Faculdade de Engenharia da Universidade do Porto

Implementação iterativa do algoritmo Counterfactual Regret Minimization

Nuno Velho

VERSÃO FINAL

Dissertação realizada no âmbito do
Mestrado Integrado em Engenharia Eletrotécnica e de Computadores
Major Automação

Orientador: Professor Eugénio Oliveira (PhD)
Co-Orientador: Luís Teófilo (Msc)

13-09-2014

U.PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

MIEEC - MESTRADO INTEGRADO EM ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES | **2013/2014**

A Dissertação intitulada

"Implementação Iterativa do Algoritmo Counterfactual Regret Minimization"

foi aprovada em provas realizadas em 10-10-2014

o júri

Presidente Professora Doutora Maria Teresa Magalhães da Silva Pinto de Andrade
Professora Auxiliar do Departamento de Engenharia Eletrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto

Professora Doutora Maria Benedita Campos Neves Malheiro
Professora Adjunto do Departamento de Engenharia Eletrotécnica do Instituto Superior de Engenharia do Porto

Professor Doutor Eugénio da Costa Oliveira
Professor Catedrático do Departamento de Engenharia Informática da Faculdade de Engenharia da Universidade do Porto

O autor declara que a presente dissertação (ou relatório de projeto) é da sua exclusiva autoria e foi escrita sem qualquer apoio externo não explicitamente autorizado. Os resultados, ideias, parágrafos, ou outros extratos tomados de ou inspirados em trabalhos de outros autores, e demais referências bibliográficas usadas, são corretamente citados.

Autor - Nuno José da Rocha Velho

Faculdade de Engenharia da Universidade do Porto

Resumo

Nos últimos anos, a Inteligência Artificial tem-se aliado à investigação em jogos com o objectivo de produzir novas formas de criação de estratégias. A área dos jogos é, por si só, extremamente importante pois representa uma das maiores indústrias do mundo da Informática. Os jogos possuem um domínio simples, no qual podem ser testadas novas abordagens de resolução de problemas, passíveis de serem posteriormente adaptadas a problemas da vida real. A investigação neste tema reside sobretudo na criação de agentes com estratégias inteligentes, capazes de ultrapassar os melhores jogadores humanos.

O *Counterfactual Regret Minimization* (CFR) é um algoritmo recursivo recorrentemente utilizado para criação de estratégias, capaz de gerar um conjunto de estratégias para 2 jogadores que estão em equilíbrio de *Nash*. Estas estratégias são depois utilizadas em jogo e, apesar de não maximizarem os ganhos, servem como óptimo ponto de partida para jogar contra adversários cuja estratégia é desconhecida.

No entanto, o CFR exige elevados recursos a nível de memória e tempo de processamento para garantir a convergência das estratégias para o referido equilíbrio. De forma a minimizar a utilização de recursos, foi desenvolvida uma implementação iterativa deste algoritmo, testada em jogos de *Poker* simples (*Kuhn e Leduc*). Os resultados obtidos demonstram uma melhoria no tempo de processamento na ordem dos 40% com um pequeno aumento da utilização de memória. Desta forma, a nova versão do CFR permitirá computar equilíbrios de *Nash* para jogos com espaço de pesquisa maior.

Abstract

In recent years artificial intelligence has been allied to the games research with the aim to produce new ways of creating strategies. The field of games is itself extremely important because it represents one of the largest industries in the world of informatics.
Games provide simple domains where new approaches to solve problems can be easily tested and these approaches can be subsequently adapted to other real life problems. Research in this subject is based mainly on the creation of intelligent agents with strategies able to overcome the best human players.

The Counterfactual Regret Minimization (CFR) is a recursive algorithm recurrently used to create strategies, capable of generating a set of Nash Equilibrium strategies for 2 players. These strategies are then used in the game by game playing agents; although these strategies not maximize gains, they serve as a great starting point to play against opponents whose strategy is unknown.

However, the CFR algorithm requires high-level memory resources and processing time to ensure the convergence of strategies for the Nash equilibrium. In order to minimize the use of resources, an iterative implementation of this algorithm was developed for simple *Poker* games (Kuhn and Leduc). The results show an improvement of the processing time in the order of 40% with a slightly memory usage increase. This way, the new version of the CFR will allow compute Nash equilibria for games with larger search space.

Agradecimentos

Esta dissertação não teria sido possível sem a ajuda e colaboração de muitas pessoas.

Em primeiro lugar, gostaria de agradecer ao meu Orientador, Msc Luís Teófilo, por toda a paciência que teve na supervisão deste trabalho e por todas as críticas contrutivas que foi apontando durante o decorrer do mesmo.

Em segundo lugar, gostaria de mencionar neste espaço e em exclusividade a minha prima Carla Rocha que sempre se mostrou interessada e disponível para me ajudar naquilo que precisasse.

Agradeço aos meus pais, à minha irmã e à minha namorada por todo o apoio que me deram e por não me terem deixado ir abaixo quando as tarefas me pareciam impossíveis.

Menciono também todos aqueles que estiveram por perto durante este período, desde amigos a familiares que, cada uma à sua maneira, foram importantes nesta etapa da minha vida.

Índice

Lista de figuras

Lista de tabelas

Abreviaturas e símbolos

Lista de abreviaturas

ACPC	*Anual Computer Poker Competition*
CFR	*Counterfactual Regret Minimization*
CFR-BR	*Counterfactual Regret Minimization - Best Response*
CPU	*Central Processing Unit*
CS	*Chance-Sampling*
DBBR	*Deviation Based Best Response*
DEEC	Departamento de Engenharia Eletrotécnica e de Computadores
E[HS]	*Expected Hands Strength*
FEUP	Faculdade de Engenharia da Universidade do Porto
GPU	*Graphics Process Unit*
IA	Inteligência Artificial
MCCFR	*Monte Carlo Counterfactual Regret Minimization*
OPCS	*Opponent-Public Chance Sampling*
PCS	*Public Chance Sampling*
RAM	*Random Access Memory*
RPS	*Rock-Paper-Scissors game*
SPCS	*Self-Play Chance Sampling*
2NL	*2 players No-Limit Texas Hold'em*

Capítulo 1

Introdução

1.1 - Contexto

Atualmente, a investigação em jogos é uma área de elevado interesse para a inteligência artificial, existindo um número variado de jogos onde a aplicar, que podem ser úteis na resolução de outros problemas dos mais diversos domínios. Os jogos podem ser classificados em dois tipos fundamentais: jogos de informação completa e jogos de informação incompleta, sendo estes últimos o objeto de interesse desta dissertação.

Um jogo de informação incompleta pode ser definido como um jogo entre dois ou mais participantes, no qual pelo menos um não tem acesso a todos os dados referentes ao jogo (ex.: as cartas dos adversários num jogo de *poker*), ou seja, os participantes não têm conhecimento total do estado do jogo, tornando-o desta forma aleatório. Este tipo de jogos são um desafio atual e significativo para a inteligência artificial e existem alguns algoritmos para os resolver e criar estratégias.

No entanto, o melhor algoritmo atual utilizado para a criação de estratégias de *poker* é pesado em termos de recursos computacionais. Existe contudo uma solução que consiste na utilização de um par de estratégias de equilíbrio de *Nash*. O algoritmo atual para obter essas estratégias nos jogos sequenciais é o *Counterfactual Regret Minimization* (CFR). Porém, apresenta problemas pois é lento (elevados tempos de processamento) e necessita de uma elevada memória computacional, tornando-se complicado aplicar o referido algoritmo em jogos cujo o espaço de pesquisa seja elevado. De modo a diminuir o impacto destes efeitos negativos, foi desenvolvida uma versão iterativa do CFR.

1.2 - Motivação

A aplicação do algoritmo CFR iterativo é a motivação capital desta dissertação. O CFR tem uma importância fundamental na criação de estratégias que convergem para o equilíbrio de

Nash, resolvendo jogos de grande espaço de pesquisa. É o melhor algoritmo atual na criação de estratégias robustas baseadas na minimização do arrependimento entre dois jogadores.

No entanto, sendo este um algoritmo exigente em termos de recursos computacionais, se lhe for reduzido o tempo de processamento e a memória necessária, pode ser aplicado mais facilmente a jogos mais complexos. O objectivo passa por reduzir o nível de abstração (ou seja, a redução do espaço de pesquisa do jogo) e, consequentemente, a diminuição da complexidade do jogo (ex.: *Texas Hold'em Poker*).

Uma vez que é desconhecido qualquer trabalho sobre uma aplicação do CFR iterativo, espera-se que esta dissertação possa contribuir na procura de uma nova solução para melhorar o desempenho deste algoritmo, um dos mais populares na criação de estratégias de jogos deste tipo.

1.3 - Objetivos e hipótese

Nesta dissertação pretende-se verificar se uma implementação iterativa do algoritmo CFR melhora o desempenho em termos de complexidade temporal e espacial, comparativamente à sua versão recursiva.

Os objetivos deste trabalho são a implementação do CFR, a implementação do CFR iterativo e a comparação entre as duas versões deste algoritmo.

1.4 - Estrutura da dissertação

O presente documento encontra-se estruturado em 7 capítulos. Segue a descrição resumida do conteúdo dos restantes capítulos:

- Capitulo 2 (Conceitos fundamentais): neste capítulo são apresentadas definições sobre teoria de jogos e métodos de resolução dos mesmos. São também apresentados exemplos de jogos simples tais como o Dilema do prisioneiro, Pedra-Papel-Tesoura e algumas variantes do Poker.

- Capitulo 3 (Revisão Bibliográfica): este capítulo faz uma abordagem bibliográfica de elementos chave deste trabalho, como jogos e algoritmos, teoria dos jogos e *poker*, CFR e modelação de oponentes, baseada em estudos recentes.

- Capitulo 4 (Trabalho realizado): neste capítulo é apresentado o algoritmo desenvolvido ao longo desta dissertação.
- Capitulo 5 (Resultados): neste capítulo são mostrados os resultados obtidos, comparando dados entre a aplicação do CFR recursivo e iterativo.

- Capitulo 6 (Conclusões e trabalho futuro): neste capítulo é feita uma análise final ao trabalho realizado e são apresentadas propostas de melhorias para proximas investigação.

Capítulo 2

Conceitos fundamentais

Com o objectivo de expor alguma informação sobre os temas tratados nesta dissertação, é realizada, numa primeira fase, a análise do*s conceitos fundamentais* dos mesmos. Inicialmente é apresentada a Teoria dos Jogos e, posteriormente, são apresentados os jogos abordados.

2.1 Teoria de jogos

"A Teoria dos Jogos é então o ramo da matemática aplicada que estuda situações estratégicas onde os jogadores escolhem diferentes ações para melhorar o seu retorno[1]*"*. Analisando o livro [1] verificam-se diferentes formas de representar jogos de informação completa, onde cada jogador conhece todos os movimentos que já ocorreram e os que poderão ocorrer a seguir bem como o estado do jogo. Este livro mostra formas de representar e definir jogos assim como formas para os resolver.

O conceito de Equilíbrio de *Nash* é definido como uma combinação de estratégias em que a estratégia de cada jogador é a melhor possível para ele próprio, tendo em conta a estratégia escolhida pelo adversário. Assim, o resultado do jogo é satisfatório para ambos os jogadores, razão pela qual nenhum deles tem argumentos para alterar a estratégia pela qual optou. O Equilíbrio de *Nash* verifica-se independentemente do facto de os jogadores terem ou não uma estratégia dominante[2].

[1] A enciclopédia livre. Wikipédia, "Teoria dos Jogos," 2013. [Online]. Available: http://pt.wikipedia.org/wiki/Teoria_dos_jogos.

2.1.1 Representação de jogos

2.1.1.1 Jogo na forma normal

Nesta vertente, as instâncias do jogo são representadas por tuplos que contêm os jogadores, conjuntos de estratégias e preferências dos jogadores. Tendo em conta esta definição, um jogo é composto por [1]:

- Um número finito de jogadores $N = \{1,2,\ldots n\}$;
- Cada jogador $i \in N$ escolhe ações do conjunto de ações Si;
- Os ganhos são definidos pelos perfis de estratégias que consistem em todas as estratégias escolhidas pelos diferentes jogadores;
- Os jogadores têm influência nos ganhos; esta influência é normalmente definida por uma função de *payoff* (resultado/prémio).

Tabela 1 - Exemplo de um jogo na forma normal (retirado de: A enciclopédia livre. Wikipédia, "Teoria dos Jogos,")

		Jogador 2	
		Esquerda	Direita
Jogador 1	Cima	4,3	-1,-1
	Baixo	0,0	3,4

2.1.1.2 Jogo na forma extensiva

Neste tipo de representação de jogos são utilizadas árvores de decisão. Esta árvore é representada por nós e ramos (ou ligações), onde os nós representam estados de jogos, os ramos representam as ações possíveis de um dado jogador e as folhas da árvore representam a pontuação do jogo. Os nós podem ser terminais (onde se definem quanto cada jogador ganha/perde), ou podem ser nós de decisão (onde é feito um teste para apurar o vencedor). Este tipo de jogos é representado da seguinte forma [1]:

- Um número finito de jogadores $N = \{1,2,3,\ldots,n\}$;
- Um conjunto de sequências ou histórias do jogo, onde uma sequência não pode ser uma sub-história de outra sequência.
- Uma função que ajusta um jogador a cada subsequência de cada história final.
- Para cada jogador $i \in N$, a influência nos ganhos está acima do conjunto de histórias finais.

Na Figura 1 está representado um exemplo de um jogo na forma extensiva, onde as letras dos ramos representam as ações dos jogadores que serão adicionadas à história do jogo a medida que este evolui.

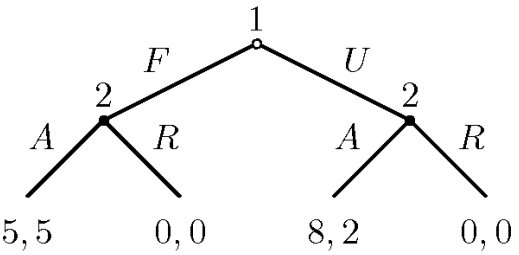

Figura 1 - Exemplo de um jogo na forma Extensiva (retirado de: A enciclopédia livre. Wikipédia, "Teoria dos Jogos,")

2.1.2 Tipos de jogos

É possível classificar os jogos mediante variados parâmetros. As classificações mais utilizadas são as seguintes:

- Simétricos e Assimétricos;
- Soma Zero e Soma Diferente de Zero;
- Não Sequenciais e Sequenciais;
- Informação completa ou incompleta.

Em termos de definições, um jogo é considerado simétrico se os pagamentos/resultados (recompensas que os jogadores recebem a cada possível resultado do jogo derivando de decisões estratégicas) finais apenas dependerem da estratégia escolhida; são considerados assimétricos no caso inverso.

Os jogos de Soma Zero são jogos em que a soma dos resultados de todos os jogadores é igual a zero, isto é, no caso de jogos entre dois participantes o vencedor ganha tanto quanto o jogador vencido perdeu.

Jogos não sequenciais ou simultâneos são aqueles onde os jogadores se movem simultaneamente; os jogadores jogam ao mesmo tempo, sem terem consciência do que cada um fez.

Jogos sequenciais são jogos em que os jogadores jogam um a seguir ao outro, sendo que excluindo o iniciador do jogo, os outros possuem alguma informação, isto é, conhecendo o histórico de movimentos dos adversários. Assim sendo, é este o tipo de jogo estudado nesta tese.

Os jogos não sequenciais são tipicamente representados na forma normal (onde se presume que cada jogador atue simultaneamente, ou, pelo menos, sem conhecer a ação dos outros); já os sequenciais são representados na forma extensiva (os jogadores têm alguma informação acerca das escolhas dos outros jogadores).

Jogos de informação completa definem-se quando todas as jogadas forem conhecidas por cada um dos jogadores envolvidos (todos os participantes conhecem todas as jogadas

efectuadas). Desta forma, os jogos de informação completa são simultaneamente sequenciais.

2.1.3 Exemplos de jogos

Um dos conceitos fundamentais desta dissertação é o equilíbrio de *Nash* e, para o podermos integrar de forma eficaz, é necessário abordar alguns jogos exemplo como é o caso do jogo do Dilema do Prisioneiro e do jogo Pedra-Papel-Tesoura.

2.1.3.1 Dilema do Prisioneiro

Este jogo é muito popular entre os investigadores pois, para além de ser muito simples, ainda não existem soluções perfeitas para a sua resolução. O jogo inicia-se da seguinte forma: existem dois suspeitos sob custódia policial e, como a polícia não tem provas suficientes para condenar ambos, separa-os, colocando-os em salas diferentes, e oferece-lhes o mesmo acordo [3]:

- Se um dos prisioneiros confessar (trair o outro) e o outro permanecer em silêncio, o indivíduo que confessou sai livre enquanto que o outro cumpre dez anos de prisão.
- Se ambos ficarem em silêncio (colaborarem um com o outro), a polícia só pode condená-los a um ano de prisão cada um.
- Se ambos confessarem (traírem-se mutuamente), cada um apanha cinco anos de prisão.

Este problema seria simples de resolver se os suspeitos conhecessem a decisão que cada um iria tomar; como estão separados e as ações são tomadas simultaneamente, este problema torna-se num dilema, sendo classificado como um jogo de informação incompleta. O objetivo de cada suspeito é ficar preso o menor tempo possível; para isso terá de escolher uma das duas alternativas, sendo o resultado dessa escolha dependente da decisão do outro prisioneiro. Para exemplificar, atribuíram-se letras aos suspeitos (suspeito A e suspeito B).

Do ponto de vista do suspeito A, se o suspeito B escolher colaborar e o A escolher também colaborar, apanha um ano de prisão; mas se escolher trair, sai em liberdade. Então, neste caso, a melhor opção para o suspeito A será trair. Se o suspeito B escolher trair, e se o suspeito A escolher colaborar, apanha dez anos de prisão. Por outro lado, se escolher trair apanha cinco anos de prisão. Neste caso, a melhor opção é de novo trair. Ou seja, independentemente da decisão do suspeito B, para o suspeito A, trair é a melhor opção (como já foi acima referido). Do ponto de vista do suspeito B, a mesma coisa acontece. Desta forma, conclui-se que a opção trair-trair é a opção mais segura para os dois prisioneiros, uma vez que representa a opção de equilíbrio (Equilíbrio de *Nash*). Ao escolherem desviar-se desta estratégia, sabendo que o adversário não o faz, sairão prejudicados.

Um dos grandes problemas deste dilema é que a melhor opção não é trair-trair, mas sim se ambos os suspeitos colaborarem, pois só ficariam presos um ano cada um. O problema desta ação é que é necessária confiança mútua pois, se um indivíduo colaborar e o outro trair, o indivíduo que trai sairá livre e o outro ficará preso dez anos. Desta forma, o Equilíbrio de *Nash* é representado pelas ações trair-trair, que são as que trazem mais regalias para ambos os suspeitos. Na Tabela 2 encontra-se uma representação na forma normal do jogo.

