatabaseManager is created in one script, `manage_database.py`. Simply running this script will fully update the database with all existing cards. Pauses are built into the script using Python's `time` module to avoid overburdening ScryFall's servers.

To facilitate debugging during development, ScryFall is never queried for `all` cards. Instead, a for-loop is used to systematically acquire cards in order of mana cost, and add them to the database only after all have been downloaded. This means that, when errors occurred during download, I can resume the download from midway through the loop rather than starting from the beginning.

4.5 Representing Multiple-Faced Cards

Some cards have two faces, each of which may be considered two different cards with their own text and mana cost. Although most cards have only one face, I sought to keep the database in 1st Normal Form, a database normalization standard which stipulates each column in a databae contains only a single value per entry [17]. I therefore treat Faces as having a many-to-one relationship with cards. Broadly, the Face object has attributes used to determine its interaction with a given game state: ie, its casting cost and potential basic landtypes (stored in the `faces._typeline` attribute), while the Card object holds attributes relating to the viability of a card within a deck and a given user's preferences (IE, what colours of mana it produces, the cost of the card).

The rules on how to play a card's different face varies between cards: some maybe played with either face, while others have a single playable face and swap to the other under certain game conditions. This is determined by the `cards._layout` attribute. A card's layout is one of several strings extracted from the SCO, each of which denotes a different way of formatting face relationships (ie, cards with the "transform" layout have one playable face; cards with the "adventure" and "mdfc" layout have two). The playabiltiy of a given face is stored in the `faces._playable attribute.`

Some cards are a spell card on one side and a land card on another. A card is considered a land if it has at least one playable face that is a land, represented by the `card._overall_land` attribute.

4.6 Cycles

Recall from Fig. 2 land cards within the same cycle are mechanically identical but refer to different colours in their text. A card's memebership in a given cycle is, unfortunately, not included in a SCO. For this reason, each cycle has a string attribute, `cycle._regex`, consisting of a Regular Expression. If DatabaseManager matches the regex of a cycle to a card, that card is connected to that cycle via Foreign Key. Since some cycles are mechanically identical to each other but are distinguished by the presence of basic land types or whether the constituent cards have the "snow" card type, these are both also stored in the cycle object as Boolean values, to sort cards whose text matches multiple cycles.

Although some spell cards are arranged into cycles - mechanically similar cards where each costs a different colour or set thereof - this is irrelevant to LandFill, and the cycles table therefore stores only land cycles.

4.7 Database Verification Testing

A secondary script, `test_db.py`, conducts both Unit Tests and Property Tests on the database.

In Unit Testing, the output of code is compared to a pre-computed result [20]. In this case, Unit Tests check that there are an expected number of total cards and an expected number of total lands in each cycle. Unit testing cycles is important in ongoing maintenance, to ensure that any new regex patterns added for new cycles does not accidentally capture existing ones.

In Property Testing, random inputs are generated for code, and outputs are checked to ensure they fall within broad parameters [20]. In this case, the Property tests randomly selects a sample of database entries and checks that none have more than two faces, and that none have two faces with the same text. This is more expedient than testing all cards, and allows reasonable confidence that the database is populated as expected.

5 The Simulator

5.1 The Commander Format

In the Commander format, deckbuilders choose a creature to be the "Commander" of the deck. Rather than being included in the deck, the Commander is placed in the "Command Zone", allowing them to be reliably cast in every game. Some cards are marked to allow a "Partner", a secondary commander also placed in the command zone. Because of this, Commander decks generally include cards that work synergystically with the commander in gameplay.

Analysis by the Command Zone podcast suggests that the average Commander game runs for 10 turns per player [9].

Commander games utilize the "London" mulligan rule, with a single "free" mulligan. This means that at the beginning of a game, a player must put N cards on the bottom of their library, where N is the number of mulligans they have taken after the initial free one [5].

5.2 Classes

5.2.1 GameCard and subclasses

During simulation, a land must exhibit the unique behaviour of its cycle. LandFill, in order to execute the method overriding required here, requires a class hierarchy beginning with a general Card object, of which Spells and Lands are behaviourally distinct child classes, and each cycle represents a further child class of Land.

SQLAlchemy does support Single Table Inheritance, in which subclasses with distinct methods are stored in the ancestor class's shared table; it is also possible to temporarily store game state information via cached properties without updating the database. However, in practice, it proved simpler to create a new abstract class, GameCard, for reprsentation of the Card object in game. The initiation method of each GameCard takes a Card ORM object as an argument, and extracts relevant data from it. This removes any risk of persisting data in between games. It also allows me to create SubLands, a descendant class of Land that, rather than deriving any of its attributes


```
def live_prod(self, game):  LeahFrancesca
    active = False
    for needed in self.needed_types:
        if needed in game.battlefield.landtypes_list():
            active = True
            break
    if not active:
        return [self.primary]
    else:
        return [self.primary, self.secondary]
```

Figure 10: Examle of a land belonging to the Verge cycle and the implementation of `land.live_prod(game)` in that cycle

from information stored in the Database, can be assigned manually to produce any colour of mana when it is instantiated. This allows me to model objects that behave like Lands in simulation but do not represent real land cards, and assign them as properties of other Land objects, which is useful when modelling more complex lands. This will be detailed in 5.7.

- `GameCard.mandatory` - if `True`, this card must be included in the deck.
- `GameCard.permitted` - if `False`, this card, although retrieved from the Database for player approval, was considered undesirable by the player and should not be tested or included.

The simulation makes heavy use of two methods belonging to the Land subclass, both of which take a Game (see 5.2.3) as an argument:

- `land.live_prod(game)` - given the current game state, returns the colours of mana a land can produce, expressed as a list of strings (ie, [”U”, ”W”] for a UW land.)
- `land.enters_untapped(game)` - returns True if the land would enter untapped given the current game state, and False if not.

For examples of how these are overridden to reflect mechanically distinct cycles, see Fig. 10 and Fig. 11. When a Card is returned from the ORM, the relevant GameCard subclass for it is determined via a match statement taking its attribute `Card.cycle` as an argument.

A Spell object stores its mana cost as a list of objects corresponding to the differing costs of its faces. Each cost is a dict object relating a pip to a quantity. A Spell with one face that costs WUU2 would have `Spell.cost = [{"W":1, "U":2, "B": 0, "R": 0, "G": 0, "C": 0, "Gen": 2}]`.

5.2.2 CardCollection and Subclasses

CardCollection is an abstract class representing any Zone into which a GameCard can be moved such that it is never in multiple CardCollections. CardCollection subclasses are as follows:

- Deck.
- Hand - a player’s hand drawn at the start of each game, from which cards can be played.

Figure 11: Examle of a land belonging to the Check cycle and the implementation of `land.enters_untapped(game)` in that cycle

- Battlefield - a zone into which cards are played from the hand during a game.
- Graveyard - a zone containing cards in a game that are no longer in play.
- MonteCarlo - this object tests different decks and runs the Hill Climbing algorithm. It is modelled as a CardCollection in that it may be thought of as a virtual "player", who swaps their cards into and out of a deck before each game.

CardCollections contain a list attribute, `CardCollection.card_list`, and the specialized `CardCollection.give(GameCardCollection)`, which removes a GameCard from a CardCollection's card list and adds it to the card list of another, ensuring a card is never in two zones at once.

5.2.3 Simulation and Subclasses

The Simulation abstract class covers objects that perform actions with a specific deck, and thus take a Deck as an attribute, and feature an overwritten method, `Simulation.run()`, which performs a simulation and assigns data from that simulation as attributes of the object.

- Game - simulates a game of MTG with its deck.
- Trial - creates many Game objects using its deck and runs them.

5.2.4 "Lump"

Consider the below game state:

$$\text{Battlefield} = \big[\text{Land(Basic Forest)} \quad \text{Land(Basic Forest)} \quad \text{Land(Basic Island)}\big]$$

$$\text{Hand} = \big[\text{Spell}(G) \quad \text{Spell}(GG)\big]$$

Although any individual spell in the player's hand is castable, including the largest spell, the current selection of lands is suboptimal, as if the player had access to three green mana, they could cast two spells this turn. For this reason, rather than assessing the castability of spells, LandFill assesses the castability of "lumps". A Lump is an aggregation of Spell objects, which unlike in a CardCollection are not remved from any other lists when assigned. On a given turn, a Game will

have a Hand, a CardCollection, and a series of Lumps, which represent different permutations of cards in that hand. `Lump.cost` is formatted similarly to `Spell.cost`, and returns the combined cost of all the Spells making it up. A Lump only stores one face of a spell; the face to be included is specified at the Lump's instantiation. `Lump.cmc` returns the sum of the CMC of all its constituent cards.

5.3 Initiating a Game

At the start of each game, the deck is shuffled. The first seven cards from the Deck are moved to the Hand.

As covered in 3.2.5, it is not practical to implement realistic mulligan behaviour, especially since adherence to the London Mulligan approach would require strategic decision making about what cards to put on the bottom. LandFill therefore mulligans at most one time (the "Free" mulligan), and only if their initial hand contains fewer than three lands.

