Universidade Federal De Santa Catarina
Centro Tecnológico
Bacharelado em Ciências da Computação

“Algoritmos para um jogador inteligente de Poker”

Autor: Vinícius Sousa Fazio

Florianópolis/SC, 2008

Universidade Federal De Santa Catarina
Centro Tecnológico
Bacharelado em Ciências da Computação

“Algoritmos para um jogador inteligente de Poker”

Autor: Vinícius Sousa Fazio

Orientador: Mauro Roisenberg

Banca: Benjamin Luiz Franklin

Banca: João Rosaldo Vollertt Junior

Banca: Ricardo Azambuja Silveira

Florianópolis/SC, 2008

AGRADECIMENTOS

Agradeço a todos que me ajudaram no desenvolvimento deste trabalho, em especial ao professor Mauro Roisenberg e aos colegas de trabalho João Vollertt e Benjamin Luiz Franklin e ao Ricardo Azambuja Silveira pela participação na banca avaliadora.

RESUMO

Poker é um jogo simples de cartas que envolve aposta, blefe e probabilidade de vencer. O objetivo foi inventar e procurar algoritmos para jogadores artificiais evolutivos e estáticos que jogassem bem poker.

O jogador evolutivo criado utiliza aprendizado por reforço, onde o jogo é dividido em uma grande matriz de estados com dados de decisões e recompensas. O jogador toma a decisão que obteve a melhor recompensa média em cada estado. Para comparar a eficiência do jogador, várias disputas com jogadores que tomam decisões baseados em fórmulas simples foram feitas.

Diversas disputas foram feitas para comparar a eficiência de cada algoritmo e os resultados estão demonstrados graficamente no trabalho.

Palavras-Chave: Aprendizado por Reforço, Inteligência Artificial, Poker;

SUMÁRIO

LISTA DE FÓRMULAS

LISTA DE TABELAS

LISTA DE GRÁFICOS

LISTA DE FIGURAS

1. INTRODUÇÃO

Ninguém sabe ao certo a origem do Poker (WIKIPEDIA POKER, 2008). O jogo mais antigo conhecido que tenha as características de blefe, "vence quem tem a melhor mão" e aposta é do século XV e de origem alemã, chamado *Pochspiel*. Há registros de pessoas jogando um jogo chamado *As Nas* no século XIX, que é muito parecido com poker e usa vinte cartas. Alguns historiadores descordam da origem do poker como uma variação do As Nas e acreditam vir de um jogo francês chamado *Poque.* O ator inglês Joseph Crowell relatou que o jogo era jogado em New Orleans em 1829, com um baralho de vinte cartas e quatro jogadores apostando qual "mão" era a mais valiosa. Logo após isso, o baralho inglês de 52 cartas foi utilizado e durante a Guerra Civil Americana as regras ficaram mais parecidas com as que são utilizadas hoje. Em torno de 1925 começou-se a jogar o poker com cartas comunitárias, que é a modalidade usada neste trabalho. O poker e seus jargões fazem parte da cultura americana. Em 1970 começaram os campeonatos mundiais de poker nos Estados Unidos. Em 1987, o poker com cartas comunitárias foram introduzidos em cassinos da Califórnia e se tornaram a modalidade de poker mais popular até hoje. Em 1998, um filme com o tema de poker chamado "Cartas na Mesa", em inglês "Rounders", foi lançado nos Estados Unidos.

Poker é um jogo de risco ou blefe. O espírito do jogo é conseguir convencer o adversário que o seu jogo é mais forte que o dele e tentar adivinhar se o jogo dele é mais forte que o seu. O convencimento é através de apostas. Se você não apostar que seu jogo é melhor que o do seu adversário, o seu adversário vence sem precisar mostrar o jogo. Também é conhecido como jogo de mentiroso, assim como o "truco" no Brasil.

As regras do jogo são bem simples e o jogador não possui informações suficientes para tomar a decisão ótima. Ou seja, é um jogo de informação incompleta e essa é uma característica de muitos problemas relacionados a área de Inteligência Artificial.

Um jogador com muita sorte consegue ganhar todos os jogos sem precisar tomar decisões inteligentes, mas esses casos são raros. Para a maioria das jogadores, Poker envolve alguma decisão inteligente. Isso explica porque bons jogadores de poker ganham mais que jogadores

medianos e porque existem jogadores de Poker profissionais. Algumas pessoas afirmam que Poker é um jogo puramente de sorte mas para outras, que escreveram muitos livros a respeito, como David Sklansky, não. Existem muitos livros e artigos publicados a respeito, dezenas de softwares sendo comercializados em que o único propósito é tomar decisões inteligentes em Poker e muitos sites de Poker Online proíbem o uso destes softwares por dar muita vantagem aos jogadores que os utilizam.

O objetivo deste trabalho é procurar e formular algoritmos que tomem decisões inteligentes em poker, especificamente na modalidade Poker Texas Hold'em, que é a modalidade mais popular de Poker. Para esse objetivo ser cumprido foi necessário implementar diversos algoritmos e como resultado foi mostrado neste trabalho cinco diferentes estratégias. A primeira estratégia foi um jogador aleatório, que decide qualquer coisa sem levar em conta o estado do jogo. A segunda foi o jogador constante, que toma uma decisão constante, também sem se importar com o estado do jogo. A terceira estratégia, nomeada MFS, foi utilizar uma fórmula que faz uma relação linear com algumas informações do jogo, que são: com a quantidade de dinheiro apostado, a quantidade de dinheiro que possa vir a receber caso vença, a probabilidade de vencer e a probabilidade de perder. A quarta estratégia, nomeada RH, foi também utilizar outra fórmula que é uma relação linear de outras informações do jogo, que são: quantidade de dinheiro que receberá caso vença, a quantidade de jogadores que ainda vão tomar alguma decisão, a quantidade de jogadores não fugiram, a quantidade de dinheiro que precisa pagar para continuar no jogo e a quantidade de vezes que alguém aumentou a aposta. A última estratégia, nomeada jogador de aprendizado por reforço, foi armazenar milhões de decisões e recompensas em diferentes estados do jogo através de simulações e, quando for tomar uma decisão, consultar essa base de dados por todas as decisões feitas em um estado semelhante e tomar a decisão que obteve o melhor resultado, na média. Esta última estratégia precisava de muitas constantes para ajustar a “semelhança do estado do jogo” e para encontrar estas constantes foi utilizado Algoritmo Genético.

