Optimizing Board Control in Digital Card Games: A Linear Programming Approach to Enhanced AI Decision-Making

by
Barney Guo

My *Cool* Thesis Advisor: Dr. William B. Lassiter
Committee Member: Dr. Daniel A. Kessler

A thesis submitted in partial fulfillment
of the requirements for the
Degree of Bachelor of Arts with Honors
in Statistics and Analytics

UNIVERSITY OF NORTH CAROLINA AT CHAPEL HILL
Chapel Hill, North Carolina
April 28, 2025

Contents

List of Tables

Abstract

Decision-making in digital collectible card games (CCGs) poses a richly combinatorial challenge for artificial intelligence. We model each turn as an Integer Linear Program (ILP) and solve it exactly via Gurobi. The playing style arises from the weight vector in the ILP's objective, allowing the same deck to emphasize face damage, board control, minion preservation, or a weighted blend thereof. Using fixed Paladin and Warrior lists, we defined seven weight profiles each, yielding seven strategic variants per deck. These agents then competed in a 1,000-game round-robin with uniformly random pairings. Results show clear, statistically significant performance gaps across weight profiles, confirming that objective-function design strongly influences in-game behavior and match outcomes. Our study demonstrates ILP's viability as a practical engine for generating strategically diverse CCG agents and offers a reproducible framework for analyzing optimization-driven heuristics.

Summary

Digital collectible card games such as *Hearthstone* involve sequential, resource-limited decisions under hidden information. We frame each turn as an integer linear program: binary variables capture card plays, attack assignments, and survival outcomes, while linear constraints enforce mana availability, board capacity, and keyword logic (Charge, Rush, Taunt). Because the model is entirely linear, Gurobi solves each turn precisely and within practical time.

Strategic variety arises from eight objective weights (lethal damage, board clearing, face pressure, threat removal, friendly survival, minion damage, trade efficiency, card value). Seven weight profiles—Keep-Alive, Board-Clear, Aggro-Face, Value-Trading, Aggro-Tempo, True-Balanced, plus a favored hybrid—were combined with two 30-card decks in a 7×7 round-robin, each pairing running 1,000 matches, totaling about 50,000 ILP solves. After every turn, we update the simulated board until a hero's health reaches zero.

The results show that varying these weights alone shifts play styles along offense–control lines: Aggro-Face often dominates if it goes first against slower lists, Keep-Alive shines in longer scenarios, and the hybrid profile secures the highest overall win rate by balancing lethal aggression with board management. Mirror matchups exhibit a major first-player edge (up to 87% wins), emphasizing tempo's impact even under exact optimization. We find that integer linear programming yields tractable, interpretable, and configurable AI agents for CCGs, pointing to a broader decision-support template for adversarial domains. Future extensions include multi-turn horizons, explicit hidden-card modeling, and evolutionary searches over weight vectors.

Acknowledgments

The author would like to especially express sincere gratitude to Dr. William B. Lassiter for his invaluable guidance, insightful suggestions, and consistent support throughout this research project. The author particularly appreciate his enthusiasm for pushing him to explore the application of integer-programming techniques to model complex decision-making and board-control optimization in simulated card games. The author also acknowledge the resources and computational facilities provided by the Department of Statistics and Operations Research at the University of North Carolina at Chapel Hill; this work made use of the Gurobi Optimizer under its academic license program.

The author would also like to extend heartfelt thanks to Elisabeth Guo, in all honesty, though he can no longer pinpoint exactly why she deserves this credit. She never proof-read a draft, but she did remark that Hearthstone was the cool game the boys played back in her middle-school days, which promptly reminded the author that he is, in fact, three years older than she is. Perhaps it was her off-hand quip about Transformers, delivered moments after a linear-programming model had crashed, that convinced the author even the ugliest errors can, with a little re-tooling, transform into something useful; or perhaps she merely reminded him that commas and semicolons are not interchangeable. Whatever the reason, her presence proved indispensable: the project's objective value increased, the author's sanity constraint remained (mostly) feasible, and she even made him forswear musical forever. He also hopes she never feels depressed—life is genuinely beautiful, she herself is remarkable, and there will always be someone who cares for her. May all her future endeavors unfold smoothly.

Chapter 1

Introduction

In today's world of rapid technological progress, artificial intelligence has emerged as a transformative force across numerous domains, ranging from autonomous vehicles to automated financial trading. As an undergraduate, I am still in the early stages of exploring these intricate fields, yet I often find myself wondering how AI might extend beyond advanced, specialized applications and play a more practical role in everyday life. This line of thinking led me to consider areas where personal interests intersect with AI's potential for innovation. Since I have always enjoyed card games, I began to reflect on how AI could be applied in this familiar context. Card games offer a unique set of challenges: they involve strategic decision-making based on hidden information, probabilistic events, stochastic processes, and efficient resource management. These complexities make them an ideal testbed for AI-driven decision-making and strategy optimization. With this inspiration in mind, I decided to focus my honor thesis on exploring how AI can automate decision-making in card games and enhance players' strategic thinking.

This thesis explores the development of an automated decision-making framework for digital card games, with a primary focus on *Hearthstone*. By framing decision-making in these games as an optimization problem, we seek to develop AI strategies that improve gameplay by maximizing board control—a critical factor in card games. Board control often dictates the flow of a match: when in a disadvantaged position, effective decision-making is essential to mitigate losses and regain stability, while in an advantageous position, players must find ways to secure a decisive victory. This dual consideration of managing board presence in both offensive and defensive scenarios highlights the complexity of strategic decision-making in such games. Through this approach, we aim to demonstrate how advanced optimization techniques—such as linear programming—can be applied to enhance AI performance in complex, dynamic environments. Ultimately, this work aspires to contribute not only to AI-driven strategy development in digital card games but also to the broader understanding of

applying optimization methods in similar real-world scenarios involving uncertainty and strategic decision-making.

Chapter 2

Background

Hearthstone is a widely popular strategic, turn-based digital card game that requires players to outthink and outmaneuver their opponents through careful planning and tactical decision-making. In the game, players deploy a combination of minions, spells, and weapons to reduce their opponent's hero health to zero, while managing limited resources known as *mana*. Each turn presents a dynamic mix of opportunities and constraints, forcing players to balance offense and defense while vying for control of the board.

What sets *Hearthstone* apart is its strategic depth, driven by key elements such as *minion abilities*, *attack mechanics*, *damage types*, and the *effective utilization of resources*. These complexities not only make the game engaging but also position it as an excellent platform for studying artificial intelligence. The interplay of hidden information, randomness, and resource management makes *Hearthstone* an ideal candidate for exploring AI-driven decision-making and strategy optimization in uncertain and dynamic environments.

2.1 Foundations of Research

2.1.1 Relevance of *Hearthstone* to AI Research

- **Hidden Information and Stochasticity**: Unlike perfect information games such as chess, *Hearthstone* involves hidden information, as players cannot see the cards in their opponent's hand or deck. Additionally, the randomness introduced by card draws and certain card effects makes the game highly stochastic, posing unique challenges for AI models.

- **Complex Decision-Making**: *Hearthstone* requires players to make strategic decisions involving multiple variables, such as resource management, board control, and predicting the opponent's moves. This complexity makes it an ideal platform to test AI techniques for decision-making in dynamic environments.

2.1.2 Optimization in Games

- **Role of Optimization**: Optimization techniques play a central role in improving decision-making in games. Mathematical models can help determine the best course of action by maximizing or minimizing a particular objective, such as damage dealt, resource efficiency, or board control.
- **Applications in Digital Card Games**: In digital card games like *Hearthstone*, optimization can be applied to deck construction, in-game decision-making, and strategic planning. For example, linear programming can optimize mana usage, while heuristic-based methods can improve gameplay strategies.

2.1.3 AI and Digital Card Games

- **AI Applications in Card Games**: AI research has made significant strides in games such as poker, which shares some similarities with *Hearthstone* in terms of hidden information and probabilistic outcomes. Notable AI systems like DeepStack and Libratus have demonstrated the potential for advanced decision-making in stochastic environments.
- **Gaps in Research**: While AI has been successfully applied to games like poker and StarCraft, there is relatively limited research on AI applications in digital card games. This thesis aims to address this gap by focusing on *Hearthstone* and developing an AI-driven optimization framework for strategic decision-making.

2.1.4 Why *Hearthstone*?

- **Unique Mechanics and Popularity**: *Hearthstone*'s combination of hidden information, randomness, and complex decision-making makes it a unique and challenging environment for AI research. Additionally, its popularity ensures a wide audience and a well-documented set of mechanics, facilitating robust experimentation and analysis.
- **A Testbed for AI Development**: The game's well-defined rules and manageable scale make it an excellent testbed for developing AI models that can be applied to other domains involving uncertainty and strategy, such as finance, logistics, or autonomous systems.

2.1.5 Structure of the Thesis

- **Introduction**: Provides an overview of the problem, objectives, and scope of the research.
- **Background**: Explains the mechanics of *Hearthstone* and situates the game within the context of AI research and optimization.
- **Model Development**: Details the proposed optimization framework and the mathematical models used to represent decision-making in *Hearthstone*.
- **Results and Analysis**: Presents experimental results, evaluates the performance of the framework, and discusses findings.
- **Conclusion and Future Work**: Summarizes the contributions of the thesis and outlines potential avenues for further research.

To ensure that this thesis is accessible to readers without prior knowledge of *Hearthstone*, the following section introduces the game mechanics and terms relevant to this study. All terms and concepts are presented in alphabetical order (A-Z) for ease of reference. If you are already familiar with *Hearthstone*, you may skip ahead to Chapter 4 to continue reading.

2.2 Classes

In *Hearthstone*, each class has a unique set of abilities, keywords, and hero powers that define its playstyle. The ten classes are listed below.

- **Death Knight**: A versatile class that leverages a unique rune system, allowing players to customize their deck with Blood, Frost, or Unholy runes. Death Knights use a combination of powerful minions, spells, and undead synergies to control the battlefield. Common keywords associated with Death Knight include *Lifesteal*, *Corpse Generation*, and *Reborn*, providing both offensive and defensive options.
- **Demon Hunter**: A fast-paced, aggressive class focusing on direct attacks and high-damage combos. Commonly uses keywords like *Lifesteal* and *Outcast* to maintain tempo and capitalize on aggressive strategies.
- **Druid**: Known for its flexibility, using spells and minions to adapt between offense and defense. Druids frequently use keywords such as *Taunt* and *Choose One* for versatile strategic options.

- **Hunter**: Primarily an aggressive class that focuses on quick, direct damage to the opponent's hero. Common keywords include *Rush* and *Deathrattle*, supporting fast, powerful attacks.
- **Mage**: Specializes in spellcasting with strong board control and direct damage spells. Often utilizes keywords like *Freeze* and *Spell Damage* to control opponents and maximize damage output.
- **Paladin**: A balanced class that combines minion swarming with powerful buffs. Common keywords include *Divine Shield* and *Taunt*, focusing on building and sustaining a strong board presence.
- **Priest**: A control-oriented class with a focus on healing, board control, and card manipulation. Frequently employs *Lifesteal* and *Silence* to extend the game and neutralize key threats.
- **Rogue**: Known for its versatility and combos, often relying on high-damage spells and weapons. Keywords such as *Combo* and *Stealth* support quick, calculated attacks.
- **Shaman**: A class that leverages totems, elemental synergy, and powerful board control. Common keywords include *Overload* and *Totem*, enabling unique minion summons and spells.
- **Warlock**: A high-risk, high-reward class, often sacrificing health to gain powerful effects. Keywords such as *Lifesteal* and *Discard* allow Warlocks to generate significant value through resource management.
- **Warrior**: Known for strong defensive abilities, armor, and powerful weapons. Frequently uses keywords like *Taunt* and *Rush* to protect the hero while applying pressure on opponents.

2.3 Hero Powers

Each class in *Hearthstone* has a unique Hero Power that supports its playstyle and strategy.

- **Death Knight (Ghoul Charge)**: Summons a 1/1 Ghoul with Charge, allowing immediate attacks. The specific effects of the hero power can vary based on the Death Knight's rune setup, adapting to different strategic needs.
- **Demon Hunter (Demon Claws)**: Grants +1 Attack to the hero for the turn, supporting aggressive, weapon-based strategies.

- **Druid (Shapeshift)**: Provides +1 Attack for the turn and +1 Armor, offering flexibility for both offense and defense.
- **Hunter (Steady Shot)**: Deals 2 damage to the enemy hero, aligning with aggressive strategies focused on consistent hero damage.
- **Mage (Fireblast)**: Deals 1 damage to any target, providing direct damage useful for control and removal.
- **Paladin (Reinforce)**: Summons a 1/1 Silver Hand Recruit, supporting a board-centric strategy with minion synergies.
- **Priest (Lesser Heal)**: Heals any character for 2 health, fitting into a control strategy by sustaining board presence and longevity.
- **Rogue (Dagger Mastery)**: Equips a 1/2 dagger, enhancing the hero's ability to deal damage directly and activate combo cards.
- **Shaman (Totemic Call)**: Summons a random totem from a selection of four, providing various support abilities to maintain board control.
- **Warlock (Life Tap)**: Draws a card and deals 2 damage to the hero, enabling card advantage at the cost of health—a key feature in high-risk, high-reward strategies.
- **Warrior (Armor Up!)**: Grants 2 Armor to the hero, fitting into defensive and control playstyles by increasing survivability.