Tabela 2 - Representação do Dilema do Prisioneiro

		Prisioneiro B	
		Colabora	Trai
Prisioneiro A	Colabora	Ambos condenados a 1 ano	A Condenado a 10 anos; B sai Livre
	Trai	A Sai Livre; B condenado a 10 anos	Ambos condenados a 5 anos

2.1.3.2 Pedra Papel Tesoura

Outro jogo simples que mostra igualmente a força do Equilíbrio de *Nash* é o RPS (*Rock-Paper-Scissor*, em português, Pedra-Papel-Tesoura). É um jogo de dois jogadores onde cada um escolhe a sua aposta entre pedra, papel e tesoura. Esta escolha é feita simultaneamente entre os dois jogadores, e o vencedor é conhecido usando as regras de que a tesoura bate o papel, o papel bate a pedra e a pedra bate a tesoura (Figura 2). Se ambos os jogadores apostarem o mesmo objeto, considera-se que houve um empate. A habilidade neste jogo vem de um jogador conhecer a estratégia do adversário; esta estratégia seria a aposta aleatória. Assim, neste simples jogo existe um equilíbrio de *Nash* que visa escolher aleatoriamente entre as três possíveis apostas. Se um jogador inexperiente se desvia desta estratégia, um oponente forte pode jogar melhor ao descobrir e explorar a nova estratégia. No entanto, se o jogador inexperiente nunca se desvia desta estratégia, não poderá ser observado pelo jogador forte e este não poderá prever a ação ele irá tomar. Desta forma, o jogador inexperiente garante um empate (em vez de perder o jogo) aplicando os princípios básicos do Equilíbrio de *Nash*.

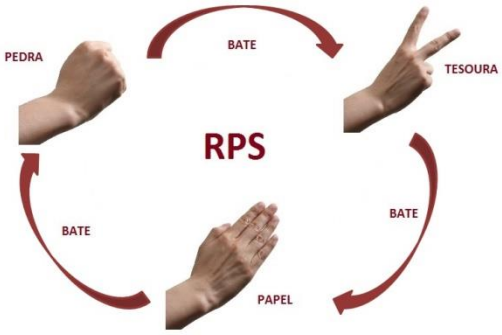

Figura 2 - Exemplo do jogo Pedra-Papel-Tesoura

2.2 Poker

O *Poker* é um dos jogos mais interessantes na investigação em jogos e na inteligência artificial por ser um jogo de informação incompleta e onde vários agentes competem precisando conhecer diversas probabilidades. No campo de estudo desta dissertação este é um dos jogos mais complexos ao qual se pode aplicar o CFR (*Counterfactual Regret Minimization*), algoritmo central neste trabalho, detalhadamente explicitado no capítulo 3. Existem vários tipos de *Poker*, mas o mais conhecido e mais jogado mundialmente é o Texas *Hold'em*. Nesta dissertação são utilizados o *Kuhn Poker* e o *Leduc Poker*, que são variantes mais simples.

2.2.1 Kuhn Poker

O *Kuhn Poker* é um jogo considerado simples comparativamente ao *Texas Hold'em*. É um jogo de soma zero com dois jogadores. Existe um baralho de três cartas, geralmente um Rei, uma Dama e um Valete. Os jogadores apostam às cegas (uma aposta forçada antes dos jogadores terem acesso às cartas chamada de *blinds*) e recebem uma carta cada um. A partir daí têm duas ações possíveis para o desenrolar o jogo: passar ou apostar. Se os dois jogadores passarem consecutivamente, é verificada a carta maior de entre as duas que os jogadores possuem e determina-se o vencedor, realizando-se o *pay-off* (resultado/prémio); acontece o mesmo se ambos apostarem consecutivamente. Como se trata de um jogo bastante curto e com um número reduzido de conjuntos de informação, o *Kuhn Poker* pode ser representado numa árvore binária.

A árvore apresentada na Figura 3 mostra todos os estados de um jogo de *Kuhn Poker*. Pode-se desta forma verificar que existem quatro estados de decisão, onde o jogador em questão escolhe o que fazer a seguir (se passar ou apostar), e cinco estados de resultados, onde após terminado o jogo, se determina quem ganha e quem perde e quanto recebe/perde cada jogador.

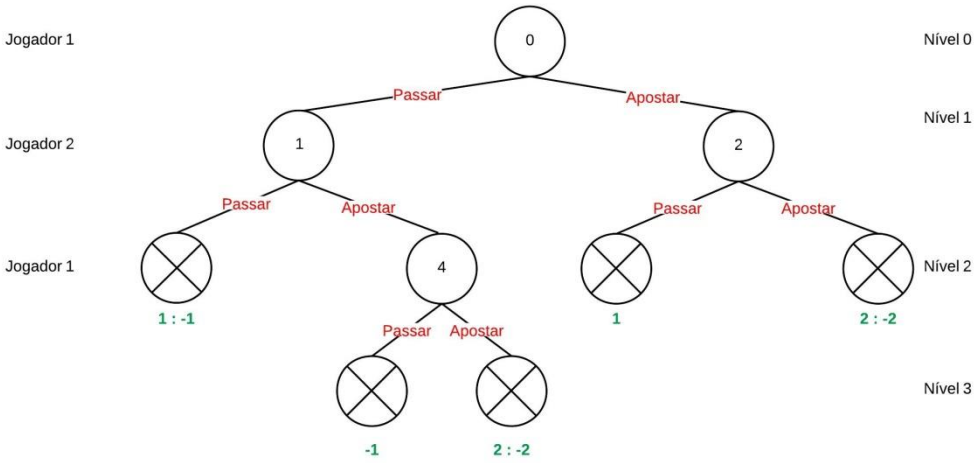

Figura 3 - Árvore de jogo do Kuhn Poker

2.2.2 Leduc Poker

No intuito de encontrar outro jogo, um pouco mais complexo, para verificar se a construção de um algoritmo CFR iterativo traz vantagens para jogos mais complexos, o *Leduc Poker* também foi matéria de estudo. Este jogo, que possui as mesmas características do *Kuhn Poker*, é jogado também por dois jogadores, existindo neste caso duas rondas de apostas. Neste jogo, o baralho é constituído por seis cartas, sendo estas pares de naipes diferentes (exemplo: Valete espadas e Valete ouros, Rainha espadas e Rainha ouros, Rei espadas e Rei ouros). O jogo começa com os jogadores a apostarem às cegas e, em seguida, é dada uma carta a cada jogador, iniciando-se a primeira ronda de apostas; após serem feitas as apostas e tomadas as ações, se ninguém tiver desistido, é revelada a carta seguinte do topo do baralho, uma espécie de "*mini-flop*", comparando com o *Texas Hold'em*. Desta forma, inicia-se a segunda ronda de apostas. Tendo uma carta na mesa e uma na mão, os jogadores poderão ganhar o jogo com a carta mais alta, como no Kuhn Poker, ou fazendo par entre a sua carta e a que está na mesa. Uma particularidade deste jogo é que poderá ocorrer um empate caso os jogadores possuam a mesma carta.

Para jogos como o *Texas Hold'em* é inviável representar a árvore de jogo, visto ser um jogo muito extenso com milhões de conjuntos de informação (cerca de $3{,}194 \times 10^{14}$) [2]. Para

[2] M. Johanson,(2013) Measuring the size of large no-limit poker games.

o *Leduc* ainda é possível e, na Figura 4, está representada a árvore de jogo completa. Comparando-a com o do *Kuhn Poker*, esta apresenta 7 níveis de jogo, do 0 ao 6, e é importante salientar que os nós 3, 6 e 10 representam uma nova ronda de apostas, ou seja, é apresentada a carta da mesa.

Assim sendo, e segundo a classificação de jogos abordada neste capítulo, o Kuhn poker e Leduc podem ser classificados como jogos simétricos de soma zero, e jogos sequenciais de informação incompleta, que se representam na forma extensiva.

Neste capítulo, foi feita uma análise detalhada à Teoria de jogos e definidos enumeros conceitos desta área. Foram também apresentados jogos exemplo que serão utilizados ao longo da dissertação.

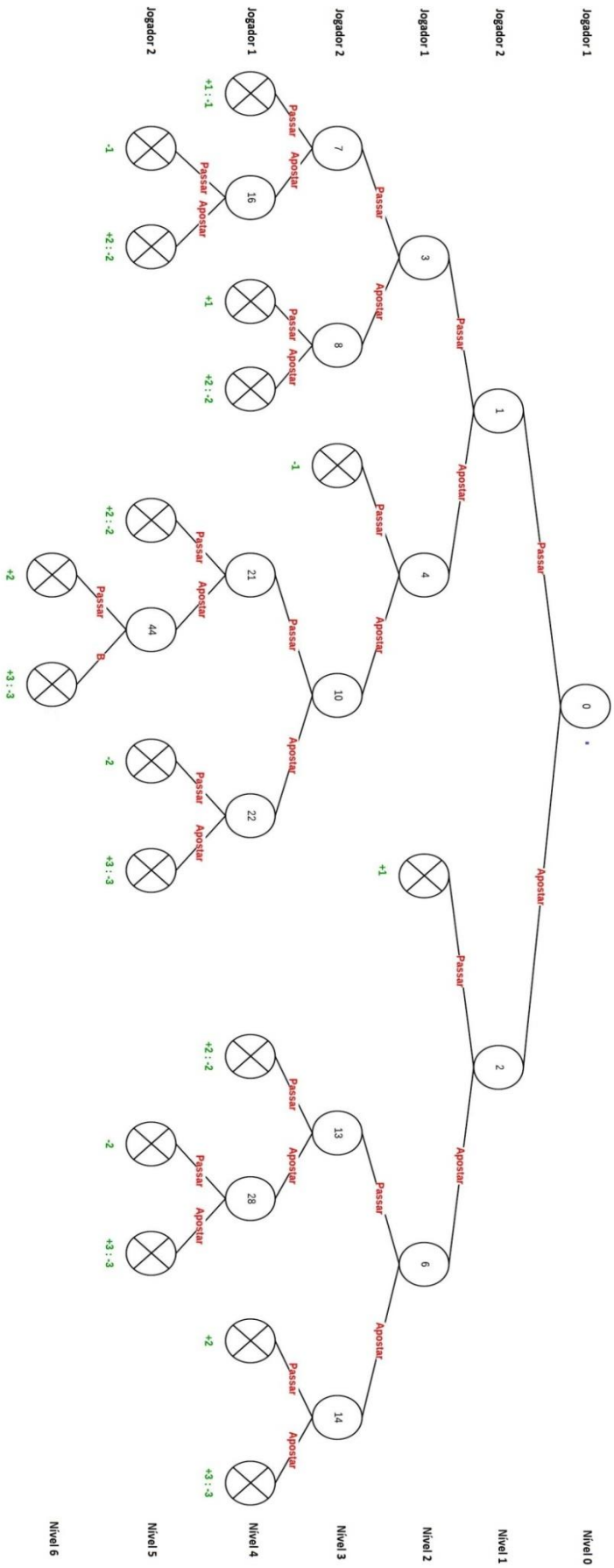

Figura 4 - Árvore de jogo do Leduc Poker

Capítulo 3

Revisão bibliográfica

No sentido de ir de em encontro aos objetivos apresentados, e para completer o capítulo anterior, foi realizada uma revisão bibliográfica no ambito do tema desta dissertação, onde o CFR é o elemento central.

3.1 Jogos e algoritmos

Segundo o artigo [4], a investigação em jogos tem sido uma das áreas de interesse da Inteligência Artificial. Os progressos observados relativamente aos jogos de informação completa, levaram a IA a vencer humanos considerados *experts* internacionais.

No entanto, a investigação em jogos de informação incompleta é considerada como um desafio maior, devido à complexidade introduzida pela informação desconhecida.

Neste estudo, os autores mencionam o *OOS* (*Online Outcome Sampling*) como sendo o primeiro algoritmo de busca de informação incompleta, em que é garantida a convergência para um equilíbrio de estratégias (em jogos de dois participantes e de soma zero). O *OOS* é, assim, descrito como um algoritmo capaz de evitar problemas comuns noutros algoritmos de criação de estratégias, como, abstrações da árvore de jogo.

Este mesmo estudo sugere ainda que, ao contrário do *ISMCTS* (*Information Set Monte Carlo Tree Search*), a exploração das estratégias produzidas pelo *OOS* diminui, e o tempo de procura aumenta. Na prática, o OOS tem o mesmo desempenho do *ISMCTS* em situações de jogo com dois participantes, produzindo no entanto estratégias com menor possibilidade de exploração, durante o mesmo tempo de busca.

Sendo o estudo de jogos de *Poker* um ponto central neste trabalho, convém reforçar a importância das técnicas de pesquisa em árvores de jogo, muito aplicadas nesta área. Nos últimos anos, tem sido esta a abordagem mais comum para criar um agente de *Poker* robusto. Têm-se registado progressos no que diz respeito ao uso de algoritmos de equilíbrio de aproximação para programar um equilíbrio próximo do ideal, num contexto de jogo abstrato. Este conceitos referem-se a uma abordagem *offline*, uma vez que requerem a concepção de um domínio específico de abstrações.

Segundo o artigo [5], é reforçada uma vez mais a importância da investigação na teoria dos jogos desde o início da era computorizada. A teoria despertou a atenção não só da ciência da computação, que a vem utilizando em avanços na inteligência artificial e cibernética, mas também das ciências políticas, ciências militares, ética, economia, filosofia, biologia e jornalismo.

Os resultados da teoria podem ser aplicados a jogos de entretenimento, mas também a aspectos significativos da vida real, da vida em sociedade, sendo o dilema do prisioneiro um bom exemplo.

Quando se faz referência à teoria dos jogos, mencionar o seu mais célebre co-fundador, John von Neumann (um dos matemáticos mais importantes do século XX), torna-se incontornável. A sua obra publicada em 1944 *Theory of Games and Economic Behavior*, em parceria com o economista Oscar Morgenstern, contém a teoria matemática dos jogos baseada na teoria dos jogos de estratégia.

Ainda de acordo com o artigo [5], os jogos matemáticos de estratégia têm sido os mais cobiçados para estudo e, geralmente, apresentam, parcial ou totalmente, as seguintes propriedades do Poker:

- regras bem definidas e lógica simples;
- estratégias complexas;
- objectivos específicos bem definidos;
- resultados mensuráveis.

Para além da fundação da teoria dos jogos de John von Neumann, os aspetos estratégicos do *Poker* não foram detalhadamente estudados pelas ciências da computação antes de 1992. Este jogo possui vários atributos que o tornam um bom território para o estudo de novos desafios. No que diz respeito à estrutura matemática subjacente e à taxonomia dos jogos, são salientadas as seguintes propriedades:

- é um jogo de informação incompleta;
- tem resultados estocásticos;
- as informações ocultas são parcialmente observáveis;
- é um jogo competitivo entre dois ou mais jogadores.

O mesmo estudo refere que a informação incompleta e estocástica, com observação parcial, está entre os maiores desafios teóricos da ciência da computação.

3.1.1 TEOREMA DO MAX-MIN

Segundo [6], John von Neumann provou a existência do teorema do max-min em 1928; o teorema estabelece que na matriz dos jogos de soma zero, com informação completa, existe um par de estratégias para ambos os jogadores que mostra a cada um deles como minimizar as suas perdas máximas (daí o nome "max-min"). Ao examinar todas as estratégias executáveis, o jogador deve considerar todas as respostas possíveis do adversário, apostando na estratégia que resultará na minimização da sua perda máxima. O matemático mostrou-nos que estes "max-minis" eram iguais , mas contrários, entre os dois jogadores.
O teorema foi melhorado pelo autor e alargado aos jogos de informação incompleta com mais de dois jogadores; o resultado foi publicado em 1944 em *Theory of Games and Economic Behavior*.

3.1.2 PROBLEMA NP-COMPLETO

Caracteriza-se pelo subconjunto dos problemas de decisão em NP (tempo polinomial não determinístico). Todo o problema em NP se pode reduzir, com uma redução de tempo polinomial, a um dos problemas NP-completo. Este tipo de problemas é considerado o mais difícil de NP. Se fosse encontrada uma forma de resolver rapidamente qualquer problema NP-Completo (em tempo polinomial), então poderiam ser utilizados algoritmos para resolver todos os problemas NP rapidamente.

3.2 Counterfactual Regret Minimization

De acordo com [7], um estudo realizado na Universidade de Alberta, instituição de referência no âmbito da investigação em jogos, evidencia uma nova técnica na resolução de jogos extensos, baseando-se na diminuição do arrependimento. Desta forma, deu-se origem ao conceito de arrependimento contra-factual, que, ao ser minimizado, diminui também o arrependimento geral, a partir do qual podem ser criadas estratégias que atinjam o equilíbrio de *Nash*. Este artigo apresenta o conceito de minimização do arrependimento contra-factual, conceito base desta dissertação, e demonstra esta técnica aplicada a jogos extensos como o *Texas Hold'em Poker* com resultados promissores no que diz respeito a estados de jogo calculados.

Em suma, o conceito de arrependimento contra-factual é definido para conjuntos de informação individuais, que serão iterativamente minimizadas para cada um dos conjuntos de informações. A soma dos arrependimentos contra-factuais dá origem ao arrependimento total.

Ainda com base nesta referência, os autores evidenciam o aparecimento, no ano 2000, de um importante algoritmo da teoria dos jogos baseado no conceito de minimização do arrependimento. Os jogadores alcançam o equilíbrio jogando através da detecção do arrependimento de jogadas passadas e construindo jogadas futuras proporcionais a arrependimentos positivos. Esta técnica é descrita como tendo suscitado uma revolução em versões de jogos de computador de alguns dos mais difíceis jogos de *bluff*.

Os mesmos autores salientam ainda que, sendo este conceito relativamente recente, existe ainda assim algum material curricular disponível para a introdução de algoritmos baseados no arrependimento. No entanto, este material é ainda considerado, segundo os autores, modesto.

Neste algoritmo, a estratégia utilizada é a análise do arrependimento, ou seja, o cálculo da diferença entre a melhor utilidade que poderia ter ocorrido e a utilidade real obtida. Assim, arrependimento pode ser sucintamente definido como a diferença entre o que o jogador ganhou e aquilo que poderia ter ganho.

Ainda de acordo com com a mesma referência, a minimização do arrependimento pode ser alargada a jogos sequenciais, onde os jogadores devem jogar segundo uma sequência de ações para atingir uma fase de jogo terminal (como é o caso do *Kuhn Poker*).

Atendendo ao artigo [8], o CFR é definido como uma técnica recente de dois jogadores e de soma zero que calcula estratégias convergentes para o equilíbrio de Nash. Uma estratégia de equilíbrio de Nash é útil em jogos de dois participantes desde que maximize a sua utilidade numa situação de jogo com o pior adversário. No entanto, em jogos com vários participantes (três ou mais) todas as evidências teóricas deixam de ser válidas. Desta forma, o CFR não garante convergência para equilíbrio de Nash para mais de dois jogadores. Apesar deste facto, os autores reforçam a ideia de que agentes criados a partir do CFR podem ter um bom desempenho em jogos de três participantes (foi o primeiro trabalho de investigação a apresentar resultados de bom desempenho com o CFR em jogos de *multiplayers*).

Assim, e com base neste estudo, afirma-se que o CFR foi criado para calcular estratégias que convergem para o equilíbrio de *Nash* em jogos com dois participantes, de soma zero de informação completa. No entanto, foi demonstrado que o CFR tem potencial para calcular estratégias vencedoras em situações de jogos com três participantes e de soma zero, utilizando estratégias de informação completa e incompleta.