Para medir o desempenho dos jogadores foi feito uma comparação de quantidade de vitórias disputando as diferentes estratégias uma com a outra além de jogadores humanos testarem os jogadores artificiais. As disputas foram feitas utilizando diferentes configurações dos jogadores artificiais.

A justificativa para esse trabalho não é apenas criar um bom jogador de poker. A motivação é testar e comparar diferentes algoritmos de para jogar poker e testar um novo algoritmo de aprendizado para solucinar um problema com informação incompleta em um domínio relativamente simples que é o poker. Uma boa solução pode servir de inspiração para resolver problemas mais genéricos relacionados a barganha, que é o espírito do poker, como a bolsa de valores.

O trabalho está estruturado em cinco capítulos, sendo este - a introdução – o primeiro. O próximo capítulo fará uma revisão bibliográfica de teorias e métodos que serão usadas no trabalho. No terceiro capítulo, há todos os métodos utilizados em detalhes. O quarto capítulo mostra os resultados obtidos pelos métodos utilizados e uma interpretação destes resultados. O último capítulo contém uma revisão dos resultados obtidos e sugestões de trabalhos futuros.

2. FUNDAMENTAÇÃO TEÓRICA

A área da inteligência artificial que estuda jogos é uma das áreas mais antigas da computação. Segundo (LUGER; STUBBELFIELD, 1998):

"Game playing is also one of the oldest areas of endeavor in artificial intelligence. In 1950, almost as soon as computers became programmable, the first chess programs were written by Claude Shannon (the inventor of information theory) and by Alan Turing. Since then, there has been steady progress in the standard of play, to the point where current systems can challenge the human world champion without fear of gross embarrassment.
(...)
But what makes games really different is that they are usually much too hard to solve. Chess, for example, has an average branching factor of about 35, and games often go to 50 moves by each player, so the search tree has about 35^{100} *nodes (although there are 'only' about* 10^{40} *different legal positions). Tic-Tac-Toe (noughts and crosses) is boring for adults precisely because it is easy to determine the right move. The complexity of games introduces a completely new kind of uncertainty that we have not seen so far; the uncertainty arises not because there is missing information, but because one does not have time to calculate the exact consequences of any move. Instead, one has to make one's best guess based on past experience, and act before one is sure of what action to take. In this respect, games are much more like the real world than the standard search problems we have looked at so far.*

Jogar é uma das áreas de esforço da inteligência artificial. Em 1950, logo após os computadores se tornarem programáveis, o primeiro programa de xadrez foi escrito por Claude Shannon (o inventor da teoria da informação) e por Alan Turing. Desde então, tem ocorrido um sólido progresso nos padrões de jogar, ao ponto em que os sistemas atuais podem desafiar campeões mundiais humanos sem medo de constrangimento.
(...)
Mas o que faz os jogos realmente diferentes [de outros problemas da inteligência artificial] é que eles são geralmente muito mais difíceis de resolver. Xadrez, por exemplo, tem um fator de expansão de 35, e jogos frequentemente vão a 50 movimentos por jogador, então a árvore de

pesquisa tem aproximadamente 35^{100} nodos (apesar de existir 'apenas' 10^{40} diferentes posições permitidas). Jogo da velha é entediante para adultos precisamente porque é fácil determinar o movimento correto. A complexidade dos jogos introduz um tipo de incerteza completamente novo que não vimos ainda; a incerteza acontece não por falta de informação mas porque não há tempo de calcular a exata consequência de qualquer movimento. Ao invés disso, alguém deve fazer o melhor tentativa baseado em experiência passada, e agir antes de ter certeza de qual ação tomar. Dessa maneira, jogos são muito mais como o mundo real que os problemas de busca comuns que vimos até agora.“.

Portanto, jogos são um grande estímulo na área de inteligência artificial para descobrir novas soluções para problemas que podem ser generalizados e aplicado para resolver outros problemas do cotidiano já que os problemas do cotidiano tem características de problemas de jogos. Decidir o preço de um produto é um exemplo de um problema que poderia ser resolvido através de uma solução semelhante à solução do Poker porque ambos precisam tomar decisões que podem ser rentáveis ou não e ambos contêm informações incompletas.

2.1. Regras do Poker Texas Hold'em

Poker é um jogo com regras simples e isso foi um dos motivos de ter se tornado tão popular (POKERLOCO, 2008). O objetivo do jogo é ganhar mais dinheiro. Todos os jogadores apostam dinheiro e o jogador que tiver o melhor jogo no final recebe todo o dinheiro. O jogo deve ter no mínimo dois jogadores e é possível até vinte e dois jogadores. O aconselhável é até dez jogadores.

Existem diversas modalidades de aposta, divididas em dois grandes grupos: **Limit** e **No-Limit**. A modalidade Limit tem algum tipo de limite pré-estabelecido de aposta enquanto a No-Limit a aposta pode ir até a quantidade de dinheiro disponível para o jogador apostar.