2.4 Cards

In *Hearthstone*, cards are the primary tools players use to execute strategies and achieve victory. Each card belongs to a specific type, with distinct functions and gameplay implications. The types of cards relevant to this thesis are outlined below, along with a brief mention of locations, which are excluded from this study.

- **Minions**: These are creatures summoned onto the battlefield. Minions have attack and health values, and many possess additional abilities, such as *Taunt*, *Rush*, or *Divine Shield*, that can influence the game in various ways. They play a central role in maintaining board control and implementing offensive or defensive strategies.
- **Spells**: Spells are cards that generate immediate effects when played. These effects can range from dealing damage to enemies, healing allies, drawing additional cards, or even altering the state of the battlefield. Spells are vital for strategic flexibility and impactful plays.

- **Weapons**: Equipped by the player's hero, weapons allow direct attacks against minions or the opposing hero. They have an attack value and durability, which determines how many times they can be used before breaking. Weapons often complement aggressive or control-oriented strategies.
- **Hero Cards**: Hero cards transform the player's hero into a new one with enhanced abilities and a new hero power. They often provide an immediate effect that can shift the momentum of the game. Hero cards are pivotal in late-game strategies and adapting to specific situations.
- **Locations (Excluded)**: Locations are unique cards placed on the battlefield that provide activatable effects every other turn. These effects can enhance minions, deal damage, or provide other strategic advantages. While important in gameplay, locations are not included in the scope of this thesis as the focus is on cards more directly tied to minion, spell, weapon, and hero card interactions.

Each turn, players are allocated resources known as *mana*, which are used to play cards. Strategic gameplay also involves managing mana efficiently while balancing offensive and defensive actions to maintain board control. Critical mechanics such as *attack mechanics*, *damage types*, and *minion abilities* further enhance the depth of gameplay, making *Hearthstone* an ideal platform for exploring AI-driven decision-making models.

2.5 Core Mechanics

2.5.1 Deck

In *Hearthstone*, a deck is a collection of 30 cards selected by the player prior to the start of a match. This deck serves as the player's arsenal throughout the game, with each card drawn at random during play. Constructing an effective deck involves strategic selection of cards that synergize well together, considering factors such as mana curve, card draw potential, and overall game plan. The composition of the deck significantly influences the player's strategy and adaptability during matches.

2.5.2 Mana

Mana is the primary resource in *Hearthstone* used to play cards and activate Hero Powers. Represented by Mana Crystals, each player starts with one Mana Crystal and gains an additional one at the beginning of each turn, up to a maximum of

ten. The mana cost of each card is indicated by a number within a blue crystal icon on the card, denoting the amount of mana required to play that card. Effective mana management is crucial, as it dictates the pacing of gameplay and the ability to execute strategies. Certain cards and effects can modify the number of available Mana Crystals temporarily or permanently, adding layers of strategic depth to resource management. :contentReference[oaicite:0]index=0

2.6 Phases

A match in *Hearthstone* consists of several phases that structure gameplay and determine the sequence of actions for each player:

- **Mulligan Phase**: At the start of the match, players draw an initial hand of cards and are given the opportunity to replace any unwanted cards. This phase allows players to optimize their starting hand for their strategy.
- **Player Turns**: Each match alternates between players taking turns. A turn is divided into three key stages:
    1. *Start of Turn*: Players draw a card and gain one additional Mana Crystal (up to a maximum of ten). Certain effects or card abilities that activate at the start of a turn occur here.
    2. *Main Phase*: Players can play cards, attack with minions or the hero, and activate abilities. Actions are constrained by the player's available mana and strategic considerations.
    3. *End of Turn*: The player's actions conclude, and any effects or abilities triggered at the end of the turn are resolved. Control passes to the opponent.

Understanding these phases is essential for effectively planning and executing strategies throughout the game.

2.7 Objectives

The primary objective in *Hearthstone* is to reduce the opponent's hero health from 30 to 0 before they can do the same to you. To achieve this goal, players must balance offense and defense, leveraging card synergies, board control, and resource management. The gameplay is driven by several key strategic objectives:

- **Maintaining Board Control**: Ensuring the dominance of minions on the battlefield to pressure the opponent and mitigate their actions.
- **Maximizing Damage Output**: Optimizing the use of minions, spells, and weapons to deal consistent and impactful damage to the opponent.
- **Resource Management**: Efficiently using mana and cards to sustain pressure and adaptability throughout the match.
- **Anticipating Opponent Moves**: Predicting the opponent's actions and adapting strategies to counter their plans.

By focusing on these objectives, players can navigate the complexities of the game and outmaneuver their opponents to secure victory.

Chapter 3

Literature Review

The application of artificial intelligence (AI) to digital card games such as *Hearthstone* has been extensively explored in recent years, with researchers employing diverse methodologies to tackle challenges like decision-making, deck optimization, and strategic gameplay. One prominent approach, as demonstrated in [1], utilizes evolutionary algorithms (EAs) to optimize AI agents. EAs simulate natural selection by iteratively refining parameters based on a fitness function, enabling the development of competitive Hearthstone agents. This method has proven effective in addressing the stochasticity and hidden information inherent in the game. However, evolutionary algorithms focus on predefined parameters and lack flexibility in adapting to dynamic in-game states. By contrast, our research addresses these limitations by employing integer programming to optimize decisions in real time, incorporating board states, resource allocation, and multi-objective optimization into a unified framework.

Other works have examined the evolving dynamics of Hearthstone gameplay. In [2], researchers explored how evolutionary algorithms can simulate the metagame by generating competitive decks and analyzing their impact on the broader meta. While this study provided valuable insights into deck composition and metagame evolution, it primarily concentrated on pre-game deckbuilding rather than in-game decision-making. Similarly, [8] focused on optimizing decks in Hearthstone's arena mode, offering innovative methods for deck construction but neglecting the complexities of turn-by-turn gameplay.

Machine learning approaches have also gained traction in Hearthstone research. In [3], the authors leveraged historical gameplay data to predict opponents' actions, enabling agents to adapt their strategies accordingly. This predictive approach enhances strategic foresight but does not directly address action sequencing or resource management during gameplay. Neural network-based approaches, such as those presented in [4] and [5], demonstrated the utility of deep learning in evaluating board states and predicting optimal plays or win rates. While neural networks excel at capturing

complex patterns in data, these studies primarily focused on prediction rather than optimization, leaving room for methods that directly influence decision-making.

Beyond predictive methods, hybrid models combining neural networks and adaptive systems have also been explored. For instance, [6] introduced a system that integrates fuzzy ART with neural networks to adapt to changing game conditions. This adaptive approach successfully enhances agent flexibility but does not explicitly incorporate multi-objective optimization or resource constraints, which are critical for making optimal in-game decisions.

Another combinatorial approach was discussed in [7], where the authors proposed optimizing card selection and gameplay strategies through combinatorial methods. While this work aligns with our objective of enhancing strategic gameplay, it primarily focuses on isolated elements rather than integrating constraints like mana usage, attack priorities, or survival probabilities. Our model builds upon these foundations by using integer programming to holistically address resource allocation, decision-making, and strategy optimization.

Despite these advances, no previous work has attempted to model Hearthstone decision-making using integer programming. This novel approach enables the precise formulation of game constraints, such as mana limits, survival probabilities, and strategic value maximization, within a mathematical framework. By addressing the limitations of evolutionary algorithms, neural networks, and combinatorial methods, our research provides a robust solution to optimizing turn-by-turn decisions, offering a new perspective on Hearthstone AI and broader applications in strategic AI development.

Chapter 4

Model

In this chapter, we present our integer linear model for automating turn-by-turn decisions in Hearthstone. We introduce parameter, indices, binary variables to encode minion attacks, survival, and card plays, alongside constraints capturing mana usage, lethal checks, and other key mechanics. By adjusting the objective function's weights, this framework accommodates a range of strategic priorities through a unified and easily extended approach.

4.1 Notation

4.1.1 Parameters

- m: Total number of friendly minions on the battlefield, where $0 \leq m \leq 7$.
- n: Total number of enemy minions on the battlefield, where $0 \leq n \leq 7$.
- h: Total number of cards (minions, spells, etc.) in hand, where $0 \leq h \leq 10$.
- k: Represents $m + h$, the total number of cards at the player's disposal, where $0 \leq k \leq 17$.
- H_{hero}: The current health of the enemy hero, where $0 \leq H_{\text{hero}} \leq 30$.
- D_{hero}: Total damage dealt to the enemy hero by friendly minions.

4.1.2 Indices

- i: Index of each friendly minion on the battlefield, where $i \in \{1, \ldots, m + h\}$.

- j: Index of each enemy minion on the battlefield, where $j \in \{1, \ldots, n\}$.

4.1.3 Minion Attributes

- A_i: Attack value of friendly minion i, representing how much damage friendly minion i can deal when attacking.
- B_i: Health value of friendly minion i, indicating how much damage it can take before being eliminated.
- P_j: Attack value of enemy minion j, representing how much damage enemy minion j can deal when attacking.
- Q_j: Health value of enemy minion j, indicating how much damage it can take before being eliminated.

4.1.4 Strategic and Resource Parameters

- S_i: A fixed value assigned to each card (minions, spells, etc.) in the hand and deck, representing its strategic importance.[I]
- C_i: The mana cost assigned to each card p, representing the amount of mana required to play it.
- M: The total mana available, which increases incrementally each turn but is capped at a maximum of 10 mana.
- $W_1, W_2, W_3, W_4, W_5, W_6, W_7, W_8$: Weights assigned to different terms in the objective function, representing their relative importance in maximizing overall efficiency.

4.2 Decision Variables

4.2.1 Attack Variables

- $x_{(i,\text{hero})}$: Binary decision variable indicating whether friendly minion i is used to attack the enemy hero.

$$x_{(i,\text{hero})} = \begin{cases} 1 & \text{if friendly minion } i \text{ attacks the enemy hero,} \\ 0 & \text{otherwise.} \end{cases}$$

[I] This value remains constant to ensure consistent evaluation and prioritization across turns within the optimization model.

- $x_{(i,j)}$: Binary decision variable indicating whether friendly minion i attacks enemy minion j.

$$x_{(i,j)} = \begin{cases} 1 & \text{if friendly minion } i \text{ attacks enemy minion } j, \\ 0 & \text{otherwise.} \end{cases}$$

4.2.2 Survival Variables

- z_j: Binary decision variable indicating whether enemy minion j survives.

$$z_j = \begin{cases} 1 & \text{if enemy minion } j \text{ survives,} \\ 0 & \text{if enemy minion } j \text{ dies.} \end{cases}$$

- z_{hero}: Binary decision variable indicating whether the enemy hero survives.

$$z_{\text{hero}} = \begin{cases} 1 & \text{if the enemy hero survives,} \\ 0 & \text{if the enemy hero dies.} \end{cases}$$

- y_i: Binary decision variable indicating whether friendly minion i survives.

$$y_i = \begin{cases} 1 & \text{if friendly minion } i \text{ survives,} \\ 0 & \text{if friendly minion } i \text{ dies.} \end{cases}$$

4.2.3 Play Variable

- u_k: Binary decision variable indicating whether card k (minion, spell, etc.) is played during the current turn.

$$u_k = \begin{cases} 1 & \text{if card } k \text{ (minion, spell, etc.) is played,} \\ 0 & \text{if card } k \text{ (minion, spell, etc.) is not played.} \end{cases}$$

4.2.4 Clear Variable

- c: A **binary** decision variable indicating whether *all* enemy minions on the board are cleared during the current turn.

$$c = \begin{cases} 1, & \text{if every enemy minion is killed (board is fully cleared),} \\ 0, & \text{otherwise.} \end{cases}$$

4.3 Baseline Full Model

$$\begin{aligned}\text{Maximize:}\quad & W_1 \cdot (1 - z_{\text{hero}}) + W_2 \cdot c \\ & + W_3 \cdot \sum_{i=1}^{m+h} A_i \cdot x_{(i,\text{hero})} - W_4 \cdot \sum_{j=1}^{n} P_j \cdot z_j \\ & + W_5 \cdot \sum_{i=1}^{m} B_i \cdot y_i + W_6 \cdot \sum_{i=1}^{m+h} \sum_{j=1}^{n} A_i \cdot x_{(i,j)} \\ & - W_7 \cdot \sum_{i=1}^{m+h} \sum_{j=1}^{n} P_j \cdot x_{(i,j)} + W_8 \cdot \sum_{i=m+1}^{m+h} S_i \cdot u_i\end{aligned}$$

Subject to:

$$\sum_{j=1}^{n} x_{(i,j)} + x_{(i,\text{hero})} \;\leq\; 1, \qquad \forall\, i \in \{1, \ldots, m\} \tag{1}$$

$$z_j \;\geq\; 1 - \left[\frac{\sum_{i=1}^{m+h} A_i\, x_{(i,j)}}{Q_j}\right], \qquad \forall\, j \in \{1, \ldots, n\} \tag{2}$$

$$z_{\text{hero}} \;\geq\; 1 - \left[\frac{\sum_{i=1}^{m+h} A_i\, x_{(i,\text{hero})}}{H_{\text{hero}}}\right] \tag{3}$$

$$c \;\leq\; 1 - z_j, \qquad \forall\, j \in \{1, \ldots, n\} \tag{4}$$

$$y_i \;\leq\; 1 - \sum_{j=1}^{n} \left[\frac{P_j - B_i + 1}{P_j} \cdot x_{(i,j)}\right], \qquad \forall\, i \in \{1, \ldots, m\} \tag{5}$$

$$\sum_{i=1}^{m} y_i \;+\; \sum_{i=m+1}^{m+h} u_i \;\leq\; 7 \tag{6}$$

$$y_i \;\leq\; u_i, \qquad \forall\, i \in \{m+1, \ldots, k\} \tag{7}$$

$$\sum_{i=m+1}^{m+h} C_i\, u_i \;\leq\; M, \qquad \forall\, i \in \{m+1, \ldots, k\} \tag{8}$$

x, y, c, z, and u are all binary decision variables.