To avoid having to model the Command Zone, the Commander and Partner are added to the Hand after mulligans.

5.4 Running a Turn

The below process is repeated ten times per game. Many decisions made in running an individual turn refer to the number L, which is the number of lands on the battlefield at the start of the turn. The most mana accessible on any given turn, then, is $L + 1$, contingent on having a land in hand that will enter the battlefield untapped.

5.4.1 Untap and Draw

Every Land object has a Boolean value, `Land.tapped`. At the start of every turn, all lands in the Battlefield object are set to `Land.tapped = False` In practice, since all spells are cast simultaneously each turn as part of a lump and tapped lands are simply treated as lands that cannot be used the turn they enter, this is not significant. Nonetheless, modelling this aspect fo game behaviour ensures a common design pattern, and may facilitate modelling other land designs.

5.4.2 Determining Lumps

LandFill here makes use of the `combinations` function belonging to Python's `itertools` module. This allows it to determine all possible spell combinations in the hand. Each of these combinations are added to a Lump, to check whether they can be played with the current lands.

As this has to be done every turn to accommodate newly drawn cards, I immediately identified it as a potential runtime bottleneck. A non-mulliganned initial opening hand containing 2 lands and 6 spells (after the draw during the first turn) produces $2^6 - 1 = 63$ combinations. While the only relevant combinations are those with combined CMC $< L + 1$, this cannot be determined without checking individual combinations as per the Knapsack Problem (see 3.2.6). The simulator therefore uses the following heuristics:

- There is no need to search for any combinations of more than L+1 spells, as (notwithstanding spells of CMC zero, which are irrelevant for manabase optimization), there is no way to play more than this number of spells in a single turn using only lands.

- Any spells of $CMC > L + 1$ are ineligible, as they would not be castable in the first place.

Code profiling, carried out throughout development via the `line_profiler` library, shows that, with these heuristics in place, playing lumps takes up around 1/3rd of the runtime of `game.run_turn()`, making it an area for further improvement. Notably, it is *not* viable here to resort to Lazy Evaluation, which in this context would mean ceasing generation of new combinations after a playable lump has been identified. Consider the following opening hand ($L = 0$):

$$\text{Hand} = \big[\text{Spell}(U) \quad \text{Spell}(G) \quad \text{Spell}(GB) \quad \text{Land}(Island) \quad \text{Land}(Forest) \quad \text{Land}(Swamp)\big]$$

Were Lumps to be generated and tested lazily, the simulator would identify the spell costing U as playable, and play the Basic Island as the land for the turn. This would be a suboptimal play, however, as playing the Basic Forest would allow the same mana expenditure while also allowing the playing of a spell on turn 2 (if the Swamp were then played).

5.4.3 Playing a Land and a Lump

If the Hand object contains no Lands, the playability of each Lump is determined, and largest playable Lump is "played" - its constituent Spell objects are moved to the Battlefield. Determining the playability of a Lump is a complex process outlined in 5.4.4.

If the Hand contains at least one Land, the Lumps umps are ranked by the value of `Lump.cmc`, in descending order. The Simulator then sequentially plays each land, determines the first (largest) lump that can be played with it on the battlefield, assigns it to the Land as a temporary variable, and then returns it to the hand. The list of land objects (`lands`) is then progressively refined by the below functions:

```
allows_largest = self.filter_by_largest(lands)
filtered_as_taplands = self.filter_as_taplands(allows_largest)
filtered_by_most_produced = self.filter_by_most_produced(filtered_as_taplands)
```

Where:

- `filter_by_largest()` returns a list of lands capable of playing a Lump of CMC M, where M is the highest CMC of any Lump.
- `filter_by_taplands()` returns its input if none of the inputted lands would enter tapped this turn, and otherwise produces the ones that will. Placing this after `filter_by_largest()` ensures that the deck aims to play lands that enter tapped on a turn when that makes no difference to the amount of mana cast, thus preventing them from interfering at more meaningful moments later on.
- `filter_by_most_produced()` assesses the non-generic combined pips of every spell in the hand, and returns the land or lands which remove the largest number of these pips which are not already produced by a land on the battlefield. For example, if a hand's spells have the combined pips R B U G G, and the battlefield contains a single swamp, then Ketria Triome (producing RUG) would have a score of 1, while Zagoth Triome (producing BUG) would have a score of 2. Lands with the lowest score are returned, and out of these lands, lands which produce the greatest variety of mana are prioritized (ie, for a hand requiring G and U, a BUG land would be prioritized over a GU land)

A land is played from the returned lands at random, increasing the value of L by one, and a lump is played with values as close to L as possible. When the Lump is played, each land used in its casting is set to to `land.tapped = True`. If relevant, as in the case of Fetch Lands (see 5.7.1), the Land is informed what color of mana it will be producing, as per the mapping returned by the Linear Assignment Function outlined in the following section.

5.4.4 Assessing Lump Playability

The question of whether a given Lump can be played with a given set of lands is non-trivial. This is partly to do with the inbuilt complexity of some lands; complex designs such as Filter Lands, Check Lands and Dual-Faced Lands are discussed in 5.7. However, even setting these aside, the fact that some lands produce many colors of mana while others produce few or one means that a mapping between land and pip must be established such that multicolor lands are not "wasted" on pips whch are already well served by lands offering fewer colours. Several approaches to this were explored during development.

My initial approaches involved generating a list of all combinations of mana that the Lands in the Battlefield object could produce. If an entry on this list included all pips in `Lump.cost`, then that Lump is playable. The runtime bottleneck that this produces should be quickly obvious, especially as there are no comparable heuristics to those used when determining Lumps. Consider the below battlefield:

$$\text{Battlefield} = \begin{bmatrix} \text{Land}(BUG) & \text{Land}(BUG) & \text{Land}(UG) & \text{Land}(UG) \\ \text{Land}(UG) & \text{Land}(GB) & \text{Land}(GB) & \\ \text{Land(Basic Island)} & \text{Land(Basic Island)} & & \end{bmatrix}$$

In this case, there are $3*3*2*2*2*2*2*1*1 = 288$ combinations, which must be re-calculated on every played land. While lazy evaluation is an option here, it does not save much time, as any half-completed lazy calculations must be needlessly re-commenced if another lump is played before a new land is played.

To save recalculating combinations each turn, my initial solution was to store the list of combinations as an attribute of the Battlefield Object. Whenever a new land, capable of producing N different colours was played, its first colour would be added to all existing combinations. The Battlefield would then generate $N-1$ shallow copies of each combination, adding a different colour produced by the new land to each. Even without new combination generation, however, simply iterating through the existing combinations to update them proved far too slow. It additionally made it impractical to simulate removing lands from the battlefield. Some lands, such as Verge Lands and Filter Lands (see Fig. 2 in 2.1.2), produce different colours of mana depending on what other lands on the battlefield. This means that, when determining what land to play, the simulator must account for not only what mana that land can produce, but what mana it enables pre-played lands to produce. The easiest way to simulate this is to "play" each land, assess lump playability with it on the battlefield, and then return it to the hand before testing the next land.

A more promising solution was to establish a function capable of generating all combinations of colored mana from a an input set of lands, and memoize the output of this function. In memoization, the output of a function from a given set of arguments is stored in a cache, and a new output is calculated only if those arguments are not already cached [4]. In Python, this cache can be persisted across sessions via the Pickle module. Because any two untapped lands that produce (for example) UB are functionally identical in this context, the number of combinations to memoize appears

Figure 12: Two potential arrangements of lands that can be canonicalized identically

at first comparatively small, as each land can be canonicalized only as the colours it produces. Moreover, basic lands do not need to be canonicalized, and simply reattached to each permutation afterwards (see Fig. 12).

This functioned acceptably for a 3-colour deck. Memoization via Pickle produced a .pkl file of around 1000 megabytes. However, when I trialled a 5-color deck, this expanded by a factor of five (and may have continued to expand). The cache in memory, meanwhile, became so lengthy that a single cache miss took multiple seconds, and even a comparatively small number of cache misses increased the runtime, at this stage in development, from running 1000 games in about five seconds to running that same number over forty minutes.

Ultimately, I settled on modelling the question as a Linear Assignment Problem using SciPy's `linear_assignment()` function. Koopmans and Beckman [13] set out the Linear Assignment problem in the following layman's terms: paraphrasing, if a set of factories are to be built on a set of plots of land, and the suitabilities of a given plot to a factory's production processes means that each factory will return a specific profit at a specific plot, how can plots be assigned to factories to maximize the overall profit? In this context, it is helpful to invert the example. If a factory incurs a specific *cost* at a specific plot, how can plots be assigned to minimize costs?