No começo do jogo, cada jogador recebe duas cartas que somente o dono das cartas pode ver. Um jogo é dividido em quatro rodadas. Em todas as rodadas, o primeiro a jogar é o jogador à esquerda de quem distribuiu as cartas. O segundo a jogar é a esquerda do primeiro a jogar, e assim sucessivamente. O que fazer em cada rodada é explicado a seguir.

*2.1.1. Primeira rodada – **Pre-flop***

O primeiro a jogar é o **Small Blind** que significa que ele é obrigado a apostar a aposta mínima. O valor da aposta mínima é combinada entre os jogadores antes de começar o jogo. Segue o jogo para o segundo a jogar, a esquerda do último que jogou. Ele é o **Big Blind** que significa que ele é obrigado a apostar o dobro da aposta mínima. Segue o jogo para o próximo a jogar, a esquerda do último a jogar. Agora ele tem três alternativas:

1. Fugir - **Fold**: Ele abandona o jogo, não tendo direito a nenhum prêmio e perde o que já apostou. Além disso, esse jogador não pode mais tomar nenhuma decisão no jogo.

2. Continuar no jogo - **Call** ou **Check**: Para continuar no jogo, ele tem que ter apostado a mesma quantidade de dinheiro que o jogador que mais apostou. O termo Call é quando você precisa pagar alguma coisa para continuar no jogo. O termo Check é quando você pode continuar no jogo sem pagar nada. *Por exemplo: se o primeiro jogador apostou 5, o segundo 10, o terceiro, para continuar no jogo, precisa apostar 10. Nesse caso é um Call.*

3. Aumentar a aposta - **Raise** ou **Bet**: Para aumentar a aposta, ele deve apostar o necessário para continuar no jogo e mais o que ele quiser aumentar. O termo Bet é quando você não precisa pagar nada para aumentar a aposta. O termo Raise é quando você precisa pagar alguma coisa antes de aumentar a aposta. *Por exemplo: se o primeiro jogador apostou 5, o segundo 10, o terceiro deve apostar mais que 10 para estar aumentando a aposta. Nesse caso é um Raise.* Existe um tipo especial de aumento de aposta chamado **All-In.** Acontece quando o jogador aposta tudo o que tem. Ele pode usar esse recurso quando a quantidade de dinheiro que ele precisa apostar para continuar no jogo é maior que a quantidade de dinheiro disponível para ele. Apostando tudo, ele pode continuar no jogo, mas a distribuição do prêmio é diferenciada. Será explicada na sessão 'Vencedor'.

Todos os jogadores tem a vez de jogar, em ordem. Toda vez que algum jogador aumenta a aposta, o jogo continua em ordem até que todos tenham pagado a aposta ou fugido. Toda vez que um jogador vai pagar a aposta, ele pode tomar qualquer uma das três decisões acima, que

é fugir, pagar a aposta ou aumentar ainda mais a aposta, fazendo com que todos tenham que pagar novamente esse aumento de aposta. Se nenhum jogador precisar pagar nada para continuar no jogo e todos os jogadores já jogaram, a rodada acaba. Senão, segue o próximo jogador.

Exemplo da primeira rodada: Um jogo de 4 jogadores. O primeiro é Small Blind e apostou 5. O segundo é "Big Blind" e apostou 10. O terceiro decidiu continuar no jogo: apostou 10. O quarto decidiu aumentar a aposta: apostou 30. O primeiro, para continuar no jogo deve apostar 25 (como ele já apostou 5 no small blind, 5 + 25 = 30). Ele aposta 40, totalizando uma aposta de 45. O segundo, para continuar no jogo deve apostar 35 (10 + 35 = 45). Ele decide fugir, deixando os 10 que ele já apostou na mesa. O terceiro deve apostar 35 para continuar no jogo (10 + 35 = 45). Ele decide continuar no jogo, apostando 35. O quarto deve apostar 15 para continuar no jogo (30 + 15 = 45). Ele decide aumentar, apostando 20, totalizando 50 (30 + 20 = 50). O primeiro, para continuar no jogo, deve apostar 5 (45 + 5). Ele aposta 5, continuando no jogo. O segundo já fugiu e não tem direito a nada. O terceiro deve apostar 5 para continuar no jogo. Ele aposta 5, continuando no jogo. Como ninguém precisa pagar mais nada para continuar no jogo, vai para a próxima rodada

*2.1.2. Segunda rodada – **Flop***

No começo da rodada, o jogador que deu as cartas coloca três cartas do baralho na mesa para que todos possam ver.

O jogador à esquerda de quem deu as cartas tem as mesmas três alternativas da primeira rodada. Como ninguém apostou nada ainda, se ele quiser continuar no jogo ele não precisa pagar nada, fazendo um Check. Os critérios para acabar a rodada são os mesmos da primeira rodada. O primeiro a aumentar a aposta fará um Bet, pois ele estará aumentando a aposta sem precisar pagar nada para continuar no jogo. O resto da rodada segue como na primeira rodada.

*2.1.3. Terceira rodada – **Turn***

No começo da rodada, o jogador que deu as cartas coloca mais uma carta do baralho na mesa

para que todos possam ver. Segue o jogo como no Flop.

*2.1.4. Quarta rodada - **River***

No começo da rodada, o jogador que deu as cartas coloca mais uma carta do baralho na mesa para que todos possam ver. Segue como no Turn.

*2.1.5. Fim do jogo - **Showdown***

Todos os jogadores que não fugiram mostram suas cartas e declaram o seu jogo. Seu jogo é o melhor jogo formado por cinco cartas das sete disponíveis (cinco cartas da mesa mais duas cartas que recebeu no começo do jogo).