4.4 Description

4.4.1 Description of Objective Function

The objective function maximizes the overall effectiveness of gameplay by balancing offensive, defensive, and strategic actions while optimizing resource usage. It is structured as follows:

$$\begin{aligned}\text{Maximize:}\quad & W_1 \cdot \big(1 - z_{\text{hero}}\big) + W_2 \cdot c \\ & + W_3 \cdot \sum_{i=1}^{m+h} \Big(A_i \cdot x_{(i,\text{hero})}\Big) - W_4 \cdot \sum_{j=1}^{n} \Big(P_j \cdot z_j\Big) \\ & + W_5 \cdot \sum_{i=1}^{m+h} \Big(B_i \cdot y_i\Big) + W_6 \cdot \sum_{i=1}^{m+h} \sum_{j=1}^{n} \Big(A_i \cdot x_{(i,j)}\Big) \\ & - W_7 \cdot \sum_{i=1}^{m+h} \sum_{j=1}^{n} \Big(P_j \cdot x_{(i,j)}\Big) + W_8 \cdot \sum_{k=1}^{h} \Big(S_{m+k} \cdot u_{m+k}\Big).\end{aligned}$$

The components of the objective function are:

- $+ W_1 \, (1 - z_{\text{hero}})$

  Rewards killing the enemy hero. If $z_{\text{hero}} = 0$ means the hero is dead, then $(1 - z_{\text{hero}}) = 1$. A higher W_1 stresses hero lethality.

- $+ W_2 \, c$

  Rewards clearing all enemy minions. If $c = 1$ indicates a full board clear, a large W_2 values complete removal of enemy threats.

- $+ W_3 \sum_{i=1}^{m+h} A_i \, x_{(i,\text{hero})}$

  Rewards attacks on the enemy hero. The variable $x_{(i,\text{hero})} = 1$ if minion i attacks the hero; multiplied by its A_i.

- $- W_4 \sum_{j=1}^{n} P_j \, z_j$

  Penalizes leaving high-attack enemy minions alive. If $z_j = 1$, minion j survives, costing $P_j \times W_4$ in the objective.

- $+W_5 \sum_{i=1}^{m+h} B_i \, y_i$

  Rewards friendly minion survival. If $y_i = 1$, minion i stays alive, adding $B_i \times W_5$ to the objective.

- $+W_6 \sum_{i=1}^{m+h} \sum_{j=1}^{n} A_i \, x_{(i,j)}$

  Rewards dealing damage to enemy minions. Each attack from minion i on minion j adds $A_i \times W_6$.

- $-W_7 \sum_{i=1}^{m+h} \sum_{j=1}^{n} P_j \, x_{(i,j)}$

  Penalizes the damage your minions risk taking when they trade. Attacking an enemy minion j with attack P_j induces a cost $P_j \times W_7$.

- $+W_8 \sum_{k=1}^{h} S_{m+k} \, u_{m+k}$

  Rewards playing high-value cards from your hand. If $u_{m+k} = 1$ for card k, you gain $S_{m+k} \times W_8$, incentivizing impactful plays.

Altogether, these terms produce a composite objective that tries to: *(a)* kill the enemy hero, *(b)* clear enemy minions, *(c)* preserve one's own board, *(d)* trade into enemies efficiently, *(e)* and leverage high-value cards from the hand. Varying each weight $W_1, \ldots, W_8$ tunes the overall strategy to favor face damage, board control, or other priorities.

Overall, the objective function optimizes gameplay by balancing offensive actions (attacking the hero), minimizing enemy threats, enhancing friendly minion survival, and maximizing strategic value from card play. Each term's weight (W_1, W_2, W_3, W_4) allows for fine-tuning the model based on different strategic priorities.

4.4.2 Description of Constraints

Constraint I: Friendly Minion Attack

$$\sum_{j=1}^{n} x_{(i,j)} + x_{(i,\text{hero})} \leq 1 \quad \forall i \in \{0, 1, \ldots, m\}$$

This constraint ensures that each friendly minion can attack at most one enemy minion or the enemy hero per turn.

Constraint II: Enemy Minion Survival

$$z_j \geq 1 - \sum_{i=1}^{m+h} \left[\frac{A_i}{Q_j} \cdot x_{(i,j)} \right] \quad \forall j \in \{0, 1, \ldots, n\}$$

Where:

- z_j: Binary decision variable indicating whether enemy minion j survives (1 if survives, 0 if not).
- Q_j: Health value of enemy minion j.
- $x_{(i,j)}$: Binary variable indicating if friendly minion i attacks enemy minion j (1 if attacking, 0 otherwise).
- A_i: Attack value of friendly minion i.

This constraint ensures that an enemy minion survives only if its health is sufficient to withstand the cumulative damage from all friendly minions attacking it.

Constraint III: Enemy Hero Survival

$$z_{\text{hero}} \geq 1 - \sum_{i=1}^{m+h} \left[\frac{A_i}{H_{\text{hero}}} \cdot x_{(i,\text{hero})} \right], \quad \forall j = 1, \ldots, n$$

Where:

- z_{hero}: Binary decision variable indicating whether the enemy hero survives (1 if survives, 0 if not).
- H_{hero}: The current health value of the enemy hero.
- $x_{(i,\text{hero})}$: Binary variable indicating if friendly minion i attacks the enemy hero (1 if attacking, 0 otherwise).
- A_i: Attack value of friendly minion i.

This constraint ensures that the enemy hero survives only if their remaining health is sufficient to withstand the cumulative damage from all attacking friendly minions.

Constraint IV: Enemy Minion Clearance

$$c \leq 1 - z_j \quad \forall j \in \{1, \ldots, n\}$$

Where:

- c: Binary decision variable indicating whether *enemy minion j is fully killed* (1 if killed, 0 if not).
- Q_j: Health value of enemy minion j.
- $x_{(i,j)}$: Binary variable indicating if friendly minion i attacks enemy minion j (1 if attacking, 0 otherwise).
- A_i: Attack value of friendly minion i.

This constraint ensures that an enemy minion is considered *cleared* ($c = 1$) if the total damage from all attacking friendly minions is sufficient to exceed that minion's health. If $\sum_{i=1}^{m+h} (A_i/Q_j)\, x_{(i,j)} \geq 1$, then c_j can be forced to 1 in the solution, reflecting a lethal amount of assigned damage.

Constraint V: Friendly Minion Survival

$$y_i \leq 1 - \sum_{j=1}^{n} \left[\frac{P_j - B_i + 1}{P_j} \cdot x_{(i,j)} \right] \quad \forall i \in \{0, 1, \ldots, m\}$$

Where:

- y_i: Binary variable indicating whether friendly minion i survives (0 if survives, 1 if dead).
- P_j: Attack value of enemy minion j.
- B_i: Health value of friendly minion i.

This constraint ensures that a friendly minion survives only if its health is sufficient to withstand the attack from an enemy minion it engages with.

Constraint VI: Minion Survival and Play

$$y_i \leq u_i \quad \forall i \in \{1, \ldots, m+h\}$$

Where:

- y_i: Binary decision variable indicating whether friendly minion i survives (1 if it survives, 0 if it does not).
- u_i: Binary decision variable indicating whether minion i is played during the turn (1 if played, 0 if not).

This constraint ensures that the survival status (y_i) of a minion is only relevant if the minion is played ($u_i = 1$). If the minion is not played ($u_i = 0$), its survival status is automatically set to $y_i = 0$.

Constraint VII: Minion and Card

This constraint ensures that the total number of surviving minions on the battlefield plus the cards played from hand does not exceed a set limit (e.g., 7 in this case).

$$\sum_{i=1}^{m} y_i + \sum_{i=m+1}^{m+h} u_i \leq 7$$

Where:

- y_i: Binary variable indicating whether friendly minion i survives (1 if it survives, 0 if it dies).
- u_k: Binary variable indicating whether card k in hand is played (1 if playable, 0 if not).
- m: Total number of friendly minions on the battlefield.
- h: Total number of cards in hand.

This constraint models the limitation that only a certain number of minions and cards can be placed on the battlefield at the same time.

Constraint VIII: Mana

$$\sum_{k=1}^{h} u_k \cdot C_k \leq M$$

Where:

- u_k: Binary variable indicating whether minion or spell k is played (1 if played, 0 otherwise).
- C_k: Mana cost of playing minion or spell k.
- M_t: Total mana available on turn t, where $M_t = \min(t, 10)$.

This constraint ensures that the total mana cost of the cards played during a turn does not exceed the available mana for that turn, respecting the resource limitations of the game.

Constraint IX: Attack Eligibility for Newly Played Minions [II]

$$x_{(i,j)} \leq (CH_i + R_i) \cdot u_i \quad \forall i = m+1, \ldots, m+h,\ j = 1, \ldots, n$$
$$x_{(i,\text{hero})} \leq CH_i u_i \quad \forall i$$

The first constraint ensures that a friendly minion i can only attack an enemy minion on the turn it is played if it has either the Charge or Rush keyword. The second constraint restricts attacks on the enemy hero to minions with Charge. Without Charge or Rush ($CH_i = 0$ and $R_i = 0$), the minion cannot attack on the turn it is summoned.

- $x_{(i,j)}$: Binary decision variable indicating whether friendly minion i attacks enemy minion j.
- $x_{(i,\text{hero})}$: Binary decision variable indicating whether friendly minion i attacks the enemy hero.
- CH_i: Binary indicator for whether minion i has Charge (1 if Charge is active, 0 if not).

[II] Rush and Charge are keywords in *Hearthstone* that will be formally introduced in the next chapter. Briefly, minions with the Rush keyword can attack enemy minions on the turn they are played, while minions with the Charge keyword can attack both enemy minions and the enemy hero. Since our model considers the sum of $m+h$ for assessing lethal conditions, we introduce these keywords here for clarity.

- R_i: Binary indicator for whether minion i has Rush (1 if Rush is active, 0 if not).

These constraints ensure that:

1. **Minions with Charge** ($C_i = 1$) can attack both enemy minions and the hero on the turn they are summoned.
2. **Minions with Rush** ($R_i = 1$) can attack enemy minions immediately on the turn they are summoned, but not the hero.
3. **Minions without Charge or Rush** ($C_i = 0$ and $R_i = 0$) cannot attack on the turn they are summoned.

Chapter 5

Keywords

In this chapter, we aim to integrate our model with specific classes in Hearthstone. By aligning the mechanics of class abilities and each class's specialized card keywords with our previously established mathematical model, we can explore how these elements interact within a variety of strategic contexts.

Earlier, we outlined that Hearthstone decks commonly belong to archetypes such as Aggressive, Midrange, or Control. Here, we focus on Paladin and Warrior to demonstrate how class-specific synergies and specialized keywords can be applied in a more balanced deck setting. We employ a standard, middle-of-the-road deck—one similar to the Innkeeper's list—providing a straightforward showcase of minion interactions and offensive maneuvers without leaning heavily into any single playstyle.

5.1 What are Keywords?

As discussed, cards in Hearthstone often come with specific keywords that grant them unique abilities and influence gameplay strategy. In this section, we introduce some of the most common keywords relevant to our model and illustrate how each can be represented mathematically within the framework of our constraints. By examining these keywords, we gain insight into how special abilities are incorporated into an playstyle's decision-making processes.

Minions in Hearthstone are associated with a variety of keywords that define their abilities and interactions. These keywords can be divided into general keywords (applicable to all classes) and class-specific keywords. Examples include:

- **General Keywords:**

  - Charge
  - Taunt
  - Windfury
  - Battlecry
  - Deathrattle
  - ...

- **Class-Specific Keywords:**
  - Rush (Warrior)
  - Divine Shield (Paladin)
  - Overload (Shaman)
  - Combo (Rogue)
  - ...

5.2 Common Keywords

Common keywords in *Hearthstone* appear on minions available to any class and significantly shape the flow of play. These abilities—such as **Charge**, **Taunt**, and **Windfury**—determine how, when, and against whom each minion can attack or be attacked. By influencing both offensive and defensive maneuvers, they add depth to the board state and create further decision points for resource management and combat sequencing.

In addition to these common keywords, two other prominent mechanics are *Battlecry* and *Deathrattle*. A Battlecry triggers the moment a minion is played (often drawing extra cards or dealing immediate damage), whereas a Deathrattle resolves upon that minion's death (e.g., summoning tokens or applying buffs/debuffs). Because these effects vary widely among individual cards, implementing them in a solver requires detailed, card-specific logic. At present, we do not include Battlecry or Deathrattle constraints in our Python code, leaving them for possible future enhancements of the model.

5.2.1 Charge

Minions with the Charge keyword are allowed to attack enemy minions and the enemy hero immediately upon being played. The following condition enables this behavior by allowing attacks on the same turn a minion with Charge is summoned.

The parameter CH_i is a binary indicator for whether friendly minion i has the Charge keyword:

- $CH_i = 1$: The friendly minion i has Charge, allowing it to attack immediately after being summoned.
- $CH_i = 0$: The friendly minion i does not have Charge, so it cannot attack on the same turn it is summoned.

If a minion i has Charge ($CH_i = 1$), it is eligible to attack immediately upon being played. Otherwise, minions without Charge must wait until the following turn to initiate attacks.