No artigo [9], os autores definem uma família de variantes do algoritmo *Monte-Carlo Counterfactual Regret Minimization* (MCCFR) para cálculo do equilíbrio aproximado em jogos extensivos que difere na forma de representar a árvore de jogo a cada iteração. Os autores salientam ainda que o MCCFR executa as mesmas atualizações a nível de arrependimento que o CFR.

Neste estudo, introduziram-se dois membros adicionais às variantes do algoritmo: o *outcome-sampling*, onde apenas uma única jogada é exposta em cada iteração, e o *external-sampling*, que mostra a possibilidade de um nó da árvore de jogo assim como as ações do adversário correspondente a esse mesmo nó.

O número de iterações para calcular o *external-sampling* é ampliado por um fator constante, mas, apesar disso, alcança uma diminuição do custo de tempo por iteração, o que resulta numa melhoria temporal no cálculo do equilíbrio. Uma vez que com o *outcome-sampling* não é necessário o conhecimento total da estratégia do adversário (para além de amostras de jogo derivadas da estratégia), os autores mostram como este pode ser utilizado na minimização do arrependimento *online*.

No estudo, os autores apresentaram evidências de que ambos (*outcome-sampling* e *external-sampling*) têm alta probabilidade de limitar o arrependimento geral. Assim, um equilíbrio aproximado pode ser programado utilizando *self-play*. Provam ainda a existência de um limite menor em termos de arrependimento para o algoritmo do CFR e relacionam este novo limite com os do MCCFR.

Ainda de acordo com os mesmos autores, Zinkevich em colaboração com colegas, apresentou a "*vanilla form*", uma forma que requer a árvore de jogo completa para que esta possa ser transposta em cada iteração (sendo possível evitar a pesquisa total da árvore). A mesma equipa aborda também uma variante do CFR específica para o *Poker* em que as amostras possibilitam o aparecimento de resultados em cada uma das iterações. No entanto, os autores concluem que o *external-sampling* demonstra uma melhoria no tempo de cálculo comparativamente ao *Vanilla CFR*.

Os autores demonstram empiricamente e em domínios diferentes que a diminuição temporal nas iterações supera o aumento de iterações necessárias, conduzindo a uma convergência mais rápida. É ainda mencionada uma desvantagem do CFR considerada relevante, que é o facto de necessitar que a política do adversário seja conhecida, sendo este um factor de inconveniência para a minimização do arrependimento *online* num contexto de jogo extensivo.

Os autores do artigo [10] apresentam uma variante do CFR designada por *Chance-Sampled* (CS), que apresenta amostras de um conjunto de resultados para uma determinada iteração, atravessando apenas a porção da árvore de jogo que lhe corresponde. Comparativamente ao algoritmo de base, este procedimento resulta numa mais rápida, mas menos precisa, atualização de estratégias. Em jogos extensos, a drástica redução temporal por iteração supera o aumento do número de iterações exigidos na convergência para uma estratégia ideal.

No mesmo artigo, é estudada a aplicação de uma nova árvore de jogo ao CFR, que resulta em três novas variantes: *Self-Public Chance Sampling* (SPCS), *Opponent-Public Chance Sampling* (OPCS) e *Public Chance Sampling* (PCS). Com estas variantes, um pequeno número de iterações lentas, cada uma delas atualizando um grande número de conjuntos de informação, produz atualizações precisas da estratégia, utilizando valores calculados. Ou seja, estas variantes operam de forma mais lenta, mas, ao mesmo tempo, de forma mais eficaz e com iterações precisas. Os autores provam a convergência destas novas técnicas e

demonstram empiricamente que a PCS converge mais rapidamente para um equilíbrio do que a CS, nomeadamente no *Poker*.

Os autores do artigo [11] definem o CFR-BR como uma forma modificada do CFR, para utilização em jogos abstratos. Em vez de se utilizar *self-play*, o CFR-BR utiliza o algoritmo do CFR na abstração para um jogador, enquanto o outro jogador é substituído por um oponente que joga a melhor resposta possível. Enquanto o CFR converge para um equilíbrio num contexto de jogo abstrato, o CFR-BR converge para a estratégia menos explorável (no jogo completo) de entre todas as estratégias possíveis.

O artigo [12] apresenta pela primeira vez o CFR-BR, introduzindo-o como a primeira técnica para superar os desafios da abstração em jogos de dois participantes e soma zero; demonstram ainda a eficiência do algoritmo no domínio do *Texas Hold'em Poker* com dois participantes, onde foi usado para criar aproximações à informação desconhecida da estratégia do outro jogador, conduzindo à diminuição da utilização da memória.

No artigo [13] é colocada a hipótese de acelerar o cálculo das abstrações para acelerar também o cálculo do CFR para o *Poker*. Tratando-se de um jogo muito complexo, a probabilidade de atingir ao Equilibrio de *Nash* sem utilizar abstração é reduzida. A técnica utilizada, consiste no tratamento de um conjunto de mãos de cartas semelhantes, isto é utilizando o E[HS] (*expected hands strength*). O objetivo seria acelerar este processo utilizando o *Average rank strength*, um método que utiliza tabelas de jogo já calculadas para acelerar o E[HS]. Este método apresentou melhorias evidentes em relação ao MCCFR (aproximadamente o triplo da rapidez).

3.3 Modelação de oponentes

Os autores do estudo [14], salientam o facto de os jogos de informação incompleta serem muito menos estudados comparativamente aos jogos de informação completa, onde já se registou muito progresso nos últimos anos. Realçam ainda a importância da utilização de algoritmos para desenvolver modelos de oponentes em jogos de informação incompleta, destacando esta problemática como uma das áreas de investigação mais importantes em *machine learning*. Este mesmo estudo sugere que a modelação de um oponente que se adapta rapidamente às mudanças nas condições do jogo é possível e efetiva.

De acordo com o artigo [15], foi desenvolvido um algoritmo para a modelação de oponentes em formas extensas de jogos de informação incompleta. Este algoritmo (*DBBR – Deviation-Based Best Response*) baseia-se na observação da frequência de ações do oponente, construindo um modelo de oponente através da combinação de informação de uma estratégia de equilíbrio pré-calculado, e de observações. Desta forma, o algoritmo determina a melhor resposta para o modelo de oponente em questão. Neste exemplo, tanto o modelo de oponente como as melhores respostas são atualizados continuamente e em tempo real.

Através deste estudo, os autores concluem que o *DBBR* é um algoritmo híbrido, pois combina raciocínio teórico de jogo com modelação de oponentes pura, e pode explorar oponentes após um número reduzido de interações. O DBBR é descrito como um algoritmo eficaz e efetivo na prática.

De acordo com [16], nalguns casos, a modelação de oponentes é considerada uma das partes mais importantes do processo de jogo. O *Poker* é um bom exemplo explicativo desta afirmação, pois existe um *bluffer* que finge ter uma boa mão com más cartas, e onde os outros participantes devem ser capazes de avaliar o estilo do adversário (sendo ele *bluffe*r ou não), o mais rápido possível, para poder estar apto a tomar as melhores decisões em seguida. Analogicamente, em jogos de estratégia que decorrem em tempo real, os jogadores mais experientes utilizam os seus próprios recursos individuais para tentar identificar as estratégias dos adversários o mais precocemente possível. Ultimamente, tem-se demonstrado interesse em imitar esta característica humana.

Os autores formularam a modelação de um oponente baseando-se naquilo a que chamaram "problema de engenharia reversa". Assim, o oponente tem um modelo oculto para a tomada de decisões, e o jogador pode apenas observar as jogadas que este estabelece. Para estudar o modelo oculto, o jogador pode interagir com o oponente através de jogos, mas o número de interações deve ser limitado. O jogador não está autorizado a jogar um número demasiado elevado de partidas contra o seu oponente, de forma a impossibilitá-lo de reconhecer a sua estratégia. Neste estudo, é proposta a utilização do algoritmo da "engenharia reversa", no sentido de conferir um mecanismo de controlo interno em agentes robóticos.

Neste capitúlo, foram abordados temas relacionados com o CFR, foi definido e explicado o algoritmo em questão e feita a revisão a alguns trabalhos relacionados, com especial foco em algumas variantes dos CFR.

Capítulo 4

Implementação

Ao longo deste processo, foram criadas estratégias para jogos aplicando o algoritmo *counterfactual regret minimization*. Os jogos para os quais foram criadas estratégias são jogos simples com números reduzidos de conjuntos de informação. Utilizaram-se tanto o algoritmo recursivo como o iterativo para comparar as estratégias criadas por ambos. Os jogos para os quais foram realizadas as estratégias são o Kuhn e o Leduc Poker.

4.1 CFR recursivo

Para calcular a estratégia para o *Kuhn* e *Leduc Poker* utilizando o CFR foi feito um treino (conjunto de jogos para calcular uma estratégia média que convirja para o Equilíbrio de *Nash*) em C# baseado num tutorial existente *online* [7] que utiliza o CFR como algoritmo recursivo.

A aplicação inicia-se com a definição de algumas variáveis, começando por atribuir o valor 0 e 1 às ações *PASS* e *BET* respetivamente e contabilizando quantas ações existem durante um jogo (*PASS* e *BET*). É criado também o gerador de números aleatórios, que será utilizado para baralhar as cartas, e é iniciada uma árvore binária que será indexada com vetores contendo os conjuntos de informação de cada nó do jogo; em seguida, a árvore é completada com uma classe Nó onde constarão todos os valores relativos ao conjunto de informação.

<Definição de variáveis Globais>

```
PASS ← 0, BET ← 1; NUM_AÇÕES ← 2;
Random;
HashMap ← <ConjuntosInformacao, Node>
```

A classe Nó contém atributos onde são guardados os valores para cada conjunto de informação. A classe contém igualmente dois métodos: um para o cálculo da estratégia e outro para o cálculo da estratégia média. A diferença entre as duas funções é que a primeira calcula a estratégia instantânea para cada conjunto de informação durante o jogo e a segunda calcula a estratégia média total para cada conjunto de informação no final de todos os jogos, utilizando os valores guardados na classe Nó.

<Classe Nó>

<Atributos>

```
ConjuntoInformação;
SomaArrependimento[NUM_AÇÕES];//Vetor que armazena a soma dos
arrependimentos de um dado nó
Estratégia[NUM_AÇÕES];//vetor que armazena a estratégia de um dado nó
SomaEstratégias [NUM_AÇÕES];//vetor que armazena a soma dos valores da
Estratégia[];
acoesPossiveis[NUM_AÇÕES];//Ações possíveis para o jogo em questão
```

<Métodos>

```
ObterEstratégia ← probabilidadeJogador
SomaNormalizada ← 0;
DE i=0 → NUM_AÇÕES
SE SomaArrependimento[i] > 0
Estratégia[i] ← SomaArrependimento[i];
SENÃO
Estratégia[i] ← 0;
normalizingSum ← SomaNormalizada + Estratégia[i];


DE i=0 → NUM_AÇÕES
              SE SomaNormalizada > 0
                   Estratégia[i] ← Estratégia[i]/SomaNormalizada;
SENÃO
                   Estratégia[i] ← 1.0 / NUM_AÇÕES;
             SomaEstratégias[i] += probabilidadeJogador * Estratégia[i];
DEVOLVE Estratégia;
```

```
ObterEstratégiaMédia
EstratégiaMédia [NUM_AÇÕES];
SomaNormalizada ← 0;

DE i=0 → NUM_AÇÕES
            SomaNormalizada += SomaEstratégias[i];
      DE i=0 → NUM_AÇÕES
               SE SomaNormalizada > 0
                  EstratégiaMédia[i]      ←      SomaEstratégias[i]      /
SomaNormalizada;
SENÃO
                    EstratégiaMédia[i] ← 1.0 / NUM_AÇÕES;
   DEVOLVE EstratégiaMédia;
```

Para treinar as estratégias para os jogos de *Kuhn* e *Leduc* é necessário uma função que iterativamente chame o algoritmo. Nesta função é definido o conjunto de cartas a usar ({2,3,4} para o *Kuhn Poker* e {2,2,3,3,4,4} para o *Leduc*). Em seguida são baralhadas com a ajuda da função aleatória criada no inicio do programa e é então chamado o CFR para calcular a utilidade do jogo, adicionando esta utilidade a uma variável no final a utilidade média do treino. Concluindo as iterações, são apresentadas as estratégias médias para cada conjunto de informação através da função de estratégia média e é exibido o valor médio do jogo, dividindo a utilidade pelo número de iterações; para a convergência ocorrer são necessárias cerca de 100 000 iterações para o *Kuhn Poker* e 1 000 000 para o *Leduc Poker*.

<Treino da estratégia>

```
Treino ← iterações
SE jogo = kuhn
Cartas = cartasKuhn ← { 2, 3, 4};
SE jogo = Leduc
Cartas = cartasLeduc ← { 2, 2, 3, 3, 4, 4};
SENAO
   //determinar cartas para outros jogos de Poker

utilidade ← 0;

DE i=0 → iterações
       BARALHAR CARTAS;
       utilidade += CFR(Cartas, "", 1, 1);

 //Para cada conjuntoInformação na class Nó
 ESCREVER (ConjuntoInformação + ObterEstratégiaMédia);
 ESCREVER (Valor médio esperado = utilidade/ iterações)
```

A função CFR é iniciada com as cartas do jogo em questão, uma vetor com a história vazia e as probabilidades relativas a cada jogador. Esta probabilidade é calculada com base na ação escolhida pelo jogador na sua jogada anterior; como no início do jogo não existem ações passadas, a probabilidade para ambos é 1.

A primeira fase é verificar se o nó é terminal; se for o caso, é retornado o resultado consoante o vencedor. Se o nó não for terminal, é calculada a estratégia para o nó em questão. Depois, recursivamente, para cada ação serão calculadas os resultados/utilidades, e no final serão multiplicados pela estratégia tomada (PASS ou BET), obtendo assim o resultado do nó em questão. Tendo a utilidade do nó atual e as utilidades dos nós seguintes, é calculado o arrependimento subtraindo, para cada ação, a utilidade do nó seguinte pelo nó atual, multiplicado pela probabilidade do oponente.

<Função CFR>

```
CFR ← (cartas,história,p0,p1)
jogadas ← Tamanhohistória;
jogador ← jogadas % 2;
oponente ← 1 - jogador;

       VERIFICA SE É ESTADO TERMINAL
             SE SIM → DEVOLVE RESULTADO

       conjuntoInformação ← cartas[jogador] + história;

       VERIFICAR SE NA ARVORE EXISTE conjuntoInformação
             SE NÃO → CRIAR

      //CÁLCULO DA ESTRATÉGIA E DA UTILIDADE
Estratégia ← ObterEstratégia(SE jogador = 0 ENTÃO p0 SENÃO p1);
       utilidade [NUM_AÇÕES];
UtilidadeNó ← 0;

DE i=0 → NUM_AÇÕES
              HistoriaSeguinte = historia + acoesPossiveis[i];

              utilidade[i] ← SE jogador = 0
      -cfr(cartas, HistoriaSeguinte,p0 * Estratégia[i], p1)
    SENÃO
       -cfr(cartas, HistoriaSeguinte,p0, p1 * Estratégia[i]);

              UtilidadeNó += Estratégia [i] * utilidade [i];

       //CÁLCULO DO ARREPENDIMENTO
DE i=0 → NUM_ AÇÕES
arrependimento = utilidade [i] - UtilidadeNó;
SomaArrependimento[i]  +=  (SE  jogador  =  0  ENTÃO  p1  SENÃO  p0)  *
arrependimento;

  DEVOLVE UtilidadeNó;
```

4.2 Implementação iterativa do CFR

O algoritmo tratado anteriormente (CFR recursivo) pode ser facilmente confundido como sendo iterativo; no entanto, o que é iterativo no programa anterior é o treino da estratégia e, para cada iteração, a função do CFR é chamada recursivamente para calcular o valor esperado de um dado jogo. Neste caso, pretende-se que a própria função do CFR não se

chame para obter o valor esperado, mas que calcule esse valor iterativamente; deste modo, espera-se que seja possível reduzir o valor de memória utilizado e, talvez, o tempo de cálculo da estratégia.

No programa recursivo é criada uma classe nó, que dá origem a nós com informação de um dado estado de jogo, guardando-os numa árvore binária. Assim, o espaço na memória vai ser ocupado consoante o jogo se vai realizando. Com o algoritmo iterativo, esses espaços são gerados na totalidade no início do programa e, em vez do sistema utilizado anteriormente, são utilizados simples *arrays* para guardar os valores necessários para o cálculo da estratégia. No jogo como o *Kuhn Poker* e o *Leduc Poker* serão criados os *arrays* necessários para as cartas que estão em jogo. Assim, para cada carta (ou conjunto de cartas no caso do Leduc), existirão os *arrays* de SomaArrependimento, SomaEstratégia e Estratégia, que serão preenchidos durante as chamadas do CFR, de forma a serem posteriormente utilizados para calcular a estratégia média.

Variáveis necessárias para cada carta ou conjunto de cartas (carta do jogador + carta da mesa):

- SomaArrependimento
- SomaEstratégia
- Estratégia

Variáveis necessárias durante uma iteração

- Utilidade
- ProbabilidadeJogadores

Na Figura 5 está representada a árvore utilizada para armazenar os dados refentes ao *Kuhn Poker,* no método recursivo. É utilizado este jogo por se tratar de uma árvore mais pequena e de fácil visualização comparativamente ao *Leduc Poker*. Na Tabela 3 está representado o *array* que equivale aos ramos da árvore da Figura 5, servindo para armazenar os valores de Estratégia, SomaArrependimento e SomaEstratégia. Na Tabela 4 está representado o *array* que equivale aos nós da árvore, para o armazenamento dos valores de Utilidade e ProbabilidaDeJogadores

Tabela 3 - Array de Estratégias e Arrependimentos

0	1	2	3	4	5	-	-	8	9	-	-	-	-

Tabela 4 - Array de Utilidades e Probabilidades

0	1	2	X	4	X	X	-	-	X	X	-	-	-	-

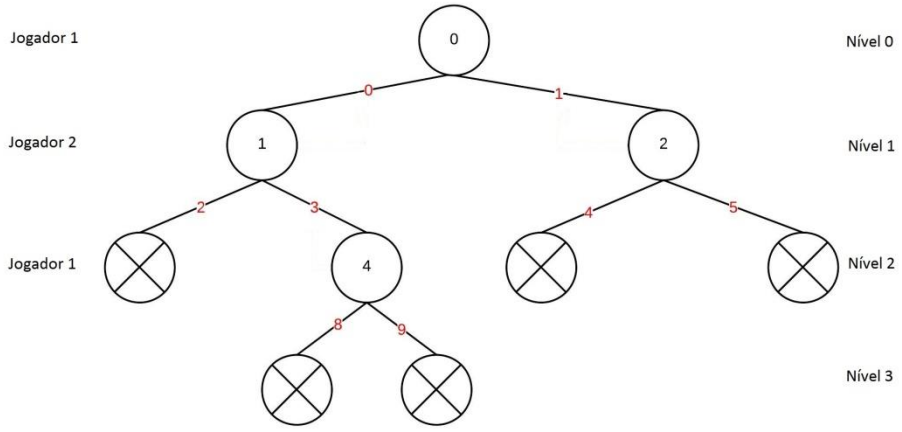

Figura 5 - Árvore de variáveis Kuhn Poker

As funções de treino e de ObterEstratégia utilizadas no método iterativo são idênticas às do método recursivo. No entanto, neste caso já não estão dentro da classe Nó uma vez que esta já não é utilizada. Para ser feita uma iteração do treino (isto é, um jogo) é definido como parâmetro da função CFR, para além das cartas baralhadas, o nível máximo da árvore. No caso do *Kuhn Poker,* o nível máximo será 3 (Figura 3) e para o *Leduc Poker* será 6 (Figura 4). Na função CFR são executadas as seguintes ações:

1. Atualização de probabilidades e estratégias para as cartas passadas por argumento;
    a. Chamada da função ObterEstrategia;
    b. Armazenamento das estratégias nos devidos *arrays*.
2. Atualização das utilidades e do arrependimento;
    a. Acesso as probabilidades e estratégias calculadas anteriormente;
    b. Cálculo da utilidade final do jogo.

