LandFill

A Magic: The Gathering Manabase Generation App

By

Leah Francesca Liddle

Department of Engineering Mathematics
University of Bristol

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Science in the Faculty of Engineering.

September 2025

Word count: ten thousand and four

Executive Summary

This report documents the development of LandFill, a website for use by players of a deck-building trading card game called Magic: The Gathering (MTG). LandFill automates the selection of resource-providing "land" cards, used to play resource-demanding "spell" cards; the resource in question is called "mana". Over 1000 land cards have been printed, only around 38 of which can be included in any given deck. If inputted a decklist with no land cards, it selects land cards for it. Some land cards produce multiple type of mana but incur a gameplay penalty when played that limits the speed or flexibility with which these mana types can be accessed. Other lands can only produce one type of mana but incur no such penalties. Selecting appropriate lands is therefore a combinatorial optimization contigent on both the distribution of how cards in the deck require mana of different types, and how well the deck is able to accommodate the penalties of different land cards. While many players select lands based on intuition and preference, some best-practices in Land Selection have been determined by professional MTG player and Mathematics Ph.D Frank Karsten, whose work has been based on experiments run using Monte Carlo search. LandFill adapts this approach into a flexible and user-friendly application that can assess cards for a specific deck. Beginning with Karsten's heuristic that a victorious MTG player is usually the one who spends the most total mana, LandFill implements an extemely simple MTG player AI who attepts to allocate land cards efficiently to spend as much mana as possible. Over many simulated games, it determines an approximate performance score for the deck. Using this performance score, LandFill is able to implement a Steepest-Ascent Hill Climbing algorithm to identify a list of high performing lands. While tests of output consistency show that LandFill is not capable of establishing a definitive optimum, user tests found it to be capable of assembling a comparable or better list of lands than they would themselves in less time and with reduced effort.

To summarize:

- I have created a stripped down Magic the Gathering Simulator capable of assigning an approximate performance score to a deck, and from there via Monte Carlo search, a Hill Climbing Optimization routine that uses this data to generate a list of lands for a deck, and a usable interface that allows this to work as a WebApp. How this system functions is outlined in PAGES.
- In doing so, I have built on existing methods used by professional Magic: The Gathering analysts, notably Frank Karsten, and existing deckbuilding support apps, outlined in PAGES.
- User testing strongly suggests that the resulting product makes it easier to build decks to a higher quality, as outlined in PAGES.

Dedication and acknowledgements

I am hugely grateful for the expert advice of my supervisor John Lapinskas and for the support of my partner Rosie Solomon, and for my parents, Elizabeth and Peter Liddle, who provided support and advice on statistical analysis. Thanks also to my tutor, Sarah Connolly, for helping me to keep a level head. And, of course, I owe a deep thank you to my Magic: The Gathering community, particularly to Ben Hammond, who pointed me to some useful algorithms, and to Ollie Gibb, who first taught me to play.

Author's declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

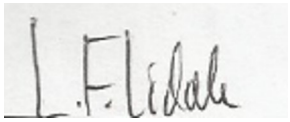
SIGNED:  DATE: .....

Table of Contents

List of Tables

Table **Page**

List of Figures

Figure **Page**

Chapter 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 [1].

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's manabase should be chosen to maximize the probability that the player has access to 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. It models the selection of lands for a deck as a combinatorial optimization problem, where a given manabase yields a measurable performance score for the deck as a whole. This performance score is necessarily approximate: Churchill *et al* have demonstrated that, as it is possible to construct a Turing Machine within an MTG game whose halting is the necessary condition for a player's victory, a deck's winning strategy is indeterminable [2]. Therefore 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 [3] provides an opening for approximating a performance score via Monte Carlo search. 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. Users may input a deck's spell cards into this system using a web interface and receive a completed decklist that has been broadly optimized within their set preferences. Initial user testing suggests that users are able to use LandFill to generate manabases of a comparable or greater quality to those they

can assemble themselves in less time and with less effort.

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 as well as areas for future development. I will conclude with an evaluation of the strengths and limitations of the app.

Chapter 2

Problems and Proposed Solutions to MTG Manabase 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.

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 are called "ramp" spells. A player wins by using spells to reduce their opponent's "life points" to zero.

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. Fig. 2.1 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, "a Forest" is any card with the Forest subtype, while "a Basic Forest" is 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

Figure 2.1: A spell card

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 game state. In the example given, a player's hand contains a spell that requires R *and* U *and* G, a land that produces R *or* U per tap, a Basic Mountain and a Basic Island, while the battlefield contains a land that produces U *or* G per tap.

$$\text{Battlefield} = \Big[\text{Land}(UG)\Big]$$

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

Lands that produce two colours are called "Dual 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, if card A is better than card B in at least one way, and worse in no ways, A is considered "strictly better" than B [4]. 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 [5]. 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, such as Fetch Lands, which produce no mana but deploy another land from the library when played. 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 points 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," while Tropical Island is an "Original Dual Land." The prevalence of cycles such as Check and Fetch lands (see Fig. 2.2), additionally means that a cycle with basic landtypes may be considered strictly better than an identical cycle without. Lands which always enter tapped are called "taplands"

Figure 2.2: 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

This makes optimization complex. Consider the lands Prairie Stream and Deserted Beach, both referenced in Fig. 2.2. Since any situation in which Prairie Stream would enter untapped would also allow Deserted Beach to enter untapped, yet the same is not true vice-versa, Deserted Beach is, taken in isolation, a stronger land. Consider, however, the following situation (all named nonbasic land cards are depicted in Fig 2.2). Two players play two identical UW decks with identical UW manabases consting of Flooded Strand (a Fetch Land), Hallowed Fountain (a Shock Land), 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:

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

Since both players need copious amounts of both U and W mana, they should use Flooded Strand (see Fig. 2.2), which as a Fetch Land allows the player to pull another land from their library and then shuffle, to find in their library 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 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 may play 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 respective

probabilites of beginning a turn with N lands on the battlefield, where either:

- N is greater than 1 and at least two lands are basic, or
- The player has 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 performance, 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 curve of 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: since there are more solutions than can be feasibly investigated, 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".

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 an opponent [6]. Simulating all possible spell card interactions is an unfeasible undertaking here, and not necessarily a helpful one, given MTG's Turing Completeness [2]. There is, however, a useful heuristic for our purposes: in a game, the winning player is typically the one who spends the most mana [3]. The simulator, therefore, need only try to spend as much mana as possible each turn. 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 S of spells with a CMC as close to M as possible.
7. Repeat steps 3-6 for each turn of the simulated game.

Complexity is introduced in any situation in which multiple lands allow for the spending of M mana, such as in the sample hand drawn in the previous section in which I analysed Prairie Stream and Deserted Beach. In the absence of any spells of CMC=1, playing either the Basic Plains or the Flooded Strand yelds $M = 0$. The simulator, therefore, must strategize for expenditure on future turns.

2.2.2 LandFill Optimization

In a Monte Carlo search, solution space is explored by taking the outputs of stochastic processes, and sampling the resulting distribution to approximate the typical values of that process [7]. Performance P of a manabase could be conducted by giving the Simulator a deck, running many games with it, and recording the deck's average performance as the sample average value of P, approximating the true average by the law of large numbers. P is then 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 and each dimension represents the quantity of a given land in the manabase. 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 $N(\underline{x}_c)$ — i.e., all solutions which differ only by some simple transformation $\underline{x}_c$ — are examined. The first solution to return a higher value is adopted as the new $\underline{x}_c$ [8]. 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.
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 [8]. However, given that the simulator itself can only loosely approximate the decision-making process of an actual MTG player, LandFill falls into the category of optimization problem for which "the optimal solution", as described by Zanakis and Evans, "[is] only academic" [9]. 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 [10]. 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 [11]. The Simulator and Optimizer are two of four component subsystems to LandFill:

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

Possibly clearer and shorter: "Since each component utilizes the one before it, LandFill naturally lends itself to an Iterative Development process, rather than "Waterfall Development", in which all testing is withheld until product completion [HYPERLINK ł"11]. Waterfall Development would preclude adjusting upstream components in the light of design shortcomings discovered a downstream component. "

Since each component utilizes the one before it, LandFill suits an "Iterative Development" process, rather than the alternative "Waterfall Development" system in which testing is done after product completion [11]; this would preclude adjusting upstream components to accommodate unexpected behaviour in downstream ones. 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 and User Research

Validation Testing was conducted via a series of user tests after development. Details of the methodology and results of this are in Chapter 8.

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 therefore be flexible enough to fit into a range of different deck construction strategies, and be able to accommodate, via user input, game-extrinsic manabase restrictions such as price and preference. To account for this during development, I conducted user research via a series of interviews and exercises with MTG players. This is detailed in Chapter 3.

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 rests on Ward and Cowling's landmark 2009 paper, "Monte Carlo Search Applied to Card Selection in Magic: The Gathering". Ward and Cowling hypothesize a that methods used to automate other imperfect information games, such as Bridge and Poker, may be applied to MTG [12][13][14]. 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 engine, features an AI opponent, "Sparky". Although useful in gameplay tutorials, Sparky presents no challenge to experienced players [14].

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 [12]. 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 allocate. This puts it somewhat outside the realm of scholarship inaugurated by Ward and Cowling, which is concerned with how to encode strategy given the complexity of gameplay and the limited information. Ward and Cowling's automated players use Monte Carlo analysis to examine possible outcomes of combat between creature spells once cast [12], while Alvin *et al* explore graph theoretic representation of card synergies [14]. Neither explore efficient use of mana, as the choice of spells in their models 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 [13]. LandFill's simulator represents a 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. However, their results are inconclusive. As their work relied on a full automated AI player — which, as established, typically underperforms 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 search to analyse a given deck [15]. A central figure here is Frank Karsten. His seminal series of articles, *How Many Sources Do You Need to Consistently Cast Your Spells?* [16], use Monte Carlo search to produce a grid (see Fig. 2.3) outlining how many lands of colour C you need depending on the most demanding spell

Sources	C	1C	CC	2C	1CC	CCC	3C	2CC	1CCC	CCCC	4C	3CC	2CCC	1CCCC	5C	4CC	3CCC	5CC	4CCC	Sources
6	58.4%	63.1%	19.4%	69.8%	25.5%	4.5%	76.6%	33.4%	7.3%	0.7%	82.4%	42.1%	11.3%	1.5%	87.4%	51.1%	16.7%	60.0%	23.3%	6
7	64.6%	69.3%	25.4%	76.0%	32.9%	7.3%	82.3%	42.0%	11.5%	1.6%	87.5%	51.9%	17.5%	3.0%	91.6%	61.5%	25.2%	70.4%	33.8%	7
8	70.3%	74.7%	31.5%	81.0%	40.2%	10.8%	86.8%	50.6%	16.5%	2.9%	91.3%	61.1%	24.7%	5.4%	94.6%	70.7%	34.3%	78.9%	44.8%	8
9	75.1%	79.3%	37.8%	85.2%	47.5%	14.8%	90.4%	58.4%	22.7%	4.7%	**94.1%**	69.3%	32.7%	8.7%	**96.6%**	78.3%	44.3%	85.7%	55.8%	9
10	79.4%	83.3%	43.9%	88.6%	54.4%	19.5%	**93.1%**	65.9%	29.2%	7.3%	96.1%	76.2%	41.0%	13.1%	97.9%	84.5%	53.7%	90.5%	65.9%	10
11	83.0%	86.5%	50.1%	91.4%	61.0%	24.7%	95.1%	72.4%	36.2%	10.4%	97.5%	82.1%	49.5%	18.3%	98.8%	89.3%	62.8%	94.0%	74.7%	11
12	86.3%	89.3%	55.9%	**93.6%**	67.1%	30.4%	96.7%	78.3%	43.5%	14.3%	98.4%	86.9%	57.8%	24.5%	99.3%	92.9%	71.1%	**96.4%**	82.0%	12
13	89.0%	**91.7%**	61.5%	95.3%	72.7%	36.2%	97.8%	83.2%	50.8%	18.8%	99.1%	90.9%	65.7%	31.5%	99.7%	**95.5%**	78.5%	98.0%	87.7%	13
14	**91.3%**	93.6%	66.7%	96.7%	77.9%	42.3%	98.6%	87.4%	58.0%	24.1%	99.5%	93.8%	72.9%	39.1%	99.8%	97.3%	84.6%	99.0%	92.3%	14
15	93.2%	95.1%	71.7%	97.7%	82.3%	48.5%	99.1%	90.9%	65.0%	30.0%	99.7%	**96.0%**	79.4%	47.0%	99.9%	98.5%	89.6%	99.5%	95.4%	15
16	94.9%	96.4%	76.2%	98.5%	86.2%	54.8%	99.5%	**93.7%**	71.5%	36.3%	99.9%	97.6%	85.0%	55.1%	100.0%	99.2%	93.4%	99.8%	**97.6%**	16
17	96.2%	97.5%	80.3%	99.1%	89.7%	61.0%	99.7%	95.8%	77.7%	42.9%	99.9%	98.6%	89.6%	63.3%	100.0%	99.7%	**96.1%**	99.9%	98.8%	17
18	97.3%	98.2%	84.0%	99.4%	**92.5%**	66.9%	99.9%	97.4%	83.2%	50.1%	100.0%	99.3%	93.3%	71.1%	100.0%	99.9%	97.9%	100.0%	99.6%	18
19	98.2%	98.8%	87.3%	99.7%	94.8%	72.7%	100.0%	98.5%	87.9%	57.3%	100.0%	99.7%	**96.1%**	78.5%	100.0%	100.0%	99.1%	100.0%	99.9%	19
20	98.9%	99.3%	90.3%	99.9%	96.7%	78.2%	100.0%	99.3%	91.9%	64.8%	100.0%	99.9%	98.0%	85.2%	100.0%	100.0%	99.7%	100.0%	100.0%	20
21	99.4%	99.6%	**92.9%**	100.0%	98.1%	83.2%	100.0%	99.7%	**95.1%**	72.3%	100.0%	100.0%	99.2%	90.8%	100.0%	100.0%	99.9%	100.0%	100.0%	21
22	99.7%	99.8%	95.2%	100.0%	99.1%	88.1%	100.0%	99.9%	97.6%	79.5%	100.0%	100.0%	99.8%	**95.2%**	100.0%	100.0%	100.0%	100.0%	100.0%	22
23	99.9%	99.9%	97.1%	100.0%	99.7%	**92.4%**	100.0%	100.0%	99.2%	86.6%	100.0%	100.0%	100.0%	98.4%	100.0%	100.0%	100.0%	100.0%	100.0%	23
24	100.0%	100.0%	98.7%	100.0%	100.0%	96.5%	100.0%	100.0%	100.0%	**93.5%**	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	24
25	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	25

Figure 2.3: 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 [16]

requiring colour C in your deck. Rather than simulate gameplay, Karsten abstracts a spell to its mana cost, and then runs a series of Monte Carlo analyses, with the percentage of lands in the deck capable of producing that spell's colour progressively increasing. Analysing each possible single-colour mana cost of CMC M this way, Karsten determines the number of lands of its colour required in a deck to have a 89% chance of being able to cast that spell on the turn during which the $M'th$ land is played.