5.2.2 Taunt

Minions with the *Taunt* keyword require opposing minions to target them first, preventing attacks on other enemy minions or the enemy hero until all Taunt minions have been eliminated. This mechanic enforces a strict targeting priority.

The parameter tt_j is a binary indicator for whether enemy minion j has the Taunt keyword:

- $tt_j = 1$: The enemy minion j has Taunt, and must be attacked before any non-taunt target.
- $tt_j = 0$: The enemy minion j does not have Taunt and can only be targeted if no Taunt minions remain.

In addition, the solver maintains a binary variable z_j to indicate whether enemy minion j is still alive:

- $z_j = 1$: The enemy minion j is alive.
- $z_j = 0$: The enemy minion j has died.

Forbidding Hero Attacks if Taunt Minions Live

To enforce the restriction against attacking the enemy hero while Taunt minions remain, the following constraint is used:

$$(z_j = 1) \implies x_{\text{hero}}(i) = 0,$$

for each enemy minion j with $tt_j = 1$ and for each friendly minion i. Equivalently, this may be expressed in code as $(z_j == 1) >> (x_{\text{hero}}(i) == 0)$. If at least one Taunt minion is alive, no friendly minion is permitted to attack the enemy hero.

Blocking Attacks on Non-Taunt Minions

Similarly, attacks on non-taunt enemy minions are restricted if any Taunt minion is alive. For every enemy minion j with $tt_j = 0$ and every Taunt minion k with $tt_k = 1$, we enforce:

$$(z_k = 1) \implies x(i, j) = 0.$$

This ensures that if a Taunt minion k is alive, a friendly minion i cannot attack a non-taunt minion j. Only when all Taunt minions have been removed do other enemy targets become valid.

5.2.3 Windfury

A minion with the *Windfury* keyword can attack *twice* each turn, rather than just once. We introduce a binary parameter w_i for friendly minion i:

$$w_i = \begin{cases} 1 & \text{if minion } i \text{ has Windfury,} \\ 0 & \text{otherwise.} \end{cases}$$

To enforce two possible attacks for minions with Windfury, and only one attack otherwise, we set:

$$\sum_{j=1}^{n} x(i, j) \; + \; x_{\text{hero}}(i) \;\; \leq \;\; 2 \cdot w_i \; + \; \big(1 - w_i\big) \quad \forall i,$$

where:

- $x(i, j)$ is a binary variable indicating whether minion i attacks enemy minion j.
- $x_{\text{hero}}(i)$ is a binary variable indicating whether minion i attacks the enemy hero.
- If $w_i = 1$, the right-hand side becomes 2, allowing up to two total attacks (on enemy minions or the hero).
- If $w_i = 0$, the right-hand side becomes 1, restricting the minion to a single attack that turn.

Behavioral Cases

1. **Windfury Active** ($w_i = 1$): Minion i may attack any combination of two targets, such as two separate enemy minions or one minion plus the enemy hero.
2. **No Windfury** ($w_i = 0$): Minion i is limited to a single attack in the turn, following the usual combat rules.

This single linear constraint succinctly captures the Windfury mechanic, shifting from an attack limit of 1 to 2 depending on whether w_i is set.

5.2.4 Battlecry

The variable b_i is a binary indicator for whether friendly minion j has the Battlecry keyword:

- $b_i = 1$: The friendly minion j has a Battlecry effect, which triggers immediately when the minion is played.
- $b_i = 0$: The friendly minion j does not have a Battlecry effect, so no additional effect is triggered upon playing it.

If a minion has Battlecry ($b_j = 1$), we introduce the following constraint to ensure that the Battlecry effect activates only once upon summoning:

$$e_j \leq u_j \cdot b_j$$

Where:

- e_j: Binary variable indicating whether the Battlecry effect of minion j is triggered.
- u_j: Binary variable indicating whether minion j is played on the current turn.
- b_j: Binary indicator for whether minion j has Battlecry (1 if Battlecry is active, 0 otherwise).

This constraint ensures that the Battlecry effect e_j only triggers if:

1. The minion j is played on the current turn ($u_j = 1$).
2. The minion j has the Battlecry keyword ($B_j = 1$).

This way, the model respects that Battlecry effects are single-use upon summoning and only activate for minions with the keyword.

5.2.5 Deathrattle for Friend Minions[III]

The variable D_i is a binary indicator for whether friendly minion i has the Deathrattle keyword:

- $D_i = 1$: The friendly minion i has a Deathrattle effect, which triggers immediately upon the minion's death.
- $D_i = 0$: The friendly minion i does not have a Deathrattle effect, so no additional effect is triggered upon its death.

If a minion has Deathrattle ($d_i = 1$), we introduce the following condition to ensure that the Deathrattle effect activates only when the minion dies:

$$e_{d_i} \leq (1 - y_i) \cdot D_i$$

Where:

- e_{D_i}: Binary variable indicating whether the Deathrattle effect of minion i is triggered.
- y_i: Binary indicator for whether minion i survives (1 if alive, 0 if dead).
- d_i: Binary indicator for whether minion i has Deathrattle (1 if Deathrattle is active, 0 otherwise).

5.2.6 Deathrattle for Enemy Minions[IV]

The variable e_j is a binary indicator for whether enemy minion j has the Deathrattle keyword:

[III] Note: This keyword is only discussed and not implemented in the Python code.
[IV] Note: This keyword is only discussed and not implemented in the Python code.

- $e_j = 1$: The enemy minion j has a Deathrattle effect, which triggers immediately upon the minion's death.
- $e_j = 0$: The enemy minion j does not have a Deathrattle effect, so no additional effect is triggered upon its death.

If an enemy minion has Deathrattle ($e_j = 1$), we introduce the following condition to ensure that the Deathrattle effect activates only when the minion dies:

$$e_j \leq (1 - z_j) \cdot e_j$$

Where:

- e_j: Binary variable indicating whether the Deathrattle effect of enemy minion j is triggered.
- z_j: Binary indicator for whether enemy minion j survives (1 if alive, 0 if dead).
- e_j: Binary indicator for whether enemy minion j has Deathrattle (1 if Deathrattle is active, 0 if not).

This condition ensures that $e_j = 1$ if:

1. The enemy minion j has died ($z_j = 0$).
2. The enemy minion j has the Deathrattle keyword ($E_j = 1$).

When $e_j = 1$, the Deathrattle effect of enemy minion j is triggered, affecting the board state accordingly.

5.3 Class-Specific Keywords

Minions in *Hearthstone* can also carry *class-specific keywords*, which appear only on cards from certain heroes. Whereas *common keywords* (e.g. Charge, Taunt, Windfury) are available to all classes and significantly affect the flow of play, these class-specific abilities introduce an additional layer of specialized strategy. In this work, we focus on two such keywords:

- **Rush** (Warrior): allows newly played Warrior minions to attack enemy minions immediately, but *not* the enemy hero.

- **Divine Shield** (Paladin): grants a minion one free instance of damage absorption.

When incorporating *Divine Shield* into our constraints, we consider both friendly minions that possess Divine Shield and enemy minions that also might have it. The model must handle these two scenarios distinctly, as each side's "shield" can prevent lethal damage once.

5.3.1 Warrior: Rush

The *Rush* keyword, commonly associated with the Warrior class, allows a newly summoned minion to attack *enemy minions* immediately on the same turn it is played. However, unlike *Charge*, Rush does *not* permit direct attacks on the enemy hero during the turn of summoning.

We represent this via a binary parameter R_i for each friendly minion i:

$$R_i = \begin{cases} 1 & \text{if minion } i \text{ has Rush (and is usually class-specific),} \\ 0 & \text{otherwise.} \end{cases}$$

Additionally, let u_i indicate whether minion i is actually played (summoned) this turn. If $R_i = 1$ and $u_i = 1$, the newly summoned minion i may immediately attack enemy *minions*, but not the enemy hero. Mathematically, if $x(i, j)$ denotes an attack by minion i on enemy minion j, we impose:

$$x(i, j) \;\leq\; \big(CH_i + R_i\big)\, u_i,$$

where CH_i is the Charge indicator. Thus:

- If $R_i = 1$ (*Rush* minion) but $CH_i = 0$, then $x(i, j) \leq u_i$. The minion can attack enemy minions if played this turn, but cannot attack the enemy hero (as that would require $CH_i = 1$).
- If neither Rush nor Charge is present ($R_i = 0$ and $CH_i = 0$), the right-hand side is zero, disallowing attacks immediately on the summoning turn.

In other words, a newly summoned minion with Rush can engage in *minion-to-minion* combat right away—crucial for Warrior's tempo plays—while a newly summoned *Charge* minion (or both) may also attack the opposing hero that turn.

5.3.2 Paladin: Divine Shield[V]

Minions with the Divine Shield keyword can absorb one instance of damage without losing health. This constraint models the protective effect of Divine Shield, ensuring that the minion does not take any damage the first time it is attacked. After the Divine Shield is removed, the minion will take normal damage from subsequent attacks.

$$y_i \leq 1 - \frac{(P_j - B_i + 1)}{P_j} \cdot x_{(i,j)} \cdot s_i + (1 - s_i) \quad \forall i, j$$

Where:

- y_i: Binary variable indicating whether the friendly minion i survives (1 if it survives, 0 if it dies).
- B_i: Health value of friendly minion i before the attack.
- P_j: Attack value of enemy minion j.
- $x_{(i,j)}$: Binary decision variable indicating whether friendly minion i attacks enemy minion j.
- s_i: Binary indicator for whether minion i has Divine Shield (1 if Divine Shield is active, 0 if removed).

This constraint ensures that Divine Shield negates damage from the first attack on a minion, allowing it to survive without health loss. Once Divine Shield is removed, the minion will be subject to standard survival conditions.

This constraint operates as follows:

- If Divine Shield is Active ($s_i = 0$): The minion i does not take any damage d_i from a single attack, effectively ignoring the impact of P_j. The Divine Shield then breaks, setting $s_i = 0$.
- If Divine Shield is Not Active ($s_i = 1$): The minion i takes normal damage according to the survival constraint, with the damage taken modeled by the term $\frac{(P_j - B_i + 1)}{P_j} \cdot x_{(i,j)}$.

[V] Note: This keyword is only discussed and not implemented in the Python code.

5.3.3 Paladin: Divine Shield for Friendly and Enemy Minions[VI]

Minions with the Divine Shield keyword can absorb one instance of damage without losing health. This constraint models the protective effect of Divine Shield, ensuring that a minion does not take any damage the first time it is attacked. After the Divine Shield is removed, the minion will take normal damage from subsequent attacks.

For friendly minions with Divine Shield, we have the following constraint:

$$y_i \leq 1 - \frac{(P_j - B_i + 1)}{P_j} \cdot x_{(i,j)} \cdot s_i + (1 - s_i) \quad \forall i, j$$

Where:

- y_i: Binary variable indicating whether the friendly minion i survives (1 if it survives, 0 if it dies).
- B_i: Health value of friendly minion i before the attack.
- P_j: Attack value of enemy minion j.
- $x_{(i,j)}$: Binary decision variable indicating whether friendly minion i attacks enemy minion j.
- s_i: Binary indicator for whether friendly minion i has Divine Shield (1 if Divine Shield is active, 0 if removed).

This constraint ensures that Divine Shield negates damage from the first attack on a friendly minion, allowing it to survive without health loss. Once Divine Shield is removed, the minion will be subject to standard survival conditions.

If an Enemy Minion Has Divine Shield

To prevent damage from being dealt to an enemy minion with Divine Shield, we add the following constraint:

$$z_j \leq 1 - x_{(i,j)} \cdot s_j \quad \forall i, j$$

Where:

[VI]Note: This keyword is only discussed and not implemented in the Python code.

- z_j: Binary variable indicating whether the enemy minion j survives (1 if it survives, 0 if it dies).
- s_j: Binary indicator for whether enemy minion j has Divine Shield (1 if Divine Shield is active, 0 if removed).
- $x_{(i,j)}$: Binary decision variable indicating whether friendly minion i attacks enemy minion j.

This constraint operates as follows:

1. **If Divine Shield is Active for Enemy Minion** ($s_j = 1$): The friendly minion i cannot deal any damage to enemy minion j, effectively protecting j from the attack. However, the enemy minion j can still deal damage to friendly minions if it attacks.
2. **If Divine Shield is Not Active for Enemy Minion** ($s_j = 0$): The friendly minion i can deal normal damage to the enemy minion j, following standard survival constraints.

This setup ensures that Divine Shield behaves correctly for both friendly and enemy minions, providing immunity from damage for one attack when the shield is active.

5.4 Extended Full Model

We now extend the Baseline Full Model (Section 4.3) to incorporate additional Hearthstone keywords that appear in our solver: *Charge*, *Rush*, and *Taunt*. We introduce new binary parameters and constraints to capture these mechanics.

5.4.1 Added Keyword Parameters

We define the following binary indicators for *friendly* minions:

$$CH_i = \begin{cases} 1 & \text{if minion } i \text{ has } \textit{Charge}, \\ 0 & \text{otherwise}, \end{cases} \qquad R_i = \begin{cases} 1 & \text{if minion } i \text{ has } \textit{Rush}, \\ 0 & \text{otherwise}. \end{cases}$$

Here, CH_i or R_i is fixed at 1 if minion i possesses that keyword; 0 otherwise. These values do not change during optimization but guide the solver's constraints on who may attack immediately upon being played.