Passando a exemplificar as ações é feito primeiro o cálculo das probabilidades e das estratégias, desde o nível 0 até ao nível máximo; serão assim calculadas as probabilidades para a Tabela 4 e estratégias (PASS/BET) para a Tabela 3. Este cálculo é feito da seguinte forma:

```
ProbabilidadeJogador[posição] = ArrayEstratégia[posição - 1];
Estratégia=ObterEstratégia(posição,carta[Jogador],ProbabilidadeJogador[pos
ição]);
ArrayEstratégia[2*posição] = Estratégia[0];
ArrayEstratégia[(2*posição)+1] = Estratégia[1];
```

O cálculo da posição é feito a partir do nível da árvore, utilizando o seguinte algoritmo:

$$posição = [(2^{Nivel} - 1)\ ;\ (2^{Nivel+1} - 2)]$$

<Atualizar nível de probabilidades>

```
AtualizarNívelProbabilidades ← (cartas,probabilidadesJogador,
ArrayEstratégia, Nível)

DE i = 2^Nivel − 1 → (2^Nivel+1 − 2)
      SE (posção não for NosDecisão)
AtualizarPorbabilidades ← (cartas, nível, i, probabilidadesJogador,
ArrayEstratégia);
```

<Atualizar nível de probabilidades>

```
AtualizarNivelUtilidades ← cartas, arrayUtilidades, ArrayEstratégia,Nível,
ProbabilidadeJogador)

DE i = 2^Nivel − 1 → (2^Nivel+1 − 2)
      SE (posção não for NosDecisão)
      arrayUtilidade ← AtualizarUtilidade(cartas,nível,i, ArrayEstratégia,
ProbabilidadeJogador);
      SENAO
      arrayUtilidade[i] = PAYOFF
```

Este algoritmo garante que todas as posições de cada nível sejam visitadas. No caso dos *ArrayEstratégia*, para cada posição da ProbabilidadeJogador, têm de existir duas posições para colocar os valores referentes às probabilidades de "passar" e "apostar" (daí a pesquisa nesses *arrays* ser diferente dos arrays das probabilidades dos nós).

O cálculo das ProbabilidadeJogador é feito consoante a estratégia e ação adotada pelo jogador anterior, com exceção das 3 posições iniciais, visto que a probabilidade é sempre 1. Na Figura 6 percebe-se porque é que:

```
PorbabilidadeJogador[posição] = ArrayEstratégia[posição - 1].
```

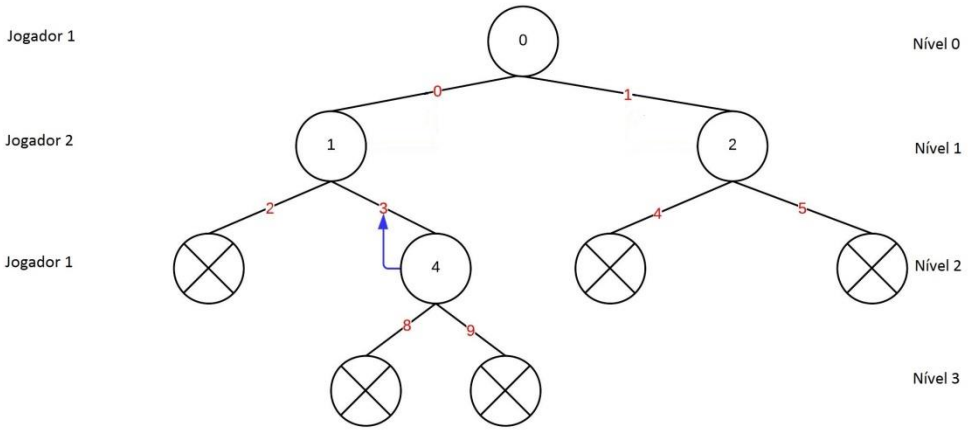

Figura 6 - Explicação do cálculo da probabilidade

Uma vez calculadas as probabilidades, é agora necessário calcular as utilidades de cada nó (os valores esperados), bem como o arrependimento. Para tal é feita a pesquisa dos *arrays* de forma inversa, ou seja, desde o nível máximo até ao 0, começando pelos nós de resultado/pagamento. A utilidade de um nó é calculada por:

$$Utilidade[posição] = (strategy[PASS] * Utilidade[posição * 2]) + (strategy[BET] * Utilidade[(posição * 2) + 1])$$

Na Figura 7 está representado um exemplo do cálculo da utilidade de um nó, mostrando a aplicação da equação da utilidade.

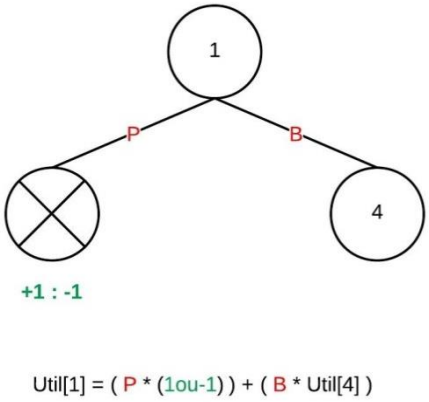

Figura 7 - Cálculo da Utilidade

Como as utilidades são calculadas de baixo para cima, as utilidades dos nós de resultados/pagamento são calculadas primeiro, pelo que foi criado um *array* de booleanos que as deteta consoante a posição e o tipo de nó; um teste é efetuado antes do cálculo das utilidades. No trecho de código seguinte pode observar-se esse vetor, idêntico ao *array* das utilidades e probabilidades, que apresenta "true" quando se trata de um nó de resultados/pagamentos.

```
NosDecisão={false,false,false,true,false,true,true,null,null,true,true,nul
l,null,null,null};
```

Quando não se trata de um nó deste tipo, é chamado o *array* de estratégia para a carta em questão para assim obter os valores de "passar" e "apostar" no cálculo da utilidade do nó acedido.

```
Estratégia[0] = ArrayEstratégia[2 * posição];
Estratégia[1] = ArrayEstratégia[(2 * posição) + 1];
```

```
Utilidade[posição]   =   (Estratégia[0]   *   Utilidade[(2*posição)+1])   +
(Estratégia[1] * Utilidade[(2*posição) + 2]);
```

Uma vez calculada a utilidade, avança-se para o cálculo do arrependimento de "passar" e "apostar" de um dado nó. Este cálculo é feito através da subtração do nó de *"pass"/"bet"* e do nó atual. Estes arrependimentos são adicionados ao *array* público de SomaArrependimentos para serem posteriormente utilizados no cálculo da estratégia média.

```
ArrependimentoPass = Utilidade[(2 * posição) + 1] - Utilidade [posição];
ArrependimentoBet = Utilidade[(2 * posição) + 2] - Utilidade [posição];
```

São estes os cálculos necessários à obtenção dos valores para o cálculo da estratégia média. Depois de serem obtidos estes valores (utilizando o algoritmo iterativo), é chamado o ObterEstratégiaMédia, apresentado no CFR recursivo, para cada carta e cada posição de que se pretende obter a estratégia média. A função CFR iterativa e a função de treino estão representadas a seguir, com as funções de atualização a executarem os cálculos que foram anteriormente explicados.

<CFR Iterativo>

```
Cfr ← (cartas, nívelMáximo)
DE i=0 → nivelMáximo
       Jogador = i % 2;
      Carta1 = cartas[Jogador];

AtualizaProbabilidades(cartas,ProbabilidadeJogador,
Estratégia[carta1],i);



DE i= nivelMáximo → 0
      Jogador = i % 2;
      Carta1 = cartas[Jogador];
       Carta2 = cartas[1-Jogador];

AtualizaUtilidades(cartas,utilidades,Estratégia[card1],
Estratégia[card2],i, ProbabilidadeJogador);
```

<Treino da estratégia>

```
Treino ← iterações

SE jogo = kuhn
      cartas = { 2, 3, 4 };
      nívelMaximo = 3
SE jogo = Leduc
      cartas = { 2, 2, 3, 3, 4, 4};
      nivelMaximo = 6;
SENAO
      //determinar cartas e níveis para outros jogos

DE i=0 → iterações
      BARALHARCARTAS
      Utilidade += cfr(cartas, nívelMaximo);
```

```
APRESENTAR ESTRATÉGIA
APRESENTAR (Utilidade/iterações)
```

4.3 Diferençãs de implementação entre Kuhn e Leduc poker

Atendendo ao facto de o algoritmo criado para o treino do Kuhn Poker ser bastante genérico, poucas alterações foram realizadas na aplicação do CFR ao Leduc. Foram criados novos vetores com as dimensões da árvore do Leduc e foram também adicionadas algumas funcionalidades que o Kuhn não possui, entre as quais, verificar se já foi posta a carta na mesa, verificar se existe par com a carta da mesa ou verificar se houve empate entre os dois jogadores. Posto isto, as alterações mais significativas são ao nível das folhas das árvores porque aparecem novos métodos de avaliação das cartas como é o caso da existência de par entre a carta de um jogador e a carta da mesa. Na Tabela 5 apresentam-se as principais diferenças a ter em consideração na aplicação do algoritmo iterativo do CFR aos jogos de Kuhn e Leduc Poker.

É possível confirmar os valores dos conjuntos de informação apresentados na Tabela 5 consultando os Anexos 1 e 3. O mesmo se aplica aos valores apresentados para a dimensão dos vetores, que podem ser comfirmados consultando as Tabelas 3 e 4 ou aplicando as fórmulas apresentadas. Relativamente ao número de iterações necessárias para a convergência de cada treino dos jogos para o equilíbrio de *Nash*, contatou-se que para o Kuhn Poker cerca de cem mil iterações seriam suficientes; já para o Leduc teriam de ser aumentadas num factor de 10 para obter uma estratégia robusta.

Neste capitúlo foi feita a implementação do algoritmo do CFR, tanto a implementação original com o método recursivo como a nova implementação com o método iterativo, sendo possível agora obter resultados para ambos os métodos, estes, que serão analisados no próximo capitulo.

Tabela 5 - Diferenças entre Kuhn e Leduc Poker

	Kuhn Poker	**Leduc Poker**
Exemplo de cartas	{2, 3, 4}	{2, 2, 3, 3, 4, 4}
Conjuntos de Informação	12	120
Nível máximo da árvore de jogo (0 → *nMax*)	*nMax* = 3	*nMax* = 6
Dimensão dos vetores de probabilidade/utilidade	$\sum_{n=0}^{nMax} 2^n = 15$	$\sum_{n=0}^{nMax} 2^n = 127$
Dimensão dos vetores de Estratégia/Arrependimento	$\sum_{n=0}^{nMax-1} 2 * 2^n = 15$	$\sum_{n=0}^{nMax-1} 2 * 2^n = 126$
Iterações	100 000	1 000 000

Capítulo 5

Resultados

Neste capítulo serão apresentados os resultados relativos aos algoritmos criados. Serão feitas comparações entre os métodos iterativos e recursivos do CFR para ambos os jogos em estudo.

5.1 Análise temporal

O *kuhn Poker* apresenta uma árvore do jogo muito pequena (com apenas 3 níveis), o que pressupõe que os jogos sejam muito rápidos. Nesse caso, não se esperam grandes melhorias nos tempos de processamento. Já o *Leduc Poker* possui uma árvore de jogo bastante mais complexa (cerca de 6 níveis) pelo que são expectáveis melhorias mais notáveis em termos temporais. Com base nesta evidência, poderá ser feita uma aproximação ao tempo que se pouparia na criação de estratégia para o jogo do *Poker Texas Hold'em* através do método iterativo do CFR.

Tabela 6 - Comparação temporal entre ambos os métodos

Jogo	Método	Iterações	Tempo (s)	Redução (%)
Kuhn Poker	**Recursivo**	1 000 000	4,017	6,1%
	Iterativo		3,774	
	Recursivo	100 000	0,399	6,7%
	Iterativo		0,372	
Leduc Poker	**Recursivo**	1 000 000	38,276	37,8%
	Iterativo		23,801	
	Recursivo	100 000	3,739	39,4%
	Iterativo		2,266	

Na Tabela 6 podem ser consultados os tempos do método recursivo e iterativo bem como as melhorias temporais entre ambos, aplicados aos dois jogos para 100 000 e 1 000 000 de iterações.

Na análise temporal do *Kuhn Poker* verificam-se pequenas alterações entre o algoritmo recursivo e o iterativo, para a mesma estratégia criada (Anexo 1 e 2). No caso do *Leduc Poker,* são verificadas melhorias notáveis entre a aplicação do algoritmo original e o algoritmo criado, o que leva a concluir que o método iterativo aumenta o desempenho temporal na criação da mesma estratégia (Anexo 3 e 4).

5.2 Análise espacial

Esperam-se também melhorias espaciais em ambos os métodos. Na Tabela 7 podem ser consultadas as quantidades de memória utilizada entre os dois algoritmos para ambos os jogos, bem como a comparação entre a memória utilizada nos diferentes métodos.

Tabela 7 - Comparação espacial entre ambos os métodos

Jogo	Método	Memória utilizada (Kbytes)	Redução (%)
Kuhn Poker	**Recursivo**	103	5%
	Iterativo	98	
Leduc Poker	**Recursivo**	120	-17%
	Iterativo	140	

Analisando a Tabela 7, verifica-se uma pequena melhoria entre a memória utilizada pelo método iterativo para o *Kuhn Poker*. No entanto, para o *Leduc Poker* já não se verificam melhorias, o que pode significar que quanto maior for o jogo (maior número de conjuntos de informação), mais espaço é necessário para a aplicação do método iterativo; no entanto, como este teste foi apenas aplicado a dois jogos, sendo obtidos resultados diferentes, é difícil prever o que esperar da memória utilizada para jogos maiores na aplicação do método iterativo. Isto deve-se provavelmente ao facto de ser necessário alocar matrizes de elevada dimensão no início do programa, onde se guardam as variáveis necessárias para a criação da estratégia média.

5.3 Valor médio esperado

Tabela 8 - Verificação dos valores esperados

Jogo	Método	Valor Médio Esperado	Desvio-Padrão	Gama de desvios
Kuhn Poker	Recursivo	-0,05565	0,001784	[-0,06027 ; -0,04963]
	Iterativo	-0,05575	0,002974	[-0,06355 ; -0,04828]
Leduc Poker	Recursivo	-0,01009	0,002550	[-0,01596 ; -0,00378]
	Iterativo	-0,01012	0,002807	[-0,01833 ; -0,00429]

Nas experiências realizadas foram geradas 100 estratégias para cada método, dando origem aos valores esperados representados na Tabela 8. Analisando o valor médio esperado para ambos os métodos dos dois jogos em estudo e tendo em conta o baixo valor do desvio-padrão (0.2%), há evidência de que as estratégias geradas são de facto muito semelhantes. As estratégias podem ser consultadas nos Anexos 1,2,3 e 4.

5.4 Análise de resultados

Verifica-se um resultado de cerca de 50% de melhorias temporais no treino iterativo de *Leduc Poker*. Sabendo-se que para o jogo de Poker *2NL Texas Hold'em*, o CFR, demora cerca de 2 semanas para calcular uma estratégia razoável (num computador idêntico ao utilizado para o cálculo do algoritmo nesta dissertação), pode ser realizado um simples cálculo de proporcionalidade direta, e espectar a redução de 1 semana ou mais no cálculo para o *Texas Hold'em,* usando o método iterativo. Foi possível notar grandes diferenças na melhoria do *Leduc* em relação o *Kuhn*, isto porque o *Kuhn* apresenta um jogo bastante rápido e com poucos *conjuntos de informação*. Deste modo, comparando o *Leduc* ao *Texas Hold'em*, poderá ser espectável que o desempenho seja superior a 50%, sendo esta uma expectativa muito otimista. Em relação à memória utilizada, visto que esta aumentou com a utilização do algoritmo iterativo, é expectável que aumente também para o jogo de *Poker* original utilizado atualmente na ACPC (*Anual Computer Poker Competition)*, que, segundo [17], necessita de cerca de $1{,}094 \times 10^{138}$ YB de RAM para ser resolvido.

Neste capitúlo foram analisados todos os resultados obtidos nas experiencias relativas aos dois métodos utilizados nesta dissertação, resultados temporais, espaciais e verificação da credibilidade das estratégias criadas, foram também comparados os resultados dos dois métodos.

Capítulo 6

Conclusão

Neste capítulo serão apresentadas as principais conclusões, assim como algumas sugestões de trabalho a desenvolver em investigações futuras.

6.1 Conclusões

Esta dissertação teve como principal objetivo, a conversão do método recursivo para o método iterativo de um algoritmo de criação de estratégias para jogos, o CFR. Desta conversão esperam-se melhorias de desempenho em termos de tempo, memória e resultados.

Foram utilizados dois jogos de *Poker* para a aplicação deste algoritmo: o *Kuhn Poker* e o *Leduc Poker* e para ambos foram criados treinos de estratégias utilizando tanto o método recursivo como iterativo do CFR para comparar os métodos e retirar conclusões.

Comparando os métodos, é possível verificar melhorias temporais na criação de estratégias para ambos os jogos, sendo que as melhorias são mais significativos no *Leduc Poker* devido ao facto do *Kuhn Poker* ser um jogo muito pequeno e de calcúlo rápido com o método recursivo (cerca de 5% melhorias temporais). Com base nos resultados obtidos no Leduc (cerca de 40% de melhorias temporais) é possível concluir que o método iterativo ao CFR apresenta um aumento significativo de desempenho. Este desempenho pode ser extrapolado para o jogo maior de Poker, o 2NL Texas Hold'em, e estimar melhorias temporais na utilização do método iterativo de mais de 50%, o que seria soberbo atendendo ao tempo espectável que o método recursivo do CFR necessita para criar uma estratégia razoavelmente boa.

No que diz respeito à memória utilizada, verifica-se aumento no jogo do *Leduc Poker* que poderá estar relacionado com a quantidade de *arrays* que têm de ser criados para armazenar a informação processada durante a criação da estratégia.

É importante salientar que não existem grandes diferenças entre as estratégias criadas pelo método original e o iterativo, e que o valor médio esperado é praticamente o mesmo, o que valida as melhorias temporais, uma vez que para criar a mesma estratégia, o tempo diminui com a aplicação do método iterativo.