While Karsten engages loosely with how to fit multi-coloured spells and lands that may enter tapped into this framework, he acknowledges that in both cases his data does not provide firm answers. To understand whether a 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 may 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) [17] and the Standard format around the release of the Ixalan Set [18]. In the latter article, Karsten's engagement is limited to one nonbasic land cycle — Check Lands — and only to their 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 article 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 to deck-building best-practices. LandFill, therefore, is an adaptation of 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 [19] (screenshot in Fig. 2.4), 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 useful in completing a deck after nonbasics have been added by the player, it has significantly

Figure 2.4: 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.

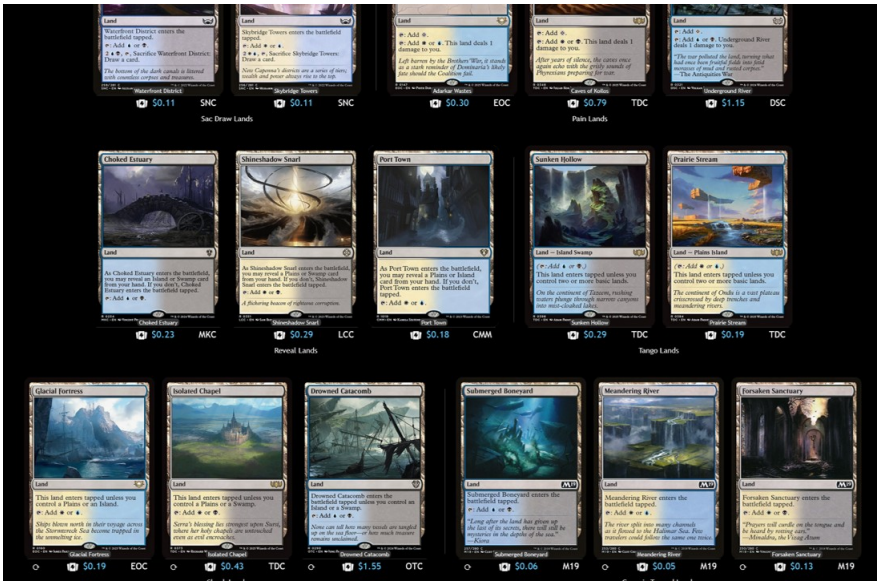


Figure 2.5: A zoomed-out screenshot of an excerpt from the ManaGathering page for a WUB deck. Note the WU, UB and BW lands arranged by cycle.

less functionality than LandFill. It is also only accessible to decks constructable via the limited set of cards available on MTG Arena.

The second is the website ManaGathering [20] (screenshot in Fig. 2.5), 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, making it easier for players to recall useful lands and helping them choose strong ones. A broad success metric for LandFill is that it should return a better result than a player using ManaGathering could in a comparable time.

Finally, the website Archidekt [21] (screenshot in Fig. 2.6) combines a basic-land allocator *à 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

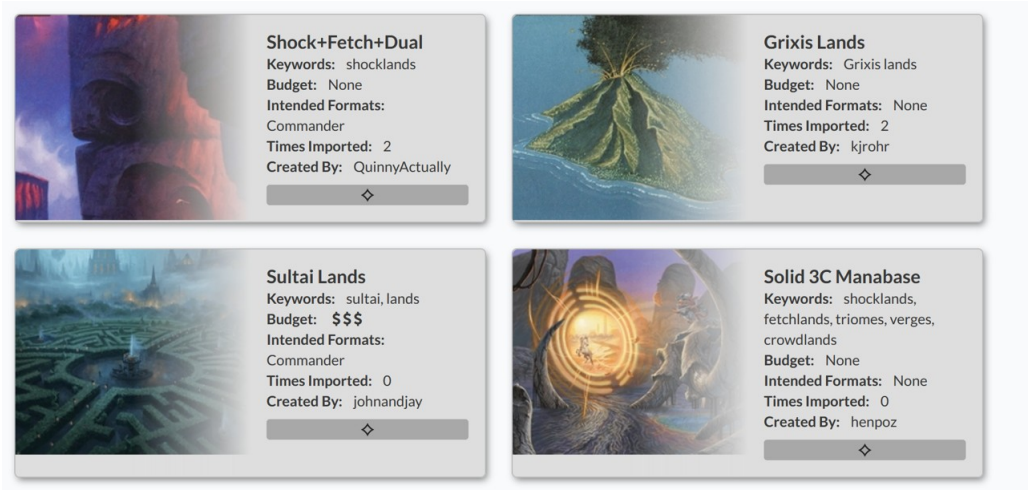


Figure 2.6: A screenshot of community created land packages on Archidekt, categorized by, among other things, price and color. These can be imported into a decklist.

colour-appropriate lands of a reasonable price by importing a 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.

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 [22], TappedOut [23] and Deckbox [24], 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 will be covered in §3.1, 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 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 [25] as a backend web framework as its simplicity made it both easy to learn and reputably faster; contenders like Django [26] are made both slower and more complex by their abundance of features (FastAPI [27], potentially lighter and faster than Flask, was discarded due to its smaller userbase and thus relative paucity of learning resources). React [28], 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.

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

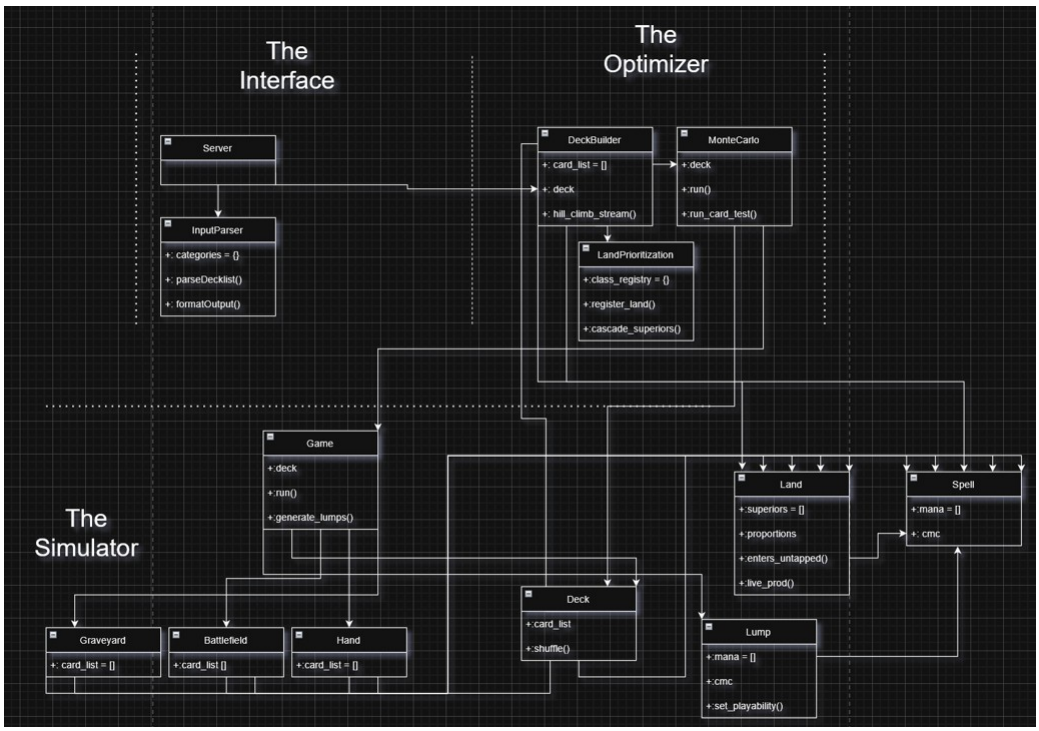


Figure 2.7: Class diagram of major classes in LandFill.

use of Object Relational Mapping to store a database of MTG cards, the user needs to only read the database. For this reason, SQLAlchemy [29], an ORM esteemed for rapid performance at the cost of easy data amendment [30], was the obvious choice.

During development, I sometimes used the Large Language Model ChatGPT [31]. The majority of my usage was to help identify bugs in my code. I also used it to suggest libraries or standardized testing methodologies (i.e., the Kano Questionnaire) after providing an outline of my use case; in all these cases, I researched its suggestions thoroughly after recommendation. Later in development, I commissioned it to write sections of code for me based on a pseudocode outline.

2.5.1 LandFill Structure

Excluding the Database, accessed by other objects via SQLAlchemy, the components outlined in §2.3.1 are displayed in Fig. 2.7. Details about the role of each constituent class, and the interactions between them, can be found at the start of the relevant sections.

Chapter 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.

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 [32]. 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 (i.e., 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 [33]. I provided users with a mockup homepage for LandFill, displayed in Fig. 3.1.

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. A generic Kano questionnaire template is displayed

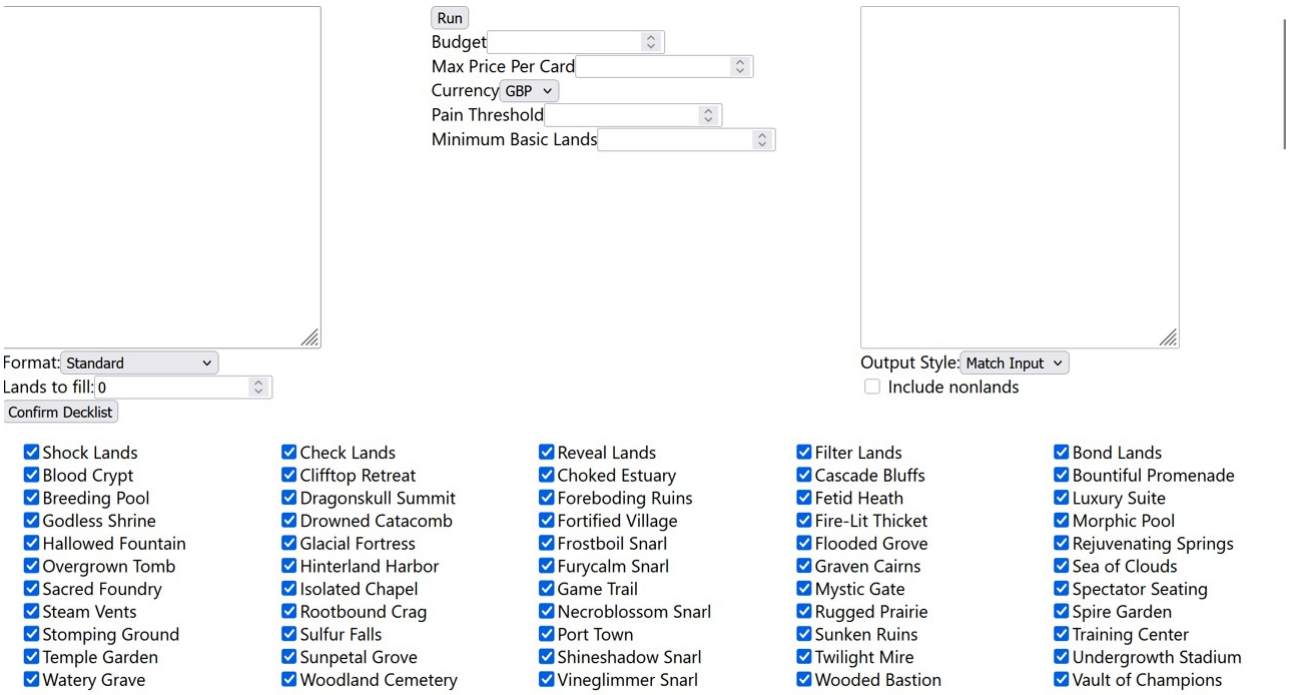


Figure 3.1: Draft front-end used for mockup testing.

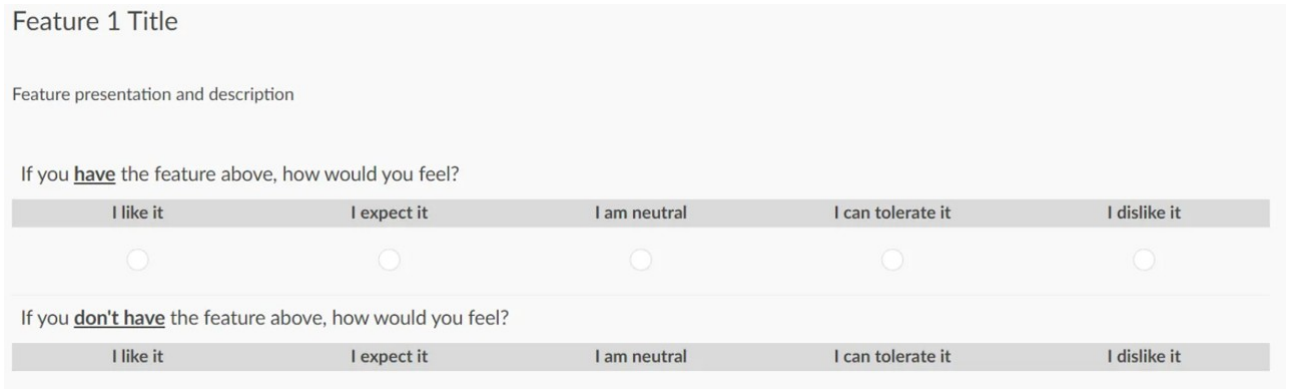


Figure 3.2: A question in a generic Kano questionnaire

in Fig. 3.2 I developed a Kano Questionnaire to determine which features requested by and offered to participants were in widespread demand. I then used a Kano Analysis Marking Table to sort features into five categories based on results: "attractive" features (presence liked, absence neutral/tolerable), "must-have" features (presence expected), "performance features" (presence liked, absence disliked, "indifferent" features (presence and absence neutral), "questionable" features (conflicting thoughts on presence and absence) and "reverse" features (presence disliked)). For my Kano Analysis results, see Fig. 3.3.