For *enemy* minions, we have:

$$tt_j = \begin{cases} 1 & \text{if minion } j \text{ has } \textit{Taunt}, \\ 0 & \text{otherwise}, \end{cases}$$

where j indexes the enemy minions. If $tt_j = 1$, minion j must be attacked before the hero or any non-taunt minions.

5.4.2 Extra or Modified Constraints

All baseline constraints remain intact: each on-board minion can attack once, lethal checks for minion survival, board limit of 7 total friendly minions, mana constraints, etc. Below, we present only the new constraints or modifications that enable *Charge*, *Rush*, and *Taunt*.

Charge and Rush

A newly played minion $i \in \{m+1, \ldots, m+h\}$ cannot normally attack unless it has either *Charge* or *Rush*. We encode this as follows:

$$x_{(i,j)} \;\le\; \big(CH_i + R_i\big)\, u_i, \quad \forall\, j = 1, \ldots, n,$$

$$x_{(i,\text{hero})} \;\le\; CH_i\, u_i,$$

where u_i indicates that minion i is actually played this turn. Thus:

- $CH_i = 1$: The minion has *Charge*, allowing attacks on both enemy minions and the hero the turn it is played.
- $R_i = 1$: The minion has *Rush*, allowing attacks on enemy minions (but *not* the hero).
- If $(CH_i + R_i) = 0$, the newly played minion i may not attack at all this turn (i.e., all $x_{(i,j)}$ and $x_{(i,\text{hero})}$ are forced to 0).

Taunt

Enemy minions with *Taunt* ($tt_j = 1$) must be killed before attacking other minions or the hero. In other words, if any taunt minion j remains alive ($z_j = 1$), we disallow attacks on non-taunt minions or the hero. Conceptually, this can be written as:

$$z_j = 1 \implies x_{(i,\text{hero})} = 0, \quad \forall i,$$

blocking hero attacks if minion j with taunt is still up. Similarly,

$$z_k = 1 \implies x_{(i,j')} = 0, \quad \forall j' \text{ where } tt_{j'} = 0,$$

so that friendly minions must not attack non-taunt minions while any taunt k remains on the board.

5.4.3 Other Keywords or Effects

Any additional behaviors (e.g. *Battlecry*, *Deathrattle*, *Divine Shield*, *Windfury*) can be modeled similarly, using binary flags to indicate whether a minion has that property and additional constraints to handle triggers or lethal checks. For instance, *Divine Shield* might allow one "free" avoidance of lethal damage, while *Windfury* enables a second attack per turn. However, we omit detailed implementation of these effects here, as they are not crucial to our current example.

5.4.4 Extended Full Model

$$\begin{aligned}\text{Maximize:}\quad & W_1 \cdot (1 - z_{\text{hero}}) + W_2 \cdot c \\ & + W_3 \cdot \sum_{i=1}^{m+h} A_i \cdot x_{(i,\text{hero})} - W_4 \cdot \sum_{j=1}^{n} P_j \cdot z_j \\ & + W_5 \cdot \sum_{i=1}^{m} B_i \cdot y_i + W_6 \cdot \sum_{i=1}^{m+h} \sum_{j=1}^{n} A_i \cdot x_{(i,j)} \\ & - W_7 \cdot \sum_{i=1}^{m+h} \sum_{j=1}^{n} P_j \cdot x_{(i,j)} + W_8 \cdot \sum_{i=m+1}^{m+h} S_i \cdot u_i\end{aligned}$$

Subject to:

$$(1) \ldots (8)$$

$$x_{(i,j)} \;\leq\; (CH_i + R_i)\, u_i, \qquad \forall\, i \in \{m+1, \ldots, m+h\}, \; \forall\, j \in \{1, \ldots, n\} \tag{9}$$

$$x_{(i,\text{hero})} \;\leq\; CH_i\, u_i, \qquad \forall\, i \in \{m+1, \ldots, m+h\} \tag{10}$$

$$x_{(i,\text{hero})} \;\leq\; 1 - \sum_{k=1}^{n} \big(\texttt{tt}_k \cdot z_k\big), \qquad \forall\, i \in \{1, \ldots, m+h\} \tag{11}$$

$$x_{(i,j)} \;\leq\; 1 - \sum_{k=1}^{n} \big(\texttt{tt}_k \cdot z_k\big), \qquad \forall\, i \in \{1, \ldots, m+h\},\; \forall\, j : \texttt{tt}_j = 0 \tag{12}$$

x, y, c, z, and u are all binary decision variables.

Chapter 6

Single Round Implementation

This chapter focuses on the practical implementation of our optimization model in Python, demonstrating how the theoretical framework becomes an interactive, AI-driven decision-making process. Our initial aim is to handle a single turn—one AI player's series of moves based on the solver's recommendations. Once we can reliably execute a single turn, we combine two consecutive turns (one per AI) to form a complete round, ensuring both players have acted. Although each AI relies on the same underlying algorithm, their decisions remain independent, reflecting any changes in the game state after each turn.

6.1 Turn-Based AI Execution

A single turn consists of six consecutive steps:

1. **Turn 1 (AI 1's Move)**:
    - AI 1 begins its turn.
    - AI 1 gains one mana crystal (if below 10).
    - AI 1 draws a card (if the deck is not empty).
    - AI 1 executes the Gurobi optimization model.
    - AI 1 applies the optimized moves to the board.
    - The game state updates, preparing for AI 2's turn.

6.2 Round-Based AI Execution

A single round consists of two consecutive turns:

1. **Turn 1 (AI 1's Move)**:
   - AI 1 begins its turn.
   - AI 1 gains one mana crystal (if below 10).
   - AI 1 draws a card if the deck is not empty; otherwise, it suffers fatigue damage (increasing by 1 each round).
   - AI 1 executes the Gurobi optimization model.
   - AI 1 applies the optimized moves to the board.
   - The game state updates, preparing for AI2's turn.
2. **Turn 2 (AI 2's Move)**:
   - AI 2 begins its turn.
   - AI 2 gains one mana crystal (if below 10).
   - AI 2 draws a card if the deck is not empty; otherwise, it suffers fatigue damage (increasing by 1 each round).
   - AI 2 executes the Gurobi optimization model.
   - AI 2 applies the optimized moves to the board.
   - The game state is updated, and the round is complete.

Each AI functions independently, ensuring fairness and eliminating information imbalance while still following an optimal strategy.

6.3 Python Workflow I: Solver Setup

At a high level, our Python code processes a single round (two turns, one per AI) through these steps:

1. **Gather Inputs:** Retrieve the current board states (friendly and enemy minions), remaining hero health, mana totals, and any relevant hand cards.
2. **Build Solver Model:** Define binary variables, constraints, and the objective function according to the selected weights ($W_1 \dots W_8$).

3. **Optimize and Extract Results:** Invoke the Gurobi solver, then parse variables indicating which minions attack and which cards are played.
4. **Apply Actions:** Update minion health, remove any that die, track used mana, and check for lethal damage.
5. **Conclude the Turn:** If the enemy hero is dead, the game ends; otherwise, proceed to the second AI's turn or finish the round.

The function `run_single_turn` in `slv.py` builds and solves a mixed-integer linear program (MILP) using Gurobi for a single turn of play. It handles decision-making about attacks, card plays, survivals, and constraints. Below, we walk through its structure and logic with embedded code fragments.

First, the function takes a number of input parameters, including game state details (minions on board, health, mana, attack/health values), card costs, and optional keyword flags like Charge or Taunt:

```
def run_single_turn(m, n, h, M, H_hero, A, B, P, Q, C, S,
                    has_charge=None, has_rush=None,
                    has_taunt=None, has_windfury=None,
                    has_ds_friendly=None, has_ds_enemy=None,
                    weights=None):
```

Default values are assigned to any missing keyword lists (e.g., Charge, Rush, Divine Shield), and if no weights are provided, equal weights are assumed:

```
if has_charge is None:
    has_charge = [False]*(m+h)
...
if weights is None:
    weights = {"W1":1,"W2":1,"W3":1,"W4":1,"W5":1,"W6":1,"W7":1,"W8":1}
W1, W2, ..., W8 = weights["W1"], ..., weights["W8"]
```

A Gurobi model is then created, with output suppressed:

```
model = gp.Model("FullModel_DS_WF_Corrected")
model.setParam("OutputFlag", 0)
```

Next, the decision variables are defined:

- `x_hero[i]`: whether friendly minion i attacks the enemy hero.
- `x[i,j]`: whether minion i attacks enemy minion j.
- `y[i]`, `z[j]`: survival of friendly and enemy minions.

- `z_hero`: survival of enemy hero.
- `c_`: whether enemy board is cleared.
- `u[i]`: whether card i is played this turn.

```
x_hero = model.addVars(m+h, vtype=GRB.BINARY, name="AttackHero")
x = model.addVars(m+h, n, vtype=GRB.BINARY, name="AttackMinion")
y = model.addVars(m+h, vtype=GRB.BINARY, name="FriendlySurvives")
z = model.addVars(n, vtype=GRB.BINARY, name="EnemySurvives")
z_hero = model.addVar(vtype=GRB.BINARY, name="EnemyHeroSurvives")
c_ = model.addVar(vtype=GRB.BINARY, name="EnemyBoardCleared")
u = model.addVars(range(m, m+h), vtype=GRB.BINARY, name="PlayCard")
```

Divine Shield bonuses are applied to health:

```
B_prime = [B[i] + (1 if has_ds_friendly[i] else 0) for i in range(m+h)]
Q_prime = [Q[j] + (1 if has_ds_enemy[j] else 0) for j in range(n)]
```

The objective function maximizes a weighted sum of goals, such as face damage, clearing the enemy board, friendly survival, and card value:

```
obj = (
    W1*(1 - z_hero)
    + W2*c_
    + W3*gp.quicksum(A[i]*x_hero[i] for i in range(m+h))
    - W4*gp.quicksum(P[j]*z[j] for j in range(n))
    + W5*gp.quicksum(B[i]*y[i] for i in range(m+h))
    + W6*gp.quicksum(A[i]*x[i,j] for i in range(m+h) for j in range(n))
    - W7*gp.quicksum(P[j]*x[i,j] for i in range(m+h) for j in range(n))
    + W8*gp.quicksum(S[i]*u[i] for i in range(m, m+h))
)
model.setObjective(obj, GRB.MAXIMIZE)
```

Constraints follow, ensuring legality:

- Each minion may attack once (or twice with Windfury).
- Survival constraints reflect total damage vs. adjusted health.
- Newly summoned minions can only act if played and have the correct keyword.
- Mana and board size limits are respected.

For example, attack limits:

```
for i in range(m):
    w_i = 1 if has_windfury[i] else 0
    attack_limit = 2 * w_i + (1 - w_i)
    model.addConstr(gp.quicksum(x[i,j] for j in range(n)) + x_hero[i] <=
    attack_limit)
```

And for board size:

```
model.addConstr(
    gp.quicksum(y[i] for i in range(m)) + gp.quicksum(u[i] for i in range(m
    , m+h)) <= 7
)
```

Taunt constraints block attacks on the hero or non-taunt minions if any taunt minion is alive:

```
if any(has_taunt[k] for k in range(n)):
    for i in range(m + h):
        for k in range(n):
            if has_taunt[k]:
                model.addConstr((z[k] == 1) >> (x_hero[i] == 0))
        for j in range(n):
            if not has_taunt[j]:
                for k in range(n):
                    if has_taunt[k]:
                        model.addConstr((z[k] == 1) >> (x[i,j] == 0))
```

Finally, the model is solved and results are extracted:

```
model.optimize()
result = {
    "status": model.status,
    "objective": None,
    "x_hero": {}, "x_minions": {},
    "z_hero": None, "z_enemy": {},
    "c_clear": None, "y_survive": {},
    "cards_played": {}
}
if model.status in [GRB.OPTIMAL, GRB.SUBOPTIMAL]:
    result["objective"] = model.objVal
    ...
```

This result dictionary includes all key decisions: attacks, survivals, whether the enemy hero survived, board clear status, and which cards were played.

In summary, `run_single_turn` builds a turn-level MILP using current board state and card info:

1. It declares binary decision variables for all possible actions.
2. It imposes constraints reflecting game rules.
3. It optimizes a weighted reward function across multiple strategic dimensions.
4. It returns an interpretable dictionary of decisions for downstream application.

This modular and weight-driven structure enables tuning for different strategies and smooth integration into turn-by-turn simulation or reinforcement learning pipelines.

6.4 Python Workflow II: Solver Results Interpret

After the solver finishes running inside the main game loop, the script interprets the result and applies the solver's chosen actions to update the game state. The key steps are:

1. **Run the solver with prepared inputs.** Inputs are generated each turn via:

```
solver_inputs = prepare_solver_inputs(game, weights_for_this_turn)
solution = run_single_turn(**solver_inputs)
```

2. **Check the solver status.** Only if the model is optimal or suboptimal do we proceed:

```
if solution and solution['status'] in [GRB.OPTIMAL, GRB.SUBOPTIMAL]:
    print("  Applying optimal/suboptimal solver results.")
    apply_solver_results(solution, game)
else:
    print(f"  Solver failed or found no solution (Status: {status_code
    }). Ending turn passively.")
```

3. **Interpret individual solver outputs.** The solver returns a dictionary like:

```
result = {
    "x_hero": {...},       # Which minions attacked the enemy hero
    "x_minions": {...},    # Minion-vs-minion attacks
    "cards_played": {...}, # Which hand cards were played
    "y_survive": {...},    # Friendly survival outcomes
    "z_enemy": {...},      # Enemy survival outcomes
```

```
    "z_hero": ...,          # Enemy hero survived?
    "c_clear": ...,         # Enemy board cleared?
    ...
}
```

Each key contains binary values (0 or 1). For example:

- If `solution["x\_hero"][i] == 1`, then minion i attacked the enemy hero.
- If `solution["x\_minions"][(i,j)] == 1`, then minion i attacked enemy minion j.
- If `solution["cards\_played"][k] == 1`, then hand card k was played this turn.