Em suma, a elaboração desta dissertação permitiu mostrar que é possível obter melhor desempenho temporal do CFR aplicando o método iterativo.

6.2 Trabalho futuro

Após a realização desta dissertação, ficam algumas recomendações e alterações que podem ser feitas futuramente.

Com a criação do método iterativo, poderiam ser facilmente obtidas melhorias através da implementação do algoritmo em GPU, permitindo a atualização concorrente dos nós de cada nível. Isto seria possível devido ao elevado poder de processamento da placa gráfica em relação ao CPU dos computadores, tirando assim partido da capacidade de cálculo aritmético que estas oferecem. Desta forma, para investigações futuras sugere-se a aplicação do algoritmo do CFR iterativo em GPU.

Capítulo 7

Bibliografia

[1] P. Morris, Introduction to games theory, springer, (1994).

[2] Von Neumann, J. and Morgenstern, O., Theory of Games and Economic Behavior (60th Anniversary Commemorative Edition), Princeton University Press, (2007).

[3] F.Barrichelo,http://www.teoriadosjogos.net/teoriadosjogos/list-trechosimprime.asp?id=29, consultado pela ultima vez dia 6/09/2014

[4] Lanctot, M., Lisy, V., and Bowling, M., Search in Imperfect Information Games using Online Monte Carlo Counterfactual Regret Minimization. In AAAI Workshop on Computer Poker and Imperfect Information.(2014).

[5] Billings, D. Algorithms and assessment in computer poker, PhD thesis, University of Alberta, 2006.

[6] "JohnVonNeumann."[Online].Available: http://pt.wikipedia.org/wiki/John_von_Neumann.

[7] Lanctot,M.; Neller, T.(2013). An Introduction to Counterfactual Regret Minimization. http://cs.gettysburg.edu/~tneller/modelai/2013/cfr/index.html, consultado pela ultima vez dia 6/09/2014

[8] Risk, N. and Szafron, D. Using counterfactual regret minimization to create competitive multiplayer poker agents, In Proceeding AAMAS '10 Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1 - Volume 1, Pages 159-166. 2010.

[9] Lancot, M.; Waugh, K; Zinkevich, M.; and Bowling, M. (2009).Monte Carlo sampling for regret minimization in extensive games. In Proc. of NIPS, volume 22, 1078-1086.

[10] Johanson, M.; Bard,N.; Lanctot,M; Gibson, R. and Bowling, M.; Efficient Nash equilibrium approximation through Monte Carlo counterfactual regret minimization, Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems, 2012.

[11] Davis, T.; Burch, N. and Bowling, M. Using Response Function to Measure Strategy Strength, Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, 2014.

[12] Johanson, M., Bard, N., Burch, N., and Bowling, M. (2012). Finding Optimal Abstract Strategies. In Extensive-Form Games. In Proceedings of the National Conference on Artificial Intelligence (AAAI), pages 1371–1379.

[13] Teo▫filo, L. F.; Reis, L. P. and Cardoso H. L., Speeding-up Poker Game Abstraction Computation: Average Rank Strength, in Computer Poker and Imperfect Information: Papers from the AAAI 2013 Workshop, 2013, pp. 59-64.

[14] K. Glocer and M. Deckert, (2007) Opponent Modeling in Poker.

[15] S. Ganzfried and T. Sandholm, (2010). Game theory-based opponent modeling in large imperfect-information games. In Proceeding AAMAS '11 The 10th International Conference on Autonomous Agents and Multiagent Systems - Volume 2, Pages 533-540.

[16] H. Park and K. Kim, "Opponent modeling with incremental active learning: A case study of Iterative Prisoner's Dilemma," … *Intell. Games (CIG), 2013 IEEE* …, 2013.

[17] M. Johanson,(2013) Measuring the size of large no-limit poker games.

[18] Zinkevich, M.; Johanson, M.; Bowling, M.; and Piccione,C. (2008). Regret minimization in games with incomplete information. In Advances in Neural Information Processing Systems 20 (NIPS 2007).

ANEXOS

Anexo 1 – Exemplo de estratégia gerada para Kuhn Poker utilizando o método recursivo do CFR

Tabela 9 - A1 - Exemplo de estratégia gerada para Kuhn Poker utilizando o método recursivo do CFR

Carta	Sequência de Acções	Probabilidades das Acções	
		Pass	Bet
2	{ }	0,748	0,252
2	{ bet }	0,999	0,001
2	{ pass }	0,667	0,333
2	{ pass, bet }	0,999	0,001
3	{ }	0,999	0,001
3	{ bet }	0,666	0,334
3	{ pass }	0,999	0,001
3	{ pass, bet }	0,412	0,588
4	{ }	0,242	0,758
4	{ bet }	0,001	0,999
4	{ pass }	0,001	0,999
4	{ pass, bet }	0,001	0,999

Anexo 2 – Exemplo de estratégia gerada para Kuhn Poker utilizando o método iterativo do CFR

Tabela 10 - A2 - Exemplo de estratégia gerada para Kuhn Poker utilizando o método iterativo do CFR

Carta	Sequência de Acções	Probabilidades das Acções	
		Pass	Bet
2	{ }	0,886	0,114
2	{ bet }	0,999	0,001
2	{ pass }	0,665	0,335
2	{ pass, bet }	0,999	0,001
3	{ }	0,999	0,001
3	{ bet }	0,665	0,335
3	{ pass }	0,999	0,001
3	{ pass, bet }	0,555	0,445
4	{ }	0,660	0,340
4	{ bet }	0,001	0,999
4	{ pass }	0,001	0,999
4	{ pass, bet }	0,001	0,999

Anexo 3 – Exemplo de estratégia gerada para Leduc Poker utilizando o método recursivo do CFR

Tabela 11 - A3 - Exemplo de estratégia gerada para Leduc Poker utilizando o método recursivo do CFR

Carta [mesa]	Sequência de Acções	Probabilidades das Acções	
		Pass	Bet
2	{ }	0,128	0,872
2	{ bet }	0,001	0,999
2	{ pass }	0,926	0,074
2	{ pass, bet }	0,013	0,987
2[2]	{ bet, bet }	0,001	0,999
2[2]	{ pass, pass }	0,999	0,001
2[2]	{ bet, bet, bet }	0,001	0,999
2[2]	{ bet, bet, pass }	0,001	0,999
2[2]	{ pass, pass, pass }	0,001	0,999
2[2]	{ pass, pass, bet }	0,001	0,999
2[2]	{ pass, bet, bet }	0,999	0,001
2[2]	{ bet, bet, pass, bet }	0,001	0,999
2[2]	{ pass, pass, pass, bet }	0,001	0,999
2[2]	{ pass, bet, bet, bet }	0,001	0,999
2[2]	{ pass, bet, bet, pass }	0,001	0,999
2[2]	{ pass, bet, bet, pass, bet }	0,001	0,999
2[3]	{ bet, bet }	0,887	0,113
2[3]	{ pass, pass }	0,999	0,001
2[3]	{ bet, bet, bet }	0,999	0,001
2[3]	{ bet, bet, pass }	0,897	0,103
2[3]	{ pass, pass, pass }	0,999	0,001
2[3]	{ pass, pass, bet }	0,999	0,001
2[3]	{ pass, bet, bet }	0,369	0,631
2[3]	{ bet, bet, pass, bet }	0,999	0,001
2[3]	{ pass, pass, pass, bet }	0,999	0,001
2[3]	{ pass, bet, bet, bet }	0,999	0,001
2[3]	{ pass, bet, bet, pass }	0,990	0,010
2[3]	{ pass, bet, bet, pass, bet }	0,999	0,001

2[4]	{ bet, bet }	0,886	0,114
2[4]	{ pass, pass }	0,999	0,001
2[4]	{ bet, bet, bet }	0,999	0,001
2[4]	{ bet, bet, pass }	0,899	0,101
2[4]	{ pass, pass, pass }	0,999	0,001
2[4]	{ pass, pass, bet }	0,999	0,001
2[4]	{ pass, bet, bet }	0,007	0,993
2[4]	{ bet, bet, pass, bet }	0,999	0,001
2[4]	{ pass, pass, pass, bet }	0,999	0,001
2[4]	{ pass, bet, bet, bet }	0,999	0,001
2[4]	{ pass, bet, bet, pass }	0,996	0,004
2[4]	{ pass, bet, bet, pass, bet }	0,993	0,007
3	{ }	0,011	0,989
3	{ bet }	0,001	0,999
3	{ pass }	0,001	0,999
3	{ pass, bet }	0,001	0,999
3[2]	{ bet, bet }	0,911	0,089
3[2]	{ pass, pass }	0,993	0,007
3[2]	{ bet, bet, bet }	0,999	0,001
3[2]	{ bet, bet, pass }	0,899	0,101
3[2]	{ pass, pass, pass }	0,831	0,169
3[2]	{ pass, pass, bet }	0,999	0,001
3[2]	{ pass, bet, bet }	0,999	0,001
3[2]	{ bet, bet, pass, bet }	0,999	0,001
3[2]	{ pass, pass, pass, bet }	0,999	0,001
3[2]	{ pass, bet, bet, bet }	0,999	0,001
3[2]	{ pass, bet, bet, pass }	0,008	0,992
3[2]	{ pass, bet, bet, pass, bet }	0,999	0,001
3[3]	{ bet, bet }	0,001	0,999
3[3]	{ pass, pass }	0,995	0,005
3[3]	{ bet, bet, bet }	0,001	0,999
3[3]	{ bet, bet, pass }	0,001	0,999
3[3]	{ pass, pass, pass }	0,022	0,978
3[3]	{ pass, pass, bet }	0,022	0,978
3[3]	{ pass, bet, bet }	0,760	0,240
3[3]	{ bet, bet, pass, bet }	0,020	0,980
3[3]	{ pass, pass, pass, bet }	0,001	0,999

3[3]	{ pass, bet, bet, bet }	0,002	0,998
3[3]	{ pass, bet, bet, pass }	0,001	0,999
3[3]	{ pass, bet, bet, pass, bet }	0,001	0,999
3[4]	{ bet, bet }	0,999	0,001
3[4]	{ pass, pass }	0,997	0,003
3[4]	{ bet, bet, bet }	0,292	0,708
3[4]	{ bet, bet, pass }	0,999	0,001
3[4]	{ pass, pass, pass }	0,917	0,083
3[4]	{ pass, pass, bet }	0,001	0,999
3[4]	{ pass, bet, bet }	0,999	0,001
3[4]	{ bet, bet, pass, bet }	0,193	0,807
3[4]	{ pass, pass, pass, bet }	0,002	0,998
3[4]	{ pass, bet, bet, bet }	0,268	0,732
3[4]	{ pass, bet, bet, pass }	0,998	0,002
3[4]	{ pass, bet, bet, pass, bet }	0,070	0,930
4	{ }	0,002	0,998
4	{ bet }	0,001	0,999
4	{ pass }	0,001	0,999
4	{ pass, bet }	0,003	0,997
4[2]	{ bet, bet }	0,999	0,001
4[2]	{ pass, pass }	0,996	0,004
4[2]	{ bet, bet, bet }	0,302	0,698
4[2]	{ bet, bet, pass }	0,999	0,001
4[2]	{ pass, pass, pass }	0,291	0,709
4[2]	{ pass, pass, bet }	0,610	0,390
4[2]	{ pass, bet, bet }	0,999	0,001
4[2]	{ bet, bet, pass, bet }	0,196	0,804
4[2]	{ pass, pass, pass, bet }	0,998	0,002
4[2]	{ pass, bet, bet, bet }	0,002	0,998
4[2]	{ pass, bet, bet, pass }	0,953	0,047
4[2]	{ pass, bet, bet, pass, bet }	0,315	0,685
4[3]	{ bet, bet }	0,999	0,001
4[3]	{ pass, pass }	0,997	0,003
4[3]	{ bet, bet, bet }	0,302	0,698
4[3]	{ bet, bet, pass }	0,999	0,001
4[3]	{ pass, pass, pass }	0,391	0,608
4[3]	{ pass, pass, bet }	0,062	0,938

4[3]	{ pass, bet, bet }	0,991	0,009
4[3]	{ bet, bet, pass, bet }	0,198	0,802
4[3]	{ pass, pass, pass, bet }	0,002	0,998
4[3]	{ pass, bet, bet, bet }	0,599	0,401
4[3]	{ pass, bet, bet, pass }	0,979	0,021
4[3]	{ pass, bet, bet, pass, bet }	0,196	0,804
4[4]	{ bet, bet }	0,001	0,999
4[4]	{ pass, pass }	0,004	0,996
4[4]	{ bet, bet, bet }	0,001	0,999
4[4]	{ bet, bet, pass }	0,001	0,999
4[4]	{ pass, pass, pass }	0,500	0,500
4[4]	{ pass, pass, bet }	0,266	0,734
4[4]	{ pass, bet, bet }	0,005	0,995
4[4]	{ bet, bet, pass, bet }	0,001	0,999
4[4]	{ pass, pass, pass, bet }	0,493	0,507
4[4]	{ pass, bet, bet, bet }	0,001	0,999
4[4]	{ pass, bet, bet, pass }	0,001	0,999
4[4]	{ pass, bet, bet, pass, bet }	0,001	0,999

Anexo 4 – Exemplo de estratégia gerada para Leduc Poker utilizando o método iterativo do CFR

Tabela 12 - - A4 - Exemplo de estratégia gerada para Leduc Poker utilizando o método iterativo do CFR

Carta [mesa]	Sequência de Acções	Probabilidades das Acções	
		Pass	Bet
2	{ }	0,092	0,908
2	{ bet }	0,001	0,999
2	{ pass }	0,836	0,164
2	{ pass, bet }	0,001	0,999
2[2]	{ bet, bet }	0,001	0,999
2[2]	{ pass, pass }	0,001	0,999
2[2]	{ bet, bet, bet }	0,001	0,999
2[2]	{ bet, bet, pass }	0,001	0,999
2[2]	{ pass, pass, pass }	0,001	0,999
2[2]	{ pass, pass, bet }	0,001	0,999
2[2]	{ pass, bet, bet }	0,988	0,012
2[2]	{ bet, bet, pass, bet }	0,001	0,999
2[2]	{ pass, pass, pass, bet }	0,001	0,999
2[2]	{ pass, bet, bet, bet }	0,001	0,999
2[2]	{ pass, bet, bet, pass }	0,001	0,999
2[2]	{ pass, bet, bet, pass, bet }	0,001	0,999
2[3]	{ bet, bet }	0,891	0,109
2[3]	{ pass, pass }	0,999	0,001
2[3]	{ bet, bet, bet }	0,999	0,001
2[3]	{ bet, bet, pass }	0,898	0,102
2[3]	{ pass, pass, pass }	0,999	0,001
2[3]	{ pass, pass, bet }	0,999	0,001
2[3]	{ pass, bet, bet }	0,411	0,589
2[3]	{ bet, bet, pass, bet }	0,999	0,001
2[3]	{ pass, pass, pass, bet }	0,999	0,001
2[3]	{ pass, bet, bet, bet }	0,999	0,001
2[3]	{ pass, bet, bet, pass }	0,989	0,011
2[3]	{ pass, bet, bet, pass, bet }	0,999	0,001
2[4]	{ bet, bet }	0,892	0,108

2[4]	{ pass, pass }	0,999	0,001
2[4]	{ bet, bet, bet }	0,999	0,001
2[4]	{ bet, bet, pass }	0,900	0,100
2[4]	{ pass, pass, pass }	0,999	0,001
2[4]	{ pass, pass, bet }	0,999	0,001
2[4]	{ pass, bet, bet }	0,001	0,999
2[4]	{ bet, bet, pass, bet }	0,999	0,001
2[4]	{ pass, pass, pass, bet }	0,999	0,001
2[4]	{ pass, bet, bet, bet }	0,999	0,001
2[4]	{ pass, bet, bet, pass }	0,988	0,012
2[4]	{ pass, bet, bet, pass, bet }	0,876	0,124
3	{ }	0,010	0,990
3	{ bet }	0,001	0,999
3	{ pass }	0,001	0,999
3	{ pass, bet }	0,001	0,999
3[2]	{ bet, bet }	0,909	0,090
3[2]	{ pass, pass }	0,998	0,002
3[2]	{ bet, bet, bet }	0,999	0,001
3[2]	{ bet, bet, pass }	0,896	0,104
3[2]	{ pass, pass, pass }	0,421	0,579
3[2]	{ pass, pass, bet }	0,999	0,001
3[2]	{ pass, bet, bet }	0,999	0,001
3[2]	{ bet, bet, pass, bet }	0,999	0,001
3[2]	{ pass, pass, pass, bet }	0,999	0,001
3[2]	{ pass, bet, bet, bet }	0,999	0,001
3[2]	{ pass, bet, bet, pass }	0,004	0,996
3[2]	{ pass, bet, bet, pass, bet }	0,999	0,001
3[3]	{ bet, bet }	0,001	0,999
3[3]	{ pass, pass }	0,997	0,003
3[3]	{ bet, bet, bet }	0,001	0,999
3[3]	{ bet, bet, pass }	0,001	0,999
3[3]	{ pass, pass, pass }	0,150	0,850
3[3]	{ pass, pass, bet }	0,001	0,999
3[3]	{ pass, bet, bet }	0,305	0,695
3[3]	{ bet, bet, pass, bet }	0,028	0,972
3[3]	{ pass, pass, pass, bet }	0,001	0,999
3[3]	{ pass, bet, bet, bet }	0,001	0,999

3[3]	{ pass, bet, bet, pass }	0,001	0,999
3[3]	{ pass, bet, bet, pass, bet }	0,001	0,999
3[4]	{ bet, bet }	0,999	0,001
3[4]	{ pass, pass }	0,346	0,654
3[4]	{ bet, bet, bet }	0,297	0,703
3[4]	{ bet, bet, pass }	0,999	0,001
3[4]	{ pass, pass, pass }	0,568	0,432
3[4]	{ pass, pass, bet }	0,015	0,985
3[4]	{ pass, bet, bet }	0,999	0,001
3[4]	{ bet, bet, pass, bet }	0,192	0,808
3[4]	{ pass, pass, pass, bet }	0,001	0,999
3[4]	{ pass, bet, bet, bet }	0,371	0,629
3[4]	{ pass, bet, bet, pass }	0,993	0,007
3[4]	{ pass, bet, bet, pass, bet }	0,084	0,916
4	{ }	0,008	0,992
4	{ bet }	0,001	0,999
4	{ pass }	0,001	0,999
4	{ pass, bet }	0,001	0,999
4[2]	{ bet, bet }	0,999	0,001
4[2]	{ pass, pass }	0,998	0,002
4[2]	{ bet, bet, bet }	0,298	0,702
4[2]	{ bet, bet, pass }	0,999	0,001
4[2]	{ pass, pass, pass }	0,429	0,571
4[2]	{ pass, pass, bet }	0,284	0,716
4[2]	{ pass, bet, bet }	0,999	0,001
4[2]	{ bet, bet, pass, bet }	0,197	0,803
4[2]	{ pass, pass, pass, bet }	0,997	0,003
4[2]	{ pass, bet, bet, bet }	0,917	0,083
4[2]	{ pass, bet, bet, pass }	0,996	0,004
4[2]	{ pass, bet, bet, pass, bet }	0,341	0,659
4[3]	{ bet, bet }	0,999	0,001
4[3]	{ pass, pass }	0,997	0,003
4[3]	{ bet, bet, bet }	0,289	0,711
4[3]	{ bet, bet, pass }	0,999	0,001
4[3]	{ pass, pass, pass }	0,161	0,839
4[3]	{ pass, pass, bet }	0,161	0,839
4[3]	{ pass, bet, bet }	0,999	0,001

