

Inteligência Artificial em *Magic: the Gathering*

Guilherme Souto Schützer

guischutzer@gmail.com

Tomás Marcondes Bezerra Paim

tomasbmp@gmail.com



IME-USP

Objetivo

Magic: The Gathering é um jogo de cartas criado em 1993 por Richard Garfield que introduziu o conceito moderno de *trading card game* (jogo de cartas colecionáveis). Com um acervo de mais de 15 mil cartas diferentes, os jogadores devem criar baralhos de 60 cartas (normalmente podendo haver repetição de até quatro cartas iguais) e competir, normalmente em jogos um contra um.

Durante o jogo, os jogadores têm que lidar com informações conhecidas (as cartas já jogadas previamente e as cartas em suas mãos) e informações desconhecidas (as cartas na mão de seu oponente e a ordem das cartas em seu baralho), o que faz com que seja praticamente impossível ter informação perfeita. Além disso, ambos os jogadores podem agir a praticamente qualquer momento dentro de um turno, o que adiciona uma camada a mais de complexidade.

Já existem inteligências artificiais feitas para jogar **Magic**, mas devido à complexidade do jogo, nenhuma inteligência artificial consegue ser realmente boa no jogo (por exemplo, elas não costumam levar em consideração as cartas que estão na sua mão, então é muito fácil fazer com que a IA sempre “morda sua isca”).

Nosso objetivo com o trabalho será entender o problema e criar a nossa versão de uma inteligência artificial para uma versão simplificada do jogo (limitando o conjunto de cartas disponíveis, adicionando restrições à construção dos baralhos e simplificando algumas regras do jogo).

Capítulo 1

Introdução

Inteligência artificial é um campo da ciéncia 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 a cada ação há uma grande quantidade de ações possíveis 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). Feitiços são cartas que têm um efeito que acontece no momento em que são jogadas e então são colocadas no cemitério (como é chamada a pilha de descarte no jogo).

Por exemplo, o feitiço Divinação tem um efeito simples: “Compre dois cards” (na versão em português do jogo, as cartas são referenciadas pela palavra em inglês *cards*). O jogador que joga esta carta pega as duas cartas do topo de seu baralho e as coloca em sua mão, aumentando o leque de possibilidades. Assim, **feitiços** podem alterar o estado do jogo de diversas maneiras (como fazer com que os jogadores comprem ou descartem cartas, alterar o total de pontos de vida de um jogador ou destruir uma criatura) e são a principal forma de interagir com o oponente ou desenvolver o seu lado do campo de batalha (como é chamada a zona do jogo em que ficam as criaturas e terrenos).

Criaturas são cartas permanentes (uma vez jogadas elas permanecem no campo de batalha até que sejam destruídas por algum feitiço ou durante o combate) que possuem poder (quantidade de dano causado em combate), resistência (quantidade de dano necessária para ser destruída) e muitas vezes habilidades que afetam o andamento do combate ou que fazem algum efeito no momento em que são jogadas.



Figura 1.1: Divinação (Feitiço)



Figura 1.2: Anjo da Misericórdia (Criatura)

Por exemplo, Anjo da Misericórdia tem a habilidade de Voar (que limita as interações do oponente durante o combate) e concede a seu controlador 3 pontos de vida ao entrar no campo de batalha. Além disso, seu poder e resistência são 3 e 3, respectivamente, conforme indicado na caixa no canto inferior direito.

Terrenos são a fonte de *mana*, que é o recurso utilizado para pagar por criaturas e feitiços. O custo de *mana* das cartas que não são terreno está indicado no canto superior direito da carta (por exemplo, para jogar Divinação é necessário usar três terrenos, sendo que um deles deve ser necessariamente uma Ilha, como na figura 1.3). Terrenos são, portanto, um dos tipos de cartas mais importantes, pois sem eles não há maneira (normalmente) de jogar suas outras cartas. Uma vez utilizado, um terreno se torna **virado** (em inglês, *tapped*): para jogar uma carta que não seja terreno, é necessário virar o número de terrenos equivalente ao seu custo. Uma vez virado, o terreno permanece virado até o começo do próximo turno de seu controlador, quando poderão ser utilizados novamente. Desta maneira, são necessários quatro terrenos para se jogar duas cartas custando duas manas cada durante o mesmo turno.

