Inteligência Artificial em Magic: the Gathering

Guilherme Souto Schützer
guischutzer@gmail.com

Tomás Marcondes Bezerra Paim
tomasbmp@gmail.com

IME-USP

Resumo

Este trabalho é um estudo do problema de Inteligência Artificial aplicado ao jogo de cartas americano **Magic: the Gathering**. O objetivo é criar uma inteligência artificial que tenha chances de ganhar em uma versão simplificada do jogo. O projeto inclui a implementação a partir do zero de um cliente próprio em modo texto de **Magic**, a modelagem do pré-problema do *mulligan* (parte inicial do jogo), resolvido pelo algoritmo de Iteração de Valor e a modelagem do problema de um jogo de **Magic**. A estratégia utilizada pela IA no jogo é uma busca local da melhor sequência de ações possível visando a recompensa imediata (ou seja, no mesmo turno – os jogadores alternam entre turnos), utilizando busca em grafo e o algoritmo Minimax em jogadas alternadas. Como resultado, temos um agente que consegue ganhar um número considerável de partidas (apesar de ser ruim contra um jogador humano com experiência) e, inclusive, tem um estilo de jogo caracterizável como "agressivo" (em grande parte por considerar somente ganhos a curto prazo).

Abstract

This work is a study of the Artificial Intelligence problem applied to the **Magic: the Gathering** trading card game. The goal is to create an artificial intelligence able to play and sometimes win in a simplified version of the game. The project includes full implementation of our own text-based **Magic** client, modelling of the mulligan pre-problem (which precedes every game), solved by the Iteration Value algorithm and modelling of a proper **Magic** game. The strategy used by the in-game AI agent is searching locally for the best possible sequence of actions in terms of immediate reward (as player take alternating turns, the immediate reward is measured in the end of a player's turn), using search in graphs and the Minimax algorithm for alternate players. As a result, we have an agent capable of winning a considerable amount of matches (despite bad performances against experienced human players) with an "aggressive"-caracterized playing style (largely due to its immediate-gains "philosophy").

Conteúdo

Capítulo 1

Introdução

Inteligência artificial é um campo da ciência da computação que estuda "agentes inteligentes", que de certa forma percebem o ambiente à sua volta e tomam ações tendo como objetivo maximizar a chance de sucesso de alguma tarefa específica. Uma utilização muito comum nos dias de hoje é a de inteligência artificial com o objetivo de criar um agente capaz de competir com humanos em jogos com alto nível de estratégia, como Xadrez e Go. Decidimos então estudar inteligência artificial para a realização deste trabalho com a ideia de modelar um outro jogo de estratégia, **Magic: The Gathering**.

Magic foi lançado em 1993, introduzindo o conceito de trading card game. Com o sucesso do jogo e popularização dos jogos digitais, eventualmente nasceram versões digitais do jogo para computadores e videogames, e com isso nasceu a necessidade de agentes que jogassem contra os jogadores. Atualmente há várias versões digitais de **Magic**, mas nenhuma tem uma inteligência artificial boa o suficiente para se provar desafiadora contra jogadores experientes, uma vez que para cada ação há uma grande quantidade de possibilidades e elementos como blefe envolvidos. Nossa intenção é entender a complexidade da representação do jogo e criar uma plataforma para jogar **Magic** que possibilite a implementação de um agente de inteligência artificial.

Na próxima seção iremos introduzir alguns conceitos e regras básicas do jogo, de modo a possibilitar a familiarização do leitor com **Magic**, facilitando a compreensão do restante do trabalho.

1.1 Conceitos básicos

Um jogo usual de **Magic: the Gathering** conta com dois jogadores munidos de um baralho de 60 cartas cada, ambos começando com 20 pontos de vida, sendo que o objetivo é reduzir o total de pontos de vida do oponente a 0. Para tanto, é preciso usar as cartas disponíveis na mão, que podem representar feitiços, criaturas ou terrenos (existem outros tipos de cartas, mas para nossa implementação iremos focar nesses três).

1.1.1 Cartas

Nesta seção explicamos a estrutura de uma carta e as diferenças entre os três tipos de carta que utilizamos no projeto. Antes de tudo, convém definir alguns termos utilizados no jogo:

- **Jogar** uma carta é um termo genérico para "retirar uma carta da sua mão e aplicar seus efeitos no jogo". Dependendo do tipo de uma carta, quando esta é "jogada", será colocada no *campo de batalha* (termo próprio para a mesa de jogo) ou no *cemitério* (termo próprio para a pilha de descarte) do jogador. Uma carta na mesa é chamada *permanente*.

- **Virar** (em inglês, *tap*) uma permanente significa fisicamente dispô-la horizontalmente (normalmente uma carta está na posição vertical). Esta ação representa uma espécie de ação das permenentes.

Quando o turno começa, o jogador ativo *desvira* suas permanentes (ou seja, as dispõe em posição vertical), portanto, usualmente, virar uma permanente é um movimento que acontecerá em até uma vez (por permanente) durante cada turno.

A seguir, examinaremos a estrutura de uma carta.

Figura 1.1: Angel of Mercy e seus principais atributos.

1. **Nome** da carta.
2. **Custo de mana**. "Mana" é o recurso necessário em **Magic** para jogar cartas que não sejam terrenos. Os símbolos da caixa representam a quantidade e tipo de mana necessários para jogar Angel of Mercy. 4, então, significa que, para jogar a carta é preciso 4 "manas" de qualquer tipo e uma "mana" branca, totalizando 5. Mana é geralmente de um dos 5 tipos – chamados *cores* – possíveis: Branco (), Azul, (), Preto (), Vermelho () e Verde (). O custo ainda determina a *cor* da própria carta: Angel of Mercy requer apenas mana branca, portanto é uma carta branca.

3. **Tipos** de cartas são escritos nesta caixa. Como já mencionado, o tipo de Angel of Mercy é Criatura. Além disso, cartas – quase sempre criaturas – também podem ter *subtipos* indicando características da criatura, no caso, um Anjo. Isto é relevante para alguns efeitos do jogo, mas nenhum deles é abordado no projeto.

4. **Habilidades** definem os efeitos próprios das cartas. Neste trabalho, usamos apenas criaturas com *habilidades de combate*, características de cada criatura que definem seu comportamento na etapa de combate, e habilidades do tipo "ao entrar em jogo", desencadeadas quando a carta se torna uma permanente.
   No caso, Angel of Mercy tem instâncias dos dois tipos: *Flying* (em portugês, Voar), que limita as interações do oponente durante a etapa de Combate, e sua segunda habilidade, que concede a seu controlador um bônus de 3 pontos de vida. Existem, também, criaturas sem habilidades (porém não existem feitiços sem habilidades). Algumas cartas contêm explicações, em parênteses, de efeitos comuns (como Voar).

5. ***Flavor text*** é um texto que ambienta a carta dentro de alguma temática. Pode ser uma fala, um comentário, ou até uma pequena história. *Flavor texts* são escritos em itálico e não influenciam em nada as regras do jogo.

6. Esta caixa contém os atributos mais importantes das criaturas (e presentes apenas neste tipo de carta): **poder e resistência**. X/Y representa uma criatura com poder X e resistência Y, ou seja, na etapa de combate, esta criatura causa X de dano total e, a qualquer momento, se a criatura tiver uma quantidade d de dano marcado tal que $d \geqslant Y$, a criatura é destruída.

Os **terrenos** citados são responsáveis por gerar mana para jogar cartas com custo. Para jogar um terreno, o jogador ativo simplesmente o retira da sua mão e coloca em campo (podendo fazer isso no máximo uma vez por turno). Neste trabalho usamos apenas terrenos com o supertipo *Básico*.[1] A cor da mana gerada pelo terreno está ligada a seu subtipo e seu símbolo associado também aparece na carta. Todos os terrenos básicos têm a habilidade "{T}: adicione C à sua reserva de mana.", onde $C \in \{$,  ,  ,  ,  $\}$, não escrita na carta.

Assim, a rotina para jogar uma carta com determinado custo é virar o número de terrenos necessários de cada tipo, remover a carta da mão e aplicar seus efeitos (caso seja uma Criatura, é colocada em campo e se torna uma Permanente, caso seja um Feitiço, seus efeitos são aplicados e é colocada no cemitério). No caso de Angel of Mercy, o jogador deve virar 4 terrenos quaisquer e uma Planície para jogá-la. Assim, que o anjo entrar em jogo, seu efeito é desencadeado Apesar de ser possível (e usual) usar mais de uma cor em um baralho, para este trabalho trabalharemos com baralhos de só uma cor.

A figura 1.3 contém, enfim, um exemplo de Feitiço. A carta possui todas as características de uma carta de Criatura, exceto pelo poder e resistência. Para jogar Volcanic Hammer, além de virar o número de terrenos necesário (ou seja, uma Montanha e um terreno qualquer para pagar o custo de ), o jogador ativo deve eleger um **alvo** para o efeito da carta. O próprio efeito descreve os alvos legais – no caso, o alvo escolhido pode ser qualquer criatura em campo *ou* qualquer jogador. Alguns efeitos tem seus alvos restritos, por exemplo, a criaturas do oponente ou somente a jogadores.

No começo do jogo é decidido aleatoriamente quem será o jogador inicial, e então os dois jogadores compram uma mão inicial de sete cartas. Antes do jogo propriamente dito começar, os jogadores podem optar por tomar uma ação chamada mulligan, que consiste em rejeitar a mão inicial, embaralhá-la de volta com o restante dos cards e comprar uma nova mão inicial, com uma carta a menos. Pode-se então repetir o processo até que cada jogador esteja satisfeito com a mão inicial ou até o jogador realizar um mulligan com apenas uma carta na mão (resultando em uma mão de zero cartas, onde não há mais a possibilidade de realizar mulligan). Uma vez que os dois jogadores tiverem escolhido manter uma mão inicial, cada jogador que realizou pelo menos um mulligan olha a carta do topo de seu *deck* (como é chamado o baralho) e decide se quer colocá-la no fundo.

[1] Geralmente, em **Magic**, um baralho tem a limitação de 4 cópias de uma mesma carta. Cartas com o supertipo *Básico* não são afetadas por esta limitação.

Figura 1.2: Os cinco terrenos básicos. Em sentido horário: Planície, Ilha, Pântano, Montanha, Floresta.

O jogo então começa, com os jogadores alternando entre turnos, onde o jogador que "controla o turno" é chamado de *jogador ativo*, com a seguinte estrutura, simplificada:

- **Início do turno**: Permanentes do jogador ativo são desviradas. Jogador ativo compra uma carta de seu deck.
- **Primeira Fase Principal**: Jogador ativo pode jogar as cartas da mão. Termina quando o jogador ativo decide *passar*, o que implica no término da fase atual.
- **Combate**: O combate é a maneira principal de um jogador ganhar o jogo, pois envolve a tentativa de diminuição dos pontos de vida de seu oponente através de ataques de suas criaturas. Será melhor explicado na próxima seção.
- **Segunda Fase Principal**: Igual à primeira Fase Principal.
- **Fase final**: Todas as criaturas tem seu dano marcado revertido para 0. Caso o jogador tenha mais de sete cartas na mão, deve descartar (colocar no *cemitério*) até ter uma mão de sete cartas.

A estrutura acima se repete até o jogo terminar, o que acontece geralmente quando algum jogador chega a 0 pontos de vida, mas também pode acontecer de outras maneiras como, por exemplo, se o baralho de um jogador acabar.

1.1.2 Exemplo de Fases Principais

Suponhamos que o jogador ativo tem 5 Planícies desviradas e 20 pontos de vida. Na sua primeira Fase Principal, joga seu Angel of Mercy (virando as 5 planícies para adicionar ✹✹✹✹✹ à sua reserva de mana),

Figura 1.3: O Feitiço "Volcanic Hammer" (Martelo Vulcânico)

que tem seu efeito desencadeado levando-o a 23 pontos de vida. Decide, então, passar o turno para a fase de Combate. Como não tem nenhuma criatura apta a atacar (melhor explicado na próxima subseção), a fase de Combate termina e começa a segunda Fase Principal. O jogador ativo decide passar a segunda fase principal. No próximo turno, o jogador ativo é trocado, compra uma carta e inicia sua primeira Fase Principal. O (novo) jogador ativo tem duas montanhas e decide jogar Volcanic Hammer alvejando Angel of Mercy (e para fazê-lo, vira as duas montanhas, gerando). Como o anjo tem 3 de resistência e o Feitiço lhe causa 3 dano, a permanente é destruída.

1.1.3 Combate

A fase de Combate é a principal maneira do jogador ativo diminuir o total de vida de seu oponente. Isto é realizado ao **atacar** com suas criaturas. O jogador defensor, como é chamado o oponente do jogador ativo, tem a chance de **bloquear** os ataques com as suas próprias criaturas. A maioria dos jogos de **Magic** termina durante esta fase.

1. Jogador ativo *declara atacantes* – escolhe quais de suas criaturas irão atacar seu oponente. Uma criatura só pode atacar se 1) estiver **desvirada** (representada na posição vertical) e 2) não estiver *enjoada* (termo que significa que a criatura entrou em jogo neste turno – uma criatura só pode atacar a partir do turno seguinte ao turno que entrou no campo de batalha). Uma vez que o jogador decidiu com quais criaturas atacar, deve **virar** as criaturas declaradas para indicar que estas estão atacando. Se o jogador ativo não tiver criaturas aptas a atacar (ou decidir não atacar com nenhuma), a fase de Combate termina.

2. O jogador defensor *declara bloqueadores* – escolhe quais de suas criaturas irão bloquear as criaturas atacantes. Mais uma vez, apenas criaturas **desviradas** podem bloquear. Diferentemente das criaturas atacantes, porém, criaturas bloqueadoras **não** se tornam viradas por executar essa ação. Algumas

habilidades restringem como criaturas podem bloquear: uma criatura atacante com Voar só pode ser bloqueada por criaturas com a habilidade Voar ou Alcance (*Reach*).

3. Caso alguma criatura atacante esteja bloqueada por mais de uma criatura, o jogador ativo decide, então, a ordem com que o dano será atribuído a cada criatura bloqueadora.

4. Por fim, há a atribuição de dano. Cada criatura atacante causa dano igual a seu poder às criaturas que a bloqueiam e, caso, não esteja bloqueada, causa dano igual ao seu poder ao jogador defensor. Cada criatura bloqueadora causa dano igual ao seu poder à criatura que bloqueou.