*2.1.6. **Rankings***

O ranking é o valor de um conjunto de cinco cartas. Quase todos os Ranking dependem da "maior carta". A ordem das cartas são, em ordem decrescente: A, K, Q, J, 10, 9, 8, 7, 6, 5, 4, 3, 2, A. O 'A' aparece duas vezes porque ele é considerado a menor carta somente no caso de 'seqüência' de A, 2, 3, 4, 5. O ranking 'seqüência' será explicado abaixo. O jogador com o melhor ranking vence o jogo. Os rankings, em ordem crescente, são:

I. Maior carta - **High Cards**: Cinco cartas diferentes. A maior carta vence. *Por exemplo: K,8,2,4,5 é mais forte que Q,10,J,8,7.* No caso de empate de maior carta, a segunda maior, e assim sucessivamente. *Por exemplo: J,7,6,5,3 é mais forte que J,7,6,5,2.* O número de combinações possíveis desse ranking é 1.302.540.

II. Um par - **One Pair**: Par é duas cartas com o mesmo número e três cartas diferentes. *Por exemplo: K,K,5,2,3 é par de K. 5,5,A,6,3 é par de 5.* No caso de empate de par, o par maior ganha. *Por exemplo: K,K,4,2,8 é mais forte 5,5,A,K,Q.* No caso de empate de par, a maior carta das 3 que restaram vence. Em caso de empate da maior, a segunda maior, e assim sucessivamente. *Por exemplo: 5,5,K,7,3 é mais forte que 5,5,Q,J,10. 7,7,K,J,3 é mais forte que 7,7,K,J,2.* O número de combinações possíveis desse ranking é 1.098.240.

III. Dois pares - **Two Pairs**: Dois pares e uma carta diferente. *Por exemplo: K,K,5,5,3 é par de K e 5. 5,5,A,A,3 é par de 5 e A.* No caso de empate de dois pares, o par maior ganha. No caso de empate de maior par, o segundo maior par ganha. *Por exemplo: K,K,4,4,8 é mais forte K,K,3,3,Q.* No caso de empate de dois pares, a maior carta da carta que sobrou vence. *Por exemplo: 5,5,3,3,4 é mais forte que 5,5,3,3,2.* O número de combinações possíveis desse ranking é 123.552.

IV. Trio - **Three Of A Kind**: Trio é três cartas com o mesmo número e duas cartas diferentes. *Por exemplo: J,J,5,J,3 é trio de J. A,A,A,4,3 é trio de A.* No caso de empate de trio, o trio maior ganha. *Por exemplo: A,A,A,4,8 é mais forte K,K,K,3,Q.* No caso de empate de trio, a maior carta das 2 que sobraram vence. No caso de empate, a segunda maior carta que sobrou vence. *Por exemplo: 5,5,5,7,4 é mais forte que 5,5,5,7,2.* O número de combinações possíveis desse ranking é 54.912.

V. Seqüência – **Straight**: Seqüência são cinco cartas na seqüência de número. *Por exemplo: A,2,3,4,5 é seqüência de 5. 7,8,9,10,J é seqüência de J.* No caso de empate de seqüência, a seqüência com a maior carta vence. *Por exemplo: 3,4,5,6,7 vence de A,2,3,4,5. O A é considerado a menor carta somente no caso de seqüência de A, 2, 3, 4, 5.* O número de combinações possíveis desse ranking é 10.200.

VI. Naipes iguais – **Flush**: Cinco cartas diferentes com naipes iguais. *Por exemplo: 3, 7, 2, 6, 8 de copas é flush de 8. 6, 9, 10, K, 2 de paus é flush de 8.* No caso de empate de flush, a maior carta vence. No caso de empate de maior carta, a segunda maior, e assim sucessivamente. *Por exemplo: A, 7, 5, 4, 3 é maior que K,K, J, Q, 10. Note que o último jogo tem também um par. Mas como o flush é maior que o par, o par é ignorado.* O número de combinações possíveis desse ranking é 5.108.

VII. Trio e par - **Full House**: Três cartas com o mesmo número e duas com o mesmo número. *Por exemplo: K,K,K,J,J é full house de K. 5,5,5,Q,Q é full house de 5.* No caso de empate de full house, o full hose com o maior trio vence. No caso de empate de trio, o maior par vence. *Por exemplo: 5,5,5,2,2 vence de 4,4,4,K,K. 8,8,8,A,A vence de 8,8,8,K,K.* O número de

combinações possíveis desse ranking é 3.744.

VIII. Quadra - **Four Of A Kind**: Quatro cartas com o mesmo número e uma carta diferente. *Por exemplo: K,K,K,K,J é quadra de K. 5,5,5,5,Q é quadra de 5.* No caso de empate de quadra, a quadra maior vence. *Por exemplo: 5,5,5,5,2 vence de 2,2,2,2,K.* No caso de empate de quadra, a maior carta restante vence. *Por exemplo: 7,7,7,7,8 vence de 7,7,7,7,6* O número de combinações possíveis desse ranking é 624.

IX. Sequência com naipes iguais - **Straight Flush**: O maior ranking do poker. Cinco cartas em seqüência com o mesmo naipe. *Por exemplo: 3,4,5,6,7 de ouro é um Straight Flush de 7. 10, J, Q, K, A de espada é um straight flush de A, também conhecido como royal straight flush, que é o maior jogo possível do poker.* No caso de empate de straight flush, a maior carta vence. *Por exemplo: 7,8,9,10,J de ouro é maior que 6,7,8,9,10 de paus.* O número de combinações possíveis desse ranking é 36.