Uma parte importante do jogo é o sistema de *cores*. As cartas do jogo são divididas entre cinco cores de mana, com um terreno associado que produz mana desta cor: Branco (Planície), Azul (Ilha), Preto (Pântano), Vermelho (Montanha) e Verde (Floresta). Cada cor tem mecânicas de jogo únicas, fazendo com que jogadores usem mais de uma cor de mana em seus baralhos para ter acesso a tipos de efeitos diferentes, ao custo de utilizar terrenos variados, criando a possibilidade de não ter o terreno certo para jogar a carta desejada.

Para este trabalho utilizaremos apenas baralhos de uma única cor.

1.2 O jogo

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.



Figura 1.3: Ilha (Terreno)

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.
- **Combate:** Jogador ativo “declara atacantes” (escolhe quais de suas criaturas irão atacar seu oponente); em seguida, seu oponente “declara bloqueadores” (escolhe quais de suas criaturas irão bloquear as criaturas atacantes). Cada criatura não-bloqueada, então, causa dano igual ao seu poder ao oponente e todas as criaturas bloqueadas e bloqueadoras causam dano entre si.
- **Segunda Fase Principal:** Igual à primeira Fase Principal.

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.

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 conhecêsssemos em detalhe a plataforma onde o jogo estaria rodando, e para isso implementamos do zero uma versão do jogo com as especificidades 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 duas cartas: Martelo Vulcânico (Volcanic Hammer) e Anjo da Misericórdia (Angel of Mercy).



Figura 2.1: Martelo Vulcânico (Feitiço)



Figura 2.2: Anjo da Misericórdia (Criatura)

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”.

```

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)

```

- **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 das cores Branco, Azul, Preto, Vermelho e Verde, respectivamente. No caso de Volcanic Hammer, o custo é “1R” e no de Angel of Mercy é “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 (um terreno ser básico significa que um deck pode conter qualquer quantidade deste terreno, ao invés do limite normal de quatro cópias por carta). 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 é do tipo “Creature” e Volcanic Hammer é do tipo “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 é “Angel”.
- **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**,

```

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)

```

uma lista com as habilidades 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.

2.2 class Permanent:

2.3 class Player:

2.4 class Game:

Capítulo 3

Modelagem

Neste capítulo abordaremos um jogo de *Magic* como um problema de Inteligência Artificial, mas para explicarmos a modelagem do problema, é necessária a introdução do conceito de Processos de Decisão de Markov, que é o que usaremos para modelar os problemas do jogo.

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.

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 $p_j, j = 0, 1, 2, \dots$. Quando o final do mês chegar, o número de unidade vendidas será $x_t = \min(D_t, s_t + a_t)$, e o lucro será dado pela função $l(x_t)$. O número de unidades que começaram 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:

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

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, 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. A seguir veremos uma representação de Magic como MDP, e em seguida como extraímos a política para o processo.

3.2 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 na experiência própria, 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 3.2 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 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

$$P_i(j) = \frac{\binom{J}{j} \binom{60-J}{i-j}}{\binom{60}{i}}, \quad j = 0, \dots, i \quad (3.1)$$

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 3.2, 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

$$P_{i-1}(j') = \frac{\binom{J}{j'} \binom{60-J}{i-1-j'}}{\binom{60}{i-1}}, \quad j' = 0, \dots, i - 1$$