1.1.4 Exemplo de Combate

Suponhamos que o jogador ativo tenha três criaturas desviradas e não-enjoadas no início da fase de combate e seu oponente tenha duas criaturas desviradas (portanto, aptas a bloquear) e 20 pontos de vida.

Figura 1.4: Criaturas do jogador ativo: dois "Grizzly Bears" (Ursos Cinzentos) em uma "Centaur Courser" (Centaura-Caçadora).

Figura 1.5: Criaturas do jogador defensor: dois "Walking Corpse" (Cadáver Ambulante).

Figura 1.6: Supondo que o jogador ativo decida atacar com Centaur Courser e um Grizzly Bears, deve então virá-los.

Figura 1.7: Por fim, o jogador defensor deve declarar quais de suas criaturas bloquearão a ofensiva. Suponhamos, então, que a escolha seja apenas bloquear Grizzly Bears com um Walking Corpse.

Assim, no final do combate, Grizzly Bears causa 2 de dano a Walking Corpse, que por sua vez também lhe causa 2 de dano. Ambas as criaturas são destruídas. Centaur Courser causa 3 de dano ao jogador defensor. O jogador defensor termina 17 de vida e um Walking Corpse desvirado, enquanto o jogador ativo tem 20 de vida e um Centaur Courser virado.

Capítulo 2

Implementação do Cliente

Para que fosse possível que implementássemos um agente de inteligência artificial que jogasse uma versão simplificada de **Magic**, era essencial que conhecêssemos em detalhe a plataforma onde o jogo estaria rodando, e para isso implementamos do zero uma versão do jogo com as especifidades desejadas usando a linguagem Python.

O programa é composto de quatro classes principais: `Game`, `Player`, `Card` e `Permanent`, representando o Jogo, Jogadores, Cartas e Permanentes (como são tratadas as cartas de Terreno e Criatura uma vez que estão em jogo) respectivamente. A seguir vamos falar de cada uma destas classes entrando em detalhes nas principais características.

2.1 `class Card:`

A classe `Card` representa as cartas do jogo. Os objetos que estarão presentes nas mãos, decks e cemitérios dos jogadores serão classes que herdam desta classe. Seus atributos representam características presentes em todas as cartas. Para exemplificar os atributos, usaremos as cartas mostradas no capítulo anterior: Volcanic Hammer e Angel of Mercy. Os atributos a seguir são comuns a todos os tipos de carta, e por isso estão presentes na classe Card:

- `name`: Uma string representando o nome da carta, por exemplo "Volcanic Hammer" ou "Angel of Mercy".
- `cost`: Uma string representando o custo de mana da carta, usando a mesma notação presente nas cartas, com W, U, B, R e G representando os símbolos de mana (Branco), (Azul), (Preto), (Vermelho) e (Verde) respectivamente. No caso de Volcanic Hammer, o custo é "1R" e no de Angel of Mercy, "4W".
- `supertype`: Uma string representando o supertipo da carta. No nosso programa, o único supertipo que irá aparecer é "Basic", que identifica os terrenos básicos, mas essa informação não é relevante para o programa. Estamos representando essa informação principalmente por formalidade para que o programa siga a mesma estrutura de tipos descrita nas regras do jogo. Nem Volcanic Hammer nem Angel of Mercy possuem supertipos, portanto este atributo é representado pela string vazia.
- `ctype`: Uma string que representa o tipo da carta. Angel of Mercy recebe o tipo `'Creature'` e Volcanic Hammer,`'Sorcery'`. Diferentemente de supertipo ou subtipo, este é um atributo que nunca estará vazio em uma carta.
- `subtype`: Uma string que representa o subtipo da carta. O subtipo serve para que algumas cartas tenham uma informação adicional que possa ser usada para ações dentro do jogo. Volcanic Hammer não tem subtipo, enquanto o subtipo de Angel of Mercy recebe `'Angel'`.

```
class VolcanicHammer(Card):
    def __init__(self, owner):
        self.name = "Volcanic Hammer"
        self.cost = "1R"
        self.supertype = ""
        self.ctype = "Sorcery"
        self.subtype = ""
        self.text = "Volcanic Hammer deals 3 damage to target creature or player."
        self.targets = [["OwnCreature", "OpponentCreature", "Player"]]
        self.owner = owner

    def effect(self, game, targets):
        targets[0].takeDamage(3)


class AngelofMercy(Card):
    def __init__(self,owner):
        self.name = "Angel of Mercy"
        self.cost = "4W"
        self.supertype = ""
        self.ctype = "Creature"
        self.subtype = "Angel"
        self.text = "Flying. When Angel of Mercy enters the battlefield, you gain 3 life. 3/3"
        self.abilities = ["Flying"]
        self.targets = []
        self.owner = owner
        self.power = 3
        self.tou = 3

    def effect(self, game, targets):
        self.owner.gainLife(3)
```

- `text`: Uma string representando o texto da carta. Essa string serve somente para interface com o jogador. O texto de Volcanic Hammer é "Volcanic Hammer deals 3 damage to target creature or player." enquanto o de Angel of Mercy é "Flying. When Angel of Mercy enters the battlefield, you gain 3 life. 3/3".
- `targets`: Uma lista contendo tuplas com as possibilidades de alvo que a carta pode ter. Angel of Mercy não tem alvos, e portanto sua lista de alvos é vazia. Volcanic Hammer pode dar alvo em criaturas ou jogadores, então sua lista de alvos é `[(''OwnCreature'', ''OpponentCreature'', ''Player'')]`. Separamos criaturas entre criaturas do mesmo dono da carta ou criaturas dos oponentes pois há cartas que só podem ter um destes conjuntos como alvo, facilitando a implementação.
- `owner`: O jogador dono da carta. Este atributo é do tipo `Player`.

Para cada carta com nome diferente, há uma classe que herda da classe `Card` e implementa as especificidades da carta. Vejamos o código da classe `VolcanicHammer`, que implementa a carta Volcanic Hammer: Podemos ver que a classe tem o método `effect()`, que implementa o efeito da carta, fazendo com que o alvo escolhido sofra três pontos de dano. Vejamos agora a implementação da carta Angel of Mercy: Além dos atributos que citamos anteriormente, Angel of Mercy tem três atributos a mais por ser do tipo Criatura: `abilities`, uma lista com as habilidades de combate da criatura; `power`, o poder da criatura; e `tou`, a resistência (toughness) da criatura. Podemos também ver o efeito da carta, que concede três pontos de vida a seu controlador.

O método `effect` de cada carta é chamado quando esta é jogada, tendo consequências distintas para cada caso.

2.2 `class Permanent:`

A classe `Permanent` serve para representar as cartas uma vez que foram jogadas e estão no campo de batalha. Seus atributos são os seguintes:

- `card`: A carta que criou a permanente. Ela contém informações relevantes para a construção do objeto e é ela, e não a permanente, que será colocada no cemitério caso deixe o campo de batalha.
- `abilities`: Uma cópia do atributo `abilities` da carta. É necessário que esse atributo seja armazenado separadamente ao da carta, pois uma vez em jogo, a permanente pode ganhar ou perder habilidades.
- `ctype`: O tipo da permanente, conforme descrito na carta que a criou.
- `owner`: O jogador dono da permanente, que será o mesmo que possúi a carta.
- `controller`: O jogador controlador da permanente. Normalmente ele será o dono, mas há efeitos que mudam o jogador que controla uma permanente.
- `tapped`: Um atributo booleano que indica se a permanente está virada ou desvirada.
- `sick`: Um atributo booleano que indica se a permanente está *enjoada*[1] ou não.
- `destroyed`: Um atributo booleano que indica se a permanente foi destruída ou não. Permanentes destruídas são colocadas no cemitério de seu controlador durante as checagens do jogo.
- `ID`: Um inteiro que serve como identificador único entre permanentes. Ele serve, por exemplo, para diferenciar permanentes que tenham sido criadas a partir de cartas iguais.

Estes atributos são alterados a partir dos métodos da classe, que não entraremos em detalhes.

Há duas classes que herdam da classe `Permanent`: `Land` e `Creature`. `Land` é a classe que representa terrenos. A única diferença desta classe para a classe mãe é que os métodos de virar e desvirar manipulam um atributo do jogador controlador: a quantidade de terrenos desvirados controlados por aquele jogador. A classe `Creature` tem alguns atributos a mais:

- `power`: Um inteiro que representa o poder da criatura.
- `tou`: Um inteiro que representa a resistência da criatura.
- `curPower`: Um inteiro que representa o poder atual da criatura. Existem efeitos que modificam o poder com duração limitada a um turno. Nestes casos, o poder volta a ser o poder armazenado no atributo `power` quando o próximo turno começar.
- `curTou`: Um inteiro que representa a resistência atual da criatura. Análogo ao atributo `curPower` relativo à resistência da criatura.
- `attacking`: Um atributo booleano que representa se a criatura está atacando ou não.
- `blocking`: Um atributo booleano que representa se a criatura está bloqueando ou não.
- `damage`: Um inteiro que armazena a quantidade de dano sofrida pela criatura até o presente momento. No início de cada turno, este atributo volta a ter o valor 0.
- `currentAbilities`: Análogo ao atributo `curPower` para as habilidades.

Criaturas têm alguns métodos especiais que possibilitam com que elas realizem ações dentro do jogo, como atacar ou causar dano, mas não entraremos em detalhes.

[1] Permanentes entram em jogo com *enjoo de invocação*, o que indica que caso ela seja uma criatura, não poderá atacar. O enjoo de uma permanente acaba no começo do turno de seu controlador.

2.3 `class Game:`

A classe `Game` representa um jogo em andamento. Devido à complexidade da classe, entrar em detalhes iria mudar o foco do trabalho, então não iremos descrever todos os métodos e atributos da classe. Há um método chamado `turnRoutine()` que é chamado repetidamente até que o jogo termine. Ele executa as ações iniciais e finais do turno (como fazer com que o jogador ativo compre uma carta no começo do turno e limpar o dano das criaturas no final do turno) e chama os métodos de fase principal e de combate, que permitem que os jogadores joguem as cartas da mão, ataquem e bloqueiem.

Além disso, há alguns métodos para definir quais são as ações legais para um jogador e para realizar as chamades *ações baseadas no estado*, que são ações performadas pelo jogo sempre que uma certa condição baseada no estado do jogo se torna verdade (por exemplo, destruir criaturas com dano igual ou superior à resistência, fazer com que jogadores com 0 ou menos pontos de vida percam o o jogo, e assim por diante). O método que cuida disso é `checkSBA()` (de "state based actions"), chamado pela classe `Game` nos seguintes momentos:

- No início e fim de cada fase.
- Após toda ação de Fase Principal.
- Durante etapas de atribuição de dano no Combate.

`Game` é responsável por chamar métodos que requerem a decisão do jogador.

2.4 `class Player:`

A classe `Player` representa um jogador na partida. Seus atributos são:

- `name`: Uma string representando o nome do jogador. Não tem nenhuma função dentro do jogo. Serve para identificar os jogadores na saída.
- `life`: Um inteiro representando o valor atual do total de pontos de vida do jogador.
- `hand`: Uma lista de objetos do tipo `Card`, representando as cartas na mão do jogador.
- `library`: Uma lista de objetos do tipo `Card`, representando as cartas no deck do jogador.
- `lose`: Um atributo booleano para identificar se o jogador perdeu o jogo. Este atributo é utilizado pela classe `Game` para decidir se o jogo acabou ou não.
- `ID`: Um inteiro representando um identificador único para o jogador, de maneira similar à que acontece com as permanentes.
- `creatures`: Uma lista de objetos da classe `Creature` com todas as criaturas controladas pelo jogador.
- `lands`: Uma lista de objetos da classe `Land` com todos os terrenos controlados pelo jogador.
- `active`: Um atributo booleano que indica se o jogador é o jogador ativo.
- `graveyard`: Uma lista de objetos da classe `Card` com todas as cartas no cemitério do jogador.
- `untappedLands`: Um inteiro representando a quantidade de terrenos desvirados que o jogador controla.
- `landDrop`: Um atributo booleano indicando se o jogador já jogou um terreno no turno atual.

```
------------------------------------------------------------
Turn 24 (Red)
Active Player: Red - 13 life
Mountain U
Mountain U
Mountain U
Mountain U
Mountain U
Mountain U
[22] Skyraker Giant U 3/4 (0)
[26] Lightning Hounds U 3/2 (0)
[28] Frenzied Raptor U 4/2 (0)

Opponent: White - 11 life
Plains U
Plains U
Plains U
Plains U
Plains U
Plains U
Plains U
Plains U
Plains U
[16] Fencing Ace U 1/1 (0)
[21] Griffin Sentinel U 1/3 (0)
[24] Heavy Infantry U 3/4 (0)
[27] Guardian of Pilgrims U 2/2 (0)
```

Figura 2.1: Representação em texto do início de um turno. A impressão das criaturas é do formato `[ID] Nome U/T poder/resistência (dano marcado)`, onde `U` significa desvirado (*untapped*) e `T` significa virado (*tapped*).

```
1) Angel of Mercy
2) Inspiring Roar
3) Griffin Sentinel
4) Plains
5) Plains
6) Siege Mastodon
7) Griffin Sentinel
Choose a card from your hand (0 will pass priority, 'p' prints the game state):
```

Figura 2.2: *Prompt* perguntando qual a ação desejada em uma fase principal. *Pass priority* significa passar para a próxima fase. Há prompts análogos para as decisões de mulligan e de combate.

A classe `Player` representa um jogador **humano** e possui, dentre outros, os métodos `mulligan()` `mainPhaseAction()`, `declareAttackers()`, `declareBlockers()` e `assignBlockOrder()`que são chamados nas respectivas fases por uma instância de jogo (classe `Game`). Tais momentos são impressos na saída padrão e esperam uma decisão em texto do jogador.