Plots of land are here equivalent to Land objects, while the factories are equivalent to the pips of a Lump's cost. `Lump.cost` is here reformatted into a list of strings corresponding to the number of each pip (treating the "Gen" key in the dict object here as a pip). The list is then given any number of strings that read "None", to ensure that the length of the list is equal to L (ensuring, with reference to the above example, that the Linear Assignment solution holds even if there are fewer factories than plots). Any lump with a total CMC greater than L is discarded from consideration. "Costs" are then set for each Land, using `Land.live_prod`, like so:

	A pip in `Land.live_prod()`	A pip not in `Land.live_prod()`	"Gen"	"None"
A Land for which `Land.tapped = True`	9999	9999	9999	2
A Land for which `Land.tapped = False`	2	9999	2	2

`SciPy.linear_assignment()` then returns a mapping of lands to pips that minimizes this total cost. If the total cost is less than 9999, the Lump is playable. While this did not perform better

than the memoization method for a 3-color deck, time penalties for 4-colour and 5-colour decks became negligible.

5.5 Concluding a Game

When a game is concluded, all cards owned by the Hand, Battlefield and Graveyard are returned to the Deck. Performance metrics for the game are set as attributes of the Game object, for querying by the Trial and MonteCarlo objects.

5.6 Heuristics

For the initial development covered in this writeup, the below heuristics have been adopted; all offer an opportunity for future improvement of the product.

5.6.1 Multiple-Faced Cards and Alternate Casting Costs

Spells are only cast via the mana cost of their first playable face, and not via any alternate mana costs. As mentioned, many spells are in fact two spells, either of which may be played; many more cards have alternate casting costs listed in their textboxes, where casting them for a different cost changes the behaviour of the card on cast. Ideally, LandFill would treat these both as seperate spells when assinging combinations in the hand, and denote a card as cast if either of its options are played. However, in addition to the difficulty of parsing the textboxes of cards with multiple costs, representing some cards in the hand as two cards which cannot both be played in the same Lump adds an extra layer of complexity to Lump creation, and has not been implemented at this stage.

5.6.2 Spells with X in their Mana Cost

A spell with mana cost GX may be cast for one Green mana plus any amount of generic mana, usually acruing more value to the player for a higher value of X. This is complicated to include in Lump combinations; moreso in the case of cards with multiple values of X (a card costing XX being includable in any lump that leaves an even quantity of leftover mana). Therefore, X is always assumed to be 1 generic mana.

5.6.3 Strategies for Future Turns

The ordering of `filter_by_largest()`, `filter_by_taplands()` snf `filter_by_largest()` encourages the Simulator to play a land that enters tapped on a turn when this does not affect mana expenditure. This allow for some forethought, relevant to situations such as the sample comparison between the Battle and Slow lands in 2.1.2. However, there are some more niche situations in which this order of priorities does not apply. Consider the below hand with $L = 0$:

$$\text{Hand} = \big[\text{Land}(BW) \quad \text{Land(Basic Island)} \quad \text{Land(Basic Island)} \quad \text{Spell}(BBWW4) \quad \text{Spell}(UU)\big]$$

Since the BW land removes more colours of mana from the hand, it would be played by the simulator. However, the spell requiring BW had a CMC of 8 and will not be played for some time; playing the Basic Island, however, would allow playing the UU spell on the following turn. Since it is

necessary for the Simulator to assess the castability of Lumps rather than Spells, planning for future turns is difficult, and indeed situations like this may only be coverable via the progressive addition of heuristics. A future refactor may replace the hand object with a "queue" of spell combinations of progressively higher mana costs, adding each newly drawn card to this queue.

5.6.4 Ramp and Draw Spells

Many spells draw additional cards, while some provide additional mana. Doing so would potentially be a very fruitful area of development, and will be discussed in the conclusion. However, in addition to the difficulties of coding ramp and draw spells, introduction of these factors bring complex strategic considerations - it may, for example, be advisable in some situations to spend less than $L+1$ mana on a ramp spell to allow for greater overall expenditure later; it is also typically advisable to play draw spells before playing lands if possible, to see if a superior land is added to the hand.

5.7 More Complex Lands

Some land cycles required significantly more complex simulation logic, outlined below. Mechanics for the respective cycles are either detailed below or can be found in Fig. 2 in 2.1.2.

5.7.1 Fetch Lands

During assessment of Lump playability, a Fetch Land (instantiated as a FetchLand object) is considered to produce mana of colour M mana if:

- The deck currently contains a land for which it can search that can produce M
- That land would return `land.enters_untapped(game) = True` for the current game state.

The search itself happens is set when the FetchLand is tapped for mana. The FetchLand calls the `game.filter_by_most_produced()` method from the current Game object to narrow down the lands it is capable of fetching, adding an extra argument to specify that it the method must also account for pips in the hand currently unaccounted for by lands in the *hand*, rather than just on the battlefield. This encourages the FetchLand to make new colours of mana available. After this, the FetchLand is moved to the Graveyard. If tapped for "None" mana, a FetchLand will search for a land that will enter tapped, if one is available in the deck.

While production of W, U, B, R, or G by a FetchLand has the standard cost of 2 during Linear Assignment, production of "Gen" is weighted at cost 1, and "None" has cost 0. This means that `SciPy.linear_assignment()` uses the FetchLand to pay for a "Gen" or "None" pip if possible. This prevents the FetchLand from being forced to search for a colour it does not need to in order to pay a generic cost, and allows it to, as much as possible, find the best land for the current hand rather than just for the spell.

To ensure that any FetchLands search for Lands even on a turn when no Lump is played - allowing them to remove from the deck lands that will enter tapped, as per the scenario in 2.1.2 - an additional "Null Lump", containing no Spells, is created each turn, forcing the Fetch Land to always provide at least one "None".

5.7.2 Filter Lands

A Filter Land (instantiated as a FilterLand) produces C and has two SubLands (see 5.2.1), each of which may produce either of the Filter Land's colours. In order to account for a situation in which multiple FilterLands may pay for the abilities of others, the following recursive algorithm is used:

1. If a Lump is castable on a battlefield containing FilterLands while tapping those FilterLands for C, it is cast as normal.
2. If not, filterlands are placed into a new list, $\underline{L}_f$. A permutation of this list is lazily generated via `itertools.permutations()`, from first Filter Land $F(1)$ to nth Filter Land $F(n)$.
3. A new list, $\underline{L}_n f$, containing all non-Filter Lands is generated, including the subset of lands capable of paying $F(1)$'s cost, $P(1)$ to $P(n)$. $P(1)$ is removed from $\underline{L}_n f$, and the sublands of $F(1)$ are added.
4. Repeat steps 2 and 3, starting with $\underline{L}_n f$ each time, until all Filter Lands have been either replaced with their sublands, or tap for colourless if there is no land capable of paying for them, with $F(n)$ being the final one so replaced.
5. If the Lump is castable with the resulting list of lands and sublands, cast it.
6. If it is not, return the land $P(1)$ that had been removed at the recursion depth when the sublands of $F(n)$ had been added, and remove $P(2)$. Assess the castability of the Lump again, casting it if possible.
7. If no lands $P(1)$-$P(n)$ in the list at the recursion depth of $F(n)$ allow for the lump to be played, return to step six for the list at the recursion depth of $F(n-1)$.
8. If the recursion depth of $F(1)$ is reached, generate a new permutation with new values of $F(1)$ and $F(n)$

Although this requires testing a large number of permutations if a lump is not castable, it does not provide a significant performance bottleneck.

5.7.3 Dual-Faced Lands

A Dual-Faced Land is given two sublands, and an attribute, `land.committed`, initialized to None. When tapped for "Gen" or "None", if `land.committed = None`, a subland is selected via (game.filter _by_most_produced()) in a similar manner to Fetch Lands. When tapped for coloured mana, `land.committed` is set to the subland capable of producing tha tcolour. If `land.committed != None`, the land is tapped as though it were the subland assigned to that attribute. The Battlefield object sets `land.committed` to None when it gives the land to another cardCollection. Like Fetch Lands, a Double Faced Land has a lower weighting to tap for non-Generic during linear assignment.

5.8 Simulator Verification Testing

Testing of the Simulator was done via optional `samplehand = []` and `sampletopdeck = []`arguments passed to `Game.run()`. Both are arrays of card names as strings, which, if included, are extracted from the deck and placed either in the hand or at the front of the deck object after it is shuffled. This allowed me to watch the Simulator play sample hands and ensure behaviour was as expected.

6 The Optimizer

6.1 Constituent Classes

6.1.1 MonteCarlo and Trial

Recall that these respectively are subclasses of CardCollection and Simulation. Their relationship may be modelled as so: MonteCarlo produces a deck, creates a single Trial object for that deck, uses that Trial's `trial.run()` method to run many games, and then accesses performance data for those games via the Trial's attributes.

6.1.2 LandPrioritization

The LandPrioritization class simultaneously ameliorates three problems in the design:

- As will be discussed in FORTHCOMING SECTION, fluke results in which a land performs equivalently to a strictly better one, as long as the betterness is narrow, are not uncommon.
- As noted in 3.2.6, some lands are balanced via a life point penalty, and that this is difficult to quantify within an optimization context.
- Since LandFill uses a Hill Climbing algorithm, at each increment, a deck may be tested as many as N times, where N is the number of neighbouring decklists (ie, decklists with one card different). This is time consuming.