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

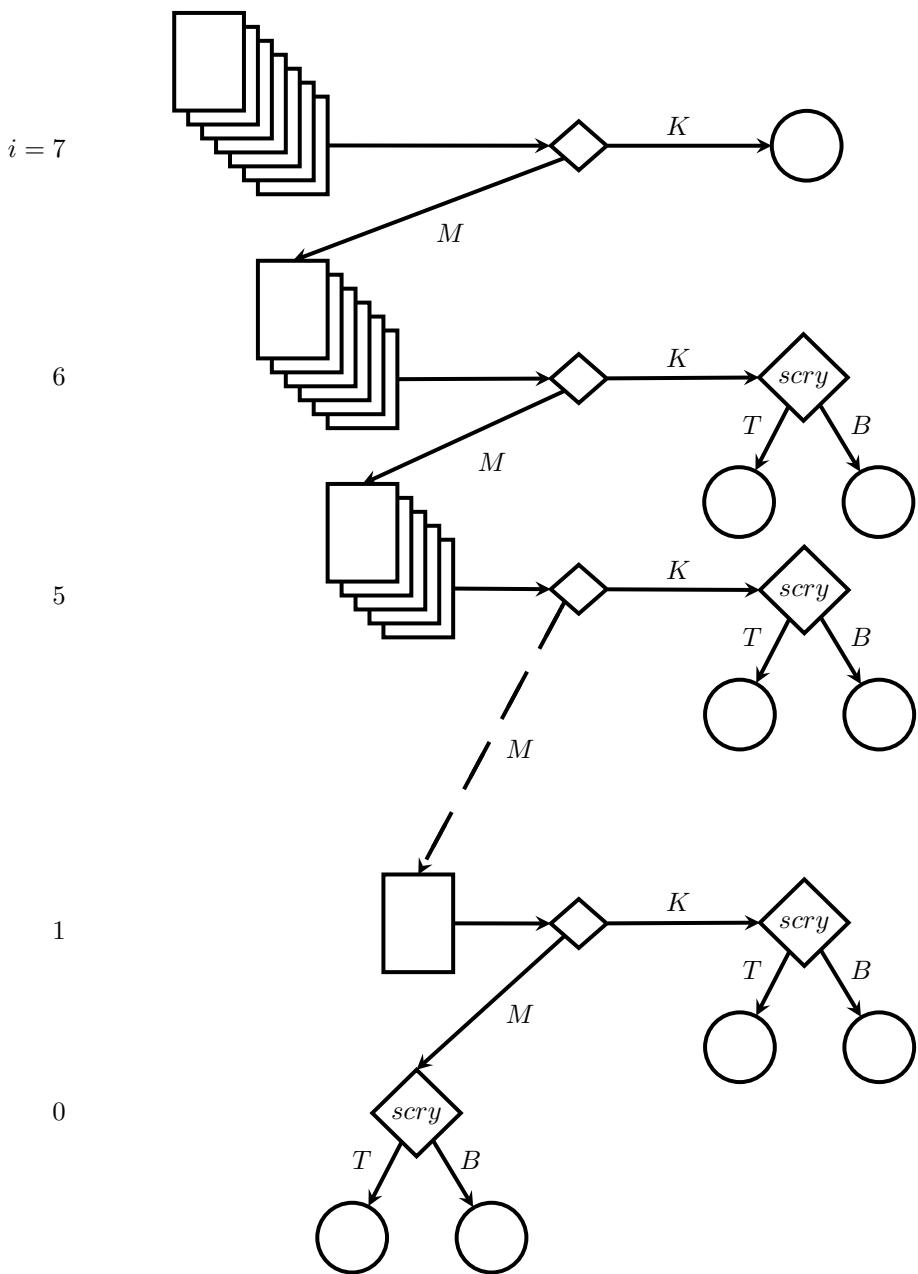


Figura 3.1: Representação visual de um problema de *mulligan*. Cada nível i representa o número de cartas na mão inicial do jogador. K significa a decisão de manter a mão, M realizar um *mulligan*, T deixar a carta no topo em um *scry* e B colocá-la no fundo.

3.2.1 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 intenção 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 3.1.

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

- $A(0, 0) := \{Keep\};$

$$P((i-1, j')|(i, j), Mulligan) = \mathcal{P}_{i-1}(j'), \quad j' \in \{0, 1, \dots, i-1\},$$

$$P(S_F|(i, j), Keep) = 1$$

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:

$$R_k(i_0, 3) > R_k(i_0, 2) > R_k(i_0, 4) > R_k(i_0, 5) > R_k(i_0, 1) > R_k(i_0, 6) > R_k(i_0, 7) \gg R_k(i_0, 0), \quad (3.2)$$

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 3.2. Para $\alpha = 3$ e $r(i)$ determinado pela tabela ??, podemos atribuir as recompensas de acordo com a tabela ??.

Assim, temos um MDP de horizonte finito (sempre acabará na época 9) e podemos determinar uma política ótima para a determinação da mão inicial.