Há mais três classes herdadas (diretamente ou não) da classe `Player`. A primeira delas é `MulliganAgent`, cuja única diferença funcional em relação a `Player` é a aplicação de um algoritmo para a decisão automática

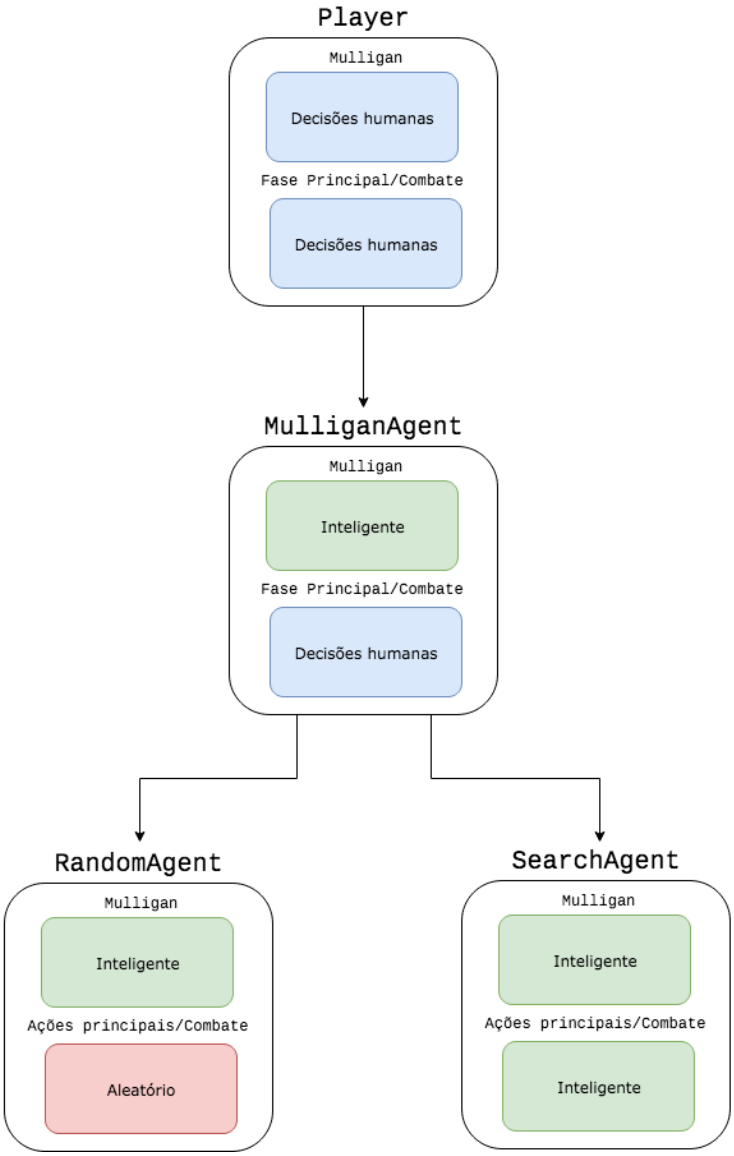

Figura 2.3: Diagrama de classes de jogadores e agentes.

do mulligan. Um agente `MulliganAgent` não é 100% autônomo, pois suas decisões para o resto do jogo ainda são requisitadas ao usuário, e não é chamado pelas opções disponíveis no programa. A existência da classe é justificada pois serve de base para os dois outros agentes (e potencialmente mais): o agente aleatório e o agente inteligente, diferentes entre si. A intenção é que qualquer agente (mesmo aleatório) tenha chance de jogar suas cartas e, para isso, deve definir uma mão inicial suficientemente "jogável" (melhor explicado em 4.1).

A classe `RandomAgent`, herdada de `MulliganAgent`, sobrescreve todas as decisões humanas de `Player` (exceto `mulligan()`) por métodos que sorteiam e submetem uma ação sorteada dentre a gama de possibilidades. Para isso, usam em cada um de seus métodos um conjunto `legalActions` de ações legais computado pela instância de `Game` que os chamou. A obtenção de ações legais é explicada no capítulo 4.

Por fim, a classe `SearchAgent` também é herdada de `MulliganAgent` e sobrescreve os métodos de decisões humana por ações decididas por algoritmos. Tais decisões serão examinadas no capítulo 4.

Capítulo 3

Fundamentos

Neste capítulo introduzimos alguns conceitos sobre inteligência artificial com o objetivo de apresentar ao leitor conceitos que serão utilizados em seções futuras.

3.1 Processo de Decisão de Markov

Processos de Decisão de Markov (em inglês, *Markov Decision Process*, ou **MDP**) são uma forma de representar alguns problemas de decisão sequenciais.

Figura 3.1: Na história do jogo, Markov é um poderoso clã de vampiros.

Para descrever um MDP, usaremos um exemplo com o intuito de tornar a explicação mais didática. Imagine que um gerente de um galpão de um produto tem como objetivo maximizar o lucro esperado para o próximo ano. A cada mês, o gerente observa quanto há em estoque do produto e decide quanto irá pedir ao distribuir para o próximo mês. A demanda mensal do produto é desconhecida, mas segue uma distribuição de probabilidade conhecida. Se o gerente pedir produto demais, terá custos para manter o estoque, e se pedir de menos, estará perdendo vendas por falta de inventário. Suponha que o galpão tem capacidade para armazenar M unidades de produto e que os custos e a distribuição da demanda não muda de mês para mês.

O primeiro conceito de MDP que iremos introduzir é o de **época de decisão**. Uma época de decisão é um momento onde uma decisão é tomada. No nosso exemplo, cada época de decisão ocorre no começo de

um mês, quando o gerente deve decidir quanto produto irá pedir ao distribuidor. Chamamos a quantidade de épocas de decisão de um MDP de **horizonte**, que pode ser finito ou infinito. No exemplo, o horizonte é finito de tamanho 12, sendo uma época para cada mês do ano no qual o gerente deseja maximizar seus lucros.

Um MDP pode ser representado por uma tupla (S, A, P, R). S é o conjunto de **estados** s_t do problema, onde um estado é uma configuração do problema em uma determinada época de decisão t. No exemplo, cada estado representa o espaço disponível no galpão, que pode variar de 0 a M.

A é o conjunto de **ações** a_t, onde t é a época de decisão. No exemplo, uma ação pode ser comprar de 0 a $M - s_t$ unidades no mês.

Para falar sobre os outros elementos da tupla do MDP, precisamos entrar em detalhes sobre os custos para pedir o produto ao distribuidor e manter o produto em estoque. Um mês começa com s_t unidades em estoque, e são encomendadas mais a_t unidades, o que gera um custo c_{pedido} dado pela função $c_p(a_t)$. Vamos assumir que as unidades sejam pedidas no começo do mês e sejam vendidas no final do mês. Há, portanto, um custo $c_{estoque}$ para manter o produto em estoque dado pela função $c_e(s_t + a_t)$. A probabilidade de que a demanda D_t no mês seja de j unidades é $p_j, j = 0, 1, 2 \ldots$. Quando o final do mês chegar, o número de unidade vendidas será $x_t = minD_t, s_t + a_j$, e o lucro será dado pela função $l(x_t)$. O número de unidades que começarão o próximo mês em estoque será $s_{t+1} = s_t + a_t - x_t$.

$R : S \times A \mapsto \mathbb{R}$ é a função que dá a **recompensa** esperada por tomar a ação a_t quando o processo está no estado estado s_t. No exemplo, $r_t(s_t, a_t) = E[l(x_t)] - c_{estoque} - c_{pedido}$.

$P : S \times A \times S \mapsto [0, 1]$ é uma função que dá a **probabilidade** do sistema passar de um estado para outro, dado uma determinada ação. No nosso exemplo:

$$Pr\{s_{t+1} = j | s_t = s, a_t = a\} = \begin{cases} p_{s+a-j}, & j \leqslant s + a \\ \sum_{i=s+a}^{\infty} p_i, & j = 0 \\ 0, & j > s + a \end{cases}$$

Uma vez que temos o nosso MDP definido, queremos chegar em uma **política**, que é o nome dado ao conjunto das **regras de decisão**. Uma regra de decisão $d_t(s)$ é uma função $d : S \mapsto A$ que que, dado um estado s na época de decisão t, retorna uma ação a que deve ser tomada a partir daquele estado de modo a maximizar o ganho final.

3.2 Busca em grafo

Podemos representar um problema de Inteligência Artificial como um grafo.[1] Um grafo G, resumidamente, é um conjunto $G = (V, E)$, onde V é chamado conjunto de vértices e E, conjunto dos arcos. Cada arco $vw \in E$ "leva" o vértice v ao vértice w, $v, w \in V$.

Um problema genérico de busca em grafo pode ser postulado como "encontrar o conjunto C de vértices que podem ser alcançados a partir de um vértice v" (alcançar, no caso, refere-se a percorrer os vértices usando os arcos). O conjunto C é chamado *território* de v. Assim, para $v = a$ no grafo em 3.2, temos $C = \{a, b, c, d, e\}$, pois a não alcança apenas o vértice f. Podemos adicionar algumas especificações para o problema, como por exemplo encontrar os *caminhos simples*[2] para os vértices no território de um vértice v (por exemplo, para o território de c em 3.2, temos o conjunto de caminhos simples $\mathcal{P} = \{(c), (c, cb, b, bd, d), (c, ce, e)\}$). Outra especificação envolve uma *função-custo*, que atribui um número real c(e) para cada arco $e \in E$. Pode ser interessante, então, encontrar os caminhos *mínimos* de um vértice ao seu território, onde o custo $c(P)$ de um caminho é a soma $\sum_e c(e)$ dos custos de cada um de seus arcos.

[1] Neste texto, sempre que mencionarmos *grafos* estaremos nos referindo *grafos dirigidos*.

[2] Um caminho é uma sequência alternante de vérices e arcos em que cada arco tem pontas nos vértices anterior e posterior na sequência, respectivamente. Um caminho simples é um caminho que não contém *laços*, ou seja, não visita um vértice mais de uma vez.

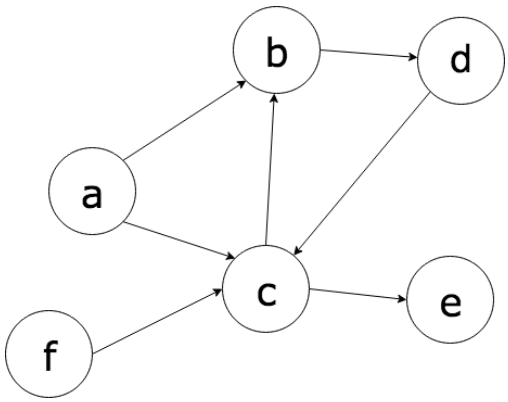

Figura 3.2: Exemplo de grafo G. $V(G) = \{a, b, c, d, e, f\}$ e $E(G) = \{ab, ac, cb, bd, dc, fc, ce\}$.

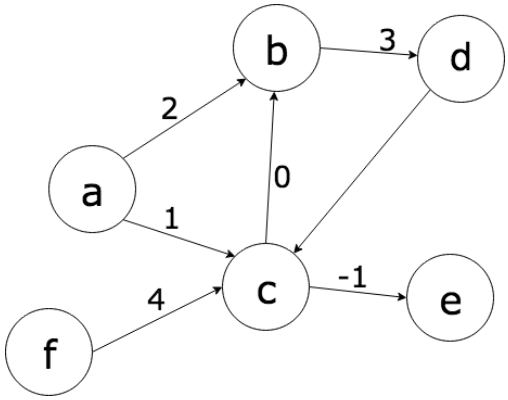

Figura 3.3: Grafo G com uma função-custo c associada. Por exemplo, um caminho mínimo de a a b é $P = (a, ac, c, cb, b)$, onde $c(P) = 1 + 0 = 1$.

Os dois tipos básicos de algoritmo de busca genérica são a **busca em largura** (que utiliza *filas* como estrutura de dados) e a **busca em profundidade** (que, por sua vez, manipula *pilhas*). Ambas fazem uso do conceito de *vizinhos* de um vértice v, ou seja, os vértices w tais que exista arco $vw \in E$. Supondo que queiramos encontrar o caminho mínimo de um vértice v até os vértices de um grafo com custos (V, E, c), a **busca em largura** funciona da seguinte maneira:

Fica como o exercício para o leitor aplicar o algoritmo ao grafo do exemplo 3.3.

3.3 Minimax

Nas modelagens de problemas de busca em jogos individuais, normalmente a solução ótima é obtida a partir de uma sequência de ações levando a um estado desejado (que normalmente representa a vitória do agente). Quando o jogo envolve mais de um jogador, é possível que as ações dos outros jogadores influenciem o estado e quais ações podem ser tomadas. Neste caso, chamamos de Max o agente que está querendo maximizar a recompensa final, enquanto o adversário de Max que está tentando fazer com que este não atinja seu objetivo é chamado de Min. O objetivo de Max é, então, encontrar uma estratégia para conseguir a maior recompensa possível, levando em consideração que Min irá sempre tomar a ação que minimiza a recompensa final.

Para exemplificar, vamos supor um jogo trivial representado pela árvore da imagem 3.4 onde Max escolhe uma de três ações alterando o estado inicial, e então Min escolhe uma de três ações alterando o estado obtido

Busca em largura (V, E, c, v):

fila $q \leftarrow \{v\}$
custo $c_w \leftarrow \infty, \forall w \in V$
caminho $P_w \leftarrow \emptyset, \forall w \in V$
enquanto $q \neq \emptyset$:
    $w \leftarrow$ elemento mais antigo de q, remove w da q
    para cada vizinho u de w:
        se o custo até u, $c_u > c_w + c_{wu}$:
            $c_u \leftarrow c_w + c_{wu}$
            $P_u \leftarrow P_w + (wu, u)$
        se u não foi expandido **então**:
            marque u como expandido
            insira u no final de q
devolva conjunto dos caminhos $P_w, \forall w \in E$

pela ação de MAX, levando a um estado final com uma recompensa associada.