2.1.7. Vencedor

Se sobrar apenas um jogador quando todos os jogadores fugirem, o jogador que não fugiu, em qualquer rodada, vence o jogo. Senão, o jogador com o jogo mais forte vence. Toda aposta é colocada em um monte comum chamado **Pot.** O jogador que vence recebe o Pot. Em caso de empate, os vencedores dividem todo o dinheiro Pot. Em caso de algum jogador ter apostado tudo o que tinha, ou seja all-in, o pot é dividido. Em um pot fica apenas o dinheiro que foi apostado até o all-in. Em outro monte fica o restante, em aposta separada.

Por exemplo: O jogador A aposta 50, o jogador B tem apenas 30 e anuncia 'all-in' apostando todo o seu dinheiro, que é 30. Nesse momento, o pot é dividido. O primeiro pot fica todas apostas até o momento mais 30. No segundo pot ficará as apostas que o jogador B não pode apostar, ou seja, 50-30 = 20. O jogador C quer continuar no jogo, então ele coloca 30 no primeiro pot e 20 no segundo pot. Digamos que o jogador B venceu. Ele leva todo o pot que ele tinha direito, ou seja, o primeiro pot. O segundo pot, que tem 40, é disputado pelos jogadores que apostaram nele, ou seja jogador A e C. Dentre os dois, digamos que A tem o melhor jogo. Então A ganha o segundo pot.

2.2. Análise do Poker

Poker é um jogo de informação incompleta. Isso significa que você precisa tomar decisões com informações que são insuficientes para garantir que você tomou a melhor ou a pior decisão. Jogos de informação completa seriam o xadrez, a damas ou o jogo da velha: todas as informações para fazer a melhor jogada estão disponíveis para os jogadores. Os jogadores de xadrez só não tomam a melhor decisão possível no jogo porque exigiria muitos cálculos, mas todas as informações para fazê-lo estão disponíveis. No Poker, é impossível garantir que a decisão tomada é a melhor decisão possível devido a informação incompleta, mas pode ser possível tomar mais decisões boas que ruins. Um grande jogador de poker chamado Sklansky escreveu o Teorema Fundamental do Poker de Sklansky (SKLANSKY, 1995):

"Sklansky's Fundamental Theorem of Poker:
Every time you play a hand differently from the way you would have played it if you could see all your opponents' cards, they gain; and every time you play your hand the same way you would have played it if you could see all their cards, they lose. Conversely, every time opponents play their hands differently from the way they would have if they could see all your cards, you gain; and every time they play their hands the same way they would have played if they could see all your cards, you lose.

Teorema Fundamental do Poker de Sklansky:

Toda vez que você joga uma mão diferentemente da maneira que você jogaria caso pudesse ver as cartas de seus oponentes, eles ganham. E toda vez que você joga sua mão do mesmo jeito que você teria jogado caso pudesse ver todas as cartas de seus oponentes, eles perdem. Inversamente, toda vez que seus oponentes jogam suas mãos diferentemente da maneira que eles teriam jogado caso pudessem ver todas suas cartas, você ganha, e toda vez que eles jogam suas mãos da mesma maneira que eles teriam jogado caso pudessem ver todas suas cartas, você perde. "

Ou seja, se você tivesse a informação completa em poker, você sempre venceria. E se o

adversário tivesse a informação completa, ele venceria. A melhor forma de vencer, portanto, é tentar obter informação completa e dificultar o máximo o adversário de obter essa informação.

As maneiras de se obter mais informação são através de estatística, através de subjetividade e através de trapaças. Criar um jogador artificial que colete informações subjetivas é extremamente complexo, só é válida quando se joga contra humanos e mesmo assim pode ser inviável. Seria criar um algoritmo que tivesse empatia de como está o jogo do adversário ou percebesse que o adversário está mentindo, nervoso, muito alegre, com medo, desatento ou outras emoções através de uma entrada de vídeo e som. É realmente complexo porque muitos seres humanos não conseguem distinguir essas informações, muito menos criar algoritmos que as identifique, não sejam enganados e saiba como lidar com elas. É inviável obter esse tipo de informação quando se joga na internet, onde não se tem acesso a um vídeo nem a voz dos adversários. A solução de trapaça é uma solução ilegítima pois quebra as regras do jogo. A solução de obter informações estatísticas seria utilizar a experiência de muitos jogos, a experiência do comportamento mais comum de determinado jogador para saber a probabilidade de blefar ou não e informações concretas como as cartas, as cartas na mesa, o dinheiro apostado na mesa, os jogadores que ainda estão em jogo, a quantidade de jogadores, dinheiro necessário para continuar no jogo.

2.3. Jogadores Artificiais Estáticos

Jogadores que tomam decisões baseado em uma função imutável são os jogadores estáticos. São simples e são ótimas referências para comparação com jogadores treinados.

2.3.1. Aleatórios

Esse jogador é um dos mais simples possíveis. Simplesmente jogam aleatoriamente, não se importando com o que está acontecendo no jogo.

2.3.2. Constantes

Esses também são bem simples. Eles tomam sempre a mesma decisão, não se importando com o que está acontecendo no jogo.