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

Tabela 3.1: recompensas-base para $j = 0, \dots, 7$

Tabela 3.2: recompensas de cada estado calculadas com $R_k(i, j) = r(j) + 3i$

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

Tabela 3.3: recompensas-base para $j = 0, \dots, 7$

Tabela 3.4: recompensas de cada estado calculadas com $R_k(i, j) = r(j) + 3i$

3.2.2 Extração da Política

Com o MDP definido, desejamos agora extraír 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:

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

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 extraír a política para o mulligan, utilizamos o algoritmo de **iteração de valor**, modificado para atender as especificidades 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:

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

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).

3.2.3 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”.

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). Os valores V_0^* iniciais são iguais às recompensas $R_k(i, j)$ associadas às ações *Keep*, portanto, nas linhas 2-4 do código 3.2.3 temos a inicialização de $V^*(i, j)$ para cada estado (i, j) do problema. O horizonte é fixo em 8 épocas de decisão (correspondentes de 7 a 0 cartas na mão do agente), portanto há 8 iterações do trecho 6-12, que na k -ésima iteração calcula a soma $\sum_{j'=0}^{i-1} V_{k-1}^*(i-1, j') \times \mathcal{P}_{i-1}(j')$. Se a soma calculada - relacionada à ação *Mulligan* - é maior do que a recompensa $R_k(i, j)$, atualizamos a entrada i, j na tabela de valores com o valor da soma. Dessa maneira, é possível determinar a política ótima $\pi^*(i, j)$ em qualquer estado (i, j) no método `mulligan()` ao verificar se $V^*(i, j)$ é igual à recompensa $R_k(i, j)$ (caso não seja, $\pi^*(i, j) \leftarrow \text{Mulligan}$). O método `getKeepReward(i, j)` retorna a recompensa de acordo com os valores em `landRewards`, inicializada na criação do agente com os valores-base de recompensa determinados na tabela `??`. Por fim, `getMullProb(i, j)` retorna a probabilidade $\mathcal{P}_{i-1}(j')$.

¹O 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.

```

class MulliganAgent(Player):

    def __init__(self, ID, onThePlay, verbosity=False):
        self.name = 'Unknown Player'
        self.life = 20
        self.hand = []
        self.library = []
        self.lose = False
        self.ID = ID
        self.creatures = []
        self.lands = []
        self.active = False
        self.graveyard = []
        self.untappedLands = 0
        self.landRewards = []
        self.landDrop = False
        self.verbosity = verbosity
        self.onThePlay = 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]

        self.initMulliganTables()

    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): ...

```

3.3 O Jogo

Um jogo de *Magic: the Gathering* é um problema obviamente mais complexo do que o *Mulligan*, mas podemos modelá-lo também como um MDP, (na maioria dos casos², de horizonte finito) tratando também das jogadas do oponente e da informação incompleta presente em sua mão.

3.3.1 *Magic* como um MDP

Assim como detalhado em (3.1) e feito com o problema do *Mulligan*, precisamos definir os conjuntos S , A , P e R para caracterizar o problema como um MDP.

- S : 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

²Um jogo normal de *Magic* não tem um horizonte finito, pois há inúmeras maneiras de retardar indefinidamente o fim do baralho (como voltar cartas da pilha de descarte para o baralho). Neste trabalho, a versão simplificada do jogo com certeza terminará, no máximo, em 60 turnos, quando algum baralho acabar.

```

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
6     for epoch in range(0, 9):
7         for i in range(7, -1, -1):
8             for j in range(i + 1):
9                 mullValue = 0
10                for jLine in range(i):
11                    prob = self.getMulliganProb(i - 1, jLine)
12                    value = self.mulliganValue[7 - (i - 1)][jLine]
13                    mullValue += prob * value
14
15                if mullValue >= self.getKeepReward(i, j):
16                    self.mulliganValue[7 - i][j] = mullValue

```

Figura 3.2: Método de iteração de valor na classe `MulliganAgent`

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).