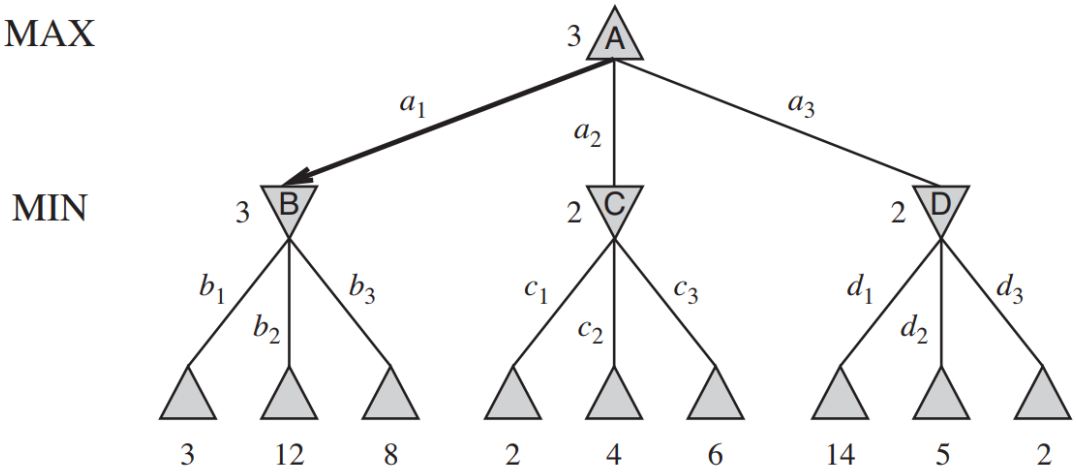

Figura 3.4: O jogo trivial com as ações possíveis de MAX e MIN. Fonte: [3, p. 164]

Dada a árvore de jogo, a estratégia ótima pode ser determinada pelo **valor minimax** de cada nó, que escreveremos como $Minimax(n)$. O valor minimax de um nó é a utilidade, do ponto de vista de MAX, de estar no estado correspondente, assumindo que ambos os jogadores joguem otimamente. Dada uma escolha, MAX prefere mover para um estado de valor máximo, enquanto MIN prefere mover para um estado de valor mínimo:

$$Minimax(s) = \begin{cases} Utilidade(s), & \text{se s é um estado terminal} \\ \max\limits_{a \in A(s)} Minimax(T(s,a)), & \text{se o jogador é MAX} \\ \min\limits_{a \in A(s)} Minimax(T(s,a)), & \text{se o jogador é MIN} \end{cases}$$

onde $A(s)$ é uma função que devolve as ações possíveis a partir de um estado s, e $T(s, a)$ é uma função de transição que devolve o estado resultante de se tomar a ação a a partir do estado s.

Aplicando as definições acima na figura 3.4, os nós no nível mais baixo da árvore recebem seu valor a partir da função de *Utilidade* do jogo. O primeiro nó de MIN, rotulado por B, tem três estados sucessores com valores 3, 12 e 8, então seu valor minimax é 3. Podemos também identificar a **decisão minimax** a partir da raiz: a ação a_1 é a escolha ótima para MAX pois leva ao estado com maior valor minimax.

Capítulo 4

Modelagem

Neste capítulo iremos detalhar a modelagem do jogo como um problema de inteligência artificial. O jogo está dividido em três partes, cada uma usando uma implementação diferente para encontrar a melhor jogada: o mulligan, que acontece uma vez no começo do jogo, foi modelado como um MDP; as fases principais foram modeladas como um problema de busca, utilizando a busca em largura para chegar na melhor ação; e o combate foi modelado como um problema adversarial utilizando minimax. A seguir entraremos em detalhes sobre cada uma dessas modelagens.

4.1 *Mulligan*

A estrutura de um turno, como descrita no primeiro capítulo, será repetida algumas vezes até o jogo acabar, porém é necessário determinar a mão inicial de cada jogador e, por isso, trataremos este problema em separado. Baseando-se em experiências próprias, a estrutura do problema do **mulligan** é notavelmente diferente do resto de um jogo de **Magic**. A principal diferença é que, apesar de ambos os jogadores tomarem decisões alternadamente nessa etapa, as ações do oponente não têm nenhuma influência direta sobre as ações do agente, que deve se concentrar em obter uma **mão inicial** que possibilite jogadas nos primeiros turnos do jogo.

A figura 4.1 representa as escolhas que o jogador poderá fazer para determiná-la, com cada conjunto de cartas representando um conjunto de estados.

Definimos vagamente uma mão como jogável se esta contém tanto cartas que impactam o estado de jogo e recursos necessários para jogá-las. Para este efeito, separaremos as cartas do deck em duas categorias: terrenos (recursos) e não-terrenos (impactam o jogo). Outro fator importante é o tamanho da mão, pois cada carta perdida representa um turno de atraso em relação a uma mão com sete cartas (uma vez que cada jogador compra uma carta por turno). Dessa maneira, em uma mão com i cartas, temos $i+1$ possibilidades, cada uma com probabilidade

$$\mathcal{P}_i(j) = \frac{\binom{J}{j}\binom{60-J}{i-j}}{\binom{60}{i}}, \quad j = 0, \ldots, i \tag{4.1}$$

onde J é o número de terrenos no deck e j o número de terrenos na mão. Note que $\sum_{j=0}^{i} P_i(j) = 1, \forall i$. Assim, a cada nível, como é mostrado na figura 4.1, o agente deve decidir entre realizar um mulligan (denotado por M), que resultará em uma mão com $i-1$ cartas, sendo j' terrenos com probabilidade

$$\mathcal{P}_{i-1}(j') = \frac{\binom{J}{j'}\binom{60-J}{i-1-j'}}{\binom{60}{i-1}}, \quad j' = 0, \ldots, i-1$$

ou manter a mão (denotado por K), o que termina o problema, seguido (para $i < 7$) de uma ação *scry*, que permite uma pequena "filtragrem" do deck para os jogadores que já acumularam alguma desvantagem (em

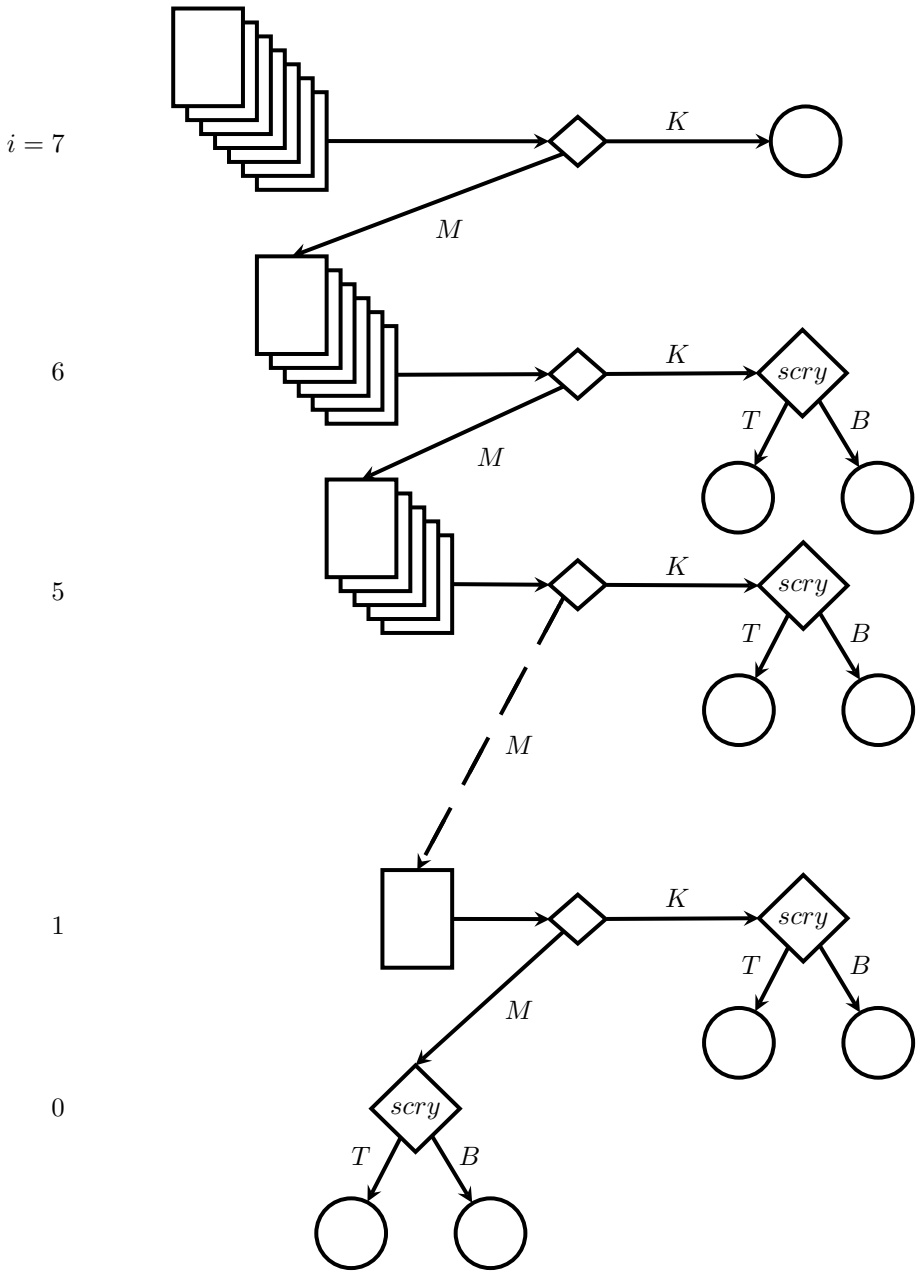

Figura 4.1: Representação visual de um problema de mulligan. Cada nível i representa o número de cartas na mão inicial do jogador. K significa a decisão de manter a mão, M realizar um mulligan, T deixar a carta no topo em um *scry* e B colocá-la no fundo.

relação ao número de cartas na mão) no processo. Por fim, a decisão também é influenciada pela informação de quem é o jogador inicial, pois ele tem virtualmente uma carta a menos, dado que no primeiro turno do jogo não se compra uma carta.

4.1.1 Mulligan como um MDP

Dada a estrutura bem conhecida do problema e sua natureza fixa (temos certeza que acabará em um determinado horizonte), podemos determinar todos os parâmetros de um MDP de horizonte finito para que o agente siga uma política com a inteção de alcançar uma mão "jogável" para uma partida de **Magic**.

O conjunto de estados S é composto por um estado inicial S_I (que representa o estado antes do jogador comprar a sua primeira mão inicial), um estado final absorvente S_F (que representa o jogador após ter decidido manter uma mão) e os estados intermediários, dados por um par $(i, j), i \in \{0, 1, \dots 7\}, j \in \{0, 1, \dots, i\}$ (que representam os estados onde o jogador tem i cartas na mão, sendo que j são terrenos). Assim, as ações $A(s)$ e as probabilidades $P(s'|s, A(s))$ de cada estado s são definidas da seguinte maneira:

- $A(S_I) := \{Start\}$;
$$P((7, j)|S_I, Start) = \mathcal{P}_7(j);$$
A partir de S_I é possível tomar a ação *Start*, que leva para o estado $(7, j)$ com probabilidade $\mathcal{P}_7(j)$, descrita na equação 4.1.

- $A(i, j) := \{Mulligan, Keep\}, i \in \{1, \dots, 7\}, j \in \{0, \dots, i\}$;

- $A(0, 0) := \{Keep\}$;
$$P((i-1, j')|(i, j), Mulligan) = \mathcal{P}_{i-1}(j'), \quad j' \in \{0, 1, \dots, i-1\},$$
$$P(S_F|(i, j), Keep) = 1$$
A partir de $(i, j), i \neq 0$, analogamente à ação *Start*, temos a ação $Mulligan$, que leva a um estado $(i-1, j')$. Um detalhe importante é a impossibilidade de $Mulligan$ quando $i = 0$. A outra ação, $Keep$, leva ao estado final.

- $A(S_F) := \{Wait\}$
$$P(S_F|S_F, Wait = 1)$$
No estado final só há a ação $Wait$, um artifício para que o MDP sempre tenha o mesmo número de épocas de decisão (e constitua um problema de horizonte finito).

Por fim, ainda precisamos determinar as recompensas $R(s'|s, a \in A(s))$ associadas a cada transição. A notação adotada é $R_k(i, j) = R(S_F|(i, j), Keep)$, ou seja, $R_k(i, j)$ é a recompensa associada à ação $Keep$ a partir estado (i, j) (em alto nível, é a decisão de manter uma mão com i cartas, sendo j delas terrenos). Como a única decisão que realmente importa para a pontuação é a ação $Keep$ (pois esta garante que a mão atual será a mão inicial), determinamos que todas as outras transições tem recompensa $R_0 = 0$. A partir de experiências pessoais, podemos determinar R_k a partir das seguintes restrições:

- Com $i = i_0$ fixo e $j \in \{0, 1, \dots, i_0\}$, as recompensas $R_k(i_0, j)$(se existem) devem seguir a seguinte relação:
$$R_k(i_0, 3) > R_k(i_0, 2) > R_k(i_0, 4) > R_k(i_0, 5) > R_k(i_0, 1) > R_k(i_0, 6) > R_k(i_0, 7) > R_k(i_0, 0), \quad (4.2)$$
pois na média, uma mão ideal tem 3 terrenos, em segundo lugar 2 terrenos, etc. Uma mão sem terrenos é considerada muito ruim.

- $R_k(i, j) > R_k(i-1, j), \forall (i, j)$: uma mão com o mesmo número de terrenos e uma carta a menos deve sempre ser considerada pior do que a alternativa.

- $R_k(i,j) > R_k(i-1,j-1), \forall (i,j)$: uma mão com o mesmo número de não-terrenos e uma carta a menos deve sempre ser considerada pior do que a alternativa.

Dessa maneira, chegamos a uma fórmula geral, $R_k(i,j) = r(j) + \alpha i$, onde $r(j)$ assume valores pré-determinados de acordo com as desigualdades 4.2.

4.1.2 Extração da Política

Com o MDP definido, desejamos agora extrair uma política, ou seja, um conjunto de regras de decisão, que digam ao agente qual ação tomar a partir de um dado estado, mas para isso precisamos da definição da **função de valor**. Uma função de valor para uma política π, denotada por $V^\pi : S \mapsto \mathbb{R}$, é tal que $V^\pi(s)$ dá o valor esperado da recompensa para esta política.

Para o problema do Mulligan com uma política π com regra de decisão d, temos que a função de valor é:

$$V_k^\pi = R(s, d(s)) + \sum_{s' \in S} P(s'|s, d(s)) V_{k+1}^\pi(s') \tag{4.3}$$

onde k é a época de decisão, e $V_z^\pi(s) = 0$, onde z é a última época de decisão, pois não há mais ações possíveis.

Para extrair a política para o mulligan, utilizamos o algoritmo de **iteração de valor**, modificado para atender as especifidades do nosso problema. O algoritmo utiliza programação dinâmica para determinar o valor $V^*(s)$ para cada $s \in S$ (o valor da função de valor para a melhor política a partir do estado s), e assim construir "de trás para frente" a melhor política para o problema.