It is useful, therefore, to encode "strict betterness" in such a way that a land is never tested against a strict superior capable of producing its colours at any hill climb increment, and that equivalent lands that differ only in life point commitment are considered strict superiors/inferiors to each other.

MonteCarlo is therefore given a LandPrioritization object, which maps out strict superiority/inferiority relationship. Land A is considered is considered strictly better than Land B if it is capable of producing all colours that Land B does and either:

- Never enters tapped, while Land B sometimes does.
- Sometimes enters untapped, while Land B never does.
- Requires a life point investment that Land B does not.
- Has basic landtypes, whlle Land B does not.
- Always produces its colours, while Land B only produces some of them under certain conditions.

Strict Betterness is mapped in 13

Land objects belonging to the MonteCarlo are "registered" with the LandPrioritization. After all have been registered, each land is assigned its strict betters according to its cycle. While this was initially modelled as a simple dict object comparing a given cycle to its superiors, a more complex class was deemed necessary to accommodate situations in which a player requested a cycle with both inferiors and superiors be removed from consideration; in such cases, LandPrioritization "cascades" through, applying the superiors of the removed land to its inferiors.

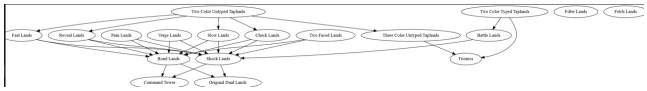

Figure 13: Digraph of strict betterness relationhips between all cycles currently handled by LandFill, where each arrow points to the superior land. Note that some labels (ie, Three Colour Untyped Taplands) refer not to single cycles but to groups of mechanically equivalent cycles which differ in ways that do not pertain to mana generation

6.2 Optimization Algorithm

6.2.1 Outline of Steepest Ascent Hill Climbing Algorithm

LandFill improves on the "Simple" Hill Climb algorithm set out in 2.2.2 by implementing instead a "Steepest Ascent" Hill Climb, also known as "method descent." At every increment, rather than adopting as a new value for $\underline{x}_c$ the first input in its neighbourhood $N(\underline{x}_c)$ to return a higher value, as in Simple Hill Climbing,, it examines all inputs in the neighbourhood and chooses the one that representing the highest rate of improvement. Since all lands must be entered into the deck in discrete quantities, this can be calculated easily by modelling all exchanges and selecting the one offering the highest improvement. This is, however, time consuming to calculate, adding a search space of

$$N \times M$$

to each increment, where $N =$ is the number non-mandatory lands in the deck and $M =$ number of lands in the MonteCarlo with all strict superiors in the deck, so that all possible exchanges can be tested.

Heuristically this can be divided into two steps: first, identifying the worst land in the deck, and second, identifying the best land to replace it with. The manner in which LandFill does this is outlined in **??**. This divison offers Steepest Ascent Hill Climbing a key advantage over other optimization methods such as Simulated Annealing, Evolutionary Algorithms, or even Simple Hill Climbing: "LandFill repeatedly swaps out the worst land in your deck for the best land that could replace it" is an intuitive summary of the process. Since LandFill makes decisions in place of a human user at a level of statistical/algorithmic abstraction beyond what could be readily offered to them for customization, it needs to operate at what THE SUMBARINE GUYS described as a high level of automation, which, as has become a topic of considerable interest given the rise of recommendation and advertising algorithms, raises issues of trust. See & Lee's 2004 examination of the topic is instructive here:

"Trustworthy automation is automation that performs efficiently and reliably. Achieving this performance sometimes requires very complex algorithms that can be extremely hard to understand. To the extent that system performance depends on appropriate trust, there may be some circumstances in which making automation simpler but less capable outweighs the benefits of making it more complex and trustworthy but less trustable"

How this informs frontend design is covered in 7.2.3.

6.2.2 Other Options

Simulated Annealing is a variant of the Neighbourhood Search algorithm that avoids becoming trappedlocal maxima by introducing an element of randomness. At any given increment, the

function may return a higher score but decline to accept this as a new value, with the probability of this occurrance dependent on a "temperature" variable that decreases as the search space is traversed.

As will be demonstrated in 6.3.4, within a number of simulated games that can be run in a user-appropriate timeframe, the chance of two decks scoring comparably when there is only one card different between them, in a way that does not reflect which of the two cards is stronger, is non negligible, essentially introducing a stochastic element on its own.

One approach I did trial, however, applied the "temperature" concept from Simulated Annealing to a Neighborhood Reduction schema. In Neighborhood Reduction, moves that are unlikely to lead to an optimal solution are discarded, saving processing time. Recall that the main synergy by which lands improve each other involves synergy between basic lands or lands with basic land types, and that the algorithm starts with a manabase consisting entirely of basic lands. As will be seen in 6.3.4, a Check Land is functionally identical to a Shock Land when they are the only nonbasic cards in a manabase, which prompts the question as to whether it is worthwhile to check all lands at every decrement of the number of basic lands in the deck. In later iterations, however, when the deck contained fewer basic lands, the performance of Shock Lands in the deck diminished rapidly. To take advantage of this, I trialled an algorithm in which a declining temperature governed the number of lands swapped out of the deck at each increment: initial steps swapped out the worst N lands in the deck for the best performing N lands tested individually. As the value of N decreased, each swap became smaller, and thus each land when subjected to cardtrial was tested against a more accurate representation of the decklist in the form to which it would be introduced. This meant, essentially, that while initial iterations largely tested lands on general quality, later iterations took more time to test lands on their synergy with the existing manabase. While this did offer runtime improvements, it was ultimately not chosen, given that Neighborhood Reduction methods carry some inherent risk of suboptimality [22], and the same runtime improvement could ultimately be achieved more simply, and with greater user consent, by simply advising them to automatically mark high performing cycles such as Shocklands and Fetchlands for mandatory inclusion.

6.3 Choice of Performance Metrics

6.3.1 Overview

To test different performance metrics, four sample decks containing the same nonland cards in colours BUG were submitted to Trial objects outside the context of the Hill Climbing algorithm:

- BasicDeck - a deck containing only basic lands in the deck's colours.
- PartialDeck - a deck containing basic lands, shock lands and fetch lands, representing a deck part way through the optimization process.
- OverDeck - a deck containing no basic lands, reflecting hypothetical over-application of the optimization principles.
- ExpectedDeck - a deck whose manabase was chosen by me, reflecting what I as a user of LandFill might expect an optimized deck to look like.

Deck	Mean	Median	Mode	Range	Kurtosis
BasicDeck	2.696	1.0	0	21	2.589267
PartialDeck	1.479	0.0	0	18	8.076589
OverDeck	0.895	0.0	0	10	5.604503
ExpectedDeck	0.609	0.0	0	18	31.770379

Figure 14: Outputs in terms of wasted mana for each deck

6.3.2 Use of "Wasted Mana"

In his analyses, referenced in 2.4, Karsten proposed three underlying assumptions [12] for assessing the castability of a spell of mana cost M:

- We want to cast the spell on turn M.
- We condition on drawing at least M lands by turn M.
- There are a realistic number of lands in the deck

To control for these factors, at the conclusion of each turn, the Simulator identifies $C(l)$, the combined CMC of the largest lump determined that turn (either cast or not) whose value is equal to or less than the number of lands on the battlefield at the turn's conclusion, and $C(c)$, the combined CMC of the lump that was cast this turn. The game tracks the accumulated value of $C(l) - C(c)$ each turn to determine W, the total wasted mana per game. This penalizes lands which enter the battlefield tapped, as they permit larger value of $C(l)$ without increasing the maximum value of $C(c)$, as well as game states in which there is an insufficient range of colours on the battlefield, without penalizing the curve of the player's deck relative to the number of lands they selected.

Focussing on mana *wasted* rather than mana *spent* is not without downsides. Bounce Lands, for example, are a cycle of lands which enter tapped and produce two mana per tap, requring in exchange the return of an already-played land to the player's hand. They perform strongly on a deck that has purposefully chosen to run fewer lands, as they effectively count for two lands in one draw, an advantage detectable only if the amount of mana *spent* per game is assessed. LandFill will not be in a position to recommend Bounce Lands for small manabases, although as Bounce Lands are fairly unpopular in the contemporary commander scene [19] this is considered an acceptable sacrifice. While presumably after sufficient games, mana expenditure would eventually converge on a meaningful value, tests of the aforementioned decks using mana expenditure as the metric returned minimal performance differences over multi-hour long trials.

6.3.3 Initial Analysis

Data from a trial of four decks, summarized in Figure 14, prompts several immediate observations. First, the modal wasted mana is consistently zero, while BasicDeck's median value of 1 is the highest of any deck, indicating that even at a comparatively low level of optimization, any performance considerations that are not simply the probability of wasting no mana in a game are only relevant in a minority of cases. Second, while the decline in mean wastage between PartialDeck and ExpectedDeck is comparable in size to the mean decline between BasicDeck and PartialDeck, the rate of increase of Kurtosis increases dramatically at higher levels of optimization. Third, such measures of central tendency hint that OverDeck may be a more viable deck than ExpectedDeck, having only a slightly higher mean wastage but a significantly lower range (wasting at most 10 mana per game, as opposed to ExpectedDeck's 18).