3.2 Analysis

To analyse the qualitative data from the Think-Aloud Evaluation and the Interview, I implemented a lightweight version of the thematic analysis principles outlined by Braun and Clarke [34]. I coded interview transcripts to identify recurring themes, which in this context are deckbuilding habits that an app like LandFill should support. As this analysis is intended solely to inform included features, within Braun and Clarke's framework I favour semantic, inductive and essentialist readings of the

No	Feature	Concept Source	Attractive	Must-Have	Performance	Indifferent	Questionable	Reverse
1	Support for decks in the Commander format.	Offered in mockup			3	1		
2	Support for decks in the Legacy format	Offered in mockup	1	1		1	1	
3	Support for decks in the Limited format	Offered in mockup	1	1	1	1		
4	Support for decks in the Modern format	Offered in mockup	1	1		1	1	
5	Support for decks in the Pauper format	Offered in mockup	1	1		2		
6	Support for making a "round trip" between LandFill and Deckbox without reformatting the decklist.	Suggested by participant			1	3		
7	Above but for Archidekt	Offered in mockup			1	3		
8	Above but for Moxfield	Offered in mockup			1	3		
9	Above but for Tappedout	Offered in mockup	1	1	1	1		
10	The ability to mark out lands that a user definitely does not want in the deck.	Offered in mockup	1		2	1		
11	The ability to mark out lands that a user definitely does want in the deck.	Offered in mockup	2	1		1		
12	The ability to set a maximum price for the manabase.	Offered in mockup	2	1			1	
13	The ability to specify how much life a player is happy to lose to lands requiring a life point investment.	Offered in mockup	3		1			
14	A FAQ answering questions about how LandFill makes decisions.	Suggested by participant	2	1		1		
15	A single tickbox excluding all lands which invariably enter tapped.	Suggested by participant	3			1		
16	The ability to exclude off-colour fetches.	Suggested by participant	1			3		
17	The ability to provide a list of lands, and choose recommendations solely from within them.	Suggested by participant	2			1	1	
18	The ability to see ranked performance of the lands in the deck.	Suggested by participant	3			1		
19	The ability to weight lands by preference for higher priority in the manabase generator.	Suggested by participant	3					1
20	Visible card images identifying what cycles a land belongs to when considering what to include and exclude.	Suggested by participant	2	1	1			
21	Visible performance metrics for the deck after optimization.	Suggested by participant	3		1			

Figure 3.3: Results from my Kano analysis; each number counts the number of respondents selecting that option.

data. As a player of Magic: The Gathering myself I have my own approaches to deckbuilding; that extrapolations from my own experience may weight my assessment of this data should be accounted for by the reader. I myself am a Persona B deckbuilder (see §3.2.4), and favour the Commander format with some experience in Standard and Limited.

3.2.1 Format Support

This relates to features 1 — 5 in Fig. 3.3. The formats Limited and Commander were played by all users except one in each case. Other users had experience of Pauper, Modern, Standard and Legacy. However, players in these latter formats commonly expressed doubt that an app such as LandFill would be useful here, as their engagement with those formats rested on netdecking.

A recurring theme, when I asked about deckbuilding approach in Limited, was surprise that the app would be viable in Limited at all. In Limited, decks are built at the site of play from a pool of randomly chosen cards, where there would not be expected to have computer access nor time to input a list of cards.

I will restrict LandFill to the Commander Format during initial development. Three out of four Kano respondents considered Commander to be a “Performance” feature. The inclusion of Limited was considered desirable to some degree by three out of four respondents, but only expected as a minimum by one. As Limited requires much stricter pre-setting of what lands can be included – as there are only so many cards available in the random pool – designing for Limited would be a specialized task; so too for Commander, whose rules are detailed in §5.1. I consider it better to design a honed app for one format and potentially expand in the future, with Commander being the most popular choice.

3.2.2 Support for 3rd Party Databases

This relates to features 6 — 9 in Fig. 3.3. My mockup offered support for TappedOut, Moxfield and Archidekt; one participant also introduced me to Deckbox, of which I had not been previously aware. All Kano respondents said they would dislike the absence of at least one of these services. LandFill must therefore support them all.

Interestingly, during the Think-Aloud Evaluation, while some participants used the dedicated "export" features of these apps, others simply copied the front page of the decklist from the app to LandFill - this is possible for Moxfield and TappedOut. LandFill therefore must be able to parse decklists from both the export function and the front page of these apps. For implementation of this, see §7.1.

3.2.3 Comprehensive Land Exclusion/Preference Options

This relates to features 10, 11, 15, 16, 19 and 20 in Fig. 3.3. Several players identified sub-cycles or groups of cycles of lands that they would want to exclude in all cases by category, without having to go through the lands individually. These were:

- Taplands - lands that always enter tapped under all circumstances.
- Off-Colour Fetch Lands - in, for example, a BUG deck, it may be worthwhile to include a UW Fetch Land, as it is able to get UG or UB Shock Lands. One participant found this aesthetically displeasing.

Both have been implemented. Three out of four Kano respondents considered the first "attractive"; although feedback on the second is mixed, it is simple to implement. Participants also generally found it hard to determine lands to exclude without having reminders of what the card text does. This was considered by all Kano respondants to be an "attractive", "must-have" or "performance" feature, and thus should be implemented. Implementation of all three features is discussed in §7.2.2.

Two participants suggested offering some ability to "weight" lands, to prioritize those that they liked over ones they did not, while keeping options open. Kano data on this was interesting: three respondents felt it to be an attractive feature, but one considered it undesirable. Although I did not implement it, since it was independently suggested by two participants and was otherwise popular with Kano Respondents, I used it as a framework for dealing with mechanically equivalent lands (see §6.1.2).

More broadly, most Kano respondents considered it "attractive" or a "must-have" to be able to specify both lands to *include* and lands to *exclude*. This means that lands must be categorized three ways, with the third being lands for general consideration. This informs the three-column design ultimately used for the frontend, as covered in §7.2.2.

3.2.4 Deckbuilder Personas

Participants 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 appropriate colours from this collection.
- Persona B, who develops a new deck and orders whatever lands they want for it.

This prompted me to ask participants 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 personas 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, it is a viable route for future development of LandFill. However, the initial design covered in this writeup will adhere to Model 2. Since justification of this rests on the ability of LandFill to make suggestions the user may not have thought of, that itself becomes a meaningful success criterion for LandFill. The manner in which that is tested is discussed in §8.1.2.

3.2.5 Visualization of Internal Workings

This relates to features 14, 18 and 21 in Fig. 3.3. Several participants felt that they would not automatically trust the output of LandFill, and suggested these features. The participant who suggested adding a ranked list of lands to the output page pointed out that this would allow them to choose what lands to remove if new cycles were printed, suggesting a possible appropriated use of LandFill as not just a generator of manabases, but as a way of learning more about the behaviour of one's manabase. All these features were considered "attractive" or "must-have" by a majority of Kano respondents, and have been implemented on the frontend as will be covered in §7.2. More broadly, the desire for trust informed both my implementation of live progress updates during optimization (see §7.2.3), and my choice of objective function (see §6.3.4).

3.2.6 Knapsack Problems and simultaneous optimization

This relates to features 12 and 13 in Fig 3.3; regarding question 13, recall from §2.1.2 that some lands are balanced over others by requiring a life point investment, such as Shock Lands compared to Basic Lands.

Both features 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 respectively average life damage and card cost) [35]. Since this would involve simultaneously optimizing multiple areas of deck design, it would represent a sizeable increase in computational complexity for LandFill.

They have thus not been implemented in their pure form. However, both of these features were deemed by Kano Respondents to be attractive to some degree, so LandFill implements a lightweight version of each. LandFill considers life loss to be an indicator of strict worseness when that is encoded into its decision making as covered in §6.1.2. My proxy for overall manabase prices is covered in §7.2.2.

3.2.7 Mulligans

No participant was able to describe any consistent principles on which they based their decision to mulligan. All participants 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.

Chapter 4

The Database

4.1 Database Requirements

The database in which LandFill stores card objects will need regular updating by the site owner, both with newly-printed MTG cards and to keep price data up to date. 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 but 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 [36]. 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 [37], 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 [38] has already been developed. Scrython returns dict objects listing card attributes, referred to within LandFill's 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.

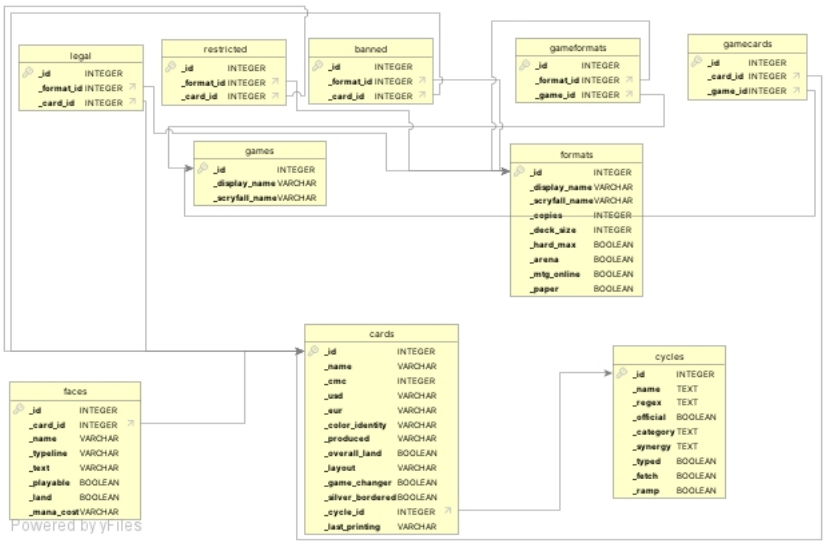


Figure 4.1: Layout of mtg.db

4.3 Database Layout

Since LandFill uses Flask, it is built not on SQLAlchemy but on the slightly more feature-heavy Flask-SQLAlchemy extension [39]. 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. 4.1.

The two upper rows of tables in Fig. 4.1 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`. 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 database contains only a single value per entry [40]. 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: i.e., 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 (i.e., what colours of mana it produces, the card price).

`cards._layout` is one of several strings extracted from the SCO, each of which denotes a different way of formatting face playability (i.e., 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. 2.2 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.

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 [41]. `test_db.py`'s 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 do 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 [41]. `test_db.py`'s 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.

Chapter 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", from which they can be repeatedly cast. Some cards are marked to allow a "partner", a secondary commander also placed in the command zone. Commander decks generally include cards that work synergystically with the commander in gameplay. All Commander decks have exactly 100 cards, including the commander.

Analysis by the Command Zone podcast suggests that the average Commander game runs for 10 turns per player [42]. Unlike other formats, Commander is typically played with more than two players per game. Commander games offer players one "free" mulligan; any further mulligans require them to sacrifice a card from their hand.

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. Each cycle is then a 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, of which Spell and Land are subclasses. The initiation method of each GameCard takes a Card 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 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.

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:

Figure 5.1: Example of a land belonging to the Verge cycle and the implementation of `land.live_prod(game)` in that cycle

Figure 5.2: Example of a land belonging to the Check cycle and the implementation of `land.enters_tapped(game)` in that cycle

- `land.live_prod(game)` — given the current game state, returns the colours of mana a land can produce, expressed as a list of strings (i.e., ["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. 5.1 and Fig. 5.2. 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}]`. `GameCard.mandatory` and `GameCard.permitted` are boolean attributes denoting cards which the user has ordered be included, and cards that may be recommended by the optimizer, respectively.

`GameCard.wasted_games` is a list attribute, usage of which will be covered in §6.4.2

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.
- 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.

- DeckBuilder — 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.

GameCards are stored in `CardCollection.card_list`. The specialized method `CardCollection.give(GameCard, CardCollection)` 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. They take a Deck as an attribute, and feature an overwritten method, `Simulation.run()`, which performs an action and extracts data from it.

- Game — simulates a game of MTG with its deck.
- MonteCarlo — 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. 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 both a Hand 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 §3.2.7, it is not practical to implement realistic mulligan behaviour. LandFill therefore mulligans at most one time (the "Free" mulligan) 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`.

5.4.2 Determining Lumps

LandFill here makes use of the `combinations` function belonging to Python's `itertools` module [43]. 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 [44], 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 ($L = 0$):

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

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 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 when it is 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 returns 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 Basic 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 (i.e., 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, and a lump is played with a CMC as close to $L + 1$ 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, the very 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. I tried this in several ways 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 $3 \times 3 \times 2 \times 2 \times 2 \times 2 \times 2 \times 1 \times 1 = 288$ 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. 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.

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 [45]. In Python, this cache can be persisted across sessions via the Pickle library [46]. Because any two untapped lands that produce (for example) UB are functionally identical in this context, the number of combinations to memoize appears 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.

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 [47]. Koopmans and Beckman [48] 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 `Lump.cost` 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 removed 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

Table 5.1: Costs set for each land.

`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 GameCards 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 MonteCarlo and DeckBuilder 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()` and `filter_by_most_produced()` 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. 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.

5.6.5 Bond Lands

Bond Lands are a cycle of dual lands without basic landtypes. They enter tapped unless a player has two or more opponents, making them extremely strong in Commander. Bond Lands, in the Simulator, always enter untapped.

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. 2.2 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 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 n'th 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, $P(1)...P(n)$. $P(1)$ is removed from $\underline{L}_n f$, and the SubLands of $F(1)$ are added in its place.

4. Repeat steps 2 and 3, starting with $\underline{L}_n f$ each time, until all FilterLands 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, return to step 2 generate a new permutation with new values of $F(1)$ and $F(n)$

9. If all permutations of $\underline{L}_f$ have been tried, the Lump is not castable.

Although this requires testing a large number of permutations if a Lump is not castable, the rarity of having a large number of FilterLands simultaneously in play means that this does not incur a noticable runtime penalty.

5.7.3 Dual-Faced Lands

A Dual-Faced Land has two faces, each of which are playable, and each of which tap for exactly one colour of mana. All Dual-Faced Lands enter the battlefield untapped. The DualFacedLand objects is assigned two SubLands, each capable of producing one of its colours, and an attribute, `DualFacedLand.committed`, initialized to `None`. When tapped for "Gen" or "None", if `DualFacedLand.committed = None`, one of the DualFacedLand's SubLands is selected via `Game.filter_by_most_produced()` in a similar manner to FetchLands. `DualFacedLand.committed` is set to that SubLand. When tapped for coloured mana, `DualFacedLand.committed` is set to the SubLand capable of producing that colour. If `land.committed != None`, the land is tapped as though it itself were the SubLand returned by that attribute. The Battlefield object sets `DualFacedLand.committed` to `None` when it returns it to another zone. Like FetchLands, a DualFacedLand has a lower weighting to force `SciPy.linear_assignment()` to map it to "Gen" or "None" mana if possible when a Lump is played.

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.

Chapter 6

The Optimizer

6.1 Constituent Classes

6.1.1 DeckBuilder and MonteCarlo

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

6.1.2 LandPrioritization

At each increment of the Hill Climbing Algorithm, when a land is removed from the deck, DeckBuilder does not test any candidate replacement land unless all lands that are strictly better than it are already either in the deck or forbidden from consideration by the player. This is for three reasons:

- As will be discussed in §6.2.1, it is not always possible to determine a conclusive performance diference between two decks with very similar card lists. Insofar as some lands can be decisively considered to be better than others, it is worth utilizing this information as a backstop against fluke results.
- As noted in §3.2.6, some lands are balanced via a life point penalty. This does not, however, impact their performance within the Simulator in any way. A land, therefore, should not be tested against a land which is only worse than it because of a life point penalty.
- Since LandFill uses a Hill Climbing algorithm, the performance P of the deck must be repeatedly determined at each increment. The number of separate P scores to be taken at each increment increases with the number of lands that could be added to the deck. This is time consuming.

DeckBuilder is therefore given a LandPrioritization object, which maps out strict superiority/inferiority relationships. Land A is considered strictly better than land B if it is capable of producing all colours that B does and meets at least one of the below criteria, as long as B does not meet any of the same criteria *vice versa*:

- Produces additional colours that B does not.
- Never enters tapped, while B sometimes does.