Começamos com a mão de 0 cartas, onde só há uma ação possível (manter a mão), então

$$V^*(0,0) = R_k(0,0)$$

e a melhor ação a partir de uma mão com 0 é $Keep$ (uma vez que não há outra ação possível). Então, calculamos V^* para mãos com 1 carta e j terrenos:

$$V^*(1,j) = \max\left\{R_k(1,j), V^*(0,0)\right\}$$

Como a recompensa por manter uma mão com 1 carta é maior do que a de manter uma mão com 0 cartas independente da configuração, neste caso $V^*(1,j) = R_k(1,j)$ e a melhor ação para esta mão será $Keep$. Para mãos com duas cartas, o valor de V^* leva em consideração as duas possibilidades de mãos com 1 carta, então a fórmula é um pouco mais complexa:

$$V^*(2,j) = \max\left\{R_k(2,j), \sum_{j'=0}^{1} V^*(1,j') \times \mathcal{P}_1(j')\right\}$$

Se o valor de $R_k(2,j)$ for maior que $\sum_{j'=0}^{1} V^*(1,j') \times \mathcal{P}_1(j')$, a ação ótima para uma mão com 2 cartas na mão, sendo que j são terrenos, será $Keep$, caso contrário, será $Mulligan$. Para mãos com $i > 2$ cartas e j terrenos, a fórmula e a lógica para decidir a ação ótima são as mesmas:

$$V^*(i,j) = \max\left\{R_k(i,j), \sum_{j'=0}^{i-1} V^*(i-1,j') \times \mathcal{P}_{i-1}(j')\right\}$$

Uma vez que V^* estiver calculado para todo estado s com uma ação ótima a_s associada, podemos construir a nossa política com a regra de decisão d tal que $d(s) = a_s$ (como cada estado só pode ocorrer em uma época de decisão específica, não há a necessidade de ter mais de uma regra de decisão em nossa política).

4.1.3 Detalhes da Implementação

O primeiro agente inteligente do projeto é `MulliganAgent`, derivado da classe `Player`. A intenção desta classe é providenciar a qualquer agente uma possibilidade maior de obter uma "mão jogável".

```
class MulliganAgent(Player):

    def __init__(self, ID, onThePlay):
        self.initializeAttributes(self, ID, onThePlay)

        if self.onThePlay:
            self.landRewards = [-7, -3, 3, 4, 2, -1, -4, -6]
        else:
            self.landRewards = [-4,  0, 5, 6, 3,  0, -3, -5]

    def initMulliganTables(self):     ...
    def setLibrary(self, deck):       ...
    def mulligan(self):               ...
    def getHandReward(self, hand):    ...
    def getKeepReward(self, i, j):    ...
    def mulliganValueIteration(self): ...
    def getMulliganProb(self, i, j):  ...
    def getMulliganValue(self, hand): ...
```

O método `mulligan()`, sobrescrito, chama o algoritmo de iteração de valor de horizonte finito em `mulliganValueIteration()` para decidir se vale a pena ou não manter a mão atual. Apesar da decisão ser feita várias vezes, o algoritmo é executado apenas uma vez, para preencher as tabelas inicializadas em `initMulliganTables()`. A tabela-base de recompensa pode assumir dois conjuntos de valores, relacionados à diferença de "pontuação" caso o jogador comece ou não o jogo (fato indicado pela *flag* `onThePlay`[1]).

```
1 def mulliganValueIteration(self):

2     for i in range(7, -1, -1):
3         for j in range(i + 1):
4             self.mulliganValue[7 - i][j] = self.getKeepReward(i, j)

5     for i in range(1, 8):

6         mullSum = 0
7         for jLine in range(i):
8             prob = self.getMulliganProb(i - 1, jLine)
9             value = self.mulliganValue[7 - (i - 1)][jLine]
10            mullSum += prob * value

11        for j in range(i + 1):
12            if mullSum >= self.getKeepReward(i, j):
13                self.mulliganValue[7 - i][j] = mullSum
```

Figura 4.2: Método de iteração de valor na classe `MulliganAgent`

[1] O termo "on the play" é uma expressão em **Magic** que significa que o jogador começa o jogo, ou seja, joga ("plays") o primeiro turno. Sobre o segundo jogador, diz-se que está "on the draw", pois este compra uma carta no seu primeiro turno enquanto seu antecessor, não. Estas diferenças representam um esforço de equilíbrio nas regras.

Os valores V_0^* iniciais são iguais às recompensas $R_k(i,j)$ associadas às ações $Keep$, portanto, nas linhas 2-4 do código 4.2 temos a inicialização de $V*(i,j)$ para cada estado (i,j) do problema. O laço nas linhas 5-13 contém a comparação entre a soma $\Sigma(i)$ (definida na seção anterior para cada quantidade de cartas na mão), calculada em 7-10 de acordo com a equação 4.3, e o valor da recompensa $R_k(i,j)$ para cada estado (i,j). A atribuição (ou não) na linha 13 definirá a extração da política posteriormente.

4.2 O Jogo

As fases principais estão modeladas como um problema de busca, onde o estado desejado é o estado final com a maior recompensa. O agente parte do estado inicial, que é o jogo no começo de seu turno, após ele comprar a carta do turno, e então testa todas as combinações de ações para descobrir qual será o melhor estado no final do turno, e quais são as ações necessárias para alcançar esse estado. É relevante mencionar que nessa modelagem o agente é "míope", ou seja, ele irá procurar ações que levem ao melhor estado próximo, mas isso não necessariamente significa que as ações serão boas a longo prazo. No caso do jogo de **Magic**, isso faz com que o agente tenha uma postura "agressiva": ao buscar qual é o melhor estado no final de seu próprio turno, o agente não levará em consideração o turno seguinte, que será controlado pelo oponente. Portanto, não irá considerar quais criaturas devem deixar de atacar para poderem bloquear durante o turno do oponente, que é uma parte tão importante do jogo quanto seu próprio turno. Sendo assim, o agente tende a atacar mais do que um jogador humano.

Para que consigamos executar a busca, é preciso definir algumas coisas: o que é um estado, quais são as ações possíveis do agente a partir de um estado, e como calcular a recompensa para comparar estados.

Cada estado no jogo representa uma diferente configuração entre pontos de vida dos jogadores (v_1 e v_2), cartas na mão do agente (lista H), quantidade de cartas na mão do agente e do oponente (h_1 e h_2) e nos dois baralhos (d_1 e d_2), cartas nos cemitérios dos dois jogadores (listas G_1 e G_2) e por fim as cartas presentes no campo de batalha (duas listas, B_1 e B_2), bem como seus estados (virado, desvirado) e quantidade de dano marcado, caso sejam criaturas. É preciso que o estado represente também algumas outras informações, como quem é o jogador ativo (representado por ac_1, ac_2), em qual fase o turno se encontra, quais criaturas entraram no campo de batalha neste turno (pois não poderão atacar) e quais criaturas estão atacando/bloqueando. Todas as possibilidades de estados formam o conjunto S (dependendo das cartas que compuserem os baralhos, o conjunto S pode ser infinito). Formalmente, então, temos

$$S := \begin{Bmatrix} v_1, v_2 \in \mathbb{Z}, \\ act_1, act_2 \in \{0, 1\}, \\ fase \in \{Principal, Combate\} \\ H \subseteq \Omega, \\ h_1, h_2 \in \mathbb{N}, \\ d_1, d_2 \in \mathbb{N}, \\ G_1, G_2 \subseteq \Omega, \\ B_1, B_2 \subseteq \Omega^* \end{Bmatrix}, \tag{4.4}$$

onde Ω é o conjunto de todas as cartas e Ω^* é o conjunto das chamadas *permanentes*, ou seja, cartas que permanecem no campo de batalha, cada uma com uma carta relacionada mais os atributos mencionados (virada ou desvirada, se pode ou não atacar e se está atacando ou bloqueando e quanto dano sofreu neste turno).

O conjunto de ações pode ser pensado como dois conjuntos: o conjunto das ações que podem ser tomadas durante uma fase principal; e o conjunto das ações tomadas durante o combate. Como estes conjuntos são disjuntos, iremos analisá-los separadamente.

Fase Principal

Durante a Fase Principal o jogador ativo pode jogar cartas. Para que o conjunto de ações A^P em uma Fase Principal seja não-vazio, é necessário que o agente seja o jogador ativo (o jogador ativo alterna todo turno). Definimos A^P da seguinte maneira, onde cada ação significa jogar uma carta (fora a ação de terminar a fase) e pode ser atribuída aos seguintes subconjuntos, cada um com condições próprias:

$$A^P := \{Land, Creature(c), Sorcery(c, T), CreatWithEff(c, T), Pass\}$$

onde cada subconjunto pode ser descrito da seguinte maneira:

- *Land* é o subconjunto de todas as ações que representam jogar uma carta de terreno. Para cada carta de terreno na mão do agente, haverá uma ação correspondente pertencente a *Land* caso o agente ainda não tenha jogado um terreno no turno.

- $Creature(c)$ é o subconjunto das ações que representam jogar uma carta de criatura com custo de mana convertido[2] c que não tenham uma habilidade que precise de um alvo. Para cada carta de criatura i com custo de mana convertido c_i e sem habilidades com alvo na mão do agente, haverá uma ação pertencente a $Creature(c_i)$ disponível caso o agente tenha pelo menos c_i terrenos desvirados.

- $Sorcery(c, T)$ é o subconjunto das ações que representam jogar uma carta de feitiço com custo de mana convertido c e que tenham o objeto T como alvo. Para cada carta de feitiço i com custo de mana convertido c_i na mão do agente, para cada permanente ou jogador t que possa ser alvo do feitiço i, haverá uma ação disponível pertencente a $Sorcery(c_i, t)$ se o agente tiver pelo menos c_i terrenos desvirados e t estiver no campo de batalha ou for um jogador. Por exemplo, o feitiço Flame Slash tem o seguinte texto, em português: "Golpe Ardente causa 4 pontos de dano à criatura alvo" e tem custo de mana . Mesmo que o agente tenha uma montanha desvirada, só será possível jogar Flame Slash se existir pelo menos uma criatura em jogo.

Figura 4.3: Ravenous Chupacabra (Chupacabra Voraz) tem uma habilidade que requer uma criatura de um oponente como alvo

- $CreatWithEff(c, T)$, de maneira análoga ao subconjunto $Sorcery(c, T)$, é o subconjunto das ações que representam jogar uma criatura de custo c com algum efeito que tenha T como alvo. Para cada criatura i de custo convertido c_i na mão do agente, para cada permanente ou jogador t que possa ser alvo da habilidade da criatura i, haverá uma ação disponível pertencente a $CreatWithEff(c_i, t)$ se o agente tiver pelo menos c_i terrenos desvirados e t estiver no campo de batalha ou for um jogador. Se o agente tiver pelo menos c_i terrenos desvirados, mas não houver nenhuma permanente que possa ser alvo da habilidade da criatura i no campo de batalha, uma ação do subconjunto $Creature(c_i)$ estará disponível ao invés disso, pois jogar uma criatura é sempre possível, indepentende da existência de alvos legais para sua habilidade. Por exemplo, a criatura Ravenous Chupacabra (4.3) tem custo de mana convertido 4 e a seguinte habilidade, em português: "Quando Chupacabra Voraz entra no campo de batalha, destrua a criatura alvo que um oponente controla." Se o oponente do agente tiver pelo menos uma criatura no campo de batalha, não haverá uma ação pertencente a $Creature(4)$ associada

[2]Custo de mana convertido é a quantidade de mana total necessária para se jogar uma carta, independente do tipo de mana requisitado. Uma carta com custo de mana tem custo de mana convertido 6.

a jogar Ravenous Chupacabra, pois seu efeito demanda que uma criatura do oponente seja destruída. Entretanto, se o oponente não controlar nenhuma criatura, a única ação associada a jogar Ravenous Chupacabra pertencerá a $Creature(4)$, pois ainda é possível jogar a criatura.

- *Pass* é um subconjunto com uma única ação, que está sempre disponível: terminar a fase principal. Se o agente escolher essa ação durante a primeira fase principal, começa então a fase de combate. Caso o agente escolha a ação durante a segunda fase principal, seu turno termina e começa o turno do oponente.

Para este projeto, restringimos o conjunto de cartas utilizadas de tal maneira que toda ação $a \in A^P$ leva s a um estado s' com probabilidade $P(s'|s,a) = 1$.[3]

Para definirmos um critério para que o agente pudesse verificar se um estado é melhor que outro, definimos a seguinte recompensa para um estado s, baseada nas nossas experiências como jogadores:

$$R(s) = \frac{v_1 - v_2}{5} + (p_1 - p_2) + \frac{(t_1 - t_2)}{2} + (1,1 \times l) + h$$

onde v_i é o total de pontos de vida do jogador i, p_i é a soma do poder de todas as criaturas controladas por i, t_i é a soma da resistência de todas as criaturas controladas por i, l é a quantidade de terrenos controladas pelo agente, e h é a quantidade de cartas na mão do agente.

O exemplo nas páginas seguintes demonstra uma sequência de estados em uma fase principal e as ações legais correspondentes.

Exemplo - Fase Principal

Para exemplificar estados e ações legais, vamos examinar uma fase principal em que o agente é o jogador ativo a partir do começo do turno. O estado s_1 é, no caso, o estado inicial. Os atributos d_1, d_2 dos estados estão suprimidos, pois são irrelevantes para o exemplo. Para este exemplo iremos supor que o oponente do agente não tem nenhuma carta em sua mão, campo de batalha ou cemitério, e ambos os jogadores têm 10 pontos de vida.

Figura E.1.1: estado s_1

$$s_1 = \{v_1 = v_2 = 10, h_1 = 4, h_2 = 0, B_2 = G_1 = G_2 = \emptyset, B_1, H\}$$
$$R(s_1) = 3,3 + 4 = 7,3$$

B_1: O campo de batalha consiste apenas em 3 terrenos (Montanhas).

[3]Efeitos que adicionariam alguma incerteza à modelagem seriam comprar cartas, efeitos que envolvem escolhas e/ou informações do oponente (como "o oponente alvo descarta uma carta"), dentre outros. Estes efeitos proporcionariam probabilidades diferentes.

H: A mão do agente consiste em um terreno (Montanha), um feitiço "Volcanic Hammer", um feitiço "Flame Slash" e uma criatura "Pensive Minotaur".