These are used inside the `apply_solver_results` function to directly update the internal `GameState` object.

4. **Prints and logging support interpretation.** Throughout the turn, the engine includes debug output to trace execution:

```
print(f">>> Turn {game.turn} - {active_player.hero.hero_class}'s Turn
    (AI Strategy {active_player_index + 1}) <<<")
print(f"  Preparing inputs with weights: {weights_for_this_turn}")
print("  Running Gurobi Solver...")
print("  Solver finished.")
```

5. **Game continues to next turn.** If the solver ran successfully, the turn ends with a call to:

```
end_turn_procedures(game)
```

which handles switching players and incrementing the turn counter.

This process ensures that every solver result is grounded in a consistent interpretation and correctly modifies the live game state. When the game ends, final prints give a detailed summary:

```
print("========== GAME OVER ==========")
print(f"Total Turns: {game.turn}")
print(f"--- Final State Player 1 ({game.player1.hero.hero_class}) ---")
print(f"--- Final State Player 2 ({game.player2.hero.hero_class}) ---")
```

These outputs help verify solver behavior and validate decision logic over time.

6.5 Python Workflow III: Results Apply to Game State

Once the solver's output is received and confirmed valid, the function `apply_solver_results` executes all selected actions by modifying the `GameState`. This includes playing cards, resolving attacks, and updating health values. The process unfolds in distinct steps:

1. **Play Cards:** The solver may choose to play one or more minion cards from hand. For each `u_k = 1`, the code deducts the mana cost and adds the card to the board, assuming legal conditions (mana available and board space):

```
for i_solver in range(m, m + h):
    hand_index = i_solver - m
    if hand_index < len(initial_hand) and solution['cards_played'].get
    (i_solver, 0) > 0.5:
        minion_to_play = initial_hand[hand_index]
        if len(active_player.board) < MAX_BOARD_SIZE and active_player
    .mana >= minion_to_play.mana_cost:
            active_player.mana -= minion_to_play.mana_cost
            active_minion = ActiveMinion(minion_to_play)
            active_player.board.append(active_minion)
```

2. **Resolve Attacks (Minion vs Minion):** If $x(i, j) = 1$, minion i attacks enemy minion j. Damage is exchanged based on the attacker's and target's stats, and Windfury is respected:

```
for i_solver in range(m + h):
    attacker_am = get_friendly_attacker(i_solver)
    for j_solver in range(n):
        if solution['x_minions'].get((i_solver, j_solver), 0) > 0.5:
            target_am = get_enemy_target(j_solver)
            if attacker_am.can_attack and attacker_am.
    attacks_this_turn < (2 if attacker_am.has_keyword("Windfury") else
     1):
                target_am.take_damage(attacker_am.attack)
                attacker_am.take_damage(target_am.attack)
                attacker_am.attacks_this_turn += 1
```

3. **Resolve Attacks (Minion vs Hero):** If $x(i, \text{hero}) = 1$, the corresponding minion attacks the enemy hero. Health is reduced accordingly:

```
if solution['x_hero'].get(i_solver, 0) > 0.5:
    if attacker_am.can_attack and attacker_am.attacks_this_turn < (2
    if attacker_am.has_keyword("Windfury") else 1):
```

```
        opponent.hero.health -= attacker_am.attack
        attacker_am.attacks_this_turn += 1
```

4. **Remove Dead Minions:** Any minion with health ≤ 0 is removed from the board:

```
active_player.board = [m for m in active_player.board if not m.is_dead
    ()]
opponent.board = [m for m in opponent.board if not m.is_dead()]
```

5. **Check for Game Over:** If a hero's health reaches 0 or below, the game ends:

```
game_state.check_game_over()
```

Before applying solver actions, the code captures the initial board and hand states:

```
initial_friendly_board = list(active_player.board)
initial_enemy_board = list(opponent.board)
initial_hand = list(active_player.hand)
```

These snapshots allow reconstruction of valid indices and ensure proper minion tracking during actions.

Turn Management

Each turn consists of a structured flow:

1. **Start of Turn:** `start_turn_procedures` restores mana and wake-up state for the active player:

```
def start_turn_procedures(game_state: GameState):
    player = game_state.get_active_player()
    if player.max_mana < MAX_MANA:
        player.max_mana += 1
    player.mana = player.max_mana
    for minion in player.board:
        minion.can_attack = True
        minion.attacks_this_turn = 0
    player.draw_card()
    game_state.check_game_over()
```

2. **End of Turn:** After applying solver decisions, the game switches to the next player:

```
def end_turn_procedures(game_state: GameState):
    game_state.switch_player()
    if game_state.current_player_index == 0:
        game_state.turn += 1
```

Once the solver's actions are resolved and any lethal checks are made, the code logs relevant data (e.g., total damage dealt, which minions remain) and ends the active player's turn. If this was the first turn of the round, control passes to the second AI. If both AIs have acted, the round finishes. Should lethal or another terminal event occur at any point, the match is recorded as completed, and no further actions are needed.

With these steps, we complete one full turn. Two consecutive turns (one per AI) form a single round. In **Chapter 7**, we will build on this workflow to showcase multi-round or multi-match experiments, testing how different weight sets ($W_1 \dots W_8$) fare against one another.

Chapter 7

Loop Implementation

This chapter culminates our work by demonstrating how single-turn and single-match logic scale up into larger simulations. Our ultimate goal is to automate repeated gameplay across multiple weight configurations, gathering robust statistics on each AI style's performance. We show how the Python code cycles through full games for every strategy pairing, accumulates outcomes (win rates, average turns, etc.), and yields empirical comparisons that validate (or challenge) our theoretical assumptions about different playstyles.

7.1 Deck

Before running multi-turn loops, each player requires a full deck of 30 cards. We construct these decks by selecting minions from Hearthstone but limiting their keywords to only *Charge*, *Rush*, or *Taunt*, removing other abilities for simplicity. Each 15-card list is then duplicated (multiplied by 2) to reach the 30-card total. At the start of the match, each hero draws 4 cards.

- **Paladin Deck** (30 Cards): Initially defined as a 15-card list of Paladin and Neutral minions, including basic *Taunt* or *Charge* examples like `Righteous Protector`, `Leeroy Jenkins`, and `Tirion Fordring`. We then double this list to achieve 30 cards:

```
p1_deck = [
    Minion("Righteous Protector", "Paladin", ["Taunt"], attack=1,
    health=1, ...),
    Minion("Argent Squire", "Paladin", [], attack=1, health=1, ...),
    Minion("Chillblade Champion", "Paladin", ["Charge"], ...),
    ...
```

```
]
paladin_deck = p1_deck * 2
```

- **Warrior Deck** (30 Cards): Similarly, the Warrior deck is made by taking a 15-card list of Warrior and Neutral minions, some with *Charge*, others with *Rush* or *Taunt*, and duplicating it:

```
p2_deck = [
    Minion("Warbot", "Warrior", ["Rush"], attack=1, health=3, ...),
    Minion("Town Crier", "Warrior", ["Rush"], ...),
    Minion("Armorsmith", "Warrior", [], ...),
    ...
]
warrior_deck = p2_deck * 2
```

All of these listed minions are legitimate Hearthstone minions, but we have removed extra keywords (e.g., Battlecry, Divine Shield) to focus on *Charge*, *Rush*, and *Taunt*. Each side begins the game by drawing four cards from their 30-card deck, with additional draws occurring at the start of each turn, unless the deck is empty (in which case fatigue damage applies). This setup offers a simplified yet varied pool of minions for testing our solver's decision-making logic in multi-turn scenarios.

7.2 Transition

Another crucial consideration, before we expand to multi-turn or multi-round loops, is ensuring that each turn's evolving board, resources, and hero attributes are properly handled. In the Python code, these updates are made in functions such as `start_turn_procedures` (for beginning-of-turn modifications) and `apply_results` (for post-combat state changes). Building on the single-turn logic from the last chapter, this additional bookkeeping ensures that, at each loop iteration, the AI always has up-to-date information and can detect if a hero's health falls to zero (or below), triggering the end of the game.

7.2.1 Board State Variables

The battlefield evolves dynamically with every turn or action, and the Python code updates these variables accordingly to reflect minion interactions:

Minion Presence and Attributes

- **Number of Minions (m, n):** As minions are summoned or destroyed, the code adjusts m (friendly) and n (enemy). If a minion's health hits zero in `apply_results`, it is removed from the board array.
- **Health Values (B_i, Q_j)**: Each minion's health changes due to attacks or spells. In Python, arrays like B (friendly health) and Q (enemy health) track these updates after each turn.
- **Exhaustion Status**: Once a minion attacks, Python code marks it "exhausted" to prevent multiple attacks (unless special keywords like Windfury are active).

7.2.2 Resource Variables

Resource management is central to Hearthstone, and the code checks or modifies these values each turn to reflect changing options:

Mana Availability

- At the *start of a turn*, Python increments the active player's mana crystals by 1 (capped at 10). This is usually done in a function like `start_turn_procedures`.
- Cards that manipulate mana temporarily can also appear, though the baseline logic simply enforces $\sum C_k \cdot u_k \leq M$ in the solver constraints.

Card Draw and Deck Size

- The active player typically **draws 1 card** from the deck at turn start (if the deck is not empty).
- The Python code decrements the deck counter or removes a card from the deck list. If the deck is empty, subsequent draws cause fatigue damage.
- Any card played ($u_k = 1$ in the solver) is removed from the player's hand array.

7.2.3 Hero Attributes

Hero status is crucial for detecting lethal and deciding between offensive or defensive play:

Hero Health and Armor

- **Hero health** H_{hero} updates when minion attacks or direct damage spells are applied. In Python, once any hero's health ≤ 0, the game ends.
- Some classes, like Warrior, track **armor**, which absorbs damage before health is reduced. If armor is used, the code deducts from armor first, then from health if armor is depleted.

Hero Power and Weapon Usage

- Each hero power can be invoked once per turn, though your baseline code may ignore these powers unless you specifically code them.
- Weapon usage modifies hero attack/durability, but the baseline solver might skip this unless explicitly modeled.

7.2.4 Checking Lethal and Proceeding in the Loop

After each turn, the Python code checks if either hero's health ≤ 0. If so, the game (or round) ends, and results are logged (winner, turn count, etc.). Otherwise, the loop proceeds to the next turn. By capturing these evolving board states, resource levels, and hero attributes, we ensure the solver's inputs remain accurate and the AI adapts to every strategic shift throughout the match.

7.3 Turn Loop in the Gurobi Code

Rather than separating the turn into distinct phases, our Python code uses a continuous loop that handles each step in a single pass. For each turn:

1. **Initialize Turn Data:** The program refreshes information about the board state, hero health, and available mana. If the active player gains a new mana crystal (up to 10) or draws a card, those updates are made here.
2. **Construct and Solve the Model:** The function `run_single_turn` (in `slv.py`) defines binary variables (e.g., $x(i, j)$, $x(i, \text{hero})$, u_k), enforces constraints (mana, attack limits, lethal checks), and sets the objective with the chosen weights $(W_1 \dots W_8)$. A call to `model.optimize()` invokes Gurobi, returning an optimal set of actions.

3. **Apply Solver Results:** Once the solution is obtained, the code interprets which minions attack, which cards are played, and whether the hero is targeted. For each chosen action, the program updates:

   - *Minion Attacks*: Deduct A_i from the target minion's health (and handle return damage P_j).
   - *Hero Damage*: Subtract A_i from the enemy hero's health if $x(i, \text{hero}) = 1$.
   - *Card Plays*: Reduce mana by the card's cost C_k and place the corresponding minion or spell effect on the board.

   If a minion's health reaches zero or less, it is removed. If the hero's health is ≤ 0, the game ends.

4. **Finalize Turn and Transition:** With all actions executed, the code logs any relevant data (e.g. total damage dealt, minions left alive). If lethal occurred, this turn loop exits and declares the winner. Otherwise, control passes to the next player's turn.

By consolidating these steps into a single loop, the Python program ensures each turn is consistently evaluated, optimized, and resolved before moving on. This loop structure can then be repeated to form rounds (two turns, one per player) and extended across multiple rounds until one hero's health is reduced to zero or another end condition is reached.

7.4 Weights

To compare different AI playing styles, seven distinct strategies were defined by assigning unique sets of weights (W_1 through W_8) to the objective function of the turn-based optimization model. These weights guide the AI's priorities when selecting actions (playing cards, attacking minions, attacking the hero). The following sections detail the weight dictionary and the strategic reasoning for each of the six implemented AI personalities.

Recall the meaning of the weights:

- W_1: **Lethal Focus.** *Reward* for killing the enemy hero. A higher W_1 prioritizes lethal damage.
- W_2: **Board Clear.** *Reward* for wiping out all enemy minions. A large W_2 strongly values an empty opposing board.