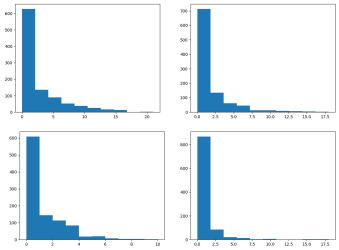

Figure 15: Histograms of the mana wasted by each deck: clockwise from top left: BasicDeck, PartialDeck, OverDeck, ExpectedDeck

Examining the histograms of ExpectedDeck and OverDeck's mana wastage clarifies this (see Figure 15). Note that between PartialDeck and OverDeck, the proportion of games with zero mana wasted declines significantly, increasing again for ExpectedDeck. This indicates that while the mean wasted mana between ExpectedDeck and OverDeck may not favour ExpectedDeck by a wide enough margin to offset the reduced risk of extreme mana wastage, OverDeck's reliability does come at the cost of a non-negligable reduction in the likelihood of a game wasting zero mana at all. Given that, as mentioned, zero mana is wasted in a plurality of games even at a low level of optimization, an amelioration of suboptimal results at the cost of optimal results is not considered a worthwhile exchange. LandFill therefore assesses a manabase based on the probability, henceforth P, of it not wasting any mana at all in a game.

6.3.4 Proportion Wasted vs Cumulative Distribution

One potential improvement over P as a metric would be C, the area under a Cumulative Distribution Function (CDF) of wasted mana per game. In such a metric, a decklist with a lower value of P would be penalized compared to one with a higher value, but two decks with equivalent values of P may return different scores depending on their probability of producing significantly sub-optimal games.

To investigate this, two additional decks were prepared: CheckDeck and ShockDeck, both identical to BasicDeck save for the replacement of a single Forest with a UB Check Land or a UB Shock Land (see Figure 2), respectively, representing decks after a first Hill Climb increment. Wheraes OverDeck is unlikely to ever be assembled by LandFill in its normal course of operations, if one metric was better able to distinguish the slight superiority of ShockDeck (which need never have a land enter tapped) to CheckDeck (which, on occasion, will), and distinguish the slight superiority of both to BasicDeck, that would be a stronger metric to use. I conducted 100 Trials of 1000 games each for all decks, summarizing the same data first with P and second with C. Box plots of the results are shown in 16

Disappointingly, neither of these methods prove more able to capture subtle performance differences at the increment level; use of P seems slightly more capable, but given the overwhelming overlap between the whiskers of BasicDeck, CheckDeck and ShockDec, this does not seem to be significant. However, with the exception of OverDeck, which is significantly more harshly penalized by P, there is not a significant difference between P and C in the proximity of the whiskers between decks that have multiple cards difference between them, suggesting that the additional information

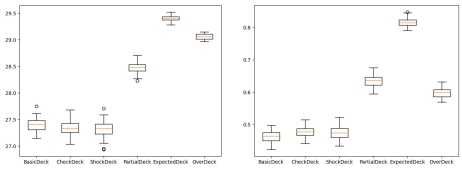

Figure 16: Box Plots representing the same Trial output data as (left to right): proportion of games in which zero mana is wasted, and area under a CMF of mana wasted.

provided by C does not produce a meaningfully more sensitive assessment.

Given this, it is worth noting that P presents several design advantages. First, it is faster: given that any value of wasted mana greater than 0 is equally discrediting, games can be abandoned the second any mana is wasted. Secondly, it is a simpler metric. While this will be discussed in more depth in **??**, this investigation thus far has been carried out under the assumption that, as these metrics are done at a level of statistical abstraction that does not translate neatly into MTG strategy, the criterion is built into the code and not customizable by the user. Accepting this, it is important to nonetheless explain to the user the criterion, and a user should not be expected to have an understanding of Cumulative Frequency Analysis in order to understand by what metric the produced decklist is considered "optimized". That a deck has been found to have a certain probability of wasting zero mana in a game is much more comprehensible.

6.4 The Hill Climbing Algorithm

6.4.1 Setup

Once the user has selected preferences, the Deck and MonteCarlo objects are instantiated. Since both are CardCollection objects, GameCards are assigned to each accordingly:

GameCard representation of..	Assgned To	`GameCard.mandatory`
All nonland cards	Deck	True
All land cards in the player's input (ie, utility lands)	Deck	True
Any land cards that the player selected from LandFill's input screen to definitely include	Deck	True
All Land cards in a recognised cycle that are not already in the deck	MonteCarlo	False
100 basic land cards of each colour required by the deck	MonteCarlo	False

Basic Lands, rather than a random sample of nonbasic lands, are used as the starting input thanks to the principle of diminishing returns set out in 2.1.2.

6.4.2 Each Increment

With a new manabase added, MonteCarlo creates a new Trial object for the deck. `trial.run()` simulates a set number $\underline{G}_r$ of games, and, for each, appends to attribute `trial.wasted_games`

a boolean value representing whether any mana was wasted this game. GameCard objects also contain a `GameCard.wasted_games` array, and this is updated with the same value as the Game on every Land object for which `GameCard.mandatory = False` and which was drawn during the game. Every land to which the simulated player has access, essentially, takes responsibility for the success or failure of the Game. The Trial then uses this information to rank all cards in the deck by performance, and then resets all values of `GameCard.wasted_games` to empty arrays. The lowest ranking card is set as `Trial.worst_performing_card`.

With the worst card so established, as outlined in 6.2.1, the next step is to identify the strongest card to replace it. This is more complicated, given the small performance differences between two decks that differ only on one card - especially since, as each candidate replacement card must be tested at each increment, it is not practical to run as many simulations as was used in `trial.run()`.Instaed, for each candidate card, MonteCarlo calls a new method: `Trial.card_test(Land)`. This runs as follows for input land L, where T refers to the number of turns for the simulated game plus the number of cards (7) in an opening hand, and $\underline{G_c}$ is the number of games simulated by the function:

1. `Trial.worst_performing_card` is removed from the deck and replaced with L.
2. A simulation is run.
3. If L or any lands capable of searching the library for L are in the top T cards of the deck, the game is run as normal.
4. If not, L is randomly added to some position within the top T cards of the deck.
5. If a mulligan occurs, steps 3 and 4 are repeated.
6. The game is run.
7. Steps 2-5 are repeated $\underline{G_c}$ times.
8. L is removed from the deck and replaced with `Trial.worst_performing_card`.

The reason for the division of step 3 and 4 is because for any reasonable size of T, the chance of drawing L in the opening hand, or before any lands capable of searching for it, is extremely significant, and once a land is in the player's hand, it becomes an invalid target for a Fetch Land. The importance of this is best illistrated with the Triome cycle, which are basic-landtyped taplands which tap for three colours. A deck forced to draw a tapland every game incurs a significant penalty which may overcome its need for colours; however, a deck with many Fetch Lands may benefit from those Fetch Lands being given the option of Fetching a three-colour land on a turn when extra mana is not needed.

`Trial.card_test(Land)` retuns to MonteCarlo the percentage of games for which zero mana was wasted with the specified substitution, and MonteCarlo replaces texttttTrial.worst_performing _card with the land that returns the highest percentage.

I assessed possible values for $\underline{G_r}$ and $\underline{G_c}$ based on their ability to produce similar manabases when a MonteCarlo was run ten times using the same initial deck. I judged this by calculating the mean of the Jaccard Indices for each combination of two manabases within each set of ten. This produced the below data:

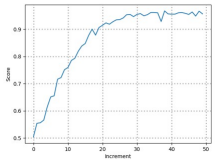

Figure 17: Rate of improvement in percentage of wasteless games for a BUG deck, where the X axis represents Hill Climb increments and the Y axis shows percentage of wateless games.

$\underline{G}_r$	$\underline{G}_c$	Average Jaccard Index	Runtime
100	50	0.678	13 seconds
100	100	0.693	16 seconds
500	100	0.717	36 seconds
1000	200	0.697	1m 23 seconds
500	1000	0.705	5m 31 seconds
2000	400	0.710	4m 46 seconds
10000	2000	0.815	21m 32 seconds

While the Jaccard Index broadly increases with higher values of $\underline{G}_r$ and $\underline{G}_c$, small alterations in these values do not produce striking differences; even the above data includes clear outliers, and is unlikely to be reliable. A substantial improvement requires a tenfold increase in the number of simulated games, which brings with it a disqualifying runtime loss (it should also be noted here that this this test was done with a three-colour deck; in a five-colour deck, with many more candidate land to be tested at each increment, the runtime penalty would be even greater). For the initial version, I used a value of 1000 $\underline{G}_r$ and 200 for $\underline{G}_c$.