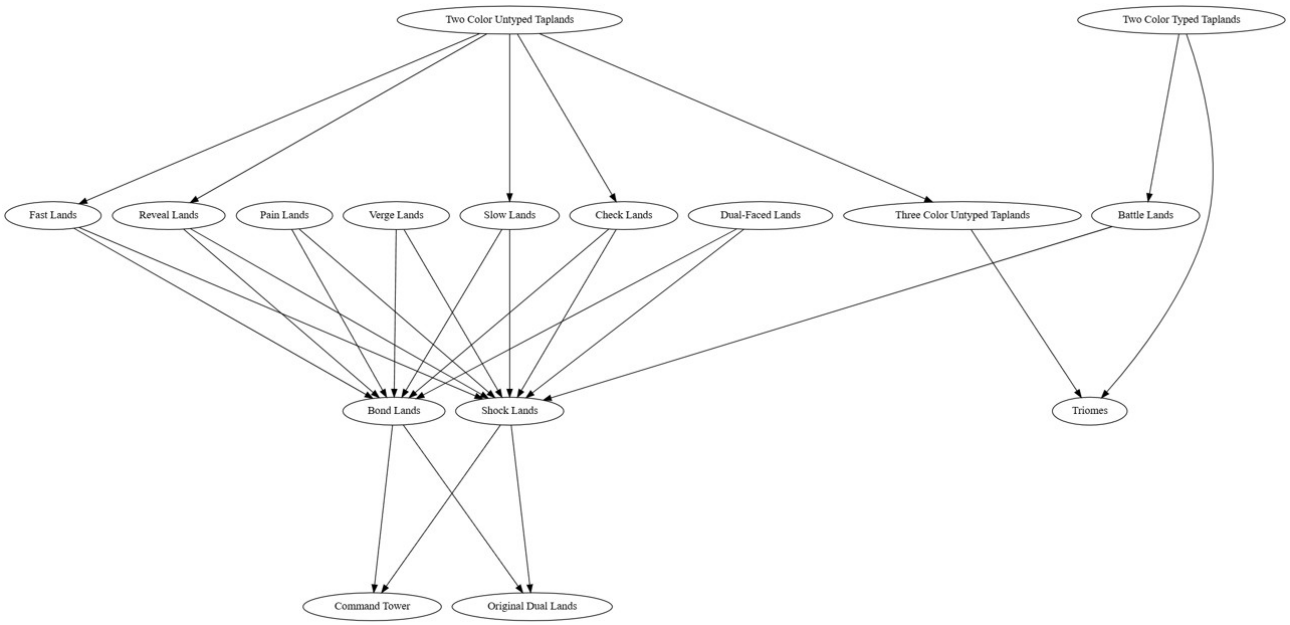


Figure 6.1: Digraph of strict betterness relationhips between all cycles currently handled by LandFill, where each arrow points to the superior land. Note that some labels (i.e., Three Colour Untyped Taplands) refer not to single cycles but to groups of mechanically equivalent cycles which differ in ways that do not pertain to mana generation.

- Sometimes enters untapped, while B never does.
- Requires a life point investment that B does not.
- Has Basic Landtypes, while B does not.
- Always produces its colours, while B only produces some of them under certain conditions.

Strict betterness is mapped in Fig. 6.1. While most of these nodes represent distinct cycles, some represent mechanically identical categories, i.e., Two Colour Typed Taplands refer to any cycle of taplands with two Basic Landtypes. Command Tower is also not a cycle but an individual land unique to the Commander Format that always enters untapped and produces any colour. Pain Lands produce C for free or one of two coloured mana for a single life point. Fetch Lands and Filter Lands, given their mechanical distinction from other lands, have no strict superiors or inferiors.

This relationship is encoded into the LandPrioritization object via a dict, where a Land child class is a key mapping to a list of all superior classes. All Land objects belonging to the DeckBuilder that are permitted for consideration by the user are fed into `LandPrioritization.register_land()` and stored within the LandPrioritization. After all Land objects are registered, `LandPrioritization.cascade_superiors()` assigns each a list of all superior Lands that have also been registered, stored as an attribute of the Land object. This process overlooks any absent lands that the user does not want LandFill to consider: if the user does not want to include Battle Lands, for example, any two-colour taplands with Basic Landtypes will recognise an equivalently coloured Shock Land as a superior.

For equivalent categories of lands — i.e., two-colour typed taplands — I implemented a stripped down version of the weighting system discussed in §3.2.3. Users are given an option on the Interface to prioritize cycles within these categories over othrs within the same category.

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. At every increment it examines *all* inputs in the neighbourhood and chooses the one representing the highest rate of improvement [8]. 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 that leads to the highest performance.

This is, however, time consuming to calculate, as it means that every candidate land must be tested once for each land in the deck it could replace. This adds a search space of $N \times M$ to each increment, where N is the number non-mandatory lands in the deck and M is the number of permitted lands in the DeckBuilder with all strict superiors in the deck. However, it is possible to divide this heuristically into two steps. First, identify the worst land in the deck. Second, identify the best land to replace it with. The manner in which LandFill does this is outlined in §6.4.2.

6.2.2 Other Options

Simulated Annealing is a variant of the Neighbourhood Search algorithm that avoids becoming trapped in local 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 [8]. Since LandFill's success criterion is simply the creation of equivalently optimized list of lands to that a human player could create with similar effort, I consider a local maxima to be an acceptable return value.

I did explore one approach that 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. The LandPrioritization object is a simple example of this. 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. Therefore, it may not be necessery to test every candidate Check Land each time one Basic Land is removed from the deck, as this will only decrease the viability of Check Lands. Check Lands will only become relevant again at a later stage in optimization, once better nonbasic options have been exhausted.

To take advantage of this, I trialled an algorithm in which a declining temperature T governed the number of lands swapped out of the deck at each increment: initial steps swapped out the worst T lands in the deck for the best performing T lands tested individually. As the value of T decreased, each swap became smaller, and thus each land 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 [49], and the same runtime improvement could ultimately be achieved more simply, and with greater user consent, by simply advising users to automatically mark high performing cycles such as Shock Lands and Fetch Lands for mandatory inclusion unless they specifically did not want them.

6.3 Choice of Objective Function

6.3.1 Overview

The objective function of the optimizer is some representation of the performance of the deck. To explore options, I ran the MonteCarlo object on four sample decks containing the same nonland cards in colours BUG:

- 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.

6.3.2 Use of "Wasted Mana"

While our initial heuristic was that a good manabase should allow the most mana to be spent, there are problems with simply assessing a deck based on total expenditure. In his analyses, referenced in §2.4, Karsten proposed three underlying assumptions [16] 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 CMC of the largest Lump determined that turn whose CMC is equal to or less than the number of lands on the battlefield at the turn's conclusion, tapped or not.
- $C(c)$, the CMC of the Lump that was cast this turn.

The sum of all values of $C(l) \dots C(c)$ in a game is 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, but always assumes enough lands have been drawn.

6.3.3 Initial Analysis

Data from a trial of four decks is presented in Fig. 6.2. Note that:

- Modal values of W are consistently zero, while BasicDeck's median W value of 1 is the highest of any deck, indicating that even at low levels 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 games. For all decks, the second most common value of W after 0 is 1.

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 6.2: Outputs in terms of wasted mana for each deck

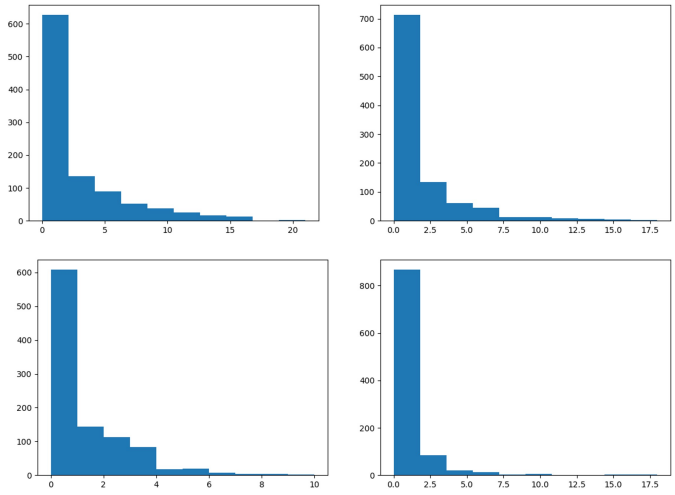


Figure 6.3: Histograms of the mana wasted by each deck: clockwise from top left: BasicDeck, PartialDeck, OverDeck, ExpectedDeck

- While the decline in mean values of W between PartialDeck and ExpectedDeck is comparable in size to the mean decline between BasicDeck and PartialDeck, kurtosis increases dramatically at higher levels of optimization. Since ExpectedDeck does not have a higher range than PartialDeck, this means that, past a certain degree of optimization, performance improves by replacing small values of W with 0, rather than by decreasing the worst-case value of W.

- Using mean and range, OverDeck could be considered a more viable deck than ExpectedDeck, having only a slightly higher mean W but a much lower range (wasting at most 10 mana per game, as opposed to ExpectedDeck's 18) — is is a lower performing but safer option.

Examining the histograms of ExpectedDeck and OverDeck's mana wastage clarifies this (see Fig. 6.3). Note that between PartialDeck and OverDeck, the proportion of games where $W = 0$ declines, 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 no 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 a worthwhile exchange. The objective function for the Hill Climbing Algorithm is therefore the probability, henceforth P, of it not wasting any mana at all in a game.

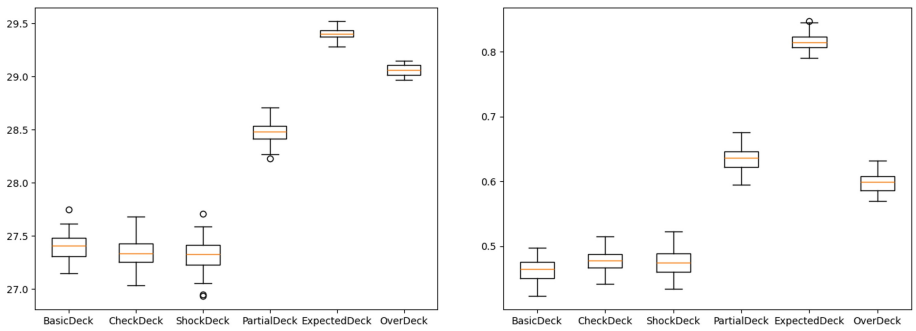


Figure 6.4: Box Plots representing the same MonteCarlo output data as (left to right): proportion of games in which zero mana is wasted, and area under a CMF of mana wasted.

6.3.4 Proportion Wasted vs Cumulative Distribution

An alternative metric to P, that may capture some subtleties to the data, is 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 games with very high values of W.

I investigated this using two new decks, CheckDeck and ShockDeck, both identical to BasicDeck save for the replacement of a single Basic Forest with, respectively, a UB Check Land and a UB Shock Land (see Fig. 2.2). This represents decks after a first Hill Climb increment. 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 MonteCarlos 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 Fig. 6.4.

Disappointingly, neither of these methods prove more able to capture subtle performance differences at the increment level; use of P seems slightly more capable; however, the widely overlapping distributions for both metrics suggest that 1000 simulated games is not enough to distinguish between the two decks. However, with the exception of OverDeck, which is significantly more harshly penalized by P, there is not a meaningful difference between P and C in the proximity of the whiskers between decks that have multiple cards different between them, suggesting that the additional information provided by C does not produce a meaningfully more sensitive assessment.

Given this, 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 once the value of W increases above zero. Secondly, it is a simpler metric. 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. See & Lee's 2004 examination of trust in complex software is instructive here. See & Lee distinguish between trust*worthy* and trust*able* software, arguing that the trustability benefit of simple processes, that may be graspable by the user, sometimes outweigh the trustworthiness benefits of more complex code that may be more accurate [50]. Accepting this, as C offers no meaningful assessment improvement despite greater nuance, I have chosen P as it is easier to explain to a lay user.

GameCard representation of..	Assgned To	`GameCard.mandatory`
All nonland cards	Deck	True
All land cards in the player's input (i.e., 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	DeckBuilder	False
100 Basic Land cards of each colour required by the deck	DeckBuilder	False

Table 6.1: Assignment of cards to different objects in the optimizer.

6.4 The Hill Climbing Algorithm

6.4.1 Setup

Once the user has selected preferences, the Deck and DeckBuilder objects are instantiated. Since both are CardCollection objects, GameCards are assigned to each as per Table 6.1.

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, DeckBuilder creates a new MonteCarlo object for the deck. `MonteCarlo.run()` simulates a set number $\underline{G_r}$ of games. For each, the boolean value $W > 0$ is added to `MonteCarlo.wasted_games`, and to `GameCard.wasted_games` for each GameCard object that 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 MonteCarlo then uses this information to rank all GameCards in the deck by performance, and then resets all values of `GameCard.wasted_games` to empty arrays. The lowest ranking GameCard is set as `MonteCarlo.worst_performing_card`.

With the worst card 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 `MonteCarlo.run()`. Instead, for each candidate land, DeckBuilder calls a new method: `MonteCarlo.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. `MonteCarlo.worst_performing_card` is removed from the deck and replaced with L.
2. A simulation is run.
3. If L or any Fetch 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 as normal, save that the value of W is not increased until after L is drawn.

7. Steps 2 — 6 are repeated $\underline{G_c}$ times.

8. L is removed from the deck and replaced with `MonteCarlo.worst_performing_card`.

Steps 3 and 4 are seperated because the chance of drawing L before any lands capable of searching for it, is very high. 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.

`MonteCarlo.card_test(Land)` returns to DeckBuilder the value of P for the specified substitution, and DeckBuilder replaces `MonteCarlo.worst_performing_card` with the land that returns the highest value.

I assessed possible values for $\underline{G_r}$ and $\underline{G_c}$ based on their ability to produce similar manabases when a DeckBuilder was run ten times using the same initial deck. The similarity of any two manabases A and B can be determined via the Multiset Jaccard Index [51]. This can be extrapolated to all ten manabases by calculating the indices pairwise and then finding the average. This produced the below data:

$\underline{G_r}$	$\underline{G_c}$	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

Table 6.2: Comparisons of Jaccard Index and runtime for different numbers of simulated games.

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 incurs a sizeable runtime penalty (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 lands 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

Fig. 6.5 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

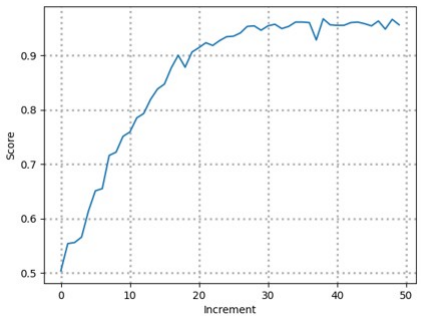


Figure 6.5: Rate of improvement in percentage of wasteless games for a BUG deck, where the X axis represents Hill Climb increments and the Y axis shows percentage of wateless games.

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 Fig. 6.5, this would be around the 33rd-40th increment. This requires smoothing of the data, to avoid triggering the halting 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 fell below zero. A Savitsky-Golay filter removes data noise by projecting a least-squares polynomial fit to sample data across a given window, and determining a new smoothed value from that polynomial [52].