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

$$S := \left\{ \begin{array}{l} v_1, v_2 \in \mathbb{Z}, \\ act_1, act_2 \in \{0, 1\}, \\ fase \in \{\text{Principal}, \text{Combate}\} \\ H \subseteq \Omega, \\ h_1, h_2 \in \mathbb{N}, \\ d_1, d_2 \in \mathbb{N}, \\ G_1, G_2 \subseteq \Omega, \\ B_1, B_2 \subseteq \Omega^* \end{array} \right\}, \quad (3.3)$$

onde Ω é o conjunto de todas as cartas e Ω^* é o conjunto das chamadas *permanentes*, ou seja, cartas na mesa, cada uma com uma carta relacionada mais os atributos mencionados (virada ou desvirada, se pode ou não atacar e se está atacando ou bloqueando).

- A : Para definir as ações possíveis, devemos pensar na estrutura de um turno: *Primeira Fase Principal* - *Combate* - *Segunda Fase Principal*. As duas fases principais funcionam do mesmo jeito. Assim, examinaremos cada fase 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 passe) e pode ser atribuída aos seguintes subconjuntos, cada um com condições próprias:

$$A^P := \left\{ \begin{array}{ll} Land, & \text{se o jogador não jogou terreno neste turno,} \\ Creature(c), & \text{se o jogador possui } c \text{ terrenos desvirados na mesa,} \\ Sorcery(c, T), & \text{se o jogador possui } c \text{ terrenos desvirados na mesa e } T \text{ são alvos válidos,} \\ CreatWithEff(c, T), & \text{se o jogador possui } c \text{ terrenos desvirados na mesa e } T \text{ são alvos válidos,} \\ Pass, & \text{termina a fase (sempre possível).} \end{array} \right\}$$

Cada terreno só pode ser jogado se o jogador ainda não jogou um neste turno. Criaturas precisam do número de terrenos desvirados descrito em seu custo para serem jogadas. Feitiços, além do custo, precisam de alvos válidos para serem jogados (por exemplo, o feitiço “Flame Slash” lê, em português, “Golpe Ardente causa 4 pontos de dano à criatura alvo.”). Dessa maneira, este feitiço só pode ser jogado se existe pelo menos uma criatura em jogo. Se uma criatura tem um efeito desencadeado quando entra em campo, por sua vez, esta criatura pode ser jogada mesmo se não existem alvos válidos para seu efeito (neste caso, o efeito simplesmente não acontece). Por fim, a ação *Pass* termina a fase: se o jogador estiver na primeira fase principal, passa para o combate, se estiver na segunda, termina o turno. Toda ação $a \in A^P$ leva s a um estado s' com probabilidade $P(s'|s, a) = 1$.

O exemplo a seguir 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.

Figura E.1.1: estado s_1

$$s_1 = \{v_1 = v_2 = 10, h_1 = 4, h_2 = 0, B_2 = G_1 = G_2 = \emptyset, B_1, H\}$$



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



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 :

$$A^P(s_1) := \left\{ \begin{array}{l} \text{Mountain} \in \text{Land}, \\ \text{PensiveMinotaur} \in \text{Creature}(3), \\ \text{VolcanicHammer}(\text{self}) \in \text{Sorcery}(2, \text{self}), \\ \text{VolcanicHammer}(\text{opponent}) \in \text{Sorcery}(2, \text{opponent}), \\ \text{Pass}. \end{array} \right\} \quad (3.4)$$

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

$$s_2 = \{v_1 = v_2 = 10, h_1 = 3, h_2 = 0, B_2 = G_1 = G_2 = \emptyset, B_1, H\}$$



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 :

$$A^P(s_2) := \left\{ \begin{array}{l} \text{PensiveMinotaur} \in \text{Creature}(3), \\ \text{VolcanicHammer}(self) \in \text{Sorcery}(2, self), \\ \text{VolcanicHammer}(opponent) \in \text{Sorcery}(2, opponent), \\ \text{Pass}. \end{array} \right\} \quad (3.5)$$

“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

$$s_3 = \{v_1 = v_2 = 10, h_1 = 2, h_2 = 0, B_2 = G_1 = G_2 = \emptyset, B_1, H\}$$



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:

$$A^P(s_3) := \left\{ \begin{array}{l} \text{FlameSlash}(PensiveMinotaur} \in B_1) \in \text{Sorcery}(1, e \in B_1), \\ \text{Pass.} \end{array} \right\} \quad (3.6)$$

“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

$$s_4 = \{v_1 = v_2 = 10, h_1 = 2, h_2 = 0, B_2 = G_2 = \emptyset, B_1, H, G_1\}$$



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:

$$A^P(s_4) := \{Pass.\} \quad (3.7)$$

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.