*2.3.3. Mathematically Fair Strategy - **MFS***

Mathematically Fair Strategy (FINDLER, 1977) é um tipo de jogador que joga justo em relação a probabilidade de vencer. Ele aposta proporcional a probabilidade e ao POT. A relação MFS é obtido por V pela fórmula:

$$V = A \cdot W - L \cdot J$$

Fórmula 1: relação MFS

Onde W é a probabilidade do jogador vencer, L é a probabilidade do jogador perder, A é a soma de aposta feita pelos adversários e J é a soma de aposta feita pelo jogador. Se V for negativa, a decisão a ser tomada é FOLD. Se V for próxima de zero, a decisão a ser tomada é CALL / CHECK. Se V for positiva, a decisão a ser tomada é RAISE / BET proporcional a V

*2.3.4. Jean Rachlin e Gary Higgins **- RH***

Esse tipo de jogador (FINDLER, 1977) foi gerado a partir de um estudo estatístico de dois estudantes: Jean Rachlin e Gary Higgins. Eles chegaram a conclusão que na maioria dos jogos, a chance de um jogador vencer não depende das cartas em suas mãos e sim de outras variáveis, algumas diretamente proporcionais e outras inversamente proporcionais. Eles chegaram na seguinte fórmula:

$$V = \frac{P}{(C+1) \cdot L \cdot F \cdot R}$$

Fórmula 2: relação RH

Onde P é o POT, R é o RAISE, L é a quantidade de jogadores que não fugiram, F é a quantidade de jogadores que precisam tomar alguma decisão antes de acabar a rodada e C é a quantidade de vezes que alguém decidiu RAISE / BET. Se V for muito baixo, a decisão a ser tomada é FOLD. Se V for razoável, deve-se decidir CALL / CHECK. Se V for alto, deve-se

decidir RAISE / BET. Ou seja, a probabilidade de vencer é proporcional ao dinheiro apostado e inversamente proporcional as outras variáveis contidas na fórmula.

2.4. Jogadores Artificiais Evolutivos

Ao contrário de jogadores estáticos, esses jogadores aprendem com a experiência. A função de decisão é mutável de acordo com um banco de dados de experiência. Essa experiência pode ter informações sobre um jogador específico, que é uma boa estratégia partindo da hipótese de que jogadores de poker tendem a jogar da mesma maneira. O banco pode ter também informações sobre situações de jogos em que o jogador adversário não é levado em conta na função de decisão, o que é o torna um jogador mais completo e genérico.

2.4.1. Aprendizado por Reforço

Aprendizado por reforço (SUTTON; BARTO, 2008, WIKIPEDIA REINFORCEMENT LEARNING, 2008) é uma forma de programar agentes para aprender baseado em recompensas. É um tipo de aprendizado não supervisionado. No aprendizado supervisionado, o agente tem uma série de exemplos de ações e suas conseqüências que o agente deve usar para aprender. No aprendizado não supervisionado, o *agente* toma decisões aleatórias e registra a reação do meio ambiente, as recompensas. Em alguns casos, as recompensas a longo prazo devem ser consideradas. Depois de alguma experiência com o meio ambiente, o agente pode tomar decisões baseado nas recompensas. Se em determinada situação o agente obteve bastante recompensa ao tomar determinada decisão, ele tomará essa decisão.

Existem dois tipos de aprendizados por reforço basicamente: o aprendizado passivo e o ativo. No aprendizado passivo, o agente sempre toma decisões aleatórias no período de aprendizado para obter bastante informação. Esse tipo de aprendizado normalmente demora mais para convergir para um bom resultado mas normalmente converge para resultado melhores. No aprendizado ativo, o agente aprende utilizando o conhecimento já adquirido para tentar agir bem desde o começo. Nesse tipo de aprendizado, o agente normalmente converge mais rápido para uma boa solução mas normalmente não converge para uma solução tão boa quanto o aprendizado passivo.

2.4.2. Algoritmo Genético

Algoritmo genético é um método de busca da computação inspirado na genética. O algoritmo genético exige uma função de fitness, que é uma função que informa o quão perto do melhor resultado aquela entrada está. Depois de definida a função é gerado uma população inicial, que é normalmente gerada aleatoriamente. A população é um conjunto de indivíduo, que é uma entrada para a função de fitness. O próximo passo é avaliar toda a população na função de fitness porque os melhores tendem a ser escolhidos para a reprodução, através do algoritmo de seleção. A reprodução é pegar dois indivíduos da população e gerar um novo indivíduo com uma parte dos valores de cada um dos dois indivíduos através do algoritmo de reprodução. Na reprodução pode haver mutação, que é um valor aleatório colocado no indivíduo gerado que não era de nenhum dos dois indivíduos geradores. A população segue reproduzindo até encontrar um indivíduo com um valor aceitável na função de fitness ou até que outra condição qualquer aconteça, como um limite na quantidade de reproduções seja atingido, e isso é a condição de parada (HOLLAND, 1972, RUSSEL; NORVIG, 1995).

As implementações variam pela escolha de representação de um indivíduo, função de fitness, algoritmo de reprodução, probabilidade de mutação, algoritmo de seleção e a condição de parada. Em geral, o algoritmo genético consegue bons resultados com uma busca muito menor que uma busca aleatória ou uma busca de por todos os casos.

2.5. Probabilidade de Vencer

Calcular a probabilidade de vencer em poker exige um método porque não é um cálculo intuitivo. É essencial que o método seja ao mesmo tempo preciso e rápido de calcular porque no treinamento de jogadores evolutivos é necessário fazer esse cálculo milhares e até milhões de vezes. Não foi encontrado um algoritmo ótimo para esse cálculo. Considerando, por exemplo, o flop. Três cartas na mesa, sete adversários. Para calcular a chance de vencer é nesssário considerar quais cartas podem ver ainda no turn e no river além das cartas de cada um dos sete jogadores.