- Calculating the mean result across a given window length 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.

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 Fig. 6.6. Halting is here triggered when the value of either of the filtering functions falls below zero. If the windows are of comparable size for the two filtering methods, the Savitsky-Golay filter retains too much data noise and triggers too soon. To trigger at an appropriate time, the Savitsky-Golay filter requires a larger window. 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.

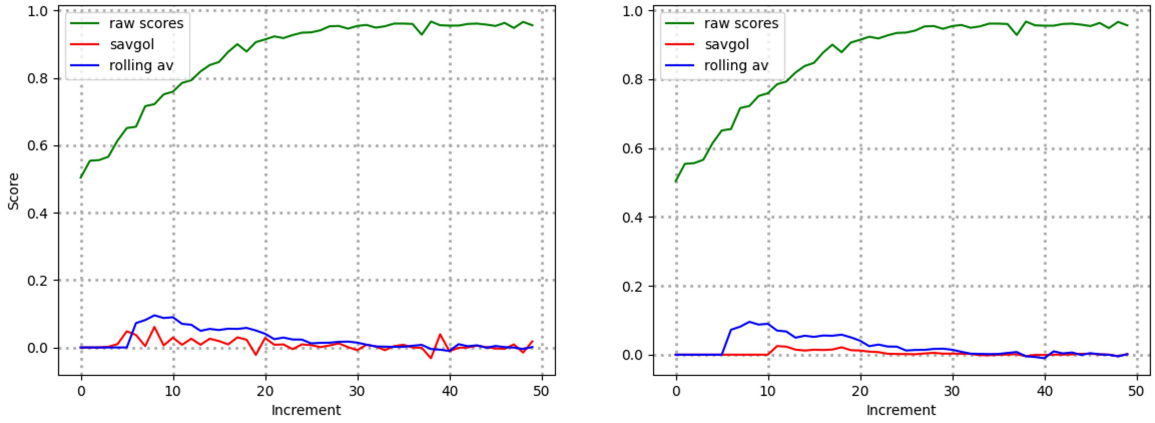


Figure 6.6: Outputs of the rolling average and the Savitsky-Golay derivative halting functions when applied to the data shown in Fig. 6.5; in both cases, the rolling average used a window size of 3, while the Savitsky-Golay filter window was set respectively at 3 (left) and 11 (right)

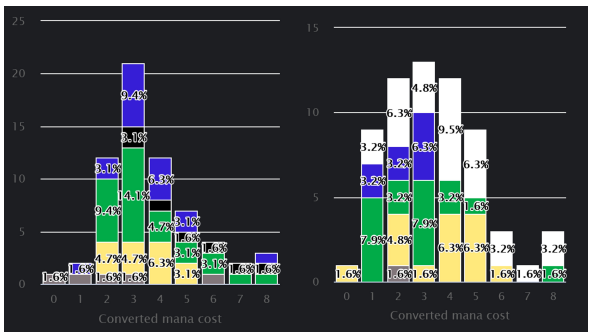


Figure 6.7: Mana curves as visualized using the TappedOut card database for a WUG and BUG deck respectively.

The first is consistency, and has been largely covered in §6.4.2. 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 meaningful cost to runtime.

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 Fig. 6.7. 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.

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 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. A Fast Land is a land that enters tapped unless a player controls 2 or fewer other lands; all other cycles are referenced in Fig. 2.2. This suggests that the Optimizer is meaningfully sensitive to deck strategy.

Chapter 7

The Interface

7.1 The Input Parser

In my initial user-research, subjects desired that LandFill support "round trips"; i.e., receive a list of cards in the formats outputted by online card databases, and output cards in the same format. For this purpose, I created the InputParser class, of which one is created at the start of each session. Test users identified four 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 (i.e., 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 returns a total of six string objects: for each database format (Deckbox and Moxfield being identical), it returns the total decklist including lands (structured into categories if permitted by the database and provided by the user), and the list of provided lands. The format output can be toggled by the user on the frontend.

7.2 App Layout

LandFill is spread across four pages, outlined below and displayed in Figs. 7.1, 7.2, 7.3 and 7.4

7.2.1 Deck Input (Fig. 7.1)

This page allows the user to list details kept constant throughout the session, i.e., cards inputted by the user, and the currency in which the session is to list card prices.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 (Fig. 7.2)

The Preferences Page allows users to customize what lands are candidates for inclusion in the deck. 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

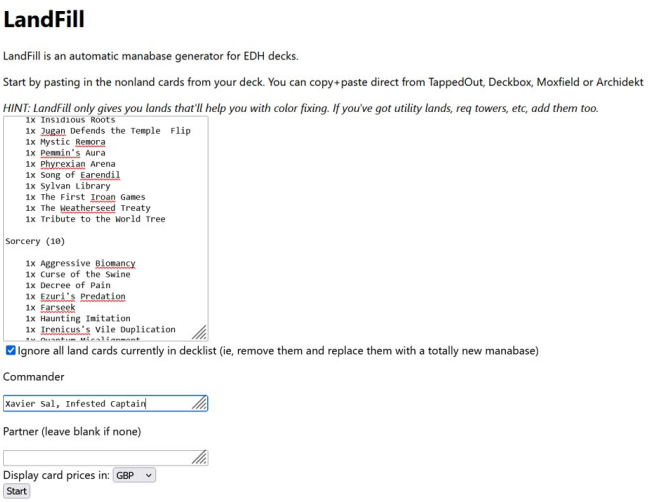


Figure 7.1: Deck Input page.

of lands, and unintuitive, as they did not always know what a given cycle did. Moreover, as LandFill can run more quickly if more cards are deemed by the player to be mandatory for inclusion, my design philosophy was to make it is easy as possible for to mark favoured cards, 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 [53], while also adding a side-panel element which would show a sample card image and card list from a draggable cycle object on mouseover. The tickboxes on this side panel allows individual lands to be included or excluded regardless of the cycle's position.

The Drag and Drop columns labelled "Rank Equivalent Lands" each represent a category of cycles that have no mechanical difference between them. It allows users to weight otherwise equivalent lands as laid out in §6.1.2. The frontend does not support dragging between these boxes, but instead records the position of cycles within them. Information from this input creates a session-specific amendment to the prioritization hierarchy used in the backend LandPrioritization object. For the preferences set in Fig. 7.2, 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 (both Bicycle and Typed Dual Lands being cycles of two-colour taplands with basic landtypes).

The panel of preferences set via tickboxes and numerical inputs above the drag-and-drop column are based on comments made by users during the initial mockup testing (see §3.2.3). Recall also 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 considered this to be a reasonable proxy.

Given the complexity of the interface, it is worth touching on the underlying logic. Since, as covered in §3.2.3, test subjects wanted to be able to remove by criterion rather than just by cycle, underlying logic needed to track not just the eligibility of a land but how that eligibility was set, so that if that criterion is unset, the land would return to its original status. Use of any of the "Exclude from Consideration" Filters, which moves multiple drag-and-drop objects to the "Exclude" column also does not prevent any individual items so moved from being dragged *out* of that column. To track this, any given land is always mapped to one of two "positiveArrays", `positiveArrays.include`

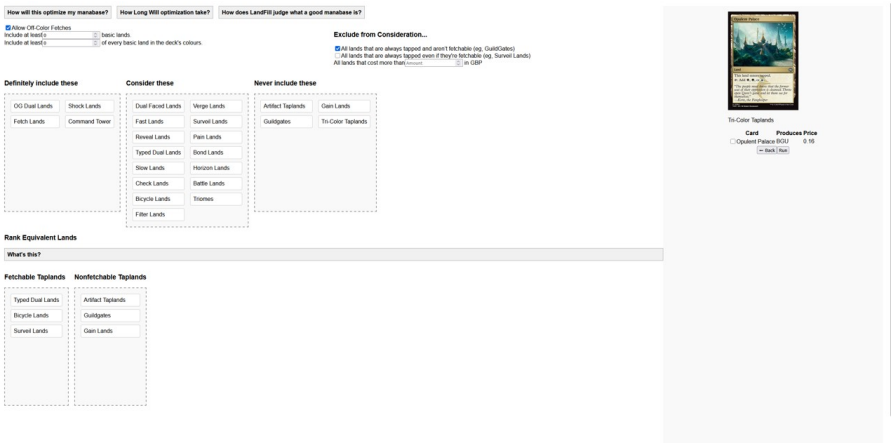


Figure 7.2: Preferences page, zoomed out to show all content; the rankings panels would normally be offscreen. The side-panel remains stationary as the user scrolls.

Figure 7.3: Progress page

or `positiveArrays.consider`. If it is excluded via any method, it is put in one or more of several "excludeArrays", corresponding to the frontend input used to exclude it (i.e., whether it was excluded with its whole cycle via the Drag and Drop panel, whether it was excluded for being an off-colour Fetch Land...). 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.

Information about the methodology and criteria used by LandFill, including an explanation of the P metric (see §6.3.3), is contained in the labelled panels, and is available on clicking.

7.2.3 Progress (Fig. 7.3)

Although initial drafts stayed on the Preferences page while the Optimizer runs, I felt that this was both inappropriate for a runtime of more than one minute, and a missed chance to establish trust in LandFill's decision making process (see §3.2.5) Moreover, during mockup testing, several users expressed a desire to see the logic being used by the backend, as this would make the results more trustworthy. The Progress page, therefore, prints each Hill Climb increment to screen. At the conclusion of the Hill Climb algorithm, the Output page loads automatically.

As this feature was added late in development, it is done crudely, by polling the backend every two seconds for new updates.

7.2.4 Output (Fig. 7.4)

The core feature of the Output Page is the textarea which shows the recommended decklist. This 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 (i.e., from TappedOut or ArchiDekt), these are re-added here, with new lands being added either under a new category, "Lands",

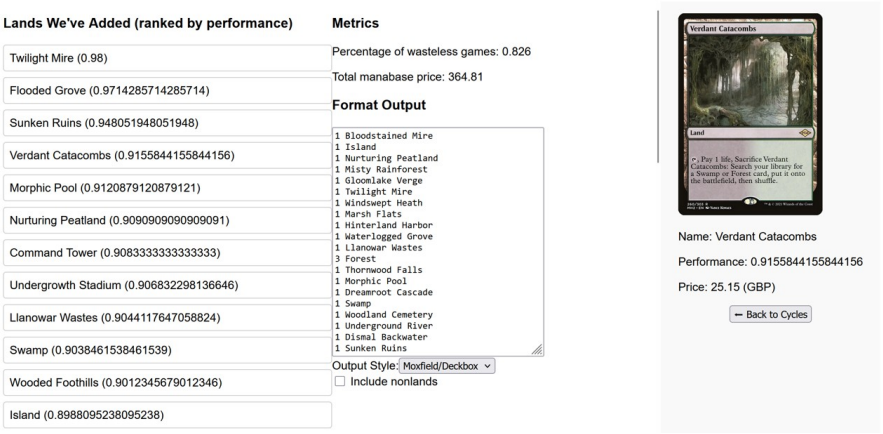


Figure 7.4: Output Page.

or under an existing category if one was detected by the InputParser 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, ordered by performance in the final Hill Climb increment, along with their individual scores.

Chapter 8

Post-Development Validation Testing

8.1 Methodology

I divided my post-development Validation Test into three broad sections, listed below. These were administered to five test subjects.

8.1.1 Effort Testing via TLX

LandFill adds value if it is able to create, with a user's input, a manabase for a deck that they consider to be of equivalent or superior quality to a manabase they could produce with an equal or lesser amount of time and effort. The NASA Task Load Index (TLX) is a questionnaire used widely in industry to measure effort, time and performance.It uses six questions, each ranked out of 100 in increments of 5 to determine a subject's perception of how much effort they expended in pursuit of a task [54]. The questions are as follows [55]:

1. How mentally demanding was the task?
2. How physically demanding was the task?
3. How hurried or rushed was the pace of the task?
4. How successful were you in accomplishing what you were asked to do? (note that for this question, a score of 0 denotes a feeling of perfect success, not total failure)
5. How hard did you have to work to achieve your level of performance?
6. How insecure, dsicouraged, irritated, stressed and annoyed were you?

A total score can be obtained by summing these scores and divided by 6, although the "raw" scores can be valuable as well.

I provided subjects with two incomplete Commander decks, a BUG deck and a WUR deck, both of which consisted solely of 62 nonland cards, and asked them to choose 38 land cards for each. Participants were allowed use of any existing deck-building apps, including Archidekt and Mana Gathering, for the WUR deck, and were asked to complete the BUG deck using LandFill. I timed participants in both tasks. After each task, subjects completed a TLX. This allowed me to compare a subject's perceived effort when building a manabase by hand to their perceived effort when using LandFill. Since this is a within-subjects analysis, each participant functions as their own control, and statistical significance can

be determined using a Wilcoxon Signed Rank test (I used the Wilcoxon calculator on Statology.org [56]). The "carryover effect' that must normally be accounted for in a within-subjects analysis [57] is here irrelevant, as both exercises are meaningfully different.

Although in some TLX analyses the user is asked to weight the six dimensions by significance, there is evidence that this negatively impacts accuracy [54]. As I am interested in comparing raw scores as well, I am not necessarily interested in weighting anyway, so did not do this.

8.1.2 Serendipity Testing

It is impractical to test the quality of manabases recommended by LandFill by simply judging the win/loss rate of decks built around its suggestions. MTG games take time, and introduce many confounding variables, including player skill. Instead, it is more feasible to test simply whether a user is satisfied with the manabase recommended. This is touched on in the TLX, which asks users to rate their "performance" at the task (in this case, building a good manabase with LandFill), but warrants dedicated investigation. Problematically, however, several subjects during User Research expressed a desire for an app that would recommend them lands that they might not have thought of otherwise (see §3.2.4). Therefore, it is important to test the degree to which, insofar as LandFill is making distinct decisions to a human player, those decisions are welcomed by the human player.

I assessed this via the following method. After users had completed both TLX exercises, I used the preferences they had set for LandFill in generation of the BUG deck to generate an alternative list of lands for the WUR deck, and used a simple Python script to categorize lands (here just string representations of the land name) into:

- Lands included in both decklists.
- Lands selected by the user not selected by LandFill.
- Lands selected by LandFill not selected by the User.

For all lands in the final two lists, asked the user's opinion on LandFill's decision to include or exclude each, grouping into cyles where possible for simplicity. Basic Lands were included for the script with their numbers — i.e., if a player chose 4 Basic Islands, and LandFill chose 6 Basic Islands, than the second list would include "4 Basic Islands" and the latter would include "6 Basic Islands."

While this does not map cleanly onto any standardized testing mechanism, it is inspired by the concept of "Serendipity", which is used in the analysis of recommender systems such as the algorithms of sites like Spotify and Instagram. Serendipity compares the "unexpectedness" of recommended items — the similarity between recommended items and a user's previous item interactions — with their "relevance", i.e., whether the user interacted with them after recommendation [58]. However, as these more formalized metrics are designed for large platforms that accrue preference data over time, I significantly adapted them. I also treated them as a prompt for discussion within the below semi-structured interview, and thus a source of qualitative information, rather than as a numerical metric.