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.³

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,

$$|A_1^C| = \sum_{i=0}^N \binom{N}{i} = 2^N, \quad (3.8)$$

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

$$|A_2^C| = (N' + 1)^M \leq (N + 1)^M, \quad (3.9)$$

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

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

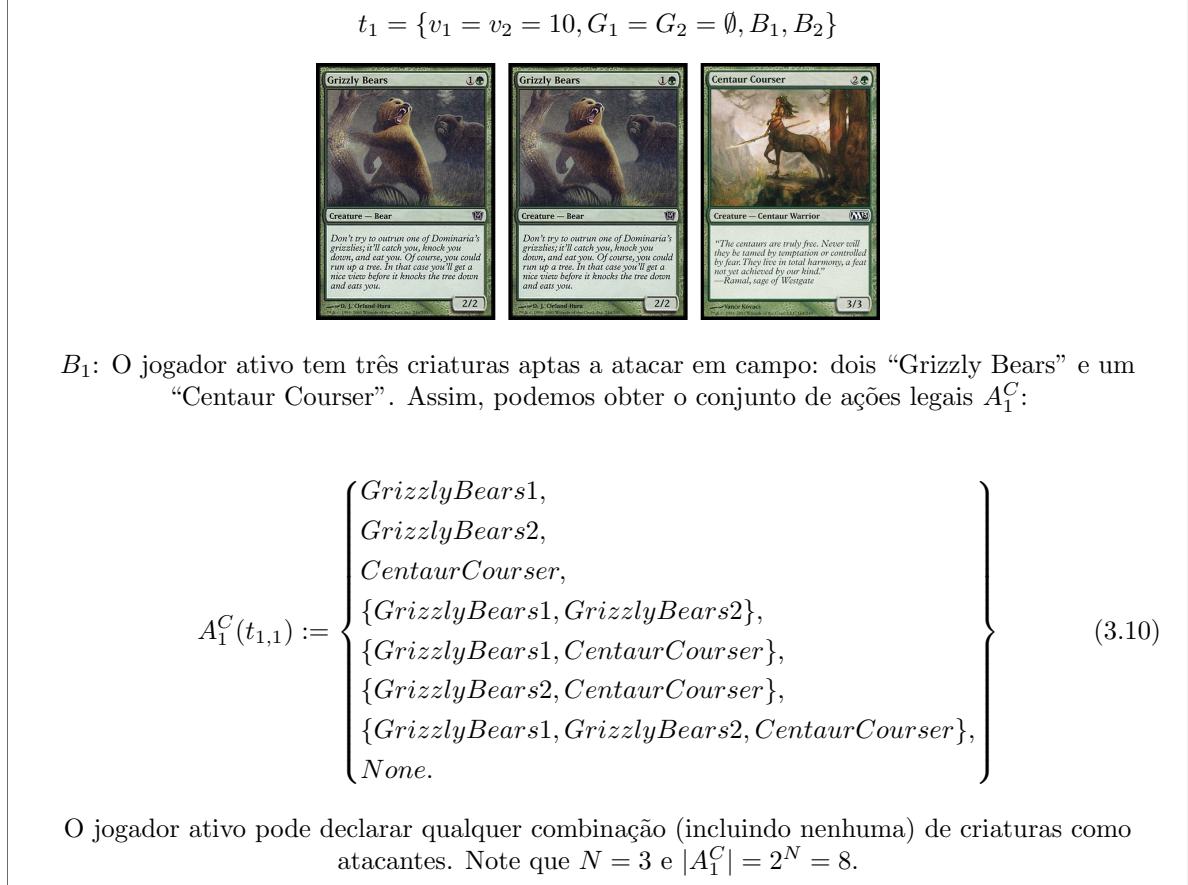
Nas próximas páginas demonstraremos por exemplos os estados e conjuntos de ações (de ambos jogadores) em uma fase de combate.

³Nesta 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



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

$$t_2 = \{v_1 = v_2 = 10, G_1 = G_2 = \emptyset, B_1, B_2\}$$



Grizzly Bears
Creature – Bear
Don't try to outrun one of Domnara's grizzlies! It'll catch you, knock you down, and eat you. Of course, you could run up a tree. In that case, you'll get a nice view before it knocks the tree down and eats you.