Dessa maneira, o conjunto de ações em s_1 pode ser definido a partir de H e B_1, B_2:

$$A^P(s_1) := \begin{Bmatrix} Mountain \in Land, \\ PensiveMinotaur \in Creature(3), \\ VolcanicHammer(self) \in Sorcery(2, self), \\ VolcanicHammer(opponent) \in Sorcery(2, opponent), \\ Pass. \end{Bmatrix} \tag{4.5}$$

Como não há criaturas em jogo, "Flame Slash" não tem alvos legais e não pode ser jogada. "Volcanic Hammer", por outro lado, admite ambos os jogadores como alvo válido, portanto pode ser jogada através de duas ações diferentes.

Suponhamos que o agente jogue a Montanha que tem na mão. Sendo assim, o estado s_2 é parecido com s_1.

Figura E.1.2: estado s_2

$$s_2 = \{v_1 = v_2 = 10, h_1 = 3, h_2 = 0, B_2 = G_1 = G_2 = \emptyset, B_1, H\}$$
$$R(s_2) = 4, 4 + 3 = 7, 4$$

B_1: O campo de batalha consiste apenas em 4 terrenos (Montanhas).

H: A mão do agente consiste em um feitiço "Volcanic Hammer", um feitiço "Flame Slash" e uma criatura "Pensive Minotaur".

O conjunto de ações em s_2 é quase igual ao conjunto de ações em s_1 :

$$A^P(s_2) := \begin{Bmatrix} PensiveMinotaur \in Creature(3), \\ VolcanicHammer(self) \in Sorcery(2, self), \\ VolcanicHammer(opponent) \in Sorcery(2, opponent), \\ Pass. \end{Bmatrix} \tag{4.6}$$

"Flame Slash" ainda não pode ser jogada.

Agora, a ação escolhida em s_2 é jogar a carta "Pensive Minotaur", uma criatura sem efeitos. Para isso, é necessário virar 3 terrenos, alterando duplamente B_1.

Figura E.1.3: estado s_3

$$s_3 = \{v_1 = v_2 = 10, h_1 = 2, h_2 = 0, B_2 = G_1 = G_2 = \emptyset, B_1, H\}$$
$$R(s_3) = 2 + 1, 5 + 4, 4 + 2 = 9, 9$$

B_1: Com "Pensive Minotaur" jogada, o campo de batalha agora contém ela, três montanhas viradas e uma montanha desvirada.

H: A mão do agente consiste em um feitiço "Volcanic Hammer" e um feitiço "Flame Slash".

O conjunto de ações legais em s_3 muda radicalmente:

$$A^P(s_3) := \begin{Bmatrix} FlameSlash(PensiveMinotaur \in B_1) \in Sorcery(1, e \in B_1), \\ Pass. \end{Bmatrix} \tag{4.7}$$

"Flame Slash" agora pode ser jogada.

Em s_3, como o agente tem apenas um terreno desvirado em jogo, jogar "Volcanic Hammer" em qualquer alvo passam a ser ações ilegais (pois são necessários dois terrenos para isso). Por outro lado, jogar "Flame Slash" passa a ser possível pois há uma criatura em campo (e a carta requer apenas uma Montanha). Apesar de parecer uma jogada péssima, para efeitos de exemplo suponhamos que esta ação seja tomada.

Figura E.1.4: estado s_4

$$s_4 = \{v_1 = v_2 = 10, h_1 = 2, h_2 = 0, B_2 = G_2 = \emptyset, B_1, H, G_1\}$$
$$R(s_4) = 4, 4 + 1 = 5, 4$$

B_1: "Pensive Minotaur" foi destruída, restam apenas as 3 Montanhas viradas usadas para jogá-la e a outra Montanha, virada, usada para jogar "Flame Slash".

H: A mão do agente consiste apenas em um feitiço "Volcanic Hammer".
G_1: A pilha de descarte do agente, por sua vez, agora contém ambos o feitiço "Flame Slash" (quando um feitiço é usado, vai para a pilha de descarte) e a criatura "Pensive Minotaur" (destruída após o efeito de "Flame Slash" lhe causar dano maior ou igual à sua resistência).

A única ação legal em s_4 é passar para o final da fase:

$$A^P(s_4) := \{Pass.\} \tag{4.8}$$

Apesar de ter um feitiço na mão, o agente não tem os recursos disponíveis para jogá-lo. Assim, a única ação legal é passar para a fase de combate, encerrando a primeira fase principal.

4.2.1 Busca local em uma Fase Principal

Para determinar em primeira instância as ações de uma fase, é possível realizar uma busca local exaustiva dentre todas as possibilidades. Assim, podemos examinar cada conjunto de ações – que levará a um estado final próprio – e determinar, pela função de recompensa definida na seção anterior – o estado terminal (ou seja, após uma ação *Pass*) desejado.

Para realizar uma busca em grafo numa fase principal, podemos definir cada estado s como um vértice do grafo e cada ação a um arco que leva um estado s a algum estado s'. Como mencionado anteriormente, todas as ações de Fase Principal levam a exclusivamente um estado sucessor, o que nos permite fazer esta transição para representação em grafo. Esta busca possui algumas diferenças em relação ao exemplo especificado em 3.2:

- Temos um vértice inicial fixo determinado pelo estado inicial de uma Fase Principal.
- A árvore de estados não está pronta de antemão – o agente precisa "testar" as ações para observar os estados sucessores.
- Não estamos interessados em encontrar caminhos para todos os vértices, mas apenas caminhos para os vértices finais (ou seja, "criados" a partir de uma ação *Pass*).
- Em vez de procurar caminhos de custo mínimo, estamos buscando um *estado final com recompensa máxima*. Pode-se considerar, então, que este problema de busca procura o caminho até um estado-objetivo, mas não sabemos de antemão se algum estado é o objetivo (já que precisamos construir a árvore e comparar todos os estados terminais possíveis.)

4.2.2 Detalhes da implementação

A classe `SearchAgent` descrita no capítulo 2 conta com um método de busca em largura. Este método é invocado, por sua vez, pelo método `mainPhaseAction()` do agente, chamado pela classe `Game` no início de cada Fase Principal.

```
1  def breadthFirstSearch(self, startState):
2      q = queue.Queue()
3      q.put(startState)
4      maxReward = -200
5      actionPath = []

6      while not q.empty():
7          state = q.get()

8          if state.isTerminal():
9              reward = state.getReward()
10             if reward >= maxReward:
11                 maxReward = reward
12                 actionPath = state.getPath()

13         else:
14             for s in self.getChildren(state):
15                 q.put(s)

16     return actionPath
```

Figura 4.4: Método de busca em largura do agente inteligente.

A função é bastante parecida com o algoritmo descrito no capítulo 3, mas adaptada para o problema da Fase Principal. No caso, as recompensas só são checadas para estados terminais, e cada estado `state` tem seu caminho a partir do estado inicial retornável por `state.getActionPath()`. A função `getChildren()` retorna a lista de estados sucessores de um estado dado s, ou seja, aplicando cada ação legal ao estado e colocando o estado resultante na lista (neste momento a árvore de busca vai sendo "desbravada"). Assim, após determinado o estado terminal desejado, a função retorna a sequência de ações escolhida pelo agente, então submetida e aplicada no cliente de jogo.

O construtor da classe recebe como parâmetro o jogo, a fase atual de jogo (em uma *string*) e o caminho de ações escolhido para chegar nele. A partir das informações do jogo é possível extrair os atributos do estado relevantes para a determinação da recompensa. Retomaremos o detalhamento da classe no final do capítulo.

```
class State:
    def __init__(self, game, phase, actionPath): ...
    def isTerminal(self):  ...
    def getReward(self):   ...
    . . .
```

Figura 4.5: Código da classe de estado.

Por fim, devemos notar que esta busca é *menos* eficiente do que uma busca em largura convencional pois *não* há checagem se um estado já foi visitado: como, na implementação, estados são objetos diferentes, seria muito custoso guardar na memória todos os estados visitados e comparar os atributos de cada estado novo com estes.

4.2.3 Combate

O combate é bem diferente da fase principal, pois não há jogada de cartas ou informação incompleta: tudo acontece com as criaturas já no campo de batalha. Sua estrutura é a seguinte: o jogador ativo *declara atacantes*, ou seja, escolhe quais das suas criaturas atacarão e, em seguida, seu oponente *declara bloqueadores*, escolhendo quais das suas criaturas bloquearão cada criatura atacante. Após isso, cada criatura atacante causa dano igual a seu poder às criaturas que a bloquearam ou ao jogador defensor (caso não tenha sido bloqueada), e cada criatura bloqueadora causa dano igual a seu poder à criatura que bloqueou. Todo dano é causado ao mesmo tempo. Por fim, cada criatura com dano maior ou igual à sua resistência é destruída: sua carta sai do campo de batalha e passa para a pilha de descarte correspondente.[4]

O conjunto A_1^C de ações do jogador ativo no combate, portanto, é o conjunto de escolhas de todas as combinações de criaturas atacantes (incluindo nenhuma). Dado que o jogador deve escolher alguma combinação,

$$|A_1^C| = \sum_{i=0}^{N} \binom{N}{i} = 2^N, \tag{4.9}$$

onde N é o número de criaturas que podem atacar na mesa do jogador ativo.

O conjunto A_2^C de ações do jogador não-ativo durante o combate, por sua vez, é o conjunto de escolhas de todas as combinações de criaturas bloqueadoras e aquelas que estas bloqueiam. Como cada criatura desvirada pode bloquear até uma criatura atacante, temos que

$$|A_2^C| = (N' + 1)^M \leqslant (N + 1)^M, \tag{4.10}$$

onde N' é o número de criaturas atacantes e M o número de criaturas aptas a bloquearem.

Assim como na fase principal, cada ação $a \in \left\{A_1^C, A_2^C\right\}$ no estado s tomada tem probabilidade $P(s'|s, a) = 1$ para o estado s' sucessor previsto pelas regras.

Nas próximas páginas demonstraremos por exemplos os estados e conjuntos de ações (de ambos jogadores) em uma fase de combate.

[4] Nesta seção não trataremos das *habilidades de criatura* relevantes para o combate, de maneira a simplificar as explicações. Trataremos melhor disso mais a frente.

Exemplo - Combate

Suponhamos um estado inicial t_1 da seguinte maneira (vamos suprimir as informações sobre mãos e baralhos pois estas são irrelevantes)

Figura E.2.1: estado t_1

$$t_1 = \{v_1 = v_2 = 10, G_1 = G_2 = \emptyset, B_1, B_2\}$$

B_1: O jogador ativo tem três criaturas aptas a atacar em campo: dois "Grizzly Bears" e um "Centaur Courser". Assim, podemos obter o conjunto de ações legais A_1^C:

$$A_1^C(t_{1,1}) := \left\{ \begin{array}{l} GrizzlyBears1, \\ GrizzlyBears2, \\ CentaurCourser, \\ \{GrizzlyBears1, GrizzlyBears2\}, \\ \{GrizzlyBears1, CentaurCourser\}, \\ \{GrizzlyBears2, CentaurCourser\}, \\ \{GrizzlyBears1, GrizzlyBears2, CentaurCourser\}, \\ None. \end{array} \right\} \tag{4.11}$$

O jogador ativo pode declarar qualquer combinação (incluindo nenhuma) de criaturas como atacantes. Note que $N = 3$ e $|A_1^C| = 2^N = 8$.

Suponhamos que o jogador ativo escolheu a ação $\{GrizzlyBears2, CentaurCourser\}$ em t_1. Assim, o jogador declarou ambas suas criaturas como atacantes. Assim como com terrenos e geração de recursos, uma criatura atacante é virada quando declarada atacante (e precisa estar desvirada previamente para poder atacar). Assim, podemos obter o estado seguinte, t_2.

Figura E.2.2: estado t_2

$$t_2 = \{v_1 = v_2 = 10, G_1 = G_2 = \emptyset, B_1, B_2\}$$
$$R(t_2) = \frac{10-10}{5} + (7-4) + \frac{7-4}{2} = 4,5$$

B_1: O jogador ativo tem um "Grizzly Bears" desvirado e um "Grizzly Bears" e um "Centaur Courser" virados, ambos atacando ($N' = 2$).

B_2: O jogador defensor tem dois "Walking Corpse" desvirados (e aptos a bloquear). Portanto, as ações legais são todas as combinações de bloqueio das criaturas atacantes:

$$A_2^C(t_2) := \left\{ \begin{array}{l} \{WalkingCorpse1 : None, WalkingCorpse2 : None\}, \\ \{WalkingCorpse1 : GrizzlyBears2, WalkingCorpse2 : None\}, \\ \{WalkingCorpse1 : CentaurCourser, WalkingCorpse2 : None\}, \\ \{WalkingCorpse1 : None, WalkingCorpse2 : GrizzlyBears2\}, \\ \{WalkingCorpse1 : None, WalkingCorpse2 : CentaurCourser\}, \\ \{WalkingCorpse1 : CentaurCourser, WalkingCorpse2 : GrizzlyBears2\}, \\ \{WalkingCorpse1 : GrizzlyBears2, WalkingCorpse2 : CentaurCourser\}, \\ \{WalkingCorpse1 : GrizzlyBears2, WalkingCorpse2 : GrizzlyBears2\}, \\ \{WalkingCorpse1 : CentaurCourser, WalkingCorpse2 : CentaurCourser\}. \end{array} \right\} \quad (4.12)$$

Assim, há $|A_2^C(t_2)| = (N'+1)^M = (2+1)^2 = 9$ ações legais.

Assim, há alguns estados sucessores possíveis. Vamos examinar algumas configurações de bloqueio distintas[5] e seus respectivos estados sucessores.

[5]Algumas ações são essencialmente a mesma, pois os dois "Walking Corpse" não têm diferenças. Como, nas regras do jogo, são considerados objetos diferentes, estamos distinguindo-os para manter a consistência.

Figura E.2.3: estado t_3

Suponhamos que a ação escolhida pelo agente tenha sido

$$\{WalkingCorpse1 : GrizzlyBears2, WalkingCorpse2 : None\},$$

ou seja, bloquear "Grizzly Bears" com um "Walking Corpse" e não bloquear "Centaur Courser".

Esta é a representação da "resolução do combate" pré-t_3. O estado t_3, portanto, refletirá as consequências das ações escolhidas. Podemos listá-las:

- "Grizzly Bears" causou um número igual a seu poder (2) de dano ao "Walking Corpse" que o bloqueou.
- "Walking Corpse" causou um número igual a seu poder (2) de dano ao "Grizzly Bears" bloqueado.
- "Centaur Courser" não foi bloqueado, causando um número igual a seu poder (3) de dano ao jogador defensor.