8.1.3 Semi-Structured Interview

In addition to the discussion prompts provided by the Serendipity Testing, I asked two additional questions:

WUR Deck (made by user)	Test Subject	1	2	3	4	5
	Mental Demand	25	50	60	10	45
	Physical Demand	25	10	10	0	20
	Temporal Demand	50	20	70	15	30
	Performance	40	20	40	65	70
	Effort	40	20	40	20	15
	Frustration	50	15	40	5	15
	Overall	38.33333	22.5	43.33333	19.16667	32.5
	Timing	8m 40	17m 40	41m 30	9m 45	11m 49
BUG Deck (made by LandFill)	Test Subject	1	2	3	4	5
	Mental Demand	15	10	10	20	10
	Physical Demand	0	0	10	0	0
	Temporal Demand	20	10	5	10	10
	Performance	20	1	40	15	30
	Effort	0	15	10	15	10
	Frustration	20	10	0	20	30
	Overall	12.5	7.666667	12.5	13.33333	15
	Timing	4m33	5m 29	11m 23	6m 59	5m 43
Differences	Test Subject	1	2	3	4	5
	Mental Demand	10	40	50	-10	35
	Physical Demand	25	10	0	0	20
	Temporal Demand	30	10	65	5	20
	Performance	20	19	0	50	40
	Effort	40	5	30	5	5
	Frustration	30	5	40	-15	-15
	Overall	25.83333	14.83333	30.83333	5.833333	17.5
	Timing	4m 7	12m 11	30m 7	2m 46	6m 6

Figure 8.1: Scores and timings from the two TLX analyses, and the differences between these values.

- Do you have any suggestions for improvements or new features?
- Do you have any other general thoughts on LandFill?

I utilized the same lightweight version of Thematic Analysis as I outlined in §3.2 and should here note my own bias as the developer. I conducted this research hoping to receive positive feedback.

8.2 Results

8.2.1 TLX Data Analysis

Scores from the TLX questionnaire are broken down by participant in Fig. 8.1. In all cases, use of LandFill resulted in:

- A lesser overall perceived workload.
- An equal or higher satisfaction with performance of the task — i.e., with the quality of the manabase generated.
- A reduction in time to complete.

Only participants 4 and 5 felt LandFill required more effort in any of the dimensions, and these were also the participants who felt that LandFill had increased their performance by the highest margin. Only participant 3 recorded no difference between their performance with LandFill and without it. This participant also saved by far the most time through use of LandFill, suggesting that this user naturally takes more time to build decks to a higher standard. According to the Wilcoxon Signed-Rank test, the overall score is statistically significant for an alpha value of 0.1. From running the Wilcoxon Signed-Rank test on the raw scores, I determined that the only raw scores that are not significant

within this alpha value are mental demand and frustration. This strongly vindicates my thesis that LandFill is a useful tool for deckbuilders.

8.2.2 Serendipity Test Analysis

Predictably, LandFill always recommended different quantities of each basic land than users. Since the deck under test was WUR, each generated manabase included a number of Basic Island, Basic Plains and Basic Mountain cards. Users said that, when choosing a manabase, they had either divided land slots equally among these after choosing nonbasic lands, or intuited the number based on the proportions required by the deck. All save one were happy to accept LandFill's estimates over their own. Because the test decks are stored on TappedOut, which lists in a pie-chart the proportions of colours required by the deck, it is also notable that even the more thoughtful players here were making decisions based on more information than they may have had access to when deckbuilding normally. In initial user research, many participants said they determine complete decklists, including manabases, before uploading them to TappedOut or an equivalent database.

The one subject who *did* favour their own quantity of basics over LandFill used far fewer basics than LandFill recommended. This is because, while users were asked not to consider utility lands, in practice, the definition of a utility land is vague; some utility lands do generate coloured mana. This particular subject chose single-colour utility lands that entered untapped over basic lands, and said that this was a general preference of theirs.

If a nonbasic land was included in LandFill's list but not in the player's, the player when asked cited one of several possible explanations, including:

- The player had forgtten about this land.
- The player habitually did not think about this land.
- This land cycle incurs a life point penalty that the player typically shies away from.

Only the third case reflects a negative assessment of LandFill's suggestion; in these cases, the player did not use the option to dismiss these lands from consideration. Given the absence of a holistic score assessing the life point damage incurred from the manabase, it may be necessary to include a more visible option to exclude lands based on life point investment in future versions.

If a nonbasic land was included in the player's list but not LandFill's, this was usually because it was a land that was not coded into LandFill. Typically, this was because it was a land that is not contained in a cycle. Currently, the only non-cycle land supported by LandFill is the Command Tower. Future iterations would thus likely benefit from inclusion of more non-cycle lands. While this is trivial to include in the backend, as these lands typically behave quite simply, it is potentially more complicaed in the frontend, the layout of which is based on cycles.

8.2.3 Semi Structured Interview Response Analysis

Since much of the interview focussed on potential new features and fixes, the bulk of the data obtained here will be covered shortly in §8.3. However, I will briefly touch here on general thoughts. While social desirability bias is a factor here, four out of the five participants described LandFill as a useful service. Subjects highlighted its relevance not only to card selection but to formatting and card recollection, as it prevented spelling mistakes and did not accidentally suggest lands in the wrong colours. Several

subjects commented that even if LandFill had not saved them time overall, automation of the process made it feel much less temporally demanding. Compared to the initial mockup, subjects found it much easier to mark cards for inclusion and exclusion, as LandFill now informs them of the behaviour of each cycle via the displayed card image.

Users, unfortunately, did experience some bugs in using the system. The InputParser did not account for formatting differences between search engines, and the toggle switch to include or remove nonland cards from the final deck output behaved temperamentally. Additionally, no user tested made use of the ability to rank mechanically equivalent lands; players either removed all (or all save one) rankable cycles from consideration or did not put much thought into which lands they wold prefer.

In several cases, it was clear in the final output that the results were only approximate. One user noted that one individual Basic Mountain performed far higher in the output than all others did, and higher than several dual lands. A second noted that, in the BUG deck, an "on-colour" Fetch Land, searching for an Island or a Forest, ranked below two "off-colour" fetches, which could only search for one basic landtype in the deck's colours. these should be considered strictly inferior in context to on-colour fetches.

In both these specific cases, this can be addressed without fundamental alteration of LandFill's methods. Regarding the latter, Off-Colour Fetch Lands can be ranked below On-Colour Fetch Lands in the LandPrioritization. Regarding the former, while it is useful during each Hill Climb increment to score each Basic Land seperately, as it prevents nonbasic lands from being ejected prematurely, it may be wise to normalize all basics of each colour in the final output. However, in a general sense, it is likely worthwhile to stress to users more clearly that LandFill's output is not totally proscriptive.

8.3 Additional Features

8.3.1 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. Two testers bought it up as an immediate next step. In both cases, the subject felt it was important not simply to persist user preferences, but fully track the lands in a user's collection. This could be done either by allowing LandFill to store a list of lands that a user owns, or by allowing LandFill to link with a user's account on another card library database. In both cases, this would require users to be able to create a secure account with LandFill.

8.3.2 Flexible Runtimes and Accuracy

Several users commented that, as the bulk of LandFill's runtime required no user input, it could afford to take longer than it currently did. Participant 3, notably, spent almost twice as much time producing their manabase as LandFill spent to conduct 10000 simulated games and 2000 card tests per run as detailed in §6.4.2, which resulted in an increased consistency of 10 percentage points compared to the amount of simulations currently run. They outlined a desirable use-case for LandFill where they specified a high degree of optimization consistency, and then "left it running while [they] had [their] lunch".

In the future, LandFill could offer users one of several accuracy level options, with projected runtimes for each. This would require significant testing in its own right, however, as in order to let

users make an informed choice, I would need to analyse the average Jaccard index improvement as a function of runtime increase across 2, 3, 4 and 5 colour decks.

8.3.3 Additional Formats and MTG: Arena Support

Users suggested the inclusion of other formats as a potential next step. Players also suggested that LandFill be formatted for use with MTG: Arena.

Given that Commander was the most popular format among my initial test subjects, and one designed for casual play, I stand by my decision to restrict initial development to this format, as it allowed me to focus on the core functionality of LandFill without worrying about handling format specifications. Nevertheless this is a natural progression in future iterations.

8.3.4 Frontend Formatting Improvements

Participants identified several usability problems. These are listed below.

- Since the ability to select or deselect individual lands within a cycle was accessible on the side panel via mouseover, players had to move their mouse carefully from the cycle label to the panel to avoid accidentally brushing over another cycle. Ideally this would be replaced by a drop-down menu accessable over the cycle label itself.
- Currency is set globally on the first page. Subjects requested the ability to change it on subsequent pages, if they missed it on the first one.
- Card prices were taken from ScryFall as downloaded by Scrython. One subject pointed out, however, that this meant that LandFill only displays the lowest price for a given land, which may be an undesirable printing. Since cards are sold on a second-hand market, and price differs by printing, prices should be given as a range, not as a single value.
- Although a FAQ on the Preferences page outlines the performance metric used, this should be re-stated on the output page in case the user ignored it.
- One subject pointed out that, if a cycle is moved to the excluded column, but some individual lands were ticked within it, the label of the cycle should be represented differently within the column to show that it is not entirely excluded.

Chapter 9

Conclusion

9.1 Summary

In this thesis, I have outlined why spontaneous generation of an optimized manabase is complicated. I broadly outlined LandFill's use case, as a flexible adaptation of the testing scripts developed by Frank Karsten so that they could be used to automate card selection for an individual deck, rather than just determine deckbuilding best-practices. I then illustrated how I organized a local database of MTG cards, and equipped it to be continously updated in response to future sets.

Equipped with this database, I have outlined how I used Python to implement an extremely stripped down MTG simulator, how to encode behaviour that effectively utilizes mana, how to model the behaviour of land cycles, and the heuristics accepted here during initial development. I then demonstrated how to use the output of these simulations in a Hill Climbing algorithm to explore the search space of possible manabases in a way that balances runtime with accuracy. I then illustrated the frontend design that would allow this apparatus to be used by lay MTG players, and how it encourages them to simplify the job of the Optimizer component by specifying lands they do and do not want.

I finally outlined preliminary test results from this system, and how they strongly suggest that LandFill, subject to de-bugging, is already capable of streamlining the deckbuilding process, making it quicker and easier to assemble manabases to a higher standard. I then outlined features that users proposed, with some examination of how they might be implemented.

Throughout this writeup, I have highlighted where LandFill is a limited product. Its optimizations, even aside from the susceptibility of Hill Climbing algorithms to local maxima, are approximate and not wholly consistent, and there are subtleties of gameplay that it simplifies even within its simplified goals (maximize mana expenditure) as an automated player. While these do not impede its utility, as user testing demonstrates, I will conclude here by exploring some areas of development where, with hindsight, I may have made different decisions.

9.2 Self Assessment

9.2.1 Use of Python as Backend

Recall from §6.4.2 that, measured by average Jaccard Index, optimizations 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, although user testing has expanded my perception of what a usable

timeframe may be. However, the increase in Jaccard Index when longer timeframes were permitted does suggest that there is room to improve LandFill simply by increasing the speed of the Simulator.

Code profiling was conducted repeatedly through development via the `line_profiler` library. It identified the main contributor to the Simulator's runtime the generation of Lumps and the assessment of Lump playability; while I adapted both processes several times to reduce this, and experimented with higher-performance libraries such as SciPy, there remains a runtime floor. Because of this, Python may not have been the best choice of a backend language. I chose Python due to its wide use in webdesign, but it is not considered a high performance language. With hindsight, it may have been wise to explore other options.

An intermediate solution may be to refactor around a piplining library such as Joblib [59] 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 have had to inform the structure of the objects at a much earlier stage of development.

9.2.2 Use of Hill Climbing Algorithm

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 [60]. Since I did not trial this algorithm, I do not know if it would have been able to reach more consistent results from an equivalent number of simulated games. This would be a potential route for experimentation in future versions.

9.2.3 Mid-Development Validation and Verification Tests

Although only one participant in my post-development validation tests openly asked for a slower and more accurate product, all users worked more slowly than LandFill. In retrospect, I would have benefitted from a mid-development test after developing the Simulator, where users simply created a manabase on a stopwatch, as this would have enabled me to make more informed decisions about what constitutes a reasonable time for the optimizer to run.

I also did not conduct sufficient verification tests of the Simulator. During development I tested each new land cycle with many sample hands and checked their behaviour, including in combination with other cycles. However, I did not formally Unit Test the Simulator. Unit Testing the Simulator is complicated, as the gameplay decisions of each sample hand play out over many turns. Moreover, as the Simulator chooses lands to play by progressively shrinking a list of possible lands, there is a stochastic element involved if mure than one land is returned by the end of this process, although there are methods to test code with stochastic outputs. Unit testing the Simulator would be an essential test before full deployment, to ensure that existing code is not broken by the introduction of new cycles.

9.3 Final Thoughts

LandFill is a very promising deckbuilder's tool with potential market applicability. Ultimately, it is more successful as an automatic manabase generator for casual use than a comprehensive optimizer,

but this is still a valid use case. It may be immediately improved via more thorough Unit Testing and debugging, and implementing more complex simulation code to avoid some of the heuristics outlined in §5.6. Broader improvements may be achieved by finding ways to let it conduct more simulated games, either by making the Simulator faster, finding a more efficient Combinatorial Optimization algorithmm or by offering users more input over how long they are content to wait for it. It is, in sum, a successful prototype for an ambitious deckbuilder support app.