Grizzly Bears
Creature – Bear
The creatures are only free. Never will they be wild again. They have no taste for meat. They also make harmonious flute music, though it's not always a treat.

*B*₁: O jogador ativo tem um “Grizzly Bears” desvirado e um “Grizzly Bears” e um “Centaur Courser” virados, ambos atacando ($N' = 2$).



Walking Corpse
Creature – Zombie
“Feeding a normal army is a problem of logistics. Feeding zombies, it is an asset. Feeding is why they’re right. Feeding is why they’re feared.” —Jadair, ghoulcaller of Nephila



Walking Corpse
Creature – Zombie
“Feeding a normal army is a problem of logistics. Feeding zombies, it is an asset. Feeding is why they’re right. Feeding is why they’re feared.” —Jadair, ghoulcaller of Nephila

*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:

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

$$(3.11)$$

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 distintas⁴ e seus respectivos estados sucessores.

⁴Algumas 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 :

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

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 relação 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 :

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

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-lo⁵.

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” 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 “Centaur Corser” bloqueado.

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:

⁵De 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.

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

Podemos perceber que, para o jogador defensor, a situação é estritamente melhor em t''_3 do que em t_3 , pois $v_2(t''_3) > v_2(t_3)$ 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.

- R : Por fim, precisamos definir as recompensas por sair de cada estado. Uma boa definição inicial leva em conta os pontos de vida do agente e do oponente, assim como as “presenças de campo”, termo vago que dá a ideia de progresso no jogo. Escolhemos medir a “presença de campo” de cada jogador com o dano que suas criaturas podem causar, ou seja, a soma do poder de todas as suas criaturas. Assim, dados pontos de vida v_1, v_2 respectivos do agente e seu oponente e seus conjuntos de permanentes B_1, B_2 , temos

$$R(s'|s, a) = \begin{cases} -\gamma, & \text{caso } v_1 \leq 0, \\ +\gamma, & \text{caso } v_2 \leq 0, \\ v_1 + v_2 + \sum_{e \in B_1} \text{pow}(e) - \sum_{e \in B_2} \text{pow}(e), & \text{cc.} \end{cases} \quad (3.12)$$

onde γ deve ser uma recompensa para o agente suficientemente alta para incentivar a derrota do oponente e evitar a própria. A função $\text{pow}(e)$ retorna o poder da permanente e .

Podemos, então, calcular as recompensas de acordo com R por sair dos estados nos exemplos demonstrados por meio das ações escolhidas:

- $R(s_2|s_1, \text{Mountain}) = v_1 - v_2 = 10 - 10 = 0$.
- $R(s_3|s_2, \text{PensiveMinotaur}(3)) = v_1 - v_2 + \text{pow}(\text{PensiveMinotaur}) = 0 + 2 = 2$.
- $R(s_4|s_3, \text{FlameSlash}(1, \text{PensiveMinotaur})) = v_1 - v_2 + 0 = 0$.

Vamos examinar agora as recompensas no exemplo de combate tomando como agente o jogador defensor (portanto, as recompensas são contrárias - v_2 e B_2 são relativos aos agentes):

- $R(t_3|t_2, \{\text{WalkingCorpse1} : \text{GrizzlyBears2}, \text{WalkingCorpse2} : \text{None}\}) = v_2 - v_1 + \sum_e (\text{pow}(e) : e \in B_2) - \sum_e (\text{pow}(e) : e \in B_1) = 7 - 10 + 2 - 5 = -6$.
- $R(t'_3|t_2, \{\text{WalkingCorpse1} : \text{GrizzlyBears2}, \text{WalkingCorpse2} : \text{CentaurCourser}\}) = 10 - 10 + 0 - 3 = -3$.
- $R(t''_3|t_2, \{\text{WalkingCorpse1} : \text{CentaurCourser}, \text{WalkingCorpse2} : \text{CentaurCourser}\}) = 8 - 10 + 2 - 2 = -2$.

Estas recompensas refletem a noção de que o estado t''_3 é mais desejável para o jogador defensor do que os outros.

3.3.2 Algoritmos para Busca Local

3.3.3 Detalhes da Implementação

Capítulo 4

Conclusão

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>