6.4.3 Halting

Figure 17 shows the rate of improvement of a deck at each step of the algorithm. The displayed pattern of a steep initial improvement that gradually shallows, with multiple small local maxima along the way, is indicative of most decks tested. Repeated application of the algorithm to an identical deck suggests that these local maxima are random performance outliers, and not true local maxima within the combinatorial optimization.

This means that rather then halting when a maxima is reached, the halting criterion must trigger when the rate of improvement drops below a certain threshold - for the data in figure 17, this would be around the 33rd-40th increment. This requires smoothing of the data, to avoid triggering the hatling criterion during the descent after a local maxima. I trialled two filtering methods:

- Smoothing the line with a Savitsky-Golay filter to remove the data noise causing the local maxima, and then halting when the derivative of the resulting line reduced below zero.
- Calculating the mean result across window length W from the most recent score generated, and comparing it with the mean result from the window of the same length before that. The system would halt when the mean of the prior window subtracted from the mean of the current window fell below zero.

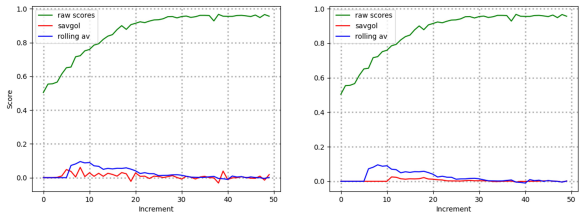

Figure 18: Outputs of the rolling average and the Savitsky-Golay derivative halting functions when applied to the data shown in Figure 17; in both cases, the rolling average used a window size of 3, while the Savitsky-Golay filter window was set respectively at 3 (left) and 11 (right)

While the Savitsky-Golay method is the more sophisticated of the two, trialling both found it to require a much larger window, as is demonstrated in **??**. Whereas a Savitsky-Golay derivative can fall below zero, and thus trigger halting, at around the same time as the rolling average does if the Savitsky-Golay window is significantly larger, if the windows are of comparable size, the Savitsky-Golay filter retains too much data noise and triggers too soon. The window-size of the halting algorithm represents a lower limit of how many iterations LandFill requires, and allowing a smaller window-size allows players who use large numbers of utility lands or who clearly specify their own preferential mandatory cycles to be rewarded with lower optimization times.

6.5 Optimizer Verification Testing

Verification Testing of the optimizer is difficult, it is impractical to assess a manabase's performance over real Commander games. For this reason, its outputs are predomenantly tested as part of validation testing, in terms of user satisfaction. However, there are two indicators that can measure the degree to which the algorithm is converging on a useful value.

The first is consistency, and has been largely covered in **??**. Given my chosen number of simulations per increment, we can say that LandFill has a consistency of around 70%, and that this can be increased at a cost to runtime; however, it would take a debilitating runtime penalty to make a meaningful improvement.

The second is the degree to which the provided manabase reflects unique properties of the deck in question beyond its colours. This is not strictly necessary for a useful product, but the prospect of a bespoke manabase offers a meaningful addition of value over resources such as those published by Frank Karsten. I tested two decks, one BUG and one WUG, which have mana curves depicted in 19. Note that for the BUG deck, many hands will contain no spells playable on the first or second turn of the game, while this is not true for the WUG deck.

On an initial run, LandFill favoured similar cycles for both decks. However, I then re-tested, removing from consideration all cycles that are:

- Strictly better than basic lands except for penalties that do not relate to mana generation (ie, Shock Lands).
- Strictly better than basic lands but lacking basic landtypes (ie, Verge Lands).

On this rerun, the difference was striking: BUG received all three candidate Slow Lands and two out of three candidate Check Lands, both of which are cycles that improve if other lands are

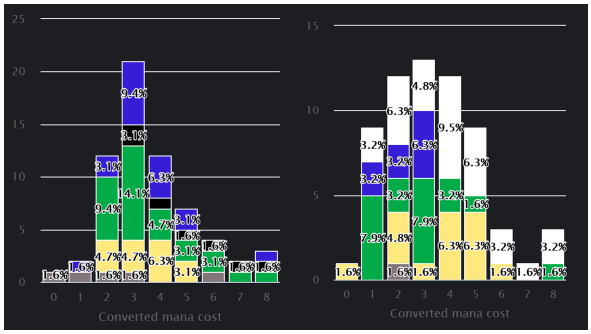

Figure 19: Mana curves as visualized using the TappedOut card database for a WUG and BUG deck respectively.

played before them in the battlefield. WUG received all three candidate Fast Lands and one out of three candidate Reveal Lands, both of which are cycles that improve if played early in the game before other lands.

7 Frontend Design

7.1 Parsing Inputs

In 3, users desired that LandFill support "round trips" (in the words of one subject), in which a list of cards was copied from a Deck database such as Tappedout or MoxField, into LandFill, and could then return the outputs without any additional formatting. For this purpose, I created the InputParser class, of which one is created at the start of each session. Test users mentioned four such databases: TappedOut, Moxfield, Archidekt and Deckbox. since Tappedout and Moxfield both allow for lists to be copied from the deck homepage rather than from the formatted export panel, and Moxfield and Deckbox take inputted cards in the same format, this means that the InputParser has been designed to distinguish between six possible input formats, and return three possible output formats.

Since Tappedout and Archidekt both support categorization of cards (ie, draw, ramp, win strategy, enemy card removal) with custom lables, the Input Parser stores all cards in a Dict object corresponding to each category, using a default key if none are specified by the user. On completion of the Monte Carlo process, the InputParser a total of six string objects: for each database format (Deckbox and Moxfield being identical), it returns the total decklist, including categories if provided by the player, and the list of provided lands.

7.2 App Layout

In the intial mockup drafted in 3, LandFill consisted of a single homepage. Users in the think-aloud evaluation generally did not understand the difference between the button that confirmed their nonland inputs and the button which commenced the MonteCarlo. On redrafting, I split the design into four pages, with purposes broadly outlined accordingly:

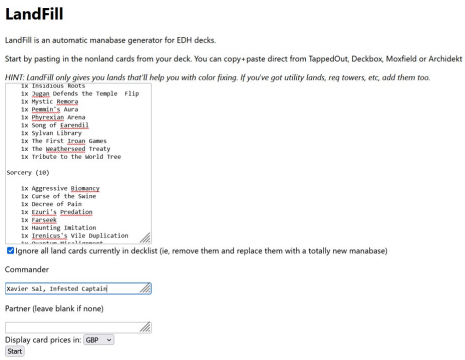

Figure 20: Deck Input page.

1. Deck Input - allows user to list details kept constant throughout the session, ie, cards inputted by the user, and the currency in which the session is to list card prices.
2. Preferences - allows user to list candidate land cards for LandFill to recommend, and prioritization thereof.
3. Progress - accepts no user input, but updates at each hill climb increment what is being swapped for what.
4. Output - lists the new decklist, including performance of the manabase and the cards therein.

Screenshots of each are set out in the below sections, with commentary.

7.2.1 Deck Input

See figure 20. The option to overwrite inputted lands with a new manabase was added in response to users in the Think-Aloud evalution who instantly copied a completed deck from a database into LandFill and manually removed lands from the list.

7.2.2 Preferences

See Figure 21. The Preferences Page allows users to customize the lands trialled by LandFill depending on their preferences and card availabilty. The initial mockup had allowed players to exclude lands or whole cycles via a series of tickboxes, but during the Think-Aloud evaluation, users had found this both overwhelming, given the sheer quantity of lands, and unintuitive, as they did not always know what a given cycle did. Moreover, given that LandFill in its current form takes a non-negligable amount of time to run, I felt it prudent to encourage the player to mark favoured cards as mandatory for inclusion, rather than placing the burden of them to list mandatory lands as part of the deck input. This required me to replace the binary input of a land being unchecked (excluded from consideration) or not with a three-way system, by which players could mark cycles as mandatory, possible or forbidden. I addressed all of these issues via a trio of drag-and-drop boxes from the React-Beautiful-DND library, while also adding a side-panel element which would show a sample card and card list from a draggable cycle object on mouseover.

Since some lands may behave identically as manabase components but may have additional mechanics that a player may prefer, I created two additional drag and drop boxes, each of which corresponds to a category of identical lands. The frontend does not support dragging between these boxes; rather, they are used to prioritize these lands accordign to thhe user's preference. Information from this input is passed to the backend LandPrioritization object. As depicted in 21, LandFill will only trial a Bicycle Land if the Typed Dual Land of its colours is either already in the deck or is excluded from consideration.

The panel of preferences still set via tickboxes and numerical inputs above the drag-and-drop column are based on comments made by users during the initial mockup testing: players were keen to set a minimum quantity of basic lands, as many non-land cards synergise with these, and price was a concern for most players in manabase construction. Several initial testers also expressed disdain towards running "off-colour fetches", especially in Commander: this might mean, for example, running a fetch land capable of finding an Island or a Mountain along with a suite of shock lands in a BUG deck, which although potentially a strategically sound decision, was felt by many players to be fundamentally unaesthetic.