Appendix A

Code Extracts

DeckBuilder Hill Climb process:

```
class DeckBuilder(CardCollection):

    def hill_climb_stream(self):
        self.halt = False
        step_output = MonteCarlo(self.deck)
        step_output.hill_climb_test()
        self.last_worst = step_output.worst_performing_card.name
        yield f"Starting␣score:␣{step_output.game_proportions}"
        scores = [step_output.game_proportions]
        iterations = 0

        while not self.halt:
            iterations += 1
            step_output = self.hill_climb_increment(step_output)
            yield f"Replaced␣{self.last_worst}␣with␣{self.last_best}␣({step_output.
                game_proportions})"
            props = step_output.game_proportions
            self.set_new_highscore(props)
            scores.append(props)
            self.halt = self.check_rolling_max(scores)

        self.deck.finalscore = step_output.game_proportions
        yield f"Final␣score:␣{self.deck.finalscore}"

     def hill_climb_increment(self, prior_test):
        worst_card = prior_test.worst_performing_card
        self.last_worst = worst_card.name
        prev_score = prior_test.game_proportions
        t = MonteCarlo(self.deck)
        if self.meets_minbasic_criteria(worst_card):
            cards_to_test = self.get_basic_spread()
        else:
            cards_to_test = self.get_cards_to_test()
        if isinstance(worst_card, BasicLand):
            cards_to_test = [x for x in cards_to_test if x.name != worst_card.name]
        self.deck.give(self, worst_card)

        champ = None
        tested_cards = []
        for trial_card in cards_to_test:
            self.give(self.deck, trial_card)
            trial_card.card_test_score = t.run_card_test(trial_card)
            self.deck.give(self, trial_card)
            if champ is None or trial_card.card_test_score > champ.card_test_score:
                champ = trial_card
            tested_cards.append(trial_card)

        tested_cards.sort(key=lambda x: x.card_test_score, reverse=True)
        tiebreaker_candidates = []
        for card in tested_cards:
```

```
            if card.card_test_score >= champ.card_test_score - 0.01:
                tiebreaker_candidates.append(card)
            else:
                break

        champ = self.break_tie(tiebreaker_candidates)
        self.last_best = champ.name

        if not self.halt:
            self.give(self.deck, champ)
            self.reset_scores(cards_to_test)
            t.hill_climb_test()
        return t

    def check_rolling_max(self, scores, window_size=3, improvement_threshold=0):
        if len(scores) < window_size *2:
            return False

        prior_window = scores[len(scores) - 2 * window_size: len(scores) -
            window_size]
        recent_window = scores[len(scores) - window_size:]
        best_recent = numpy.mean(recent_window)
        best_prior = numpy.mean(prior_window)
        improvement = best_recent - best_prior
        return improvement < improvement_threshold
```

Lump playability assessment:

```
class Lump:
    def set_playability(self, lands, game, filter_subversion = False):
        if not filter_subversion:
            if self.cmc > len(lands):
                return False
            if len(lands) == 0 and self.cmc == 0:
                return True

        invalid = 9999
        mana_required = self.mana_as_list
        weighted = [[land.set_price(game, m) for m in mana_required] for land in
            lands]
        row, col = scipy.optimize.linear_sum_assignment(weighted)
        cost = sum([weighted[row[i]][col[i]] for i in range(len(lands))])
        self.mapping = {}
        output = cost < invalid

        if output:
            for i in range(len(lands)):
                self.mapping[lands[row[i]]] = mana_required[col[i]]
        if not output:
            if not filter_subversion:
                if game.battlefield.filterlands_present():
                    return self.account_for_filterlands(lands, game)
        return output

    def account_for_filterlands(self, lands, game):
        if not self.check_filterland_feasible(lands):
            return False

        divided = self.divide_filters(lands)
        normals = divided["normals"]
        filters = divided["filters"]
        for perm in permutations(filters):
            combinations = self.assign_filter_combinations_v2(perm, normals, game)
            for combination in combinations:
                self.filterloops += 1
                if self.set_playability(combination, game, filter_subversion = True):
                    return True
        return False

    def assign_filter_combinations_v2(self, filters, normals, game):

        def _recurse(filters_left, current_lands):
```

```
            if not filters_left:
                yield current_lands
                return

            current_filter = filters_left[0]
            remaining_filters = filters_left[1:]

            activated = False
            for i, land in enumerate(current_lands):
                if land.produces_at_least_one(current_filter.required, game):
                    new_lands = current_lands[:i] + current_lands[i + 1:] +
                        current_filter.sublands
                    yield from _recurse(remaining_filters, new_lands)
                    activated = True

            if not activated:
                yield from _recurse(remaining_filters, current_lands + [
                    current_filter])

        yield from _recurse(filters, normals)
```

MonteCarlo assessment of decks and specific cards:

```
class MonteCarlo(Simulation):
    def run(self):
        self.deck.reset_card_score()
        self.run_tests()
        self.assess_deck_hc()
        self.assess_lands_hc()

    def run_tests(self, quit=True):
        for i in range(0, self._runs):
            g = Game(self.deck, turns=self.turns, verbose=self._close_examine)
            g.run(quit=quit)
            self.get_game_info(g)

    def assess_deck_hc(self):
        self.game_proportions = self.wasteless_games / self.runs

    def assess_lands_hc(self):
        for card in self.deck.lands_list():
            p = card.proportion_of_games()
        worst = None
        worst_score = 0
        after_sorting = sorted(self.deck.lands_list(), key=lambda x: x.proportions,
            reverse=False)

        for i in range(len(self.deck.lands_list())):
            rank = len(self.deck.lands_list()) - i

        candidates = self.get_worst_card_candidates(self.deck.card_list)
        for card in candidates:

            if isinstance(card, Land):
                if card.mandatory == True:
                    card.reset_grade()
                else:
                    if worst == None or card.proportions < worst_score:
                        worst = card
                        worst_score = card.proportions
                    #card.reset_grade()

        self.worst_performing_card = worst

    def run_card_test(self, card_in):
        self.deck.reset_card_score()
        card_in.options = []
        wasteless_turns = 0
        for _ in range(0, self.ct_runs):
            wasteless_turns += self.single_card_test(card_in)
        for card in self.deck.card_list:
            if isinstance(card, Land):
```

```
                card.reset_grade()
        return wasteless_turns / self.ct_runs

    def single_card_test(self, card_in):
        g = Game(self.deck, turns=self.turns)
        g.run(card_to_test=card_in)
        return self.wastelessness(g)

    def wastelessness(self, game) -> int:
        if game.leftover_mana == 0:
            return 1
        return 0
```

Determining Land and Lump to play each turn:

```
def play_land_and_spell(self):
        lands = self.lands_in_hand
        played_lands = self.battlefield.lands_list()

        for land in lands:
            self.hand.give(self.battlefield, land)
            land.tapped = not land.enters_untapped(self)
            played_lands.append(land)
            self.set_land_permit(land, played_lands)
            played_lands.remove(land)
            self.battlefield.give(self.hand, land)

        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
            , library=False)
        to_play = filtered_by_most_produced[0]

        largest = to_play.largest_lump
        self.play_land_v2(to_play)
        largest.mapping = to_play.proposed_mapping
        self.play_lump_v2(to_play.largest_lump)

        for land in lands:
            land.reset_permits()

    def filter_by_largest(self, lands):
        biggest_enabled = max([l.largest_cmc for l in lands])
        return [l for l in lands if l.largest_cmc >= biggest_enabled]

    def filter_as_taplands(self, lands):
        taplands = [x for x in lands if not x.enters_untapped(self)]
        if len(taplands) > 0:
            return taplands
        else:
            return lands

    def filter_by_most_produced(self, lands, library=False):
        if len(lands) < 2:
            return lands
        if library == False:
            relevant_lands = self.battlefield.lands_list()
        else:
            relevant_lands = []
            relevant_lands.extend(self.battlefield.lands_list())
            relevant_lands.extend(self.hand.lands_list())

        available = self.get_available_pips(relevant_lands)
        needed = [x for x in self.master_lump.colorpips]
        absent = self.calculate_absence(needed, available)

        biggest_contributors = []
        for land in lands:
            land.contribution = len(self.calculate_absence(absent, land.live_prod(
                self)))
            if biggest_contributors == [] or land.contribution < biggest_contributors
                [0].contribution:
```

```
            biggest_contributors = [land]
        elif land.contribution == biggest_contributors[0].contribution:
            biggest_contributors.append(land)

    try:
        max_colors = max([len(x.live_prod(self)) for x in biggest_contributors])
    except ValueError:
        max_colors = 0
    return [x for x in biggest_contributors if len(x.live_prod(self)) >=
        max_colors]
```

Page 2 exclusion and inclusion handling for lands:

```
function handlePlayerMoveCycleOutOfExclude(cycle) {
    const { blocked, messages } = checkCycleMoveExceptions(cycle);
    if (blocked) {
        showErrorMessages(messages);
        return;
    }
    removeCardsFromArray(cycle.cards, 'cycleMovedToExclude');
    removeCardsFromArray(cycle.cards, 'addedBecauseNotTappedButFetchable');
    removeCardsFromArray(cycle.cards, 'addedBecauseNotTappedAndNotFetchable');
    setFilters(prev => ({
        ...prev,
        addedBecauseNotTappedButFetchable: false,
        addedBecauseNotTappedAndNotFetchable: false
    }));
    moveCycleToPositiveByCardMembership(cycle);
}

function moveCycleToPositiveByCardMembership(cycle) {
    const firstCard = cycle.cards[0]; // all cards in a cycle should have the
        same positive array
    if (positiveArrays.include.some(c => c.name === firstCard.name)) {
        setColumns(prev => ({
            ...prev,
            include: [...prev.include, cycle].filter(uniqueByName),
            consider: prev.consider.filter(c => c.displayName !== cycle.
                displayName),
            exclude: prev.exclude.filter(c => c.displayName !== cycle.displayName
                )
        }));
    } else {
        setColumns(prev => ({
            ...prev,
            consider: [...prev.consider, cycle].filter(uniqueByName),
            include: prev.include.filter(c => c.displayName !== cycle.displayName
                ),
            exclude: prev.exclude.filter(c => c.displayName !== cycle.displayName
                )
        }));
    }
}

useEffect(() => {
    if (!allowOffColorFetches) {
        cycles.forEach(cycle => {
            cycle.cards.forEach(card => {
                if (card.offColorFetch) addCardToExcludeArray(card, "
                    offColorFetch");
            });
        });
    } else {
        excludeArrays.offColorFetch.forEach(card =>
            removeCardFromExcludeArray(card, "offColorFetch")
        );
    }
}, [allowOffColorFetches]);

const onDragEnd = result => {
    if (!result.destination) return;
    const { source, destination } = result;
```

```
        if (source.droppableId === destination.droppableId) return;

        const srcList = Array.from(columns[source.droppableId]);
        const [movedCycle] = srcList.splice(source.index, 1);
        const destList = Array.from(columns[destination.droppableId]);
        destList.splice(destination.index, 0, movedCycle);

        setColumns(prev => ({
            ...prev,
            [source.droppableId]: srcList,
            [destination.droppableId]: destList
        }));

        if (destination.droppableId === "include") {
            moveCycleToPositive(movedCycle, "include");
            if (source.droppableId === "exclude") {handlePlayerMoveCycleOutOfExclude(
                movedCycle) }
        } else if (destination.droppableId === "consider") {
            moveCycleToPositive(movedCycle, "consider");
            if (source.droppableId === "exclude") {handlePlayerMoveCycleOutOfExclude(
                movedCycle) }
        } else if (destination.droppableId === "exclude") {
            movedCycle.cards.forEach(card =>
                addCardToExcludeArray(card, "cycleMovedToExclude")
            );
        }

    };

    UseEffect(() => {
        if (filters.maxPrice) {
            cycles.forEach(cycle => {
                cycle.cards.forEach(card => {
                    if (parseFloat(card[currency]) > parseFloat(filters.maxPrice)) {
                        addCardToExcludeArray(card, "belowMaxPrice");
                    } else {
                        removeCardFromExcludeArray(card, "belowMaxPrice");
                    }
                });
            });
        } else {
            excludeArrays.belowMaxPrice.forEach(card =>
                removeCardFromExcludeArray(card, "belowMaxPrice")
            );
        }
    }, [filters.maxPrice]);

    const handleSidePanelTick = (e, card) => {
        const isChecked = e.target.checked;
        if (!isChecked) {
            setExcludeArrays(prev => {
                if (prev.uncheckedByPlayer.some(c => c.name === card.name)) {
                    return prev;
                }
                return {
                    ...prev,
                    uncheckedByPlayer: [...prev.uncheckedByPlayer, card]
                };
            });
        } else {
            setExcludeArrays(prev => {
                const updated = {};
                for (const key in prev) {
                    updated[key] = prev[key].filter(c => c.name !== card.name);
                }
                return updated;
            });
        }
    };
```

Appendix B

Consent Forms

PreDevelopment Consent Form

University of BRISTOL

Participant Information Sheet

Project title: MTG Manabase Optimizer

Invitation paragraph
I would like to invite you to take part in user testing for my software development project. Before you decide whether to participate, I would like you to understand what this will involve for you and the purpose of your involvement. Feel free to ask me any questions.

What is the purpose of the project?
I am working to develop a "Manabase Optimizer" for the Magic the Gathering Card Game. A deckbuilder would input a decklist of non-land cards and specify which format they are building for. The software will take optional inputs from them on a range of factors, which may include deck strategy, price limits, and land cards to include and not include. The software will then produce a list of land cards that, when added to their input cards, will meet the deck size requirements of the format. These cards will be optimized based on the mana pips of the non-land card the player provided in order to maximize the likelihood that they will have access to the mana needed to play the spells in your deck on any given turn.

I am doing this for my final project towards an MSc qualification in Computer Science. Although I will be developing the software, the actual assessed deliverable will be a report documenting my development process and computational challenges that it has raised along the way, which will include references to studies such as this one which I am running to ensure I develop a functional product.

Why have I been invited to participate?
I am asking you to participate because, as a Magic: The Gathering player, you would be a potential user of this software if it were deployed as a commercial product. I would therefore value your input about what I should be considering at this early stage in development.

Do I have to take part?
Taking part in this is entirely voluntary. I will go through this sheet with you before starting the study, and am happy to answer any questions you may have. If you agree to take part, I will ask you to sign a consent form. You are free to withdraw from the study at any time for no reason while it is being run. However, as your comments and performance will be recorded anonymously, you will not be able to remove this data after the end of the study.

What will happen to me if I take part and what will I have to do?
I will call you on a messenger app of your choosing, and record our conversation locally on a voice memo. The conversation will be divided into two parts: first, a semi-structured interview, where I will ask you some questions about what you might like to see in such a project, and then take your thoughts. Second, I will present you with a "mockup", a semi-functional html draft that resembles the proposed homepage of my web app, and conduct a "think aloud" evaluation, where I will, while providing you with minimal prompting, ask you to narrate your thoughts as you use the app. The first part could take from 10 minutes to an hour, depending on how much you are interested in talking about the matter with me, while the second will probably not take more than 5 minutes.

At some point over the next couple of weeks, once I have conducted these interviews, I will circulate a questionnaire asking people to assess the importance of various features that other people have come up with; this will follow a “Kano Questionnaire” format, where I will llist features and ask you to mark whether the presence or absence of such features would be vital to any sane use of the product, positive, neutural, tolerable or outright unpleasant. This would likely take around five minutes to complete.

You may participate in as many or few of these evaluation methods as you like, and withdraw at any time.

What are the possible disadvantages and risks involved in taking part in the project?
As the data is anonymised, and the software itself is purely an aid for playing a recreational game, I do not foresee participation in this study posing any risk of discomfort or inconvenience to the participant.

What are the possible benefits of taking part?
There is a chance that, should this design project go well, a completed version of the software will be released online as a deck-building tool. If so, you will be very welcome to use it for your own decks.

Will my participation in this project be kept confidential?
Regarding the interview, your data will be collected via an audio recording app on my smartphone. It will be kept as a saved file for a period not longer than 24 hours, and neither the contents nor the name of this sound file will contain any identifying details about you. During the 24 hours after the study, I will aggregate all data collected from all participants in this session anonymously, and classify individual comments made as positive or negative feedback, without grouping them together by individual. After this, I will delete the audio files. At no point during this process will you be identified by name, contact detail, or visually.