- W_3: **Hero Damage Pressure.** *Reward* for direct attacks on the enemy hero. Increasing W_3 encourages going face.
- W_4: **Threat Removal.** *Penalty* for leaving high-attack enemy minions alive. If $z_j = 1$ for a big minion j, a higher W_4 discourages ignoring it.
- W_5: **Self Preservation.** *Reward* for keeping friendly minions alive (multiplied by their remaining health). A large W_5 emphasizes board longevity.
- W_6: **Minion Damage Focus.** *Reward* for dealing damage to enemy minions. A higher W_6 promotes active trades or pings.
- W_7: **Trade Efficiency.** *Penalty* for the damage your minions take in trades. A large W_7 deters risky combat with high-attack foes.
- W_8: **Hand Play Value.** *Reward* for playing high-S_i cards. Raising W_8 incentivizes summoning powerful or synergistic minions/spells from your hand.

7.4.1 Strategy Definitions

Strategy I: Keep Alive

Goal: Prioritize preserving own board presence and minion health, avoiding risky trades.

Weights:

```
WEIGHTS_KEEP_ALIVE = {
    "W1": 2,  "W2": 4,  "W3": 1,  "W4": 6,
    "W5": 20, # <<< Very High
    "W6": 8,  "W7": 5,  "W8": 6
}
```

Explanation: The dominant weight is $W_5 = 20$, strongly incentivizing actions that lead to friendly minions surviving with maximum health. Face damage ($W_1 = 2, W_3 = 1$) and total board clear ($W_2 = 4$) are low priorities. Threat removal ($W_4 = 6$) and damaging enemy minions ($W_6 = 8$) have moderate values, helping preserve your own minions. Trade efficiency ($W_7 = 5$) is non-negligible but still lower, while playing cards ($W_8 = 6$) is secondary. The AI aims to maintain a durable board, avoiding adverse combat whenever possible.

Strategy II: Board Clear

Goal: Prioritize removing all enemy minions to achieve total board control.

Weights:

```
WEIGHTS_BOARD_CLEAR = {
    "W1": 1,  "W2": 20, # <<< Very High
    "W3": 1,  "W4": 15, # <<< High
    "W5": 5,  "W6": 15, # <<< High
    "W7": 10, # <<< High
    "W8": 8
}
```

Explanation: Achieving a full board clear ($W_2 = 20$) is the top priority. Closely related factors are heavily weighted: removing large threats ($W_4 = 15$), damaging enemy minions ($W_6 = 15$), and trading efficiently ($W_7 = 10$) to maximize kill potential. Hero damage ($W_1 = 1, W_3 = 1$) and preserving own minions ($W_5 = 5$) matter much less. Playing high-value cards ($W_8 = 8$) ranks moderately, but the AI primarily focuses on eliminating opposing minions.

Strategy III: Aggro Face

Goal: Maximize direct damage to the enemy hero as quickly as possible.

Weights:

```
WEIGHTS_AGGRO_FACE = {
    "W1": 20, # <<< Very High
    "W2": 2,  "W3": 18, # <<< Very High
    "W4": 3,  "W5": 1,  "W6": 3,
    "W7": 1,  "W8": 10
}
```

Explanation: This strategy strongly favors lethal ($W_1 = 20$) and face damage ($W_3 = 18$). Playing powerful cards ($W_8 = 10$) is also relevant, especially if they contribute to direct hero damage. By contrast, board control values ($W_2 = 2, W_4 = 3, W_5 = 1, W_6 = 3, W_7 = 1$) remain negligible unless forced to trade with Taunts or remove minimal threats. Overall, the AI aims to end the game rapidly by targeting the opponent's health.

Strategy IV: Value Trading

Goal: Control the board by making efficient minion trades and removing key enemy threats.

Weights:

```
WEIGHTS_VALUE_TRADE = {
    "W1": 3,  "W2": 8,  "W3": 2,
    "W4": 18, # <<< High
    "W5": 10, # Moderate
    "W6": 16, # <<< High
    "W7": 16, # <<< High
    "W8": 12  # Moderate
}
```

Explanation: High emphasis on removing large threats ($W_4 = 18$) and dealing minion damage ($W_6 = 16$) while minimizing damage to your own minions ($W_7 = 16$). Preserving your board ($W_5 = 10$) and playing high-value cards ($W_8 = 12$) support a resource advantage. Direct hero damage ($W_1 = 3, W_3 = 2$) and a complete board clear ($W_2 = 8$) rank lower. This AI aims to accumulate incremental advantages through careful trades, eventually overwhelming the opponent.

Strategy V: Aggro-Tempo

Goal: Combine aggression with enough board play to maintain tempo.

Weights:

```
WEIGHTS_AGGRO_TEMPO = {
    "W1": 15, # <<< High
    "W2": 8,  "W3": 6,  "W4": 8,
    "W5": 1,  "W6": 10, # <<< Moderate
    "W7": 1,  "W8": 3
}
```

Explanation: With $W_1 = 15$ leading the priority on lethal, the AI balances some degree of board interaction ($W_2 = 8, W_4 = 8, W_6 = 10$) to remove critical threats or maintain momentum. Preserving friendly minions ($W_5 = 1$) and trade efficiency ($W_7 = 1$) rank very low, indicating willingness to trade less favorably if it advances the aggressive plan. Hero damage pressure ($W_3 = 6$) is moderate, complementing lethal attempts. Playing cards for synergy ($W_8 = 3$) is not a major focus.

Strategy VI: True Balanced

Goal: Serve as a baseline with no strong bias toward any aspect of play.

Weights:

```
WEIGHTS_TRUE_BALANCED = {
    "W1": 5, "W2": 5, "W3": 5, "W4": 5,
    "W5": 5, "W6": 5, "W7": 5, "W8": 5
}
```

Explanation: All weights are set equally. This AI does not prefer lethal over board presence or trades over direct damage. Any outcome's relative importance arises from the raw in-game values (e.g., the actual damage or health), not from the weighting scheme. Useful for comparing with specialized strategies.

Strategy VII: Author Favorite

Goal: Achieve a balanced focus on lethal, board control, and minion damage while lightly penalizing risky trades.

Weights:

```
WEIGHTS_AUTHOR_FAVORITE = {
    "W1": 24, # <<< Very High
    "W2": 20, # <<< High
    "W3": 10, # <<< Moderate
    "W4": 20, # <<< High
    "W5": 4,  # <<< Low
    "W6": 24, # <<< Very High
    "W7": 4,  # <<< Low
    "W8": 8   # <<< Moderate
}
```

Explanation: Here the author proposes a configuration believed to best balance lethal potential, removing large enemy threats, and dealing minion damage:

- *High Lethal Emphasis:* $W_1 = 24$ ensures the AI values killing the hero if possible.
- *Strong Board Focus:* Board clear ($W_2 = 20$) and minion damage ($W_6 = 24$) remain high, compelling the AI to remove or weaken enemy minions aggressively.
- *Threat Removal:* $W_4 = 20$ penalizes leaving high-attack enemies alive, steering the solver toward timely kills.

- *Moderate Card Usage:* $W_8 = 8$ rewards playing high-value cards, but not as strongly as lethal or board presence.
- *Light Self Preservation:* $W_5 = 4$ invests little in friendly minion survival, and $W_7 = 4$ imposes only mild penalties for taking damage in trades—signaling a willingness to sacrifice minions if it secures or maintains board advantage or lethal setup.

Overall, this distribution combines the author's preference for swiftly removing threats, sustaining strong offensive pressure, and leveraging key card plays while not overly penalizing minion casualties.

7.5 Loop Implementation

After setting up our single-turn solver and incorporating keyword logic, we extend the system to run full matches and compare strategy performance across many simulations. We define *seven* weight configurations (e.g., `WEIGHTS_1` through `WEIGHTS_7`) and evaluate how they perform against each other in head-to-head games.

7.5.1 Implementation in Python

We rely on two main functions to run repeated games:

`simulate_single_game(weights_p1, weights_p2)` Runs a complete match (many turns) between two AIs using specified weight sets. Each turn alternates between players until one hero dies. The function returns the winner and number of turns:

1. **Initialize game state:** Set up decks, heroes, and draw starting hands.
2. **Main game loop:**
   (a) `start_turn_procedures`: increments mana, draws a card, and resets attack status.
   (b) **Prepare solver inputs:** collect board state, hand, health, and current weights.
   (c) `run_single_turn`: use Gurobi to compute optimal plays.
   (d) `apply_solver_results`: update the board and health values.

(e) `end_turn_procedures`: switch player and continue.

3. **End condition:** When a hero dies, return the winner and turn count.

```
def simulate_single_game(weights_p1, weights_p2, verbose=False):
    game = initialize_game()
    player_weights = [weights_p1, weights_p2]

    while not game.game_over:
        if verbose:
            game.display_full_state()

        active_player_index = game.current_player_index
        start_turn_procedures(game)
        if game.game_over:
            break

        solver_inputs = prepare_solver_inputs(game, player_weights[
    active_player_index])
        solution = run_single_turn(**solver_inputs)

        if solution and solution["status"] in [GRB.OPTIMAL, GRB.SUBOPTIMAL
    ]:
            apply_solver_results(solution, game)

        end_turn_procedures(game)

    return game.winner, game.turn
```

`run_full_experiments(num_games)` Simulates every pair of strategies against each other across many matches (default 1000). Results are stored in a nested dictionary indexed by pair (i, j).

- Loops through all pairs of weight sets using `enumerate(WEIGHTS_LIST)`.
- For each (i, j) pair, it runs `num_games` simulations using the two strategies.
- After each match, it tallies wins, draws, and turn counts.

```
def run_full_experiments(num_games):
    results = {}
    for i, w_p1 in enumerate(WEIGHTS_LIST):
        for j, w_p2 in enumerate(WEIGHTS_LIST):
```

```
            p1_wins, p2_wins, draws, total_turns = 0, 0, 0, 0

            for _ in range(num_games):
                winner, turns = simulate_single_game(w_p1, w_p2, verbose=
    False)
                total_turns += turns

                if winner == "Draw":
                    draws += 1
                elif winner == "Paladin":
                    p1_wins += 1
                elif winner in ["Warrior", "Hunter"]:
                    p2_wins += 1

            avg_turns = total_turns / num_games
            results[(i, j)] = {
                "p1_wins": p1_wins,
                "p2_wins": p2_wins,
                "draws": draws,
                "avg_turns": avg_turns
            }
    return results
```

`print_results_table(results)` Prints each result in a readable summary, one row per pairing:

```
def print_results_table(results):
    print("\n=== Strategy Pairing Results (6x6) ===\n")
    for i in range(len(WEIGHTS_LIST)):
        for j in range(len(WEIGHTS_LIST)):
            stats = results[(i, j)]
            print(f"(P1={i}, P2={j}) -> "
                  f"p1_wins={stats['p1_wins']}, "
                  f"p2_wins={stats['p2_wins']}, "
                  f"draws={stats['draws']}, "
                  f"avg_turns={stats['avg_turns']:.2f}")
        print()
```

7.5.2 Main Driver

The following block runs the full experiment and prints the results if the file is executed directly:

```
if __name__ == "__main__":
    final_results = run_full_experiments(num_games=1000)
    print_results_table(final_results)
```

7.5.3 Explanation

This experiment framework tests every strategy pairing exhaustively:

- **simulate_single_game**: Handles one full match between two strategies.
- **run_full_experiments**: Loops over all (i, j) weight combinations.
- **1000 simulations**: For each pair, this high volume smooths out randomness.
- **Result structure**: The dictionary `results[(i,j)]` records win counts, draws, and average turn length.

This setup allows us to quantitatively evaluate each strategy's performance against others, revealing which AIs dominate, draw evenly, or fall behind in different head-to-head scenarios.

7.6 Results

To evaluate each AI strategy under competitive conditions, we simulate all 7×7 matchups, where each player uses one of the seven weight configurations described earlier. Every pairing is repeated **1000 times**, and the outcomes are summarized in Table 7.1.

Each cell in the table contains:

- `p1_wins`: how often Player 1 (who goes first) wins,
- `p2_wins`: how often Player 2 wins,
- `avg_turns`: average number of turns until the game ends.

These simulations assume no draws occurred (as verified), and no second-player compensation like "The Coin" was used.

Table 7.1: Strategy Matchup Results (7x7)

P1 \P2	W1	W2	W3	W4	W5	W6	W7
W1	822/178/15.78	946/54/16.71	779/221/11.69	865/135/16.55	1000/0/14.56	703/297/12.39	998/2/15.20
W2	370/630/21.70	742/258/24.22	602/398/17.83	524/476/22.47	984/16/21.95	359/641/17.56	963/37/23.55
W3	474/526/11.91	720/280/16.37	437/563/8.41	476/524/13.24	991/9/16.34	337/663/9.11	991/9/16.99
W4	609/391/18.43	873/127/20.37	746/254/14.08	719/281/18.72	998/2/18.03	566/434/14.83	995/5/19.17
W5	56/944/20.66	384/616/26.25	138/862/18.93	190/810/22.78	872/128/26.19	35/965/17.38	796/204/27.86
W6	610/390/11.95	847/153/15.43	569/431/8.76	642/358/13.51	998/2/14.48	450/550/9.35	999/1/15.12
W7	60/940/20.65	417/583/26.16	131/869/19.48	177/823/22.70	878/122/26.20	47/953/17.52	801/199/27.87

Each cell reports the number of wins for Player 1, wins for Player 2, and the average number of turns until game end, respectively.

7.7 Analysss

7.7.1 Overall Best Performers

Across all matchups, some strategies emerge as consistently dominant—especially when played first. For example, **Weight 1** (Keep Alive) achieves overwhelming success against nearly every other configuration when used by Player 1, including a perfect 1000 to 0 score versus **Weight 5**. Similarly, **Weight 4** (Value Trading) and **Weight 7** (Author Favorite) perform exceptionally well in first-player matchups, often yielding win rates above 95%.