Recall from 3.2.6 that players did want to factor price considerations into their choices. While it is impractical, as detailed, to allow the player to set a maximum price for the manabase, LandFill gives players the option to exclude all land cards above a certain price. I consider this to be a reasonable, if imperfect, proxy.

Given the complexity of the interface, it is worth touching on the underlying logic. A common theme among test subjects in pre-development evaluation was a lack of a single consistent plan or criterion, with players generally expressing a range of broad preferences often caveatted by circumstance - price, for example, not being a factor if a player already owned an expensive land, even if they were not interested in buying more from the cycle. As another example, a player who expressed considerable hostility to any land which only ever enters tapped admitted a fondness for the "Surveil Land" cycle due to its utility effect. I felt it necessary, consequently, to always give players an override option. Individual cards can be added to consideration despite the status of the cycle as a whole via the tickboxes in the side panel, while use of any of the "Exclude from Consideration" Filters, which moves drag-and-drop objects to the "Exclude" column, does not prevent any individual items from being removed from that column. Since this means that a cycle or individual card may be categorized either by dragging and dropping or via a filter or override method, the state of any given land is always mapped to one of two "positiveArrays", positiveArrays.include or positiveArrays.consider. If it is excluded via any method, it is put in one of several "excludeArrays", corresponding to the frontend input used to exclude it. When the simulation is run, all lands in any excludeArray are marked as excluded, and all others are marked as either mandatory or possible, depending on which positiveArray they are in.

7.2.3 Progress

See Fig **??**. Although initial drafts replaced the "run" button on the Preferences page with the word "Loading", I felt that this was inappropriate for a longer optimization time - I did not want the users to think that the program had simply crashed. Moreover, during mockup testing, several users expressed a desire to see the logic being used by the backend, as this would allow them to trust the results better. Although the Preferences Page contains a contextual explanation of the Hill Climbing algorithm, the use of a Progress page allows for this to be more clearly demonstrated: each step in the Hill Climb increment is printed to screen. At the conclusion of the Hill Climb

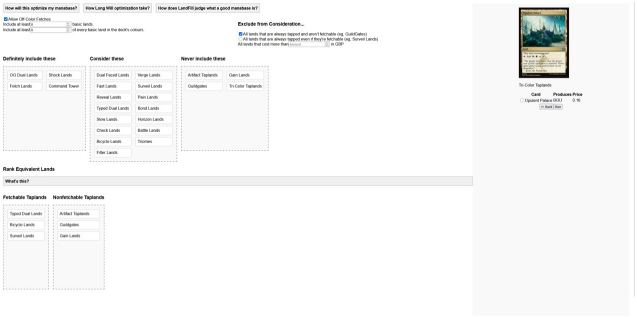

Figure 21: Preferences page, zoomed out to show all content; the rankings panels would normally be offscreen. The side-panel remains stationary as the user scrolls.

Figure 22: Progress page

algorithm, the Output page loads automatically.

7.2.4 Output

See Fig 23. The core feature of the Output Page is the textarea which shows the recommended decklist, which can display either the lands added to the deck or the entire decklist for easy export. When the entire decklist is displayed, if the input format included custom card categorizations (ie, from TappedOut or ArchiDekt), these are re-added here, with new lands being added either under a new category, "Lands", or under an existing category if one was detected that contained other land cards at input.

The left hand panel allows users to use the same mousever interface as was employed on the Preferences Page to examine the cards added to their decklist. One user, during initial testing, asked to see the lands ranked by performance, as this would inform both their choice of ramp spells and tell them what lands they should replace on the printing of a new cycle.

8 Post-Development User Tests

8.1 Areas for Improvement

8.1.1 Persistant Preference Storage and Links with Existing Databases

Storage of user data between sessions was determined early in development to be wholly outside the scope of the prototype. It was, however, identified unaninimously by testers as the feature they would most like to see added to the current form. More specifically, all users felt that LandFill would benefit from the ability to either upload a list of land cards the player owns, or link to a

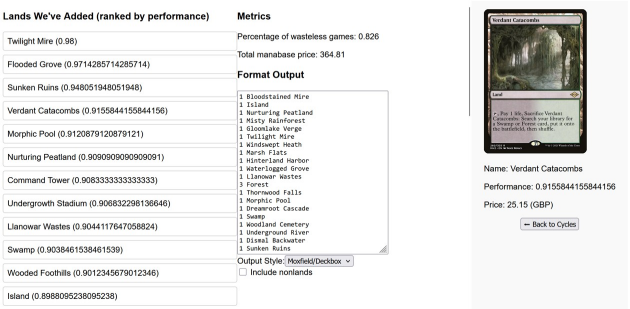

Figure 23: Output Page

user's account on a pre-existing card library database, so that lands could be excluded if the player did not currently own a copy.

9 Conclusion

9.1 Findings and Product Viability

9.1.1 Utility

LandFill earned positive user feedback. Users did not feel like the suggestions it made contradicted their own instincts as deckbuilders, and insofar as it made suggestions they may not have thought of, they considered this to be a positive addition and cause for thought. Recall that the initial role of LandFill was to produce a deck that is comparable to what a player can put together in an equivalent timeframe; to this end, it is a successful product. My broad initial hypothesis, that a goldfishing engine that cut out all elements of play strategy except the maximisation of mana expenditure would be a sufficient venue in which to test the viability of specific land cards, was correct.

In respect of the realities that, first, a consistently optimum build is not producable within a usable timeframe and, second, there is no immediately intuitive objective function for what constitutes an "optimized" deck, I designed the frontend to support uses other than the intended one, a design philosophy called "appropriation." One issue I remained mindful of through development was that even if a LandFill user did not consider a full deck optimization worthwhile, the software should still be an efficient formatting tool. A user who simply chose to list all cycles as either mandatory or excluded can still benefit from having these cycles input automatically without needing to look up individual cards on ManaGathering and input them one at a time. The use of a halting metric that only requires a minimum of 6 iterations means that such users - who will only benefit from optimization of their basic lands - will be able to use the system much more quickly. Furthermore, while it was not possible on this timeframe to synchronize LandFill with a user's card database, as widely requested in post-development testing, recall that this reflects a desire to deckbuild using only cards that they already own. Since LandFill ranks all cards in the deck by performance in the output, simply inputting a decklist with an existing manabase can inform you about which lands perform worse or better within it, and thus can be removed and used in a new deck instead.

Recall from 6.5 that, while there is room for LandFill to be made more consistent, it makes

meaningful decisions about what lands are best for a specific deck in a situation where widely acclaimed lands such as Shock Lands are not available. Given that Shock Lands are expensive, this means that, while the concept of a "deck optimizer" implies a degree of competitiveness, in practice, LandFill's algorithm may prove most relevant for players working on a budget.

While a full investigation of this is outside the scope of this project, one interesting and very consistent observation throughout design and testing of LandFill is that Filter Lands are significantly stronger and more versatile than perceived. Conventional wisdom holds that Filter Lands are best utilized in 2 and 3 colour decks; in higher colour combinations, the risk of failing to draw a land able to use in their ability is too great **??**. In my initial research, players generally criticized the cycle. However, LandFill added Filter Lands to decks of many color combination. It should be noted here that Filter Lands were by far the most complex land to encode, and this perhaps reflects a general difficulty in assessing their performance via experiments such as those ran by Frank Karsten. While I did not test this specific point thoroughly, it points to the relevance of LandFill's backend in the context of professional game analysis, were runtime not a consideration.

9.1.2 Limitations

Recall from **??** that, measured by average Jaccard Index, MonteCarlo runs that took less than ten minutes for a three-colour deck only shared around 70% of lands in common. This suggests that, with use of the simulator as outlined here, a single optimum - even a local maximum - cannot be reliably reached in a usable timeframe. However, the increase in Jaccard Index when longer timeframes were permitted suggests that structurally, LandFill works broadly as intended. Given that code profiling was conducted repeatedly through development via the `line_profiler` library, overcoming this would have likely required a total rebuild. This could involve translating the backend into a higher performance language such as C++; however, given Python's reliability as a backend programming language, an intermediate solution may be to refactor around a piplining library such as Joblib to allow multiple games to be run simultaneously. While I did explore a joblib-based implementation, the required deepcopying of deck and card objects immediately counteracted any runtime improvements; if it is feasible, it would need to inform the structure of the objects at a much earlier stage of development.

Given the extent of research necessary to determine an appropriate objective function, halting criterion and simulation count per increment for the optimizer, one area did remain underexplored - the choice of Steepest Ascent Hill Climbing itself. While I did touch on other Neighbourhood Search variations, one possibility that I did not have time to explore was the use of an Evolutionary Algorithm. In such an algorithm, pairs of high-performing manabases would be randomly shuffled together with small mutations at each increment **??**. Since I did not trial this algorithm, I do not know if it would have reached a more consistent result in a comparable time. Ultimately, given the intuitiveness of Steepest-Ascent Hill Climbing, I believe my decision to focus on rigorous testing of various adaptations of this base algorithm was an effective use of my development time; however, this would be an immediate route for further experimentation, especially if a more time-optimized refactor of the simulator as outlined above still fails to produce a consistent optimum.