The survey will be conducted anonymously, and all results will be similarly anonymised.

While my project is written, this anonymised data will be stored on my personal OneDrive account in my notes, and will be deleted at the end of the project. The final draft of my project will remain in that folder, and will also be kept by the University of Bristol. Both myself and that organization may choose to share it with others.

What will happen to the results of the research project?

Your anonymised comments may or may not be referenced in the Project Report that I write over the summer of 2025. This will be submitted as the final project of my degree, comprising 60 credits of a MSc in Computer Science. I do not currently intend to publish any of my work. Should I launch a version of the completed software after graduation, this will not contain any reference to this study in its documentation.

Participants are welcome to contact me to ask for a copy of my dissertation on its completion.

Who is organising and funding the research?
This research is overseen by the School of Computer Science of the University of Bristol. It being done as part of the Conversion Masters in Computer Science MSc.

Who has reviewed the study?

The work will be reviewed by John Lapinskas, who is a Senior Lecturer in the University of Bristol School of Computer Science.

Further information and contact details
Should you wish to contact me with any further questions, I can be reached:

By telephone or WhatsApp – 07955258430
By email – zs24945@bristol.ac.uk
On Discord – LeahUnknowing

If you have any concerns related to your participation in this study beyond what I am able to answer, please contact the University of Bristol's Research Governance team at: research-governance@bristol.ac.uk

PostDevelopment Consent Form

University of BRISTOL

Participant Information Sheet

Project title: LandFill

Invitation paragraph
I would like to invite you to take part in user testing for my software development project. Before you decide whether to participate, I would like you to understand what this will involve for you and the purpose of your involvement. Feel free to ask me any questions.

What is the purpose of the project?
I am working to develop LandFill, a "Manabase Optimizer" for the Magic the Gathering Card Game; specifically, the Commander format. A deckbuilder would input a decklist of non-land cards, and some inputs about what lands they like and do not like. It will then generate a list of land cards that will fill out the deck, roughly optimized to maximize the ability of the deck to deploy its spell cards.

I am doing this for my final project towards an MSc qualification in Computer Science. Although I will be developing the software, the actual assessed deliverable will be a report documenting my development process and computational challenges that it has raised along the way, which will include references to studies such as this one which I am running to ensure I develop a functional product.

Why have I been invited to participate?
I am asking you to participate because, as a Magic: The Gathering player, you would be a potential user of this software if it were deployed as a commercial product. I would therefore value your input about whether LandFill adds value.

Do I have to take part?
Taking part in this is entirely voluntary. I will go through this sheet with you before starting the study, and am happy to answer any questions you may have. If you agree to take part, I will ask you to sign a consent form. You are free to withdraw from the study at any time for no reason while it is being run. However, as your comments and performance will be recorded anonymously, you will not be able to remove this data after the end of the study.

What will happen to me if I take part and what will I have to do?
I will call you on a messenger app of your choosing and ask you to add lands to two Commander Decks (both of which currently have only spells). The first time you will be allowed to do this as you normally would. The second time you will be asked to use LandFill. I will time you doing both tasks. After each I will ask you to complete a short questionnaire, the "NASA Task Load Index", which will gauge how difficult and stressful you found each task. I will then move to a semi-structured interview, in which I will ask you your opinions on how the lands you chose differ from those produced by LandFill and whether you consider its opinions useful; I will also ask you for thoughts on the system as a whole and potential new features to add. You may withdraw at any time.

What are the possible disadvantages and risks involved in taking part in the project?
As the data is anonymised, and the software itself is purely an aid for playing a recreational game, I do not foresee participation in this study posing any risk of discomfort or inconvenience to the participant.

What are the possible benefits of taking part?
There is a chance that, should this design project go well, a completed version of the software will be released online as a deck-building tool. If so, you will be very welcome to use it for your own decks.

Will my participation in this project be kept confidential?
Your data will be recorded anonymously. At no point during this process will you be identified by name, contact detail, or visually.

While my project is written, this anonymised data will be stored on my personal OneDrive account in my notes, and will be deleted at the end of the project. The final draft of my project will remain in that folder, and will also be kept by the University of Bristol. Both myself and that organization may choose to share it with others.

What will happen to the results of the research project?

Your anonymised comments may or may not be referenced in the Project Report that I write over the summer of 2025. This will be submitted as the final project of my degree, comprising 60 credits of a MSc in Computer Science. I do not currently intend to publish any of my work. Should I launch a version of the completed software after graduation, this will not contain any reference to this study in its documentation.

Participants are welcome to contact me to ask for a copy of my dissertation on its completion.

Who is organising and funding the research?
This research is overseen by the School of Computer Science of the University of Bristol. It being done as part of the Conversion Masters in Computer Science MSc.

Who has reviewed the study?
The work will be reviewed by John Lapinskas, who is a Senior Lecturer in the University of Bristol School of Computer Science.

Further information and contact details
Should you wish to contact me with any further questions, I can be reached:

By telephone or WhatsApp – 07955258430
By email – zs24945@bristol.ac.uk
On Discord – LeahUnknowing

If you have any concerns related to your participation in this study beyond what I am able to answer, please contact the University of Bristol's Research Governance team at: research-governance@bristol.ac.uk

Bibliography

[1] C. Hall. "Commander: The definitive history of magic's most popular format". (2020), [Online]. Available: https://www.polygon.com/2020/5/28/21266763/magic-the-gathering-commander-origins-elder-dragon-highlander-alaska-menery/ (visited on 08/30/2025).

[2] A. Churchill, S. Biderman, and A. Herrick, "Magic: The gathering is turing complete", *arXiv preprint arXiv:1904.09828*, 2019.

[3] F. Karsten. "What's an optimal mana curve and land/ramp count for commander?" (2025), [Online]. Available: 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).

[4] P. Furtado. "What does strictly better mean in mtg?" (2024), [Online]. Available: https://draftsim.com/mtg-strictly-better/ (visited on 08/30/2025).

[5] M. Rosewater. "Get ready to dual". (2017), [Online]. Available: https://magic.wizards.com/en/news/making-magic/get-ready-dual-2017-02-27 (visited on 08/30/2025).

[6] B. Moursund. "Playing your pet: Rough-testing a magic deck". (2010), [Online]. Available: 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).

[7] N. Metropolis and S. Ulam, "The monte carlo method", *Journal of the American statistical association*, vol. 44, no. 247, pp. 335–341, 1949.

[8] E. A. Silver, "An overview of heuristic solution methods", *Journal of the operational research society*, vol. 55, no. 9, pp. 936–956, 2004.

[9] S. H. Zanakis and J. R. Evans, "Heuristic "optimization": Why, when, and how to use it", *Interfaces*, vol. 11, no. 5, pp. 84–91, 1981.

[10] E. Team. "Verification and validation in software testing". (2025), [Online]. Available: https://www.browserstack.com/guide/verification-and-validation-in-testing (visited on 08/30/2025).

[11] R. Awati. "Iterative development".

(2023),
[Online]. Available: https://www.techtarget.com/searchsoftwarequality/definition/iterative-development (visited on 08/30/2025).

[12] C. D. Ward and P. 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.

[13] A. Esche,
Mathematical Programming and Magic: The Gathering®.
Northern Illinois University, 2018.

[14] C. Alvin, M. Bowling, S. Rivers-Green, D. Siglin, and L. Alvin,
"Toward a competitive agent framework for magic: The gathering",
In *The International FLAIRS Conference Proceedings*,
Vol. 34, 2021.

[15] A. Wu. "Using monte carlo simulation to improve your manabases".
(2018),
[Online]. Available: https://article.hareruyamtg.com/article/article_en_405/?lang=en (visited on 08/13/2025).

[16] F. Karsten. "How many sources do you need to consistently cast your spells? a 2022 update".
(2022),
[Online]. Available: https://www.tcgplayer.com/content/article/How-Many-Sources-Do-You-Need-to-Consistently-Cast-Your-Spells-A-2022-Update/dc23a7d2-0a16-4c0b-ad36-586fcca03ad8/ (visited on 09/07/2025).

[17] F. Karsten. "Should you play tapped duals in 2-color limited decks?"
(2018),
[Online]. Available: 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).

[18] F. Karsten. "Mana bases in ixalan standard".
(2017),
[Online]. Available: https://web.archive.org/web/20200330115445/https:/www.channelfireball.com/articles/mana-bases-in-ixalan-standard/ (visited on 08/13/2025).

[19] "Magic: The gathering arena".
(),
[Online]. Available: https://store.steampowered.com/app/2141910/Magic_The_Gathering_Arena/ (visited on 09/03/2025).

[20] "Managathering".
(),
[Online]. Available: https://managathering.com/ (visited on 09/03/2025).

[21] "Archidekt".

(),
[Online]. Available: https://archidekt.com/ (visited on 09/03/2025).

[22] "Moxfield: A modern deck builder for magic: The gathering".
(),
[Online]. Available: https://moxfield.com/ (visited on 09/03/2025).

[23] "Tappedout".
(),
[Online]. Available: https://tappedout.net/ (visited on 09/03/2025).

[24] "Deckbox".
(),
[Online]. Available: https://deckbox.org/ (visited on 09/03/2025).

[25] "Flask".
(),
[Online]. Available: https://flask.palletsprojects.com/en/stable/ (visited on 09/03/2025).

[26] "Django".
(),
[Online]. Available: https://www.djangoproject.com/ (visited on 09/03/2025).

[27] "Fastapi".
(),
[Online]. Available: https://fastapi.tiangolo.com/ (visited on 09/03/2025).

[28] "React: The library for web and native user interfaces".
(),
[Online]. Available: https://react.dev/ (visited on 09/03/2025).

[29] "Sqlalchemy".
(),
[Online]. Available: https://www.sqlalchemy.org/ (visited on 09/03/2025).

[30] E. Venkat. "Why sqlalchemy should no longer be your orm of choice for python projects". (2023),
[Online]. Available: https://eash98.medium.com/why-sqlalchemy-should-no-longer-be-your-orm-of-choice-for-python-projects-b823179fd2fb (visited on 08/13/2025).

[31] "Chatgpt".
(),
[Online]. Available: https://chatgpt.com/ (visited on 09/03/2025).

[32] K. Manuel, "The sage encyclopedia of social science research methods.",
Reference & User Services Quarterly, vol. 44, no. 1, pp. 94–96, 2004.

[33] P. C. Wright and A. F. Monk, "The use of think-aloud evaluation methods in design",
ACM SIGCHI Bulletin, vol. 23, no. 1, pp. 55–57, 1991.

[34] V. Braun and V. Clarke, "Using thematic analysis in psychology",
Qualitative research in psychology, vol. 3, no. 2, pp. 77–101, 2006.

[35] S. Martello and P. Toth,
Knapsack problems: algorithms and computer implementations.

John Wiley & Sons, Inc., 1990.

[36] L. Riggins. "An introduction to mtgjson".
(2020),
[Online]. Available: https://triple-equals.medium.com/an-introduction-to-mtgjson-b88dbf65572 (visited on 09/01/2025).

[37] "Scryfall".
(),
[Online]. Available: https://scryfall.com/ (visited on 09/03/2025).

[38] "Scrython".
(),
[Online]. Available: https://github.com/NandaScott/Scrython (visited on 09/03/2025).

[39] "Flasksqlalchemy".
(),
[Online]. Available: https://flask-sqlalchemy.readthedocs.io/en/stable/ (visited on 09/03/2025).

[40] L. Nguyen. "A brief guide to database normalization".
(2023),
[Online]. Available: https://medium.com/@ndleah/a-brief-guide-to-database-normalization-5ac59f093161 (visited on 09/01/2025).

[41] A. Robert. "What is property-based testing?"
(2021),
[Online]. Available: https://www.mayhem.security/blog/what-is-property-based-testing (visited on 08/31/2025).

[42] J. L. K. Jimmy Wong. "Commander mythbusters — the command zone 335 — magic: The gathering commander".
(2020),
[Online]. Available: https://www.youtube.com/watch?v=Pr91Crzl7DI (visited on 08/14/2025).

[43] "Itertools - functions creating iterators for efficient looping".
(),
[Online]. Available: https://docs.python.org/3/library/itertools.html (visited on 09/03/2025).

[44] "Line-profiler 5.0.0".
(),
[Online]. Available: https://pypi.org/project/line-profiler/ (visited on 09/03/2025).

[45] G. Cocca. "What is memoization? how and when to memoize in javascript and react".
(2022),
[Online]. Available: https://www.freecodecamp.org/news/memoization-in-javascript-and-react/ (visited on 09/01/2025).

[46] "Pickle - python object serialization".
(),
[Online]. Available: https://docs.python.org/3/library/pickle.html (visited on 09/03/2025).

[47] "Scipy: Fundamental algorithms for scientific computing in python".
(),
[Online]. Available: scipy.org (visited on 09/03/2025).

[48] T. C. Koopmans and M. Beckmann, "Assignment problems and the location of economic activities",
Econometrica: journal of the Econometric Society, pp. 53–76, 1957.

[49] S. Salhi and J. 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.

[50] J. D. Lee and K. A. See, "Trust in automation: Designing for appropriate reliance",
Human factors, vol. 46, no. 1, pp. 50–80, 2004.

[51] L. d. F. Costa, "Further generalizations of the jaccard index",
arXiv preprint arXiv:2110.09619, 2021.

[52] Ç. Candan and H. Inan, "A unified framework for derivation and implementation of savitzky–golay filters",
Signal Processing, vol. 104, pp. 203–211, 2014.

[53] "React-beautiful-dnd".
(),
[Online]. Available: https://github.com/atlassian/react-beautiful-dnd (visited on 09/03/2025).

[54] E. A. Bustamante and R. D. Spain,
"Measurement invariance of the nasa tlx",
In *Proceedings of the human factors and ergonomics society annual meeting*,
SAGE Publications Sage CA: Los Angeles, CA,
Vol. 52, 2008,
Pp. 1522–1526.

[55] B. Gore. "Nasa tlx: Task load index".
(2020),
[Online]. Available: https://humansystems.arc.nasa.gov/groups/tlx/ (visited on 09/03/2025).

[56] "Wilcoxon signed rank calculator".
(),
[Online]. Available: https://www.statology.org/wilcoxon-signed-rank-test-calculator/ (visited on 09/07/2025).

[57] J. Simkus. "Https://www.simplypsychology.org/within-subjects-design.html".
(2023),
[Online]. Available: https://www.simplypsychology.org/within-subjects-design.html (visited on 09/07/2025).

[58] Z. Yan. "Serendipity: Accuracy's unpopular best friend in recommenders".

(2020),
[Online]. Available: https://eugeneyan.com/writing/serendipity-and-accuracy-in-recommender-systems/ (visited on 09/02/2025).

[59] "Joblib: Running python functions as pipeline jobs".
(),
[Online]. Available: https://joblib.readthedocs.io/en/stable/ (visited on 09/03/2025).

[60] S. J. Bjørke and K. A. Fludal,
"Deckbuilding in magic: The gathering using a genetic algorithm",
M.S. thesis, NTNU, 2017.