Interestingly, **Weight 5** (Board Clear)—though designed to remove enemy minions—only shows isolated strength. It dominates late-game decks like Weight 7 when going first, but collapses entirely against aggressive opponents when going second. These results highlight that some strategies are heavily dependent on tempo and initiative.

7.7.2 Same Weights vs. Same Weights

Mirror matchups reveal much about internal balance. Here are the diagonal results where both players use the same weight configuration:

- **Weight 1 vs. Weight 1**: P1 wins 822 of 1000 games (82.2%).
- **Weight 2 vs. Weight 2**: 74.2% win rate for P1.

- **Weight 3 vs. Weight 3**: Close to parity, with P1 at 43.7%.
- **Weight 5 vs. Weight 5**: Strong P1 advantage at 87.2%.
- **Weight 6 vs. Weight 6**: Almost balanced, P1 wins 45.0%.
- **Weight 7 vs. Weight 7**: Heavy tilt, P1 wins 80.1%.

The implication is clear: even when the strategies are symmetrical, going first yields a significant and sometimes decisive advantage. This distortion likely comes from the lack of "The Coin," which, in real Hearthstone, mitigates first-move advantage through extra mana flexibility.

7.7.3 Matchup Patterns and Strategy Classes

By viewing the table as a 7x7 dominance matrix, we observe that strategies naturally fall into *aggressive*, *value/control*, and *hybrid* classes:

- **Aggressive strategies** (e.g., **Weight 3**, **Weight 1**) tend to perform best against slower or passive decks, especially when starting first. They create early pressure that defensive decks cannot recover from in time.
- **Value-based strategies** (e.g., **Weight 4**, **Weight 5**) perform better in longer games or against decks that make inefficient trades. However, they can be dismantled before stabilizing if the opponent seizes tempo early.
- **Hybrid strategies** (like **Weight 7**) show versatility, beating both extremes under favorable conditions. While not unbeatable, they maintain decent outcomes across many pairings, especially when piloted by P1.

This suggests that no strategy is universally optimal, but some are more resilient across different board states and tempo curves.

7.7.4 Turn Length and Tempo Tempo Tempo

Average turn counts also reflect the style of play:

- Shortest games often involve **Weight 3** and **Weight 6**, both prioritizing damage and quick resolution.

- Longest games (25–28 turns) are seen in mirror matches involving **Weight 5** or **Weight 7**, where control loops extend the match.

This further reinforces that aggressive decks may win fast or lose fast, while control decks drag the game but risk never recovering against tempo loss.

7.7.5 First-Player Advantage and No "Coin"

One of the clearest findings is the overwhelming impact of going first. For many matchups, a reversal of turn order leads to a reversal in outcome. Consider:

- (Weight 1, Weight 5) = 1000 to 0, but if reversed: (Weight 5, Weight 1) = 56 to 944.
- (Weight 3, Weight 5) = 991 to 9; reversed: (Weight 5, Weight 3) = 138 to 862.

Without a second-player compensatory mechanic (e.g., The Coin, card advantage), the system strongly favors whoever plays first. Future iterations must address this to achieve fairer evaluations.

7.7.6 Influence of Deck Composition

Another limitation is the lack of spell cards and synergy mechanics. Because decks include only minions with *Charge, Rush*, or *Taunt*, certain strategic archetypes are underrepresented. In standard Hearthstone:

- Spells enable surprise lethal or reactive clears.
- Draw mechanics, discover effects, and combos change optimal play sequences.

Here, strategy expressiveness is limited, which may artificially elevate simpler strategies (like aggro face damage) and weaken midrange or combo concepts.

7.7.7 Which Strategy is "Best"?

There is no single best weight configuration across all conditions. However, from a **first-player perspective**, the top performers are:

- **Weight 1** ("Keep Alive"): Exceptional survivability and early board presence.
- **Weight 4** ("Value Trading"): Strong removal and efficiency against greedy opponents.

From a second-player perspective, these strategies fare much worse unless tailored to handle early threats. Thus, the "best" configuration is **context-sensitive**—it depends not only on your weights, but on your position and opponent's logic.

7.7.8 Next Steps and Fixes

This experiment highlights both the strengths and flaws of the current solver and simulation environment:

- **First-player bias is too strong**, requiring the addition of "The Coin" or turn-balancing logic.
- **Deck variety is too narrow**, warranting spell integration and synergy tracking.
- **Strategy comparison is meaningful**, but limited by simplified game logic and deterministic cards.

In **Chapter 8**, we will address these limitations with proposed improvements to fairness, deckbuilding realism, and long-term strategic depth. The goal is not only to test solver logic but to get closer to the complexity of real digital card games.

Chapter 8

Discussion

8.1 Limitations

Despite the progress made in modeling Hearthstone gameplay through integer programming, several simplifications and constraints limit the full expressive power of the system:

- **Limited Card Variety:** The model currently supports only a curated subset of Hearthstone's card pool, primarily basic minions with limited keywords. This excludes entire categories of gameplay, such as burn spells, disruption tools, or synergy-based win conditions (e.g., Murloc tribal, Mech synergies). As a result, the strategies tested reflect only a narrow slice of what is possible in actual play.
- **Simplified Rule Encoding:** While the solver incorporates core constraints like board limits, mana usage, and lethal damage checks, it omits many subtle game mechanics. Examples include Deathrattle chains, card draw timing, and conditional triggers. These exclusions prevent the model from simulating decks that rely on nuanced combos, sequencing, or non-linear payoff effects.
- **Static Turn Evaluation:** Each solver call models a single turn as a self-contained optimization problem. It does not simulate forward planning across multiple future turns or include uncertainty about opponent responses. Likewise, the absence of real-time triggers (e.g., Secrets or Aura effects) limits strategic foresight. The solver cannot account for mid-turn surprises that alter the board state dynamically.

These constraints are not flaws in the approach, but they highlight the tradeoff between modeling precision and tractable computation. Extending the model to include more cards and mechanics will increase realism, but also computational complexity.

8.2 Challenges

Using integer linear programming (ILP) to model turn-by-turn gameplay introduces both power and difficulty. The model is expressive and guarantees optimality—yet, several key challenges remain:

- **High Dimensionality:** Modeling Hearthstone combat involves many binary decisions: who attacks whom, which cards are played, which minions survive, and whether lethal occurs. These variables grow rapidly with larger board states or hand sizes, making the MILP problem size nontrivial even in mid-game situations.
- **Special Keyword Handling:** Even seemingly simple keywords like *Windfury* introduce nontrivial logic. While the constraint $\sum_j x(i,j) + x(i,\text{hero}) \leq 2$ captures two attacks per turn, modeling how partial damage, surviving targets, and attack sequencing interact with lethal planning is more subtle. Every keyword added to the system compounds the constraint graph.
- **Run Time and Scalability:** For small or medium-sized game states, modern solvers like Gurobi perform well. However, expanding the state space to accommodate full-featured decks with Secrets, draw engines, spell bursts, and cross-turn synergies leads to exponential growth. Without approximation or pruning, real-time performance becomes difficult to maintain.

These challenges highlight a tension between full strategic depth and computational feasibility. ILP excels at handling constrained action selection, but may need to be hybridized with heuristics or domain-specific simplifications to scale.

8.3 Outlook

Despite its limitations, this framework lays the foundation for a flexible and interpretable AI system. Several directions are worth exploring to expand both scope and applicability:

- **Hero Power Integration:** Incorporating class-specific hero powers introduces meaningful tradeoffs. For example, Paladin's token generation and Warrior's armor-up mechanics offer reliable but modest value each turn. These can be encoded as binary decisions, with corresponding mana costs and reward terms in the objective function. Hero powers would increase turn diversity and reflect the tempo/resource tradeoffs seen in real games.

- **Alignment with Meta Decks:** By tuning weights to match well-known archetypes—such as Aggro Paladin, Pirate Warrior, or Control Priest—the solver can simulate real-world decks more faithfully. Our weight configurations (e.g., "Aggro Face," "Tempo," "Value Trading") offer primitive versions of this. Scaling up the card pool and refining strategic weight tuning may allow for deck-vs-deck benchmarking or AI mirroring of actual ladder performance.

- **Extending the Keyword Set:** Currently modeled keywords (Charge, Rush, Taunt, Windfury) offer an entry point. However, Hearthstone includes many others that dramatically affect tempo, card advantage, and threat sequencing—such as Divine Shield, Lifesteal, Overload, or Spellburst. Incorporating these would require a richer model of inter-turn memory and status tracking, but would enable simulation of combo and midrange decks with much more strategic complexity.

- **Toward Real-Time Playability:** One long-term goal is to integrate the solver with a live simulation environment, enabling decisions during actual gameplay. This would require mid-turn re-optimization when a Secret triggers, or when a board state changes unexpectedly. Techniques such as constraint caching, warm starts, and partial re-solving could allow the AI to adapt while maintaining most of the MILP's structure and optimality.

- **Practical Utility for Game Developers:** With refinement, this system could be used by designers or balance teams at Blizzard or elsewhere to evaluate decks, explore counter-strategy space, or test card impact. Given a target deck, the solver could simulate how it fares across a range of tuned opponents—offering insights into matchup strength, win conditions, and deck weaknesses without thousands of human playtests.

In short, the ILP-based framework presented here shows promise as a general method for strategic decision making in collectible card games. With extensions to timing, memory, and card pool size, it could become a valuable AI tool both for automated play and analytical game design.

Chapter 9

Final Remarks

This thesis explored the use of integer linear programming (ILP) as a decision-making framework for turn-based strategy games, using a simplified model of Hearthstone as the primary testbed. By encoding core gameplay mechanics—attacks, survivals, mana constraints, and keyword effects—into a solvable optimization problem, we constructed a modular AI capable of evaluating and executing optimal plays on a per-turn basis.

Through the design of weighted objectives, the solver could emulate a wide range of strategic styles, from aggressive face-hitting to conservative board control. Simulations showed how different weight configurations performed across matchups, revealing clear patterns and a strong first-player advantage in the absence of compensatory mechanics. The results also highlighted the potential of solver-guided strategies to mimic archetypal decks from actual gameplay.

While powerful, the system as implemented comes with tradeoffs. Its simplifications—limiting card variety, excluding real-time reactivity, and treating each turn as an isolated problem—constrain its expressiveness relative to Hearthstone's full design space. Nonetheless, the solver's interpretability, extensibility, and tactical clarity make it a strong foundation for future work.

Several promising directions lie ahead. Enhancing the solver to handle real-time triggers, chain effects, and card-draw sequencing would deepen strategic realism. Integrating hero powers, a broader card pool, and second-player compensation (such as "The Coin") would more closely mirror competitive gameplay. Ultimately, connecting this solver-based AI to a live game interface could yield applications in automated playtesting, card balancing, or interactive training tools for players.

More broadly, this project demonstrates how classical optimization methods—often used in operations research or logistics—can be fruitfully applied to game design and

AI strategy. Rather than relying solely on machine learning or hand-coded rules, optimization frameworks offer a middle ground: adaptive, tunable, and grounded in explicit goals. With continued development, this approach could serve not only games like Hearthstone, but any environment where discrete, constrained decision-making is key.

In summary, this work provides a novel, practical method for simulating competitive strategy through ILP. It contributes a working prototype, a flexible architecture, and a roadmap for future research and refinement. As the boundary between AI and game design continues to blur, such tools may prove increasingly valuable—not only for modeling gameplay, but for shaping it.

Bibliography

[1] Pablo García-Sánchez et al. (2020). *Optimizing Hearthstone Agents Using an Evolutionary Algorithm*. Knowledge-Based Systems, 188, 105032. https://doi.org/10.1016/j.knosys.2019.105032

[2] Fernando de Mesentier Silva et al. (2019). *Evolving the Hearthstone Meta*. Proceedings of the IEEE Conference on Games 2019, 1-8. https://doi.org/10.48550/arXiv.1907.01623

[3] Alexander Dockhorn et al. (2018). *Predicting Opponent Moves for Improving Hearthstone AI*. Information Processing and Management of Uncertainty in Knowledge-Based Systems. Theory and Foundations (IPMU 2018), Communications in Computer and Information Science, vol 854, pp. 622–631. Springer, Cham. https://doi.org/10.1007/978-3-319-91473-2_51

[4] Łukasz Grad. (2017). *Helping AI to Play Hearthstone Using Neural Networks*. 2017 Federated Conference on Computer Science and Information Systems (FedCSIS), IEEE, pp. 121–124. https://doi.org/10.15439/2017F561

[5] Jan Jakubik. (2018). *A Neural Network Approach to Hearthstone Win Rate Prediction*. Annals of Computer Science and Information Systems, 16, 185–188. https://doi.org/10.15439/2018F365

[6] Alysson Ribeiro da Silva and Luis Fabricio Wanderley Goes. (2018). *HearthBot: An Autonomous Agent Based on Fuzzy ART Adaptive Neural Networks for the Digital Collectible Card Game HearthStone*. IEEE Transactions on Games, 10(2), 170–181. https://doi.org/10.1109/TCIAIG.2017.2743347

[7] Henry William Gorelick. (2020). *Predicting and Enhancing Hearthstone Strategy with Combinatorial Fusion*. Fordham University ProQuest Dissertations & Theses, 27833656.

[8] Jakub Kowalski and Radosław Miernik. (2020). *Evolutionary Approach to Collectible Arena Deckbuilding Using Active Card Game Genes*. 2020 IEEE Congress on Evolutionary Computation (CEC), pp. 1–8. https://doi.org/10.1109/CEC48606.2020.9185755