Assim, ambos "Grizzly Bear" e "Walking Corpse" são destruídos (pois receberam dano maior ou igual às suas resistências) e temos o seguinte estado t_3:

$$t_3 = \left\{ \begin{array}{l} v_1 = 10, v_2 = 10 - 3 = 7, \\ G_1 = \{GrizzlyBears2\}, G_2 = \{WalkingCorpse1\}, \\ B_1 = \{GrizzlyBears1(u), CentaurCourser(t)\}, \\ B_2 = \{WalkingCorpse2(u)\}. \end{array} \right\}$$

$$R(t_3) = \frac{7}{5} + (5-2) + \frac{5-2}{2} = 5,9$$

O parâmetro u de uma permanente significa que ela está desvirada, enquanto t significa que está virada.

Em t_3, ambos os jogadores tem uma criatura (essencialmente igual, pois têm os mesmos atributos) a menos em relaçnao a t_2, mas o jogador defensor perdeu pontos de vida.

Figura E.2.4: estado t'_3

Suponhamos, por outro lado, que a ação escolhida pelo agente tenha sido

$$\{WalkingCorpse1 : GrizzlyBears2, WalkingCorpse2 : CentaurCourser\},$$

ou seja, bloquear "Grizzly Bears" com um "Walking Corpse" e "Centaur Courser" com o outro.

Representação do combate. O estado t'_3 refletirá as seguintes consequências:

- "Grizzly Bears" causou um número igual a seu poder (2) de dano ao "Walking Corpse" que o bloqueou.
- "Walking Corpse" causou um número igual a seu poder (2) de dano ao "Grizzly Bears" bloqueado.
- "Centaur Courser" causou um número igual a seu poder (3) de dano ao "Walking Corpse" que o bloqueou.
- "Walking Corpse" causou um número igual a seu poder (2) de dano ao "Centaur Corser" bloqueado.

Assim, "Grizzly Bears" e os dois "Walking Corpse" são destruídos (pois receberam dano maior ou igual às suas resistências) e temos o seguinte estado t'_3:

$$t'_3 = \left\{ \begin{array}{l} v_1 = 10, v_2 = 10, \\ G_1 = \{GrizzlyBears2\}, G_2 = \{WalkingCorpse1, WalkingCorpse2\}, \\ B_1 = \{GrizzlyBears1(u), CentaurCourser(t)\}, \\ B_2 = \emptyset. \end{array} \right\}$$

$$R(t'_3) = \frac{10-10}{5} + (5) + \frac{5}{2} = 7,5$$

Em t'_3, o total de pontos de vida não foi alterado, mas a balança do combate também parece negativa para o jogador defensor, que acabou sem criaturas na mesa. Note que o dano causado a "Centaur Courser" não foi suficiente para destruí-lo[6].

Figura E.2.5: estado t''_3

Por fim, suponhamos que a ação escolhida pelo agente tenha sido

$$\{WalkingCorpse1 : CentaurCourser, WalkingCorpse2 : CentaurCourser\},$$

ou seja, bloquear "Centaur Courser" com os dois "Walking Corpse".

Representação do combate. O estado t''_3 refletirá as seguintes consequências:

- "Centaur Courser" causou um número igual a seu poder (3) de dano aos "Walking Corpse" que o bloquearam.
- Os dois "Walking Corpse" causaram um número igual a seu poder (2 + 2) de dano ao "Centaur Courser" bloqueado.
- "Grizzly Bears" não foi bloqueado, causando um número igual a seu poder (2) de dano ao jogador defensor.
- "Walking Corpse" causou um número igual a seu poder (2) de dano ao "Centaur Courser" bloqueado.

[6]De acordo com as regras, o dano causado a uma criatura permanece até o final do turno. Se, porventura, algum dano fosse causado a "Centaur Courser" na segunda fase principal, a criatura também seria destruída.

Como "Centaur Courser" recebeu dano total (4) maior ou igual a sua resistência, foi destruído. Por outro lado, "Centaur Courser" causou no total (3) dano suficiente para destruir apenas um "Walking Corpse" (e danificar em 1 o outro). O estado t''_3, portanto, é configurado assim:

$$t''_3 = \begin{Bmatrix} v_1 = 10, v_2 = 10 - 2 = 8, \\ G_1 = \{CentaurCourser\}, G_2 = \{WalkingCorpse1\}, \\ B_1 = \{GrizzlyBears1(u), GrizzlyBears2(t)\}, \\ B_2 = \{WalkingCorpse2\}. \end{Bmatrix}$$

$$R(t''_3) = \frac{10-8}{5} + (4-2) + \frac{4-2}{2} = 3,4$$

Podemos perceber que, para o jogador defensor, a situação é estritamente melhor em t''_3 do que em t_3, pois $R(t''_3) > R(t_3)$ e, apesar de ter perdido a mesma criatura em ambos os casos, em t''_3 o jogador defensor conseguiu destruir uma criatura maior do oponente no processo. Além disso, apesar de não ser estritamente pior, pois não houve perda de vida, a situação em t'_3 parece também pior para o jogador defensor do que em t''_3, já que acabou perdendo suas duas criaturas.

4.2.4 Minimax na fase de Combate

O Combate se resume em 3 decisões alternadas entre os dois jogadores, começando e terminando pelo jogador ativo. Assim, o jogador ativo é o jogador MAX, pois pretende maximizar sua recompensa no turno, enquanto seu oponente é o jogador MIN (ambos descritos na seção 3.3), que deve se esforçar para diminuir a recompensa de MAX. Assim, temos uma árvore de profundidade 2. As decisões, em ordem, são:

1. Jogador Ativo (MAX): declara criaturas atacantes.
2. Jogador Defensor (MIN): declara sua configuração de criaturas bloqueadoras.
3. Jogador Ativo (MAX): determina a ordem que o dano será atribuido se mais de uma criatura bloquear uma de suas criaturas.

Assim, os **valores minimax** no nível 2 são as recompensas máximas de ordenação para cada configuração de bloqueio, enquanto no nível 1 são as recompensas mínimas assumindo ordenações com recompensa máxima para cada configuração de ataque e, finalmente, na raiz é a recompensa máxima dentre as recompensas mínimas por cada configuração de ataque.

4.2.5 Detalhes do algoritmo

A função `combatMinimax()` de um agente iteligente (`SearchAgent`) chama mais dois métodos parecidos (representando cada nível da árvore) e tem muitos detalhes, portanto é mais útil apresentá-la em pseudocódigo.

Quando um `SearchAgent` é o jogador ativo, chamará os três níveis do algoritmo encadeados para decidir sua configuração de ataque. Por outro lado, quando um `SearchAgent` é o jogador defensor, executa a busca Minimax na subárvore relativa apenas às configurações e ordenações de bloqueio.

4.2.6 Combinando estratégias

Para definir a recompensa no final de um turno, então, devemos combinar as três buscas (busca em largura na primeira Fase Principal, busca Minimax no Combate e busca em largura na segunda Fase Principal). Isso é feito naturalmente ao expandirmos a busca na primeira fase principal para o turno todo, incorporando o estado pós-combate determinado pela busca *Minimax*. Como o combate é influenciado pelas ações tomadas no primeiro turno, cada sequência de ações terá uma previsão de combate própria. O estado pós-combate é o começo da segunda Fase Principal e, partir dele, podemos continuar a busca em largura até o final do turno, quando avaliaremos a recompensa associada a cada sequência (total) de ações.

Minimax para Combate (estado s):

$a_{max} \leftarrow \emptyset$
recompensa máxima $r^1_{max} \leftarrow -\infty$
configurações de ataque $A^C_1 \leftarrow A(s)$
para cada ação de ataque $a \in A^C_1$:
    $s' \leftarrow$ aplica ação a no estado s
    recompensa mínima $r^2_{min} \leftarrow +\infty$
    configurações de bloqueio $A^C_2 \leftarrow A(s')$

    para cada ação de bloqueio $b \in A^C_2$:
        $s'' \leftarrow$ aplica ação b no estado s'
        recompensa máxima $r^3_{max} \leftarrow -\infty$
        ordenações de bloqueio $A^C_3 \leftarrow A(s'')$

        para cada ação de ordenação $c \in A^C_3$:
            $s''' \leftarrow$ aplica ação c no estado s''
            $r \leftarrow$ recompensa(s''')
            se $r > r^3_{max}$ **então:**
                $r^3_{max} \leftarrow r$

        se $r^3_{max} < r^2_{min}$ **então:**
            $r^2_{min} \leftarrow r^3_{max}$
            $b_{min} \leftarrow b$

    se $r^2_{min} > r^1_{max}$ **então:**
        $r^1_{max} \leftarrow r^2_{min}$
        $a_{max} \leftarrow a$

devolva a_{max}

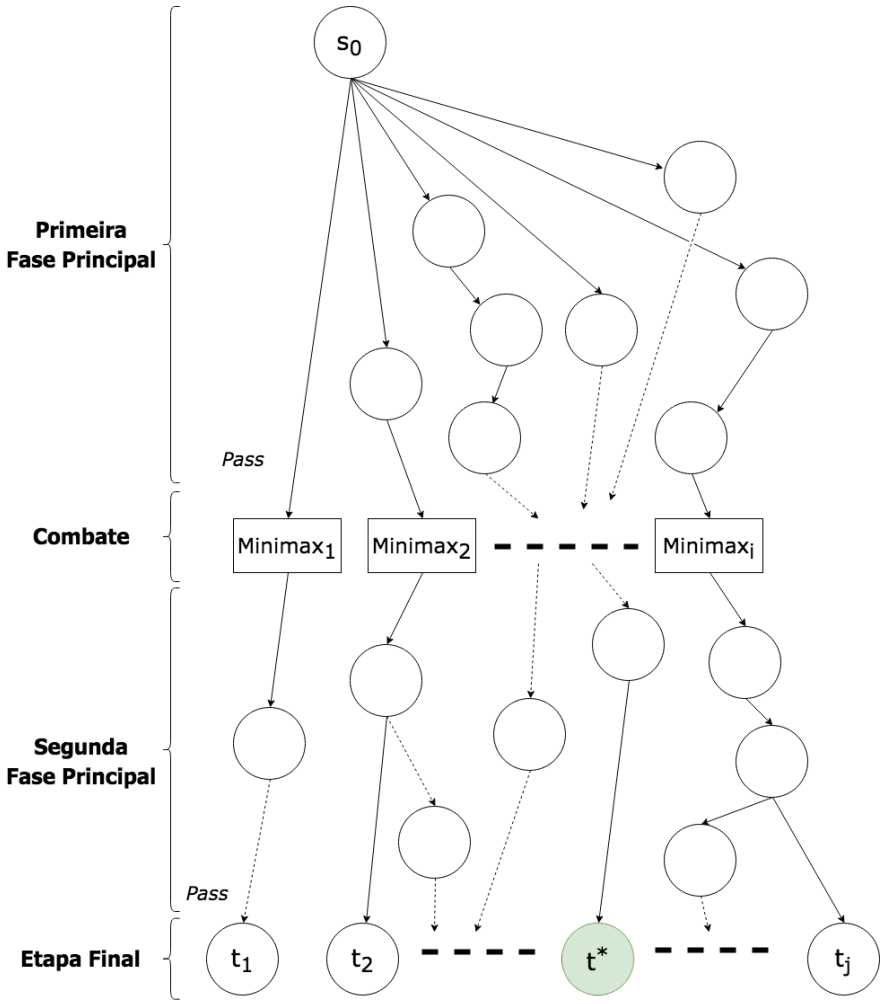

Figura 4.6: Diagrama representando espaço total de estados na busca realizada em um turno inteiro. t^* representa o estado terminal com recompensa máxima.

Como o agente pode prever incorretamente as escolhas de bloqueio de seu oponente (caso este não realize um *Minimax* ou avalie recompensas de um modo diferente, por exemplo), é possível que o estado no início da segunda Fase Principal seja diferente do previsto. Assim, a busca na segunda Fase Principal deve ser refeita para encontrar a "nova" melhor recompensa total do turno. Como isso ocorre quando o jogador defensor não escolhe sua "melhor" (segundo os critérios do agente) configuração de bloqueio, é de certa forma positivo para o agente, pois lhe dá a chance de encontrar sequência de ações com uma recompensa maior para o turno.

4.2.7 Detalhes da Implementação

O agente `SearchAgent`, quando jogador ativo, faz sua busca total em um turno no momento em que o objeto `Game` lhe "pergunta" qual será sua primeira ação na primeira Fase Principal. Assim, retorna sua sequência de ações – incluindo as ações da primeira Fase Principal, a configuração de ataque escolhida e as ações da segunda Fase Principal – determinada pelas buscas encadeadas. Examinaremos melhor como isso acontece.

O método `getChildren()` da classe `SearchAgent` (presente na figura 4.4), na verdade, pode, por sua vez, chamar dois métodos diferentes. Caso a fase do estado em questão seja alguma Fase Principal (`'First`

Main' ou 'Second Main'), é chamado um método homônimo, mas presente na classe State:

```
def getChildren(self):
    children = []
    if self.phase == 'First Main' or self.phase == 'Second Main':
        for action in self.game.getMainActions():
            game = self.game
            nextPhase = self.phase
            if action[0] != 'Pass':
                game = c.deepcopy(self.game)
                game.play(action)
                game.checkSBA()
            elif self.phase == 'First Main':
                nextPhase = 'Combat'
            elif self.phase == 'Second Main':
                nextPhase = 'End'
            children.append(State(game, nextPhase, self.actionPath + [action]))
    else:
        children = self.combatMinimax()
    return children
```

O método `getChildren()` *do estado*, então, encontra todas as ações legais de uma Fase Principal e, para cada uma, faz uma *cópia profunda*[7] do jogo. Por fim, "dentro da cópia" (pois não queremos alterar o jogo "real"), joga a ação. A partir dos resultados da ação, um novo estado é criado (com a cópia do jogo como referência) e com o caminho `actionPath` correto para se chegar no estado. Ainda, a atribuição da fase correta a ser examinada é feita.

Por outro lado, caso a fase do estado a ser expandido seja `'Combat'`, o método chamado é `combatMinimax()`. (como comentado, pouparemos o leitor de muitos detalhes do método). Assim como com `state.getChildren()`, `combatMinimax()` realiza **três laços de cópias** examinando todas as configurações de ataque, bloqueio e ordenação de bloqueio possíveis **para cada** início de combate colocado na fila da busca em largura.