2.5.1. Força Bruta

Esse método é preciso ao extremo e lento ao extremo também. Ele simplesmente considera todas as possibilidades e verifica a taxa de vitórias. No flop com sete adversários, a quantidade de possibilidades, por permutação, é:

$$\frac{47\cdot46\cdot45\cdot44\cdot43\cdot42\cdot41\cdot40\cdot39\cdot38\cdot37\cdot36\cdot35\cdot34\cdot33\cdot32\cdot31\cdot30}{2\cdot2\cdot2\cdot2\cdot2\cdot2\cdot2\cdot7\cdot6\cdot5\cdot4\cdot3\cdot2\cdot1} \simeq 10^{24}$$

Fórmula 3: exemplo de quantidade de combinações em um jogo no flop com sete adversários

Considerando que, segundo experimentos bem otimizados, um computador comum consegue calcular cerca de mil probabilidades de vencer por segundo, e supondo que fosse utilizado um bilhão de computadores em paralelo para fazer o cálculo, e cada vez que fosse necessário calcular por força bruta a probabilidade de vencer, seria necessário 31 anos para apenas um cálculo. Por isso, essa solução é inviável.

2.5.2. Monte Carlo

Esse método (WIKIPÉDIA MÉTODO DE MONTE CARLO, 2008) é parecido com o método força bruta com a vantagem de ser extremamente rápido. É um método estatístico para aproximar funções complexas. Utilizando-a na probabilidade de vencer, ao invés de testar todas as possibilidades por força bruta até chegar na proporção exata de vitórias e derrotas, é feito apenas alguns testes aleatórios até que a proporção de vitórias e derrotas se estabilize e varie até uma porcentagem tolerante. Segundo experimentos, uma simulação de apenas mil jogos tem como resultado uma probabilidade de vencer com uma margem de erro de menos de cinco porcento.

2.6. Outras Soluções

Muitos jogadores artificiais de poker, também conhecidos como pokerbots, já foram desenvolvidos. Existem até torneios (JOHANSON, 2007) de pokerbots, como o 2007 AAAI Computer Poker Competition No-Limit event ocorrido no ano de 2007. *Por exemplo Vexbot é*

um pokerbot desenvolvido pelo University of Alberta Computer Poker Research Group, que utiliza o algoritmo minimax (POKERAIWIKI, 2008). *Hyperborean07, outro bot, foi o vencedor do 2006 AAAI Computer Poker Competition No-Limit event, e utiliza* ϵ *- Nash Equilibrium, que é baseado em Nash-Equilibrium. O Nash-Equilibrium é um conjunto de estratégias utilizadas por cada jogador em que ninguém tenha vantagem em mudar (NISAN et al., 2007).*

Segue a lista de alguns pokerbots disponíveis no mercado, gratuitos ou pagos, acessado 08 de setembro de 2008 (POKERAIWIKI, 2008):

- Advanced Poker Calculator: *http://www.prohibitedpoker.com*
- BluffBot: *http://www.bluffbot.com/online/play.php*
- GoldBullion: *http://www.cbloom.com/poker/GoldBullion.html*
- Holdem Hawk: *http://www.holdemhawk.com*
- Holdem Pirate: *http://www.holdempirate.com*
- Holdem Inspector a.k.a. Online Holdem Inspector: *http://www.pokerinspector.com*
- Magic Holdem: *http://www.magicholdem.com*
- Mandraker: *http://www.cheaplabs.com/uk/Softwares/Mandraker.htm*
- Open Holdem Bot: *http://open-holdem-bot.qarchive.org*
- PokerAnalytics: *http://www.pokeranalytics.org*
- Poker Android: *http://www.pokerandroid.com*
- Poker Bot+: *http://pokerbotplus.com*
- Poker Bloodhound: *http://www.thepokerhound.com*
- Poker Crusher: *http://www.pokercrusher.com*
- Poker Edge: *http://www.poker-edge.com*
- Poker Inspector a.k.a. Online Poker Inspector: *http://www.pokerinspector.com*
- Poker Mate: *http://www.eruditesys.com/PokerMate/index.html*
- Poker Prophecy: *http://www.pokerprophecy.com*
- Poker Sherlock: *http://www.pokersherlock.com*
- Sit n' Go Brain: *http://www.pokerbrains.net/sitandgobrain.html*
- Texas Holdem Bot: *http://www.texas-holdem-bot.com*
- WinHoldem: *http://www.winholdem.net*

3. METODOLOGIA

Foi desenvolvido um programa de jogo de poker na modalidade no-limit que aceita jogadores genéricos, que pode ser um jogador humano – que pega as entradas do teclado, quatro jogadores estáticos – constante, aleatório, MFS e RH, e um jogador evolutivo – aprendizado por reforço. Todos esses jogadores jogam entre si milhares de vezes e de diversas formas possíveis: *mesas* entre 2 a 10 jogadores e com jogadores aleatoriamente escolhidos tanto no treinamento como na verificação de desempenho dos jogadores. Em todos os jogos, o valor de dinheiro inicial foi 1000 e o valor do small blind foi 10.

3.1. Ferramenta e linguagem

Todos os algoritmos foram implementados em linguagem **MatLab**® e **C**. A opção por MatLab foi pela facilidade em trabalhar com matrizes, que foi amplamente utilizado. A opção por C foi para implementar os algoritmos críticos, que exigem grande desempenho, que são o de cálculo de probabilidade de vencer e o treinamento do jogador de aprendizado por reforço. Informações sobre a linguagem MatLab pode ser encontrado em (MATLAB, 2008) e sobre a linguagem C pode ser encontrado em (ECKEL, 2000).

3.2. Treinamento

O treinamento dos jogadores deveria ser rápido para se tornar viável a tentativa de diversas formas de treinamento. A estratégia utilizada para o treinamento com este objetivo foi dividir em duas etapas onde a primeira etapa é feita apenas uma vez e a segunda etapa, que necessita de muitos ajustes, é feita muitas vezes.