9.2 Future Development

In addition to the developments requested in post-development testing, it is worth touching on some features that were consciously left out due to timeframe requirements and may be introduced in future iterations.

9.2.1 Ramp and Draw identification

Spells that ramp and draw are significant in Commander, and their lack of presence in LandFill represents a significant deviation between the Simulator and gameplay. The impact that this has on manabase decisions is non-trivial: many ramp spells reward disproportionate inclusion of Forest cards, while draw spells reduce the chance of drawing a suboptimal land at any time. Thoroughly modelling ramp and draw cards is essentially impossible: even aside from the number of extant spells, and the sequencing difficulties outlined in 9.2.1 , exact draw mechanisms often require modelling opponent behaviour - consider, for example, "Rhystic Study", one of the most popular draw spells in commander, which draws cards contingent on spells cast by the opponent.

One way to introduce ramp and draw would be to have the player identify ramp and draw spells in their deck, and give them a set of options, ie: "X card draws (1/2/3/...) cards (When played/at the start of turn/whenever you cast another spell), with the opening for the user to approximate how many cards they would expect to draw from this spell. While this would not address the sequencing issues, it may lead to more deck-specific manabase assessments. Such an approach could also be a viable way to model other strategic considerations: IE, "counterspells" (cast in response to an opponent casting a spell) could be marked as such to suggest that the deck may have no reason to cast them on an early turn, when threats are small. However, this would introduce significant frontend complexity, and given that LandFill's optimization algorithm returns a viable automatic manabase rather than a consistently optimal one, I consider incorporation of strategic minutiaie like that to be non-essential for the initial launch.

9.2.2 Queueing Lumps

Recall that the Simulator does not plan lumps to be played on future turns, and in some cases, this may lead it to make suboptimal plays. One potential solution here would be to, rather than generating lumps each turn, generate from the opening hand a queue of consecutive land plays and lump plays designed to maximize mana over forthcoming turns, recalculating this queue with the addition of each new card drawn.

9.2.3 Tiebreaker Metrics

While the LandPrioritization allows some crude tie-breaking between, for example, lands that require a life payment versus lands that do not, one way to ensure a more consistent manabase optimum for a particular deck would be to identify lands that perform within a margin of error of one another, and introduce a tie-breaker variable. This was attempted in an early model: each turn, every land was scored according to the number of playable lumps that turn, regardless of their size, and a land that facilitated the playing of a wider range of lumps could sometimes be chosen ahead of one that wasted slightly less mana if the difference was marginal. I did not implement this in the final version of the project, as exhaustively testing this after a new land was played increased runtime notably; however, it as a metric is less significant for future consideration as the general tie-breaker structure. Some Tie-Breakers - ie, the likelihood of being able to cast a commander of cost N when N lands are drawn, or something as simple as card price - could be ranked by the user in terms of preference. Others could be used to expand LandFill's sensitivity to the advantages of specific builds. As mentioned in SECTION, LandFill does not reward lands that ramp the player; yet this is on occasion very relevant, and would become moreso if ramp spells were included in the manner modelled above (in which, for example, the average mana expenditure of a land may

increase significantly if it facilitates early playing of ramp spells). A future version may revert to mana expenditure as an initial objective function with the understanding that this will be typically tied, and then use wasted mana as a tie-breaker from there.

References

[1] Chris Alvin et al. "Toward a competitive agent framework for magic: The gathering". In: *The International FLAIRS Conference Proceedings*. Vol. 34. 2021.

[2] Rahul Awati. *Iterative Development*. 2023. URL: https://www.techtarget.com/searchsoftwarequality/definition/iterative-development (visited on 08/30/2025).

[3] Alex Churchill, Stella Biderman, and Austin Herrick. "Magic: The gathering is Turing complete". In: *arXiv preprint arXiv:1904.09828* (2019).

[4] German Cocca. *What is Memoization? How and When to Memoize in JavaScript and React*. 2022. URL: https://www.freecodecamp.org/news/memoization-in-javascript-and-react/ (visited on 09/01/2025).

[5] Jason Coles. *Do You Get A Free Mulligan In Commander?* 2021. URL: https://mtgrocks.com/do-you-get-a-free-mulligan-in-commander/ (visited on 09/01/2025).

[6] Alexander Esche. *Mathematical Programming and Magic: The Gathering®*. Northern Illinois University, 2018.

[7] Pedro Furtado. *What Does Strictly Better Mean in MTG?* 2024. URL: https://draftsim.com/mtg-strictly-better/ (visited on 08/30/2025).

[8] Charlie Hall. *Commander: The definitive history of Magic's most popular format*. 2020. URL: https://www.polygon.com/2020/5/28/21266763/magic-the-gathering-commander-origins-elder-dragon-highlander-alaska-menery/ (visited on 08/30/2025).

[9] Josh Lee Kwai Jimmy Wong. *Commander Mythbusters — The command Zone 335 — Magic: The Gathering Commander*. 2020. URL: https://www.youtube.com/watch?v=Pr91Crzl7DI (visited on 08/14/2025).

[10] Frank Karsten. *Mana Bases in Ixalan Standard*. 2017. URL: https://web.archive.org/web/20200330115445/https:/www.channelfireball.com/articles/mana-bases-in-ixalan-standard/ (visited on 08/13/2025).

[11] Frank Karsten. *Should You Play Tapped Duals in 2-Color Limited Decks?* 2018. URL: https://web.archive.org/web/20201109011137/https:/www.channelfireball.com/articles/should-you-play-tapped-duals-in-2-color-limited-decks/ (visited on 08/13/2025).

[12] Frank Karsten. *What's an Optimal Mana Curve and Land/Ramp Count for Commander?* 2025. URL: https://www.tcgplayer.com/content/article/What-s-an-Optimal-Mana-Curve-and-Land-Ramp-Count-for-Commander/e22caad1-b04b-4f8a-951b-a41e9f08da14/ (visited on 08/12/2025).

[13] Tjalling C Koopmans and Martin Beckmann. "Assignment problems and the location of economic activities". In: *Econometrica: journal of the Econometric Society* (1957), pp. 53–76.

[14] Silvano Martello and Paolo Toth. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., 1990.

[15] Nicholas Metropolis and Stanislaw Ulam. "The monte carlo method". In: *Journal of the American statistical association* 44.247 (1949), pp. 335–341.

[16] Beth Moursund. *Playing Your Pet: Rough-Testing a Magic Deck*. 2010. URL: https://web.archive.org/web/20100902160143/http://www.wizards.com:80/Magic/Magazine/Article.aspx?x=mtg/daily/feature/106 (visited on 08/12/2025).

[17] Leah Nguyen. *A Brief Guide to Database Normalization*. 2023. URL: https://medium.com/@ndleah/a-brief-guide-to-database-normalization-5ac59f093161 (visited on 09/01/2025).

[18] Littleton Riggins. *An Introduction To MTGJSON*. 2020. URL: https://triple-equals.medium.com/an-introduction-to-mtgjson-b88dbf65572 (visited on 09/01/2025).

[19] Dana Roach. *Superior Numbers - Let's Bounce!* 2022. URL: https://edhrec.com/articles/superior-numbers-lets-bounce (visited on 08/19/2025).

[20] Alex Robert. *What is Property-based Testing?* 2021. URL: https://www.mayhem.security/blog/what-is-property-based-testing (visited on 08/31/2025).

[21] Mark Rosewater. *Get Ready to Dual*. 2017. URL: https://magic.wizards.com/en/news/making-magic/get-ready-dual-2017-02-27 (visited on 08/30/2025).

[22] Said Salhi and Jack Brimberg. "Neighbourhood Reduction in Global and Combinatorial Optimization: The Case of the p-Centre Problem". In: *Contributions to Location Analysis: In Honor of Zvi Drezner's 75th Birthday*. Springer, 2019, pp. 195–220.

[23] Edward Allen Silver. "An overview of heuristic solution methods". In: *Journal of the operational research society* 55.9 (2004), pp. 936–956.

[24] Editorial Team. *Verification and Validation in Software Testing*. 2025. URL: https://www.browserstack.com/guide/verification-and-validation-in-testing (visited on 08/30/2025).

[25] Colin D Ward and Peter I Cowling. "Monte Carlo search applied to card selection in Magic: The Gathering". In: *2009 IEEE Symposium on Computational Intelligence and Games*. IEEE. 2009, pp. 9–16.

[26] Peter C Wright and Andrew F Monk. "The use of think-aloud evaluation methods in design". In: *ACM SIGCHI Bulletin* 23.1 (1991), pp. 55–57.

[27] Allen Wu. *Using Monte Carlo Simulation to Improve Your Manabases*. 2018. URL: https://article.hareruyamtg.com/article/article_en_405/?lang=en (visited on 08/13/2025).

[28] Stelios H Zanakis and James R Evans. "Heuristic "optimization": Why, when, and how to use it". In: *Interfaces* 11.5 (1981), pp. 84–91.