4[3]	{ bet, bet, pass, bet }	0,194	0,806
4[3]	{ pass, pass, pass, bet }	0,001	0,999
4[3]	{ pass, bet, bet, bet }	0,210	0,790
4[3]	{ pass, bet, bet, pass }	0,994	0,006
4[3]	{ pass, bet, bet, pass, bet }	0,132	0,868
4[4]	{ bet, bet }	0,001	0,999
4[4]	{ pass, pass }	0,019	0,981
4[4]	{ bet, bet, bet }	0,001	0,999
4[4]	{ bet, bet, pass }	0,001	0,999
4[4]	{ pass, pass, pass }	0,500	0,500
4[4]	{ pass, pass, bet }	0,500	0,500
4[4]	{ pass, bet, bet }	0,004	0,996
4[4]	{ bet, bet, pass, bet }	0,011	0,989
4[4]	{ pass, pass, pass, bet }	0,013	0,987
4[4]	{ pass, bet, bet, bet }	0,001	0,999
4[4]	{ pass, bet, bet, pass }	0,001	0,999
4[4]	{ pass, bet, bet, pass, bet }	0,004	0,995

Anexo 5 - Código CFR iterativo (Kuhn Poker)

```
class Program
{
public static Random random = new Random();
public static readonly int PASS = 0, BET = 1, NUM_ACTIONS = 2;
public static Stopwatch swCPU = new Stopwatch();
public static double finalUtil = 0;


public class IterativeCFR
{

public bool?[] payoffNodes = { false, false, false, true, false, true, true,
null, null, true, true, null, null, null, null };

public double[] util = { 0, 0, 0, 0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1 };
public double[] prob = { 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

public double[] regretSumTree2 = { 0, 0, 0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -
1},
strategySumTree2 = { 0, 0, 0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1},
strategy2 = { 0, 0, 0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1};

public double[] regretSumTree3 = { 0, 0, 0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -
1},
strategySumTree3 = { 0, 0, 0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1},
strategy3 = { 0, 0, 0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1};

public double[] regretSumTree4 = { 0, 0, 0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -
1},
strategySumTree4 = { 0, 0, 0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1},
strategy4 = { 0, 0, 0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1};

//atualiza todas as posoções de um dado nível da árvore das utilidades
void updateLevelUtil(int[] card, double[] utilTree, double[] stratTree1,double[]
stratTree2, int Level, double[] probTree)
{
for (int i = (int)Math.Pow(2, Level) - 1; i <= (int)Math.Pow(2 ,(Level + 1)) - 2;
i++)
{
//verifica se é um nó de payoff
if (payoffNodes[i] == true)
{
utilTree[i] = payoffNode(Level, i, card);
}
else if(payoffNodes[i] == false)
{
updateNode(card, Level, i, utilTree, stratTree1,stratTree2, probTree);
}
}

}

//atualiza todas as posições de um dado nível da arvore das probabilidades, desde
que não sejam de payoff
void updateLevelProbs(int[] card, double[] probTree, double[] stratTree1, int
Level)
{
for (int i = (int)Math.Pow(2, Level) - 1; i <= ((int)Math.Pow(2, (Level + 1)) -
2); i++)
```

```
{
if (payoffNodes[i] == false)
updateProbs(card, Level, i, probTree, stratTree1);
}
}

//faz o payoff nos devidos nós
double payoffNode(int level, int pos, int[] cards)
{
int player = level % 2;
int opponent = 1 - player;
bool isPlayerCardHigher = cards[player] > cards[opponent];

if (pos == 3)
return isPlayerCardHigher ? 1 : -1;
else if (pos == 6 || pos == 10)
return isPlayerCardHigher ? 2 : -2;
else
return 1;
}

//atualiza as probabilidades de uma dada posição da àrvore.
void updateProbs(int[] card,int level, int pos, double[] probTree, double[]
stratTree1)
{
double[] strategy = new double[NUM_ACTIONS];

if (level < 2)
{
strategy = getStrategy(pos, card[level % 2], probTree[pos]);
stratTree1[2*pos] = strategy[0];
stratTree1[(2*pos)+1] = strategy[1];
}
else
{
if (pos % 2 == 0)
probTree[pos] = stratTree1[(pos / 2) - 2];
else
probTree[pos] = stratTree1[((pos + 1) / 2) - 2];

strategy = getStrategy(pos, card[level % 2], probTree[pos]);
stratTree1[2*pos] = strategy[0];
stratTree1[(2*pos)+1] = strategy[1];
}
}

//atualiza as utilidades e os arrependimentos de uma dada posição da àrvore
void updateNode(int[] card,int level, int pos, double[] nodeUtil, double[]
stratTree1, double[] stratTree2,  double[] probTree)
{
double[] strategy = new double[NUM_ACTIONS];
double regretP = 0,regretB=0;
double[] regretsum = new double[NUM_ACTIONS];

//Apartir das probabilidades atualizadas anteriormente, não ha necessidade de as
calcular novamente, é so ir buscar as as arvores.
strategy[0] = stratTree1[2 * pos];
strategy[1] = stratTree1[(2 * pos) + 1];

//cálculo da utilidade do nó atual
nodeUtil[pos] = ((strategy[0] * (-nodeUtil[(2*pos)+1])) + (strategy[1] * (-
nodeUtil[(2*pos) + 2])));
```

```
//calculo do arrependimento de PASS/BET
regretP = (-nodeUtil[(2 * pos) + 1]) - nodeUtil[pos];
regretB = (-nodeUtil[(2 * pos) + 2]) - nodeUtil[pos];

//calculo do arrependimento do nó atual.
if (level > 0)
{
regretsum[0] = stratTree2[pos - 1] * regretP;
regretsum[1] = stratTree2[pos - 1] * regretB;
}
else
{
regretsum[0] =  regretP;
regretsum[1] =  regretB;
}


//atualização das arvores
if (card[level%2] == 2)
{
regretSumTree2[(2 * pos)] += regretsum[0];
regretSumTree2[(2 * pos) + 1] += regretsum[1];
}
else if (card[level % 2] == 3)
{
regretSumTree3[(2 * pos)] += regretsum[0];
regretSumTree3[(2 * pos) + 1] += regretsum[1];
}
else if (card[level % 2] == 4)
{
regretSumTree4[(2 * pos)] += regretsum[0];
regretSumTree4[(2 * pos) + 1] += regretsum[1];
}
}

//função que calcula a estratégia PASS/BET para uma dada posição e para uma dada
carta
public double[] getStrategy(int pos,int card,double realizationWeight)
{
double normalizingSum = 0;
double[] strategy = new double[NUM_ACTIONS];
double[] regretSum = new double[NUM_ACTIONS];
double[] strategySum = new double[NUM_ACTIONS];

if (card == 2)
{
regretSum[0] = regretSumTree2[2 * pos];
regretSum[1] = regretSumTree2[(2 * pos) + 1];
}
else if (card == 3)
{
regretSum[0] = regretSumTree3[2 * pos];
regretSum[1] = regretSumTree3[(2 * pos) + 1];
}
else
{
regretSum[0] = regretSumTree4[2 * pos];
regretSum[1] = regretSumTree4[(2 * pos) + 1];
}
```

```
for (int i = 0; i < NUM_ACTIONS; i++)
{
strategy[i] = regretSum[i] > 0 ? regretSum[i] : 0;
normalizingSum += strategy[i];
}
for (int i = 0; i < NUM_ACTIONS; i++)
{
if (normalizingSum > 0)
strategy[i] /= normalizingSum;
else
strategy[i] = 1.0 / NUM_ACTIONS;
strategySum[i] = realizationWeight * strategy[i];
}

//atualização das arvores
if (card == 2)
{
strategySumTree2[2 * pos] += strategySum[0];
strategySumTree2[(2 * pos) + 1] += strategySum[1];
}
else if (card == 3)
{
strategySumTree3[2 * pos] += strategySum[0];
strategySumTree3[(2 * pos) + 1] += strategySum[1];
}
else
{
strategySumTree4[2 * pos] += strategySum[0];
strategySumTree4[(2 * pos) + 1] += strategySum[1];
}

return strategy;
}

//função que calcula a estratégia média final para cada carta com base nos
valores guardados durante o treino
public double[] getAverageStrategy(int pos, int card)
{
double[] avgStrategy = new double[NUM_ACTIONS];
double normalizingSum = 0;
double[] strategySum = new double[NUM_ACTIONS];

if (card == 2)
{
strategySum[0] = strategySumTree2[2 * pos];
strategySum[1] = strategySumTree2[(2 * pos) + 1];
}
else if (card == 3)
{
strategySum[0] = strategySumTree3[2 * pos];
strategySum[1] = strategySumTree3[(2 * pos) + 1];
}
else
{
strategySum[0] = strategySumTree4[2 * pos];
strategySum[1] = strategySumTree4[(2 * pos) + 1];
}

for (int i = 0; i < NUM_ACTIONS; i++)
normalizingSum += strategySum[i];
for (int i = 0; i < NUM_ACTIONS; i++)
```

```
if (normalizingSum > 0)
avgStrategy[i] = strategySum[i] / normalizingSum;
else
avgStrategy[i] = 1.0 / NUM_ACTIONS;
return avgStrategy;
}

//treino do jogo
public void train(int iterations)
{
int[] cards = { 2, 3, 4 };
int MaxLevel = 3;

for (int i = 0; i < iterations; i++)
{
//shuffle cards
for (int c1 = cards.Length - 1; c1 > 0; c1--)
{
int c2 = random.Next(0, c1 + 1);
int tmp = cards[c1];

cards[c1] = cards[c2];
cards[c2] = tmp;
}
cfr(cards, MaxLevel);
}

Console.WriteLine("Card history               [Pass/Bet]");

Console.WriteLine("2:                      " + LineString(getAverageStrategy(0,
2)));
Console.WriteLine("2b:                      " + LineString(getAverageStrategy(2,
2)));
Console.WriteLine("2p:                      " + LineString(getAverageStrategy(1,
2)));
Console.WriteLine("2pb:                      " + LineString(getAverageStrategy(4,
2)));
Console.WriteLine("3:                      " + LineString(getAverageStrategy(0,
3)));
Console.WriteLine("3b:                      " + LineString(getAverageStrategy(2,
3)));
Console.WriteLine("3p:                      " + LineString(getAverageStrategy(1,
3)));
Console.WriteLine("3pb:                      " + LineString(getAverageStrategy(4,
3)));
Console.WriteLine("4:                      " + LineString(getAverageStrategy(0,
4)));
Console.WriteLine("4b:                      " + LineString(getAverageStrategy(2,
4)));
Console.WriteLine("4p:                      " + LineString(getAverageStrategy(1,
4)));
Console.WriteLine("4pb:                      " + LineString(getAverageStrategy(4,
4)));


Console.WriteLine("\n\nEV = " + finalUtil/iterations);

}

//Função CFR iterativa
private void cfr(int[] cards, int level)
```

```
{
//Ciclo que atualiza todas as probabilidades de um dado jogo
for (int i = 0; i <= level; i++)
{
int player = i % 2;
int card1 = cards[player];
int card2 = cards[1-player];

if (card1 == 2)
updateLevelProbs(cards, prob, strategy2, i);
else if (card1 == 3)
updateLevelProbs(cards, prob, strategy3, i);
else
updateLevelProbs(cards, prob, strategy4, i);
}

//Ciclo que atualiza todas as utilidades e arrependimentos de um dado jogo com
base nas probabilidades calculadas anteriormente
for (int i = level; i >= 0; i--)
{
int player = i % 2;
int card1 = cards[player];
int card2 = cards[1-player];

if (card1 == 2)
if(card2 == 3)
updateLevelUtil(cards, util, strategy2,strategy3, i, prob);
else
updateLevelUtil(cards, util, strategy2, strategy4, i, prob);
else if (card1 == 3)
if(card2 == 2)
updateLevelUtil(cards, util, strategy3,strategy2, i, prob);
else
updateLevelUtil(cards, util, strategy3, strategy4, i, prob);
else
if(card2==2)
updateLevelUtil(cards, util, strategy4,strategy2, i, prob);
else
updateLevelUtil(cards, util, strategy4, strategy3, i, prob);
}
//Somatório dos resultados dos jogos
finalUtil += util[0];
}
}

public static string LineString(double[] arr)
{
var str = "";
foreach (var i in arr)
{
str += i + " ";
}
return str;
}

static void Main(string[] args)
{
int it = 1000000;
swCPU.Start();
IterativeCFR trainergpu = new IterativeCFR();
trainergpu.train(it);
swCPU.Stop();
```

```
Process proc = Process.GetCurrentProcess();
long memory = GC.GetTotalMemory(true);
Console.WriteLine("Memory used: {0}.", memory / 1024);
Console.WriteLine("Time: {0}", swCPU.Elapsed.TotalSeconds);

Console.Read();
}

}
```