A grande quantidade de cópias utilizada na busca total acabou sendo a maior causa de lentidão no programa, aliada à natureza exponencial do crescimento da árvore de buscas. Como a gama de configurações de combate cresce exponencialmente de acordo com o número de criaturas aptas a atacar ou bloquear (lembrando da equação 4.10), a árvore pode realmente crescer "fora de controle". Dito isso, o comportamento geral dos agentes mantém o número total de criaturas em campo não muito grande.

[7] Em Python, uma cópia profunda cria, além de um objeto novo, objetos novos idênticos para cada objeto pertencente ao que queremos copiar.

Capítulo 5

Resultados e Conclusões

Neste capítulo mostraremos e analisaremos os resultados das implementações e modelagens discutidas anteriormente. Além disso, também traremos nossas conclusões e considerações finais sobre o trabalho.

5.1 Mulligan

Para cada estado (i, j), a recompensa associada à ação *Keep* nas tabelas 5.2 e 5.4 e, à direita, o valor calculado Σ da soma de valores associada à ação *Mulligan*. As recompensas em vermelho indicam estados que a política extraída é *Mulligan*.

j	$r(j)$
0	-7
1	-3
2	3
3	4
4	2
5	-1
6	-4
7	-6

Tabela 5.1: recompensas-base para $j = 0, \ldots, 7$ "on the play"

$i \backslash j$	0	1	2	3	4	5	6	7	Σ
7	17,5	21,5	27,5	28,5	26,5	23,5	20,5	18,5	22,8395
6	14,0	18,0	24,0	25,0	23,0	20,0	17,0		18,6328
5	10,5	14,5	20,5	21,5	19,5	16,5			14,0315
4	7,0	11,0	17,0	18,0	16,0				8,8240
3	3,5	7,5	13,5	14,5					3,5119
2	0,0	4,0	10,0						-1,9
1	-3,5	0,5							-7,0
0	-7,0								–

Tabela 5.2: recompensas "on the play" de cada estado calculadas com $R_k(i, j) = r(j) + 3i$

j	$r(j)$
0	-4
1	0
2	5
3	6
4	3
5	0
6	-3
7	-5

Tabela 5.3: recompensas-base para $j = 0, \ldots, 7$

$i \backslash j$	0	1	2	3	4	5	6	7	Σ
7	20,5	24,5	29,5	30,5	27,5	24,5	21,5	19,5	24,7502
6	17,0	21,0	26,0	27,0	24,0	21,0	18,0		20,8419
5	13,5	17,5	22,5	23,5	20,5	17,5			16,4394
4	10,0	14,0	19,0	20,0	17,0				11,4721
3	6,5	10,5	15,5	16,5					6,3559
2	3,0	7,0	12,0						1,1
1	-0,5	3,5							-4,0
0	-4,0								–

Tabela 5.4: recompensas "on the draw" de cada estado calculadas com $R_k(i, j) = r(j) + 3i$

A política encontrada com a iteração de valor faz bastante sentido se levarmos em consideração a simplificação de limitar a informação das cartas na mão ao fato dela ser terreno ou não. Uma maneira de deixar a modelagem mais real seria levar em consideração o custo de mana dos cards não-terreno. Isso aumentaria enormemente a quantidade de estados possíveis (supondo que as cartas não-terreno do deck tenham custo de mana entre 1 e 5, uma mão com duas cartas teria $\sum_{i=1}^{6} i = 21$ representações possíveis de estados ao invés de 3, como acontece com a modelagem que escolhemos), mas essa modelagem ainda estaria desconsiderando informações relevantes na decisão do mulligan, como por exemplo se as cartas na mão inicial se encaixam na estratégia definida pelo jogador para combater o deck adversário. De uma maneira geral, estamos satisfeitos com os resultados obtidos, uma vez que o agente consegue mitigar a aleatoriedade tomando decisões que aumentam suas chances de ter um jogo justo.

5.2 Simulações do Jogo

Após rodar os experimentos várias vezes e analisar os resultados, conseguimos descrever o comportamento do agente dentro do jogo como agressivo: já que escolhemos uma implementação que leva em consideração apenas o turno atual, o agente não leva em conta o ataque do oponente no turno seguinte, e portanto escolhe os atacantes sem pensar em quais de suas criaturas seriam boas bloqueadoras para impedir que o oponente estabeleça uma vantagem ou até mesmo ganhe o jogo. Os decks usados para os testes serão chamados Branco e Vermelho, pelas cores de suas cartas.

Rodamos 70 testes com as configurações possíveis de agente inteligente ou aleatório controlando cada um dos decks, e obtivemos os seguintes resultados:

- **Aleatório x Aleatório**

  Para esta configuração, esperávamos que o deck Branco ganhasse mais do que o Vermelho, pois o deck Branco tem poucas cartas que possibilitam que o agente faça uma escolha *muito* ruim, como destruir sua própria criatura. Já o deck Vermelho tem cartas como Lava Axe (5.2), que possibilitam que o jogador acerte a si mesmo, ou Volcanic Hammer, que pode ser usado para destruir suas próprias criaturas.

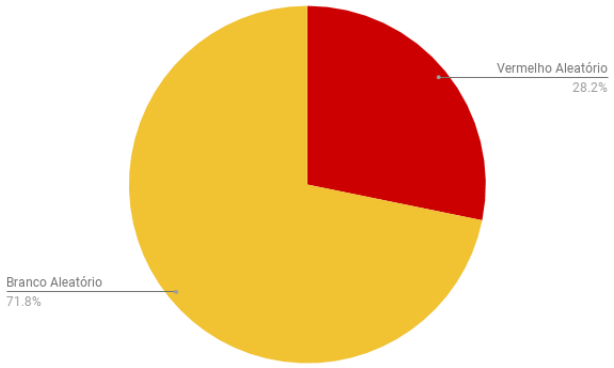

Figura 5.1: `python3 game.py -a1 random -a2 random`

  Após as simulações, o deck Branco ganhou $71,8\%$ das vezes, superando um pouco as nossas expectativas, mas num nível aceitável.

- **Aleatório x Inteligente**

  Nesta configuração, esperávamos que os agentes inteligentes ganhassem praticamente todas as partidas.

Figura 5.2: A carta Lava Axe (Machado de Lava) pode ter qualquer jogador, incluindo seu controlador, como alvo.

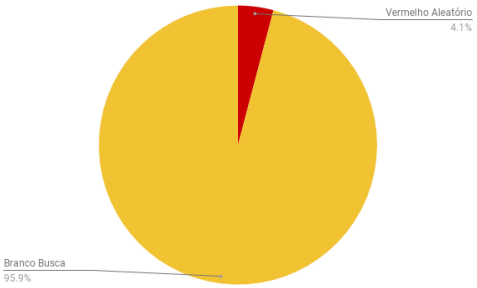

Figura 5.3: `python3 game.py -a1 random -a2 search`

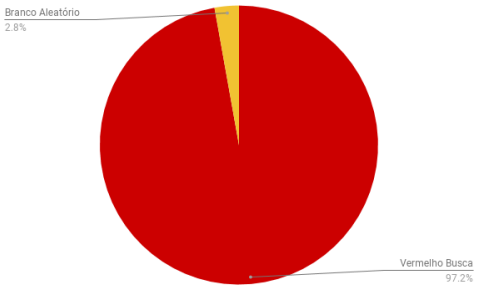

Figura 5.4: `python3 game.py -a1 search -a2 random`

Jogamos com as duas configurações possíveis, e nas duas o agente com o algoritmo de busca se saiu muito melhor que o oponente: $95,9\%$ quando no controle do deck Branco, e $97,2\%$ quando controlando o deck Vermelho. Novamente um resultado esperado, dado que o agente aleatório demora turnos para jogar seu primeiro terreno enquanto o agente inteligente joga terrenos sempre que possível, o que possibilita que jogue suas cartas muito antes.

- **Inteligente x Inteligente**

  Nesta configuração, esperávamos um desempenho um pouco melhor do deck Vermelho, uma vez que o comportamento do agente é agressivo graças a ele não olhar turnos a frente. Numa situação onde nenhum jogador está tentando preservar seu total de pontos de vida, quem tiver acesso a cartas como Lava Axe irá se beneficiar mais.

  O resultado obtido foi $76,5\%$ de vitórias para o agente com o deck vermelho, próximo do que esperávamos.

  A versatilidade providenciada por cartas que infligem dano também permitiu jogadas engenhosas da parte do agente Vermelho, como por exemplo a seguinte sequência de ações: atacar com Nest Robber (2/1) prevendo (corretamente) que o oponente o bloqueará com Siege Mastodon (3/5), o que causa com que Nest Robber seja destruído em combate. Na segunda Fase Principal, então, complementar o dano no Siege Mastodon (no momento com 2 de dano marcado) inimigo com Volcanic Hammer, causando um total de 5 de dano capaz de destruir a criatura. Como os dois agentes inteligentes decidem bloquear

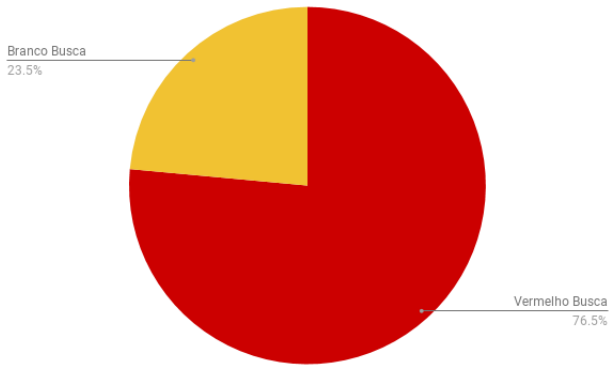

Figura 5.5: `python3 game.py -a1 search -a2 search`

baseando-se nas mesmas regras, as previsões sempre são corretas nessa categoria de testes.

5.3 Conclusões

O jogador inteligente implementado apresentou resultados interessantes de jogo e, apesar de podermos facilmente prever seu comportamento, as habilidades demonstradas são bastante valiosas em um jogo de Magic.

Um jogador (humano) por exemplo, deve balancear ganhos a curto prazo e ganhos potenciais a longo prazo, portanto definir as melhores jogadas a curto prazo faz parte do conjunto de habilidades de um bom jogador. Além disso, o jogador médio tem uma curva considerável de aprendizado para conseguir fazê-lo, enquanto esta habilidade se mostrou, de certa forma, "natural" para uma inteligência artificial, pois o algoritmo de busca local cumpre um bom trabalho em examinar e memorizar todas as possibilidades para decidir entre elas.

5.3.1 Escopo

Quando começamos o trabalho pretendíamos adicionar ao estudo o tratamento da informação incompleta (o que tornaria o problema parcialmente observável[1]), além de alguma implementação de aprendizado por reforço. Apesar de ambos terem sido descartados por uma questão de tempo, adicionar aprendizado por *Q-Learning* seria o próximo passo do projeto, pois poderíamos implementá-lo utilizando a busca local como base para as ações. Assim, uma ação para o problema de Q-Learning seria equivalente a um caminho de ações do problema da busca, e o agente de aprendizado por reforço poderia comparar dentre uma série de caminhos aquele com o maior Q-Valor, determinado dinamicamente pelo algoritmo de aprendizado. Assim, decisões a longo prazo poderiam ser contempladas. No que se refere à implementação do jogo de Magic, seria interessante continuar expandindo o problema para que contemplasse mais aspectos do jogo. Como por exemplo, a introdução de "Mágicas Instantâneas", cartas equivalente aos Feitiços, mas diferindo em que podem ser jogadas em um conjunto maior de momentos, incluindo turnos dos oponentes. O conjunto de regras de Magic: the Gathering é extremamente complexo, mas também igualmente robusto, o que possibilita definir suas mecˆanicas muito bem em uma implementação do jogo.

5.3.2 Apreciação e considerações finais

Implementar o nosso próprio cliente fez com que trabalhássemos com orientação a objetos em um projeto grande, o que não havíamos feito muito durante as disciplinas da graduação. O cliente facilitou a implementação dos agentes, uma vez que tínhamos conhecimento total da plataforma, mas por outro lado

[1] Em inglês, POMDP - "Partially Observable Markov Decision Process"

impossibilitou que testássemos os agentes corretamente até que o jogo estivesse funcionando corretamente, o que acabou se mostrando mais trabalhoso que o esperado. Ainda assim, fazer o cliente funcionar (e os algoritmos de busca também) se provou bastante satisfatório. Além disso, modelar os problemas de inteligência artificial do zero solidificou nossos conhecimentos e interesse na área, principalmente ao impor uma das maiores dificuldades do projeto: decidir quais modelagens e estratégias de resolução adotar. Como escolhemos um tema de nosso interesse pessoal (mais precisamente, um hobby), tivemos uma boa motivação para conseguir resultados (e ainda estudamos melhor as próprias nuances do jogo), mas por outro lado provou-se um desafio pois coube a nós decidir métricas e avaliar resultados.

Dentre as disciplinas cursadas no Bacharelado em Ciência da Computação relevantes para o desenvolvimento do projeto, destacamos:

- MAC0435 – Inteligência Artificial
- MAC0211/MAC0242 – Laboratório de Programação I/II
- MAC0441 – Programação Orientada a Objetos
- MAC0328 – Algoritmos em Grafos
- MAC0338 – Análise de Algoritmos

Agradecemos ao Professor Denis Deratani Mauá, que como supervisor ajudou inúmeras vezes a nortear o trabalho.

Bibliografia

[1] Andrew Schaefer - *Markov Decision Processes* - 2006 - http://egon.cheme.cmu.edu/ewo/docs/SchaeferMDP.pdf

[2] Jerônimo Pellegrini, Jacques Wainer - *Processos de Decisão de Markov: um tutorial* - 2007 - https://pdfs.semanticscholar.org/294f/694ae723a787f25043265fb3e660f62c573f.pdf

[3] Stuart Russel, Peter Norvig - *Artificial Intelligence: A Modern Approach (3rd Edition)* - 2009

[4] Wizards of the Coast - *Magic: The Gathering Comprehensive Rules* - 2018 - http://media.wizards.com/2018/downloads/MagicCompRules

[5] Paulo Feofillof - *Otimização Combinatória* - 2018 - https://www.ime.usp.br/~pf/otimizacao-combinatoria/mynotes/OtimizacaoCombinatoria.pdf