Anexo 6 - Código CFR iterativo (Leduc Poker)

```
class Program
{
//GLOBALS game
public static Random random = new Random();
public static readonly int PASS = 0, BET = 1, NUM_ACTIONS = 2;
public static Stopwatch swGPU = new Stopwatch();
public static double finalUtil = 0;

public class IterativeCFR
{
public bool?[] payoffNodes = { false,
false, false,
false, false, true, false,
false, false, true, false,    null, null, false, false,
true, false, true,  true,    null, null, false, false,    null,null,null,  null,
true, false,true,true,
null,null,true,true,null,null,null,null,null,null,null,null,true,false,true,true,
null,null,null,null,null,null,null,null,null,null,true,true,null,null,null,null,
null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,n
ull,null,null,null,null,null,null,null,null,null,true,true,null,null,null,null,nu
ll,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,nul
l,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null};

public double[] util = { 0,
0, 0,
0, 0, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1,-1, -1, 0, 0, 0, 0,
-1,-1, 0, 0, -1, -1,-1, -1,-1, -1,-1, -1,0,0,0,0,-1, -1,-1, -1,-1, -1,-1, -1,-1,
-1, 0,0, -1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1,
-1,-1, -1,0,0,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1,
-1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1};

public double[] prob = { 1,
1, 1,
1, 1,-1, 1,
1, 1,-1, 1,-1, -1,1, 1,
-1, 1,-1, -1,-1, -1,1, 1,-1, -1,-1, -1,-1, 1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, 1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -
1,-1, -1,-1, -1,-1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1,
-1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-
1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1};


public double[] regretSumTree2 = { 0, 0,
0, 0, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0,
-1, -1, 0, 0, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, -1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1, -1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-
1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1
},
strategySumTree2 = { 0, 0,
0, 0, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0,
```

```
-1, -1, 0, 0, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, -1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1, -1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-
1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1
},
strategy2 = { 0, 0,
0, 0, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0,
-1, -1, 0, 0, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, -1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1, -1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-
1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1
};
public double[] regretSumTree22 = { 0, 0,
0, 0, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0,
-1, -1, 0, 0, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, -1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1, -1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-
1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1
},
strategySumTree22 = { 0, 0,
0, 0, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0,
-1, -1, 0, 0, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, -1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1, -1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-
1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1
},
strategy22 = { 0, 0,
0, 0, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0,
-1, -1, 0, 0, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, -1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1, -1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-
1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1
};
public double[] regretSumTree23 = { 0, 0,
0, 0, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0,
-1, -1, 0, 0, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, -1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1, -1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-
1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1
},
strategySumTree23 = { 0, 0,
0, 0, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0,
```

```
-1, -1, 0, 0, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, -1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1, -1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-
1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1
},
strategy23 = { 0, 0,
0, 0, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0,
-1, -1, 0, 0, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, -1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1, -1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-
1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1
};
public double[] regretSumTree24 = { 0, 0,
0, 0, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0,
-1, -1, 0, 0, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, -1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1, -1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-
1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1
},
strategySumTree24 = { 0, 0,
0, 0, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0,
-1, -1, 0, 0, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, -1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1, -1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-
1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1
},
strategy24 = { 0, 0,
0, 0, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0,
-1, -1, 0, 0, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, -1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1,
-1,
-1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -
1, -1, -1, 0, 0,-1, -1,-1, -1, -1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -
1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1
};


//3
public double[] regretSumTree3 = { 0, 0,
0, 0, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0,
-1, -1, 0, 0, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, -1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1, -1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-
1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1
},
strategySumTree3 = { 0, 0,
0, 0, 0, 0,
```

```
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0,
-1, -1, 0, 0, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, -1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1, -1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-
1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1
},
strategy3 = { 0, 0,
0, 0, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0,
-1, -1, 0, 0, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, -1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1, -1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-
1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1
};
public double[] regretSumTree32 = { 0, 0,
0, 0, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0,
-1, -1, 0, 0, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, -1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1, -1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-
1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1
},
strategySumTree32 = { 0, 0,
0, 0, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0,
-1, -1, 0, 0, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, -1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1, -1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-
1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1
},
strategy32 = { 0, 0,
0, 0, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0,
-1, -1, 0, 0, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, -1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1, -1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-
1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1
};

public double[] regretSumTree34 = { 0, 0,
0, 0, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0,
-1, -1, 0, 0, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, -1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1, -1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-
1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1
},
strategySumTree34 = { 0, 0,
0, 0, 0, 0,
```

```
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0,
-1, -1, 0, 0, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, -1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1, -1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-
1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1
},
strategy34 = { 0, 0,
0, 0, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0,
-1, -1, 0, 0, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, -1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1, -1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-
1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1
};

public double[] regretSumTree33 = { 0, 0,
0, 0, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0,
-1, -1, 0, 0, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, -1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1, -1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-
1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1
},
strategySumTree33 = { 0, 0,
0, 0, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0,
-1, -1, 0, 0, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, -1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1, -1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-
1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1
},
strategy33 = { 0, 0,
0, 0, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0,
-1, -1, 0, 0, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, -1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1, -1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-
1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1
};
//4
public double[] regretSumTree4 = { 0, 0,
0, 0, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0,
-1, -1, 0, 0, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, -1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1, -1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-
1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1
},
strategySumTree4 = { 0, 0,
0, 0, 0, 0,
```

```
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0,
-1, -1, 0, 0, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, -1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1, -1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-
1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1
},
strategy4 = { 0, 0,
0, 0, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0,
-1, -1, 0, 0, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, -1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1, -1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-
1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1
};
public double[] regretSumTree42 = { 0, 0,
0, 0, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0,
-1, -1, 0, 0, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, -1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1, -1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-
1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1
},
strategySumTree42 = { 0, 0,
0, 0, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0,
-1, -1, 0, 0, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, -1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1, -1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-
1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1
},
strategy42 = { 0, 0,
0, 0, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0,
-1, -1, 0, 0, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, -1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1, -1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-
1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1
};

public double[] regretSumTree43 = { 0, 0,
0, 0, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0,
-1, -1, 0, 0, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, -1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1, -1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-
1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1
},
strategySumTree43 = { 0, 0,
0, 0, 0, 0,
```

```
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0,
-1, -1, 0, 0, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, -1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1, -1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-
1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1
},
strategy43 = { 0, 0,
0, 0, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0,
-1, -1, 0, 0, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, -1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1, -1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-
1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1
};
public double[] regretSumTree44 = { 0, 0,
0, 0, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0,
-1, -1, 0, 0, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, -1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1, -1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-
1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1
},
strategySumTree44 = { 0, 0,
0, 0, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0,
-1, -1, 0, 0, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, -1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1, -1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-
1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1
},
strategy44 = { 0, 0,
0, 0, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0,
0, 0, 0, 0, -1, -1, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0,
-1, -1, 0, 0, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, -1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1,
-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-
1, -1, -1, -1, 0, 0,-1, -1,-1, -1, -1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-
1, -1,-1, -1,-1, -1,-1, -1,-1, -1,-1, -1, -1, -1, -1, -1,-1, -1,-1, -1
};


//atualiza todas as posoções de um dado nível da árvore das utilidades
void updateLevelUtil(int[] card, double[] utilTree,double[]
stratTreeSING,double[] stratTreeSING2, double[] stratTree1,double[] stratTree2,
int Level, double[] probTree)
{
for (int i = (int)Math.Pow(2, Level) - 1; i <= (int)Math.Pow(2, (Level + 1)) - 2;
i++)
{
//verifica se é um nó de payoff
if (payoffNodes[i] == true)
{
utilTree[i] = payoffNode(Level, i, card);
```

```
}
else if (payoffNodes[i] == false)
{
updateNode(card, Level, i, utilTree,  stratTreeSING,stratTreeSING2,
stratTree1,stratTree2, probTree);
}
}
}
//atualiza todas as posições de um dado nível da arvore das probabilidades, desde
que não sejam de payoff
void updateLevelProbs(int[] card, double[] probTree, double[] stratTreeSING,
double[] stratTree, int Level)
{
for (int i = (int)Math.Pow(2, Level) - 1; i <= ((int)Math.Pow(2, (Level + 1)) -
2); i++)
{
if (payoffNodes[i] == false)
updateProbs(card, Level, i, probTree, stratTreeSING, stratTree);
}
}
//Verifica se já saiu a carta comunitária
bool alreadyRiver( int pos)
{
if (pos < 3 || pos == 4 || pos ==9 || pos ==5)
return false;
else
return true;
}
//faz o payoff nos devidos nós
double payoffNode(int level, int pos, int[] cards)
{
int player = level % 2;
int opponent = 1 - player;
int c1 = cards[player];
int c2 = cards[opponent];
int cc = cards[2];

bool isPlayerCardHigher = c1 >c2;
bool draw = c1 == c2;
bool havepair = c1 == cc;
bool havepairopo = c2 == cc;

if (draw)
return 0;
else if (!alreadyRiver(pos))
{
return 1;
}
else
{
if (pos == 17 || pos == 33)
return 1;
else if (pos == 29 || pos == 89 || pos == 45 || pos == 57)
return 2;
else if (pos == 15)
{
if (c1 == cc || c2 == cc)
return havepair ? 1 : -1;
else
return isPlayerCardHigher ? 1 : -1;
}
```

```
else if (pos == 18 || pos == 27 || pos == 34 || pos == 43)
{
if (c1 == cc || c2 == cc)
return havepair ? 2 : -2;
else
return isPlayerCardHigher ? 2 : -2;
}
else
{
if (c1 == cc || c2 == cc)
return havepair ? 3 : -3;
else
return isPlayerCardHigher ? 3 : -3;
}
}

}
//atualiza as probabilidades de uma dada posição da àrvore.
void updateProbs(int[] card, int level, int pos, double[] probTree, double[]
stratTreeSING, double[] stratTree)
{
double[] strategy = new double[NUM_ACTIONS];

if (level < 2)
{
strategy = getStrategy(level, pos, card, probTree[pos]);
stratTreeSING[2 * pos] = strategy[0];
stratTreeSING[(2 * pos) + 1] = strategy[1];
}
else if (level == 2)
{
if (alreadyRiver(pos))
{
if (pos % 2 == 0)
probTree[pos] = stratTreeSING[(pos / 2) - 2];
else
probTree[pos] = stratTreeSING[((pos + 1) / 2) - 2];

strategy = getStrategy(level, pos, card, probTree[pos]);
stratTree[2 * pos] = strategy[0];
stratTree[(2 * pos) + 1] = strategy[1];
}
else
{
if (pos % 2 == 0)
probTree[pos] = stratTreeSING[(pos / 2) - 2];
else
probTree[pos] = stratTreeSING[((pos + 1) / 2) - 2];

strategy = getStrategy(level, pos, card, probTree[pos]);
stratTreeSING[2 * pos] = strategy[0];
stratTreeSING[(2 * pos) + 1] = strategy[1];
}

}
else if (level == 3)
{

if (pos < 11)
if (pos % 2 == 0)
probTree[pos] = stratTreeSING[(pos / 2) - 2];
else
```

```
probTree[pos] = stratTreeSING[((pos + 1) / 2) - 2];
else
if (pos % 2 == 0)
probTree[pos] = stratTreeSING[(pos / 2) - 2];
else
probTree[pos] = stratTreeSING[((pos + 1) / 2) - 2];

strategy = getStrategy(level, pos, card, probTree[pos]);
stratTree[2 * pos] = strategy[0];
stratTree[(2 * pos) + 1] = strategy[1];
}
else if (level == 4)
{
if (pos < 19)
if (pos % 2 == 0)
probTree[pos] = stratTree[(pos / 2) - 2] * stratTreeSING[0];
else
probTree[pos] = stratTree[((pos + 1) / 2) - 2] * stratTreeSING[0];
else if (pos < 23 && pos >18)
if (pos % 2 == 0)
probTree[pos] = stratTreeSING[(pos / 2) - 2] * stratTreeSING[0];
else
probTree[pos] = stratTreeSING[((pos + 1) / 2) - 2] * stratTreeSING[0];
else if (pos < 27)
if (pos % 2 == 0)
probTree[pos] = stratTree[(pos / 2) - 2] * stratTreeSING[1];
else
probTree[pos] = stratTree[((pos + 1) / 2) - 2] * stratTreeSING[1];
else
if (pos % 2 == 0)
probTree[pos] = stratTree[(pos / 2) - 2] * stratTreeSING[1];
else
probTree[pos] = stratTree[((pos + 1) / 2) - 2] * stratTreeSING[1];

strategy = getStrategy(level, pos, card, probTree[pos]);
stratTree[2 * pos] = strategy[0];
stratTree[(2 * pos) + 1] = strategy[1];
}
else if (level == 5)
{
if (pos % 2 == 0)
probTree[pos] = stratTree[(pos / 2) - 2] * stratTreeSING[3];
else
probTree[pos] = stratTree[((pos + 1) / 2) - 2] * stratTreeSING[3];

strategy = getStrategy(level, pos, card, probTree[pos]);
stratTree[2 * pos] = strategy[0];
stratTree[(2 * pos) + 1] = strategy[1];
}

}
//atualiza as utilidades e os arrependimentos de uma dada posição da àrvore
void updateNode(int[] card, int level, int pos, double[] nodeUtil, double[]
stratTreeSING, double[] stratTreeSING2, double[] stratTree1, double[] stratTree2,
double[] probTree)
{
double[] strategy = new double[NUM_ACTIONS];
double regretP = 0, regretB = 0;
double[] regretsum = new double[NUM_ACTIONS];
```

```
//Apartir das probabilidades atualizadas anteriormente, não ha necessidade de as
calcular novamente, é so ir buscar as as arvores.
if (alreadyRiver(pos))
{
strategy[0] = stratTree1[2 * pos];
strategy[1] = stratTree1[(2 * pos) + 1];
}
else
{
strategy[0] = stratTreeSING[2 * pos];
strategy[1] = stratTreeSING[(2 * pos) + 1];
}

//cálculo da utilidade do nó atual
nodeUtil[pos] = (strategy[0] * (-nodeUtil[(2 * pos) + 1])) + (strategy[1] * (-
nodeUtil[(2 * pos) + 2]));

//calculo do arrependimento de PASS/BET
regretP = (-nodeUtil[(2 * pos) + 1]) - nodeUtil[pos];
regretB = (-nodeUtil[(2 * pos) + 2]) - nodeUtil[pos];

//calculo do arrependimento do nó atual.
if(level == 0)
{
regretsum[0] = regretP;
regretsum[1] = regretB;
}
else if (level < 3)
{
regretsum[0] = stratTreeSING2[pos - 1] * regretP;
regretsum[1] = stratTreeSING2[pos - 1] * regretB;
}
else if (level == 3)
{
if (pos < 9)
{
regretsum[0] = stratTree2[pos - 1] * stratTreeSING2[0] * regretP;
regretsum[1] = stratTree2[pos - 1] * stratTreeSING2[0] * regretB;
}
else if (pos < 11)
{
regretsum[0] = stratTreeSING2[pos - 1] * stratTreeSING2[0] * regretP;
regretsum[1] = stratTreeSING2[pos - 1] * stratTreeSING2[0] * regretB;
}
else
{
regretsum[0] = stratTree2[pos - 1] * stratTreeSING2[1] * regretP;
regretsum[1] = stratTree2[pos - 1] * stratTreeSING2[1] * regretB;
}
}
else if (level == 4)
{

if (pos < 19)
{
regretsum[0] = stratTree2[pos - 1] * stratTreeSING2[2] * regretP;
regretsum[1] = stratTree2[pos - 1] * stratTreeSING2[2] * regretB;
}
else if (pos < 23 )
{
regretsum[0] = stratTree2[pos - 1] * stratTreeSING2[3] * regretP;
regretsum[1] = stratTree2[pos - 1] * stratTreeSING2[3] * regretB;
```

```
}
else
{
regretsum[0] = stratTree2[pos - 1] * stratTreeSING2[5] * regretP;
regretsum[1] = stratTree2[pos - 1] * stratTreeSING2[5] * regretB;
}
}
else
{
regretsum[0] = stratTree2[pos - 1] * stratTreeSING2[9] * stratTreeSING2[0] *
regretP;
regretsum[1] = stratTree2[pos - 1] * stratTreeSING2[9] * stratTreeSING2[0] *
regretB;
}



if (card[level % 2] == 2)
{
if (alreadyRiver(pos))
{
if (card[2] == 2)
{
regretSumTree22[(2 * pos)] += regretsum[0];
regretSumTree22[(2 * pos) + 1] += regretsum[1];
}
else if (card[2] == 3)
{
regretSumTree23[(2 * pos)] += regretsum[0];
regretSumTree23[(2 * pos) + 1] += regretsum[1];
}
else if (card[2] == 4)
{
regretSumTree24[(2 * pos)] += regretsum[0];
regretSumTree24[(2 * pos) + 1] += regretsum[1];
}
}
else
{
regretSumTree2[(2 * pos)] += regretsum[0];
regretSumTree2[(2 * pos) + 1] += regretsum[1];
}
}
else if (card[level % 2] == 3)
{
if (alreadyRiver(pos))
{
if (card[2] == 2)
{
regretSumTree32[(2 * pos)] += regretsum[0];
regretSumTree32[(2 * pos) + 1] += regretsum[1];
}
else if (card[2] == 3)
{
regretSumTree33[(2 * pos)] += regretsum[0];
regretSumTree33[(2 * pos) + 1] += regretsum[1];
}
else if (card[2] == 4)
{
regretSumTree34[(2 * pos)] += regretsum[0];
regretSumTree34[(2 * pos) + 1] += regretsum[1];
```

```
}
}
else
{
regretSumTree3[(2 * pos)] += regretsum[0];
regretSumTree3[(2 * pos) + 1] += regretsum[1];
}

}
else if (card[level % 2] == 4)
{
if (alreadyRiver(pos))
{
if (card[2] == 2)
{
regretSumTree42[(2 * pos)] += regretsum[0];
regretSumTree42[(2 * pos) + 1] += regretsum[1];
}
else if (card[2] == 3)
{
regretSumTree43[(2 * pos)] += regretsum[0];
regretSumTree43[(2 * pos) + 1] += regretsum[1];
}
else if (card[2] == 4)
{
regretSumTree44[(2 * pos)] += regretsum[0];
regretSumTree44[(2 * pos) + 1] += regretsum[1];
}
}
else
{
regretSumTree4[(2 * pos)] += regretsum[0];
regretSumTree4[(2 * pos) + 1] += regretsum[1];
}

}

}
//função que calcula a estratégia PASS/BET para uma dada posição e para uma dada
carta
public double[] getStrategy(int level, int pos, int[] card, double
realizationWeight)
{
double normalizingSum = 0;
double[] strategy = new double[NUM_ACTIONS];
double[] regretSum = new double[NUM_ACTIONS];
double[] strategySum = new double[NUM_ACTIONS];


if (card[level%2] == 2)
{
if (alreadyRiver(pos))
{
if (card[2] == 2)
{
regretSum[0] = regretSumTree22[2 * pos];
regretSum[1] = regretSumTree22[(2 * pos) + 1];
}
else if (card[2] == 3)
{
regretSum[0] = regretSumTree23[2 * pos];
regretSum[1] = regretSumTree23[(2 * pos) + 1];
```

```
}
else if (card[2] == 4)
{
regretSum[0] = regretSumTree24[2 * pos];
regretSum[1] = regretSumTree24[(2 * pos) + 1];
}
}
else
{
regretSum[0] = regretSumTree2[2 * pos];
regretSum[1] = regretSumTree2[(2 * pos) + 1];
}

}
else if (card[level % 2] == 3)
{
if (alreadyRiver(pos))
{
if (card[2] == 2)
{
regretSum[0] = regretSumTree32[2 * pos];
regretSum[1] = regretSumTree32[(2 * pos) + 1];
}
else if (card[2] == 3)
{
regretSum[0] = regretSumTree33[2 * pos];
regretSum[1] = regretSumTree33[(2 * pos) + 1];
}
else if (card[2] == 4)
{
regretSum[0] = regretSumTree34[2 * pos];
regretSum[1] = regretSumTree34[(2 * pos) + 1];
}
}
else
{
regretSum[0] = regretSumTree3[2 * pos];
regretSum[1] = regretSumTree3[(2 * pos) + 1];
}

}
else if (card[level % 2] == 4)
{
if (alreadyRiver(pos))
{
if (card[2] == 2)
{
regretSum[0] = regretSumTree42[2 * pos];
regretSum[1] = regretSumTree42[(2 * pos) + 1];
}
else if (card[2] == 3)
{
regretSum[0] = regretSumTree43[2 * pos];
regretSum[1] = regretSumTree43[(2 * pos) + 1];
}
else if (card[2] == 4)
{
regretSum[0] = regretSumTree44[2 * pos];
regretSum[1] = regretSumTree44[(2 * pos) + 1];
}
}
```

```
else
{
regretSum[0] = regretSumTree4[2 * pos];
regretSum[1] = regretSumTree4[(2 * pos) + 1];
}
}


for (int i = 0; i < NUM_ACTIONS; i++)
{
strategy[i] = regretSum[i] > 0 ? regretSum[i] : 0;
normalizingSum += strategy[i];
}

for (int i = 0; i < NUM_ACTIONS; i++)
{
if (normalizingSum > 0)
strategy[i] /= normalizingSum;
else
strategy[i] = 1.0 / NUM_ACTIONS;
strategySum[i] = realizationWeight * strategy[i];
}

if (card[level % 2] == 2)
{
if (alreadyRiver(pos))
{
if (card[2] == 2)
{
strategySumTree22[2 * pos] += strategySum[0];
strategySumTree22[(2 * pos) + 1] += strategySum[1];
}
else if (card[2] == 3)
{
strategySumTree23[2 * pos] += strategySum[0];
strategySumTree23[(2 * pos) + 1] += strategySum[1];
}
else if (card[2] == 4)
{
strategySumTree24[2 * pos] += strategySum[0];
strategySumTree24[(2 * pos) + 1] += strategySum[1];
}
}
else
{
strategySumTree2[2 * pos] += strategySum[0];
strategySumTree2[(2 * pos)+1] += strategySum[1];
}

}
else if (card[level%2] == 3)
{
if (alreadyRiver(pos))
{
if (card[2] == 2)
{
strategySumTree32[2 * pos] += strategySum[0];
strategySumTree32[(2 * pos) + 1] += strategySum[1];
}
else if (card[2] == 3)
{
strategySumTree33[2 * pos] += strategySum[0];
```

```
strategySumTree33[(2 * pos) + 1] += strategySum[1];
}
else if (card[2] == 4)
{
strategySumTree34[2 * pos] += strategySum[0];
strategySumTree34[(2 * pos) + 1] += strategySum[1];
}
}
else
{
strategySumTree3[2 * pos] += strategySum[0];
strategySumTree3[(2 * pos) + 1] += strategySum[1];
}

}
else if (card[level % 2] == 4)
{
if (alreadyRiver(pos))
{
if (card[2] == 2)
{
strategySumTree42[2 * pos] += strategySum[0];
strategySumTree42[(2 * pos) + 1] += strategySum[1];
}
else if (card[2] == 3)
{
strategySumTree43[2 * pos] += strategySum[0];
strategySumTree43[(2 * pos) + 1] += strategySum[1];
}
else if (card[2] == 4)
{
strategySumTree44[2 * pos] += strategySum[0];
strategySumTree44[(2 * pos) + 1] += strategySum[1];
}
}
else
{
strategySumTree4[2 * pos] += strategySum[0];
strategySumTree4[(2 * pos) + 1] += strategySum[1];
}

}

return strategy;
}
//função que calcula a estratégia média final para cada carta com base nos
valores guardados durante o treino
public double[] getAverageStrategy(int pos, int card)
{
double[] avgStrategy = new double[NUM_ACTIONS];
double normalizingSum = 0;
double[] strategySum = new double[NUM_ACTIONS];
if (card == 2)
{
strategySum[0] = strategySumTree2[2 * pos];
strategySum[1] = strategySumTree2[(2 * pos) + 1];
}
else if(card == 3)
{
strategySum[0] = strategySumTree3[2 * pos];
strategySum[1] = strategySumTree3[(2 * pos) + 1];
```

```
}
else if (card == 4)
{
strategySum[0] = strategySumTree4[2 * pos];
strategySum[1] = strategySumTree4[(2 * pos) + 1];
}

else if (card == 22)
{
strategySum[0] = strategySumTree22[2 * pos];
strategySum[1] = strategySumTree22[(2 * pos) + 1];
}
else if (card == 23)
{
strategySum[0] = strategySumTree23[2 * pos];
strategySum[1] = strategySumTree23[(2 * pos) + 1];
}
else if (card == 24)
{
strategySum[0] = strategySumTree24[2 * pos];
strategySum[1] = strategySumTree24[(2 * pos) + 1];
}

else if (card == 32)
{
strategySum[0] = strategySumTree32[2 * pos];
strategySum[1] = strategySumTree32[(2 * pos) + 1];
}
else if (card == 33)
{
strategySum[0] = strategySumTree33[2 * pos];
strategySum[1] = strategySumTree33[(2 * pos) + 1];
}
else if (card == 34)
{
strategySum[0] = strategySumTree34[2 * pos];
strategySum[1] = strategySumTree34[(2 * pos) + 1];
}
else if (card == 42)
{
strategySum[0] = strategySumTree42[2 * pos];
strategySum[1] = strategySumTree42[(2 * pos) + 1];
}
else if (card == 43)
{
strategySum[0] = strategySumTree43[2 * pos];
strategySum[1] = strategySumTree43[(2 * pos) + 1];
}


else if (card == 44)
{
strategySum[0] = strategySumTree44[2 * pos];
strategySum[1] = strategySumTree44[(2 * pos) + 1];
}

for (int i = 0; i < NUM_ACTIONS; i++)
{
normalizingSum += strategySum[i];
}

for (int i = 0; i < NUM_ACTIONS; i++)
```

```
{
if (normalizingSum > 0)
avgStrategy[i] = strategySum[i] / normalizingSum;
else
avgStrategy[i] = 1.0 / NUM_ACTIONS;
}

return avgStrategy;
}
//treino do jogo
public void trainGPU(int iterations)
{
int[] cards = { 2,4,2,3,3, 4 };
int MaxLevel = 6;

for (int i = 0; i < iterations; i++)
{

for (int c1 = cards.Length - 1; c1 > 0; c1--)
{
int c2 = random.Next(0, c1 + 1);
int tmp = cards[c1];

cards[c1] = cards[c2];
cards[c2] = tmp;
}

cfr(cards, MaxLevel);
}

Console.WriteLine("Card history               [Pass/Bet]");
Console.WriteLine("2:              " + LineString(getAverageStrategy(0, 2)));
Console.WriteLine("2p:              " + LineString(getAverageStrategy(1, 2)));
Console.WriteLine("2b:              " + LineString(getAverageStrategy(2, 2)));
Console.WriteLine("2pb:              " + LineString(getAverageStrategy(4, 2)));
Console.WriteLine("2[2]pp:              " + LineString(getAverageStrategy(3,
22)));
Console.WriteLine("2[2]bb:              " + LineString(getAverageStrategy(6,
22)));
Console.WriteLine("2[2]pbb:              " + LineString(getAverageStrategy(10,
22)));
Console.WriteLine("2[2]ppp:              " + LineString(getAverageStrategy(7,
22)));
Console.WriteLine("2[2]ppb:              " + LineString(getAverageStrategy(8,
22)));
Console.WriteLine("2[2]bbp:              " + LineString(getAverageStrategy(13,
22)));
Console.WriteLine("2[2]bbb:              " + LineString(getAverageStrategy(14,
22)));
Console.WriteLine("2[2]pppb:              " + LineString(getAverageStrategy(16,
22)));
Console.WriteLine("2[2]pbbp:              " + LineString(getAverageStrategy(21,
22)));
Console.WriteLine("2[2]pbbb:              " + LineString(getAverageStrategy(22,
22)));
Console.WriteLine("2[2]bbpb:              " + LineString(getAverageStrategy(28,
22)));
Console.WriteLine("2[2]pbbpb:              " + LineString(getAverageStrategy(44,
22)));
Console.WriteLine("2[3]pp:              " + LineString(getAverageStrategy(3,
23)));
```

```
Console.WriteLine("2[3]bb:             " + LineString(getAverageStrategy(6,
23)));
Console.WriteLine("2[3]pbb:             " + LineString(getAverageStrategy(10,
23)));
Console.WriteLine("2[3]ppp:             " + LineString(getAverageStrategy(7,
23)));
Console.WriteLine("2[3]ppb:             " + LineString(getAverageStrategy(8,
23)));
Console.WriteLine("2[3]bbp:             " + LineString(getAverageStrategy(13,
23)));
Console.WriteLine("2[3]bbb:             " + LineString(getAverageStrategy(14,
23)));
Console.WriteLine("2[3]pppb:             " + LineString(getAverageStrategy(16,
23)));
Console.WriteLine("2[3]pbbp:             " + LineString(getAverageStrategy(21,
23)));
Console.WriteLine("2[3]pbbb:             " + LineString(getAverageStrategy(22,
23)));
Console.WriteLine("2[3]bbpb:             " + LineString(getAverageStrategy(28,
23)));
Console.WriteLine("2[3]pbbpb:             " + LineString(getAverageStrategy(44,
23)));
Console.WriteLine("2[4]pp:             " + LineString(getAverageStrategy(3,
24)));
Console.WriteLine("2[4]bb:             " + LineString(getAverageStrategy(6,
24)));
Console.WriteLine("2[4]pbb:             " + LineString(getAverageStrategy(10,
24)));
Console.WriteLine("2[4]ppp:             " + LineString(getAverageStrategy(7,
24)));
Console.WriteLine("2[4]ppb:             " + LineString(getAverageStrategy(8,
24)));
Console.WriteLine("2[4]bbp:             " + LineString(getAverageStrategy(13,
24)));
Console.WriteLine("2[4]bbb:             " + LineString(getAverageStrategy(14,
24)));
Console.WriteLine("2[4]pppb:             " + LineString(getAverageStrategy(16,
24)));
Console.WriteLine("2[4]pbbp:             " + LineString(getAverageStrategy(21,
24)));
Console.WriteLine("2[4]pbbb:             " + LineString(getAverageStrategy(22,
24)));
Console.WriteLine("2[4]bbpb:             " + LineString(getAverageStrategy(28,
24)));
Console.WriteLine("2[4]pbbpb:             " + LineString(getAverageStrategy(44,
24)));

Console.WriteLine("3:             " + LineString(getAverageStrategy(0, 3)));
Console.WriteLine("3p:             " + LineString(getAverageStrategy(1, 3)));
Console.WriteLine("3b:             " + LineString(getAverageStrategy(2, 3)));
Console.WriteLine("3pb:             " + LineString(getAverageStrategy(4, 3)));
Console.WriteLine("3[2]pp:             " + LineString(getAverageStrategy(3,
32)));
Console.WriteLine("3[2]bb:             " + LineString(getAverageStrategy(6,
32)));
Console.WriteLine("3[2]pbb:             " + LineString(getAverageStrategy(10,
32)));
Console.WriteLine("3[2]ppp:             " + LineString(getAverageStrategy(7,
32)));
Console.WriteLine("3[2]ppb:             " + LineString(getAverageStrategy(8,
32)));
```

```
Console.WriteLine("3[2]bbp:             " + LineString(getAverageStrategy(13,
32)));
Console.WriteLine("3[2]bbb:             " + LineString(getAverageStrategy(14,
32)));
Console.WriteLine("3[2]pppb:             " + LineString(getAverageStrategy(16,
32)));
Console.WriteLine("3[2]pbbp:             " + LineString(getAverageStrategy(21,
32)));
Console.WriteLine("3[2]pbbb:             " + LineString(getAverageStrategy(22,
32)));
Console.WriteLine("3[2]bbpb:             " + LineString(getAverageStrategy(28,
32)));
Console.WriteLine("3[2]pbbpb:             " + LineString(getAverageStrategy(44,
32)));
Console.WriteLine("3[3]pp:             " + LineString(getAverageStrategy(3,
33)));
Console.WriteLine("3[3]bb:             " + LineString(getAverageStrategy(6,
33)));
Console.WriteLine("3[3]pbb:             " + LineString(getAverageStrategy(10,
33)));
Console.WriteLine("3[3]ppp:             " + LineString(getAverageStrategy(7,
33)));
Console.WriteLine("3[3]ppb:             " + LineString(getAverageStrategy(8,
33)));
Console.WriteLine("3[3]bbp:             " + LineString(getAverageStrategy(13,
33)));
Console.WriteLine("3[3]bbb:             " + LineString(getAverageStrategy(14,
33)));
Console.WriteLine("3[3]pppb:             " + LineString(getAverageStrategy(16,
33)));
Console.WriteLine("3[3]pbbp:             " + LineString(getAverageStrategy(21,
33)));
Console.WriteLine("3[3]pbbb:             " + LineString(getAverageStrategy(22,
33)));
Console.WriteLine("3[3]bbpb:             " + LineString(getAverageStrategy(28,
33)));
Console.WriteLine("3[3]pbbpb:             " +
LineString(getAverageStrategy(44,33)));
Console.WriteLine("3[4]pp:             " + LineString(getAverageStrategy(3,
34)));
Console.WriteLine("3[4]bb:             " + LineString(getAverageStrategy(6,
34)));
Console.WriteLine("3[4]pbb:             " + LineString(getAverageStrategy(10,
34)));
Console.WriteLine("3[4]ppp:             " + LineString(getAverageStrategy(7,
34)));
Console.WriteLine("3[4]ppb:             " + LineString(getAverageStrategy(8,
34)));
Console.WriteLine("3[4]bbp:             " + LineString(getAverageStrategy(13,
34)));
Console.WriteLine("3[4]bbb:             " + LineString(getAverageStrategy(14,
34)));
Console.WriteLine("3[4]pppb:             " + LineString(getAverageStrategy(16,
34)));
Console.WriteLine("3[4]pbbp:             " + LineString(getAverageStrategy(21,
34)));
Console.WriteLine("3[4]pbbb:             " + LineString(getAverageStrategy(22,
34)));
Console.WriteLine("3[4]bbpb:             " + LineString(getAverageStrategy(28,
34)));
```

```
Console.WriteLine("3[4]pbbpb:             " + LineString(getAverageStrategy(44,
34)));

Console.WriteLine("4:              " + LineString(getAverageStrategy(0, 4)));
Console.WriteLine("4p:              " + LineString(getAverageStrategy(1, 4)));
Console.WriteLine("4b:              " + LineString(getAverageStrategy(2, 4)));
Console.WriteLine("4pb:              " + LineString(getAverageStrategy(4, 4)));
Console.WriteLine("4[2]pp:             " + LineString(getAverageStrategy(3,
42)));
Console.WriteLine("4[2]bb:             " + LineString(getAverageStrategy(6,
42)));
Console.WriteLine("4[2]pbb:             " + LineString(getAverageStrategy(10,
42)));
Console.WriteLine("4[2]ppp:             " + LineString(getAverageStrategy(7,
42)));
Console.WriteLine("4[2]ppb:             " + LineString(getAverageStrategy(8,
42)));
Console.WriteLine("4[2]bbp:             " + LineString(getAverageStrategy(13,
42)));
Console.WriteLine("4[2]bbb:             " + LineString(getAverageStrategy(14,
42)));
Console.WriteLine("4[2]pppb:             " + LineString(getAverageStrategy(16,
42)));
Console.WriteLine("4[2]pbbp:             " + LineString(getAverageStrategy(21,
42)));
Console.WriteLine("4[2]pbbb:             " + LineString(getAverageStrategy(22,
42)));
Console.WriteLine("4[2]bbpb:             " + LineString(getAverageStrategy(28,
42)));
Console.WriteLine("4[2]pbbpb:             " + LineString(getAverageStrategy(44,
42)));
Console.WriteLine("4[3]pp:             " + LineString(getAverageStrategy(3,
43)));
Console.WriteLine("4[3]bb:             " + LineString(getAverageStrategy(6,
43)));
Console.WriteLine("4[3]pbb:             " + LineString(getAverageStrategy(10,
43)));
Console.WriteLine("4[3]ppp:             " + LineString(getAverageStrategy(7,
43)));
Console.WriteLine("4[3]ppb:             " + LineString(getAverageStrategy(8,
43)));
Console.WriteLine("4[3]bbp:             " +
LineString(getAverageStrategy(13,43)));
Console.WriteLine("4[3]bbb:             " + LineString(getAverageStrategy(14,
43)));
Console.WriteLine("4[3]pppb:             " +
LineString(getAverageStrategy(16,43)));
Console.WriteLine("4[3]pbbp:             " + LineString(getAverageStrategy(21,
43)));
Console.WriteLine("4[3]pbbb:             " + LineString(getAverageStrategy(22,
43)));
Console.WriteLine("4[3]bbpb:             " + LineString(getAverageStrategy(28,
43)));
Console.WriteLine("4[3]pbbpb:             " + LineString(getAverageStrategy(44,
43)));
Console.WriteLine("4[4]pp:             " + LineString(getAverageStrategy(3,
44)));
Console.WriteLine("4[4]bb:             " + LineString(getAverageStrategy(6,
44)));
Console.WriteLine("4[4]pbb:             " + LineString(getAverageStrategy(10,
44)));
```

```
Console.WriteLine("4[4]ppp:             " + LineString(getAverageStrategy(7,
44)));
Console.WriteLine("4[4]ppb:             " + LineString(getAverageStrategy(8,
44)));
Console.WriteLine("4[4]bbp:             " + LineString(getAverageStrategy(13,
44)));
Console.WriteLine("4[4]bbb:             " + LineString(getAverageStrategy(14,
44)));
Console.WriteLine("4[4]pppb:             " + LineString(getAverageStrategy(16,
44)));
Console.WriteLine("4[4]pbbp:             " + LineString(getAverageStrategy(21,
44)));
Console.WriteLine("4[4]pbbb:             " + LineString(getAverageStrategy(22,
44)));
Console.WriteLine("4[4]bbpb:             " + LineString(getAverageStrategy(28,
44)));
Console.WriteLine("4[4]pbbpb:             " + LineString(getAverageStrategy(44,
44)));


Console.WriteLine("\n\nEV = " + (finalUtil / iterations));

}

//Função CFR iterativa
private void cfr(int[] cardsIND, int level)
{

//Ciclo que atualiza todas as probabilidades de um dado jogo
for (int i = 0; i <= level; i++)
{
int player = i % 2;
int card1 = cardsIND[player], cardRiver = cardsIND[2];
int card2 = cardsIND[1 - player];

if (card1 == 2)
{

if (cardRiver == 2)
updateLevelProbs(cardsIND, prob, strategy2, strategy22, i);
else if (cardRiver == 3)
updateLevelProbs(cardsIND, prob, strategy2, strategy23, i);
else if (cardRiver == 4)
updateLevelProbs(cardsIND, prob, strategy2, strategy24, i);
}
else if (card1 == 3)
{

if (cardRiver == 2)
updateLevelProbs(cardsIND, prob, strategy3, strategy32, i);
else if (cardRiver == 3)
updateLevelProbs(cardsIND, prob, strategy3, strategy33, i);
else if (cardRiver == 4)
updateLevelProbs(cardsIND, prob, strategy3, strategy34, i);

}
else if (card1 == 4)
{

if (cardRiver == 2)
updateLevelProbs(cardsIND, prob, strategy4, strategy42, i);
```

```
else if (cardRiver == 3)
updateLevelProbs(cardsIND, prob, strategy4, strategy43, i);
else if (cardRiver == 4)
updateLevelProbs(cardsIND, prob, strategy4, strategy44, i);

}

}

//Ciclo que atualiza todas as utilidades e arrependimentos de um dado jogo com
base nas probabilidades calculadas anteriormente
for (int i = level; i >= 0; i--)
{
int player = i % 2;
int card1 = cardsIND[player], card2 = cardsIND[1 - player], cardRiver =
cardsIND[2];

if (card1 == 2)
{
if (card2 == 2)
{
if (cardRiver == 3)
updateLevelUtil(cardsIND, util, strategy2,strategy2, strategy23,strategy23, i,
prob);
else if (cardRiver == 4)
updateLevelUtil(cardsIND, util, strategy2, strategy2, strategy24, strategy24, i,
prob);
}
else if (card2 == 3)
{
if (cardRiver == 3)
updateLevelUtil(cardsIND, util, strategy2, strategy3, strategy23, strategy33, i,
prob);
else if(cardRiver ==2)
updateLevelUtil(cardsIND, util, strategy2, strategy3, strategy22, strategy32, i,
prob);
else if (cardRiver == 4)
updateLevelUtil(cardsIND, util, strategy2, strategy3, strategy24, strategy34, i,
prob);
}
else if (card2 == 4)
{
if (cardRiver == 3)
updateLevelUtil(cardsIND, util, strategy2, strategy4, strategy23, strategy43, i,
prob);
else if (cardRiver == 2)
updateLevelUtil(cardsIND, util, strategy2, strategy4, strategy22, strategy42, i,
prob);
else if (cardRiver == 4)
updateLevelUtil(cardsIND, util, strategy2, strategy4, strategy24, strategy44, i,
prob);
}
}
else if (card1 == 3)
{
if (card2 == 2)
{
if (cardRiver == 3)
updateLevelUtil(cardsIND, util, strategy3, strategy2, strategy33, strategy23, i,
prob);
else if (cardRiver == 2)
```

```
updateLevelUtil(cardsIND, util, strategy3, strategy2, strategy32, strategy22, i,
prob);
else if (cardRiver == 4)
updateLevelUtil(cardsIND, util, strategy3, strategy2, strategy34, strategy24, i,
prob);
}
else if (card2 == 3)
{

if (cardRiver == 2)
updateLevelUtil(cardsIND, util, strategy3, strategy3, strategy32, strategy32, i,
prob);
else if (cardRiver == 4)
updateLevelUtil(cardsIND, util, strategy3,strategy3, strategy34, strategy34, i,
prob);
}
else if (card2 == 4)
{
if (cardRiver == 3)
updateLevelUtil(cardsIND, util, strategy3, strategy4, strategy33, strategy43, i,
prob);
else if (cardRiver == 2)
updateLevelUtil(cardsIND, util, strategy3, strategy4, strategy32, strategy42, i,
prob);
else if (cardRiver == 4)
updateLevelUtil(cardsIND, util, strategy3, strategy4, strategy34, strategy44, i,
prob);
}
}
else if (card1 == 4)
{
if (card2 == 2)
{
if (cardRiver == 3)
updateLevelUtil(cardsIND, util, strategy4,strategy2, strategy43, strategy23, i,
prob);
else if (cardRiver == 2)
updateLevelUtil(cardsIND, util, strategy4,strategy2, strategy42, strategy22, i,
prob);
else if (cardRiver == 4)
updateLevelUtil(cardsIND, util, strategy4, strategy2,strategy44, strategy24, i,
prob);
}
else if (card2 == 3)
{

if (cardRiver == 2)
updateLevelUtil(cardsIND, util, strategy4, strategy3, strategy42, strategy32, i,
prob);
else if(cardRiver ==3)
updateLevelUtil(cardsIND, util, strategy4, strategy3, strategy43, strategy33, i,
prob);
else if (cardRiver == 4)
updateLevelUtil(cardsIND, util, strategy4, strategy3, strategy44, strategy34, i,
prob);
}
else if (card2 == 4)
{
if (cardRiver == 3)
updateLevelUtil(cardsIND, util, strategy4, strategy4, strategy43, strategy43, i,
prob);
```

```
else if (cardRiver == 2)
updateLevelUtil(cardsIND, util, strategy4, strategy4, strategy42, strategy42, i,
prob);

}
}

}
//Somatório dos resultados dos jogos
finalUtil += util[0];
}


}

public static string LineString(double[] arr)
{
var str = "";
foreach (var i in arr)
{
str += i + " ";
}
return str;
}

static void Main(string[] args)
{
int it = 1000000;
IterativeCFR trainergpu = new IterativeCFR();
swGPU.Start();
trainergpu.trainGPU(it);
swGPU.Stop();
long memory = GC.GetTotalMemory(true);

Console.WriteLine("Time : {0}", swGPU.Elapsed.TotalSeconds);
Console.WriteLine("Memory: " + memory / 1024);
Console.Read();
}

}
```