A primeira etapa foi registrar um histórico de jogos, decisões e recompensas. Nessa base foi registrado todos as decisões de jogadores com diferentes métodos em vários jogos e a recompensa daquela decisão, que só é preenchida no final do jogo e é replicada para todas as ações que levaram àquele recompensa. A informação contida nessa base é a seguinte:

Nome	**Descrição**
POT	Soma da quantidade de dinheiro apostado por todos
ODDS	Relação do POT com o dinheiro apostado
RAISE	Quantidade de dinheiro que precisa usar para continuar no jogo
CHANCE	Probabilidade de vencer o jogo quando todos mostram as cartas
ROUND	Em que rodada - pre-flop, flop, turn ou river - o jogo se encontra
FOLLOWERS	Número de jogadores que ainda vão decidir na rodada atual
INGAME	Número de jogadores que não fugiram nem saíram
NPLAYERS	Número de jogadores
QTYRAISE	Quantidade de BET / RAISE feito por todos

Tabela 1: dimensões do estado do jogo

As informações contidas no estado do jogo ainda não comentadas são o ODDS, o FOLLOWERS, o INGAME, o NPLAYERS e o QTYRAISE. ODDS é apenas a relação do dinheiro do POT em relação ao dinheiro apostado, *por exemplo, um ODDS igual a 5 significa que tem no POT 5 vezes o valor apostado pelo jogador.* O FOLLOWERS indica quantos jogadores ainda vão decidir algo. Isso porque o último a decidir tem vantagem em relação ao primeiro a decidir porque o último já sabe quem aumentou e quem não. INGAME indica quantos jogadores ainda não fugiram. NPLAYERS indica quantos jogadores estão jogando, independente de ter fugido, falido ou ainda estar em jogo. O QTYRAISE indica a quantidade de vezes que algum jogador aumentou a aposta e é importante porque jogos que houveram muito aumento de aposta pode indicar que muitos jogadores estão com boas chances de vencer.

Depois de preenchida essa base de dados com milhares de decisões, foi iniciado a segunda etapa, que consiste em passar para todos os jogadores informações de treinamento que são essas informações do histórico de jogos. Os jogadores estáticos ignoram essa informação assim como os humanos. *Um exemplo de uma linha de informação dessa base de dados: primeira rodada, chance de vencer de 30%, POT de 50, ODDS de 3, é necessário pagar 20 para continuar no jogo, foi tomada a decisão de RAISE 40, recompensa de 50.*

3.3. Verificação de desempenho

Depois do treinamento foi feito a verificação do desempenho de todos os jogadores. Essa medição foi feita através da comparação de proporção de vitória de cada jogador em centenas de mesas. Uma mesa é um conjunto de jogadores que começam com a mesma quantidade de dinheiro e jogam diversas vezes até que sobre apenas um jogador com dinheiro ou até atingir um número limite de jogos, normalmente em torno de 20, o que acontecer primeiro. O vencedor da mesa é o jogador que tiver mais dinheiro no final.

3.4. Implementação dos Jogadores Artificiais

Cada jogador tem dois tipos de ações: 'tomar uma decisão' e 'treinar'. Na ação 'tomar uma decisão', cada jogador tem como entrada o **estado do jogo**. O 'estado do jogo' é multidimensional e suas dimensões são as mesmas informações guardadas na base de dados, que são: POT, ODDS, RAISE, CHANCE, ROUND, FOLLOWERS, INGAME, NPLAYERS, QTYRAISE.

Para simplificar a implementação e o treinamento dos jogadores, foi estabelecido que cada jogador tem como saída da ação 'tomar uma decisão' um valor discreto entre 1 e 6, que é a **decisão discretizada**:

Código	**Descrição**
1	Fugir
2	Continuar no jogo
3	Aumentar pouco: equivalente a 2 a 4 small blind
4	Aumentar médio: equivalente a 6 a 18 small blind
5	Aumentar muito: equivalente a 20 a 40 small blind
6	Aumentar tudo: all-in

Tabela 2: decisão discretizada

A ação 'treinar' não tem nenhum retorno pois ela apenas altera o estado interno de cada jogador. Note que jogadores humanos e estáticos não fazem nada nessa ação. A entrada dessa ação é um 'estado do jogo', uma 'decisão discretizada' e uma recompensa. A recompensa é definida por R :

$$R = F - I$$

Fórmula 4: recompensa de um jogo

Onde F é a quantidade de dinheiro que o jogador tinha quando acabou o jogo e I é a quantidade de dinheiro que o jogador tinha quando começou o jogo.

3.4.1. Aleatórios

Esses jogadores tomam a decisão discretizada aleatória na seguinte proporção:

Proporção	**Descrição**
11%	Fugir
44%	Continuar no jogo
11%	Aumentar pouco
11%	Aumentar médio
11%	Aumentar muito
11%	Aumentar tudo

Tabela 3: Distribuição da decisão do jogador aleatório

3.4.2. Constantes

Esses jogadores tomam sempre a mesma decisão discretizada.

3.4.3. Mathematically Fair Strategy - MFS

Esses jogadores tomam a decisão discretizada definida por D na fórmula:

$$se \;\; O = 0 \;\rightarrow\; D = 2$$

$$E = \frac{P \cdot (O-1)}{O}$$

$$M = P - E + R$$

$$V = \frac{C \cdot E - (100-C) \cdot M}{100}$$

$$\begin{aligned}
se \;\; O \neq 0 \;\; e \;\; V < -T \;\rightarrow\; D &= 1 \\
se \;\; O \neq 0 \;\; e \;\; -T \leqslant V < T \;\rightarrow\; D &= 2 \\
se \;\; O \neq 0 \;\; e \;\; T \leqslant V < 5 \cdot T \;\rightarrow\; D &= 3 \\
se \;\; O \neq 0 \;\; e \;\; 5 \cdot T \leqslant V < 10 \cdot T \;\rightarrow\; D &= 4 \\
se \;\; O \neq 0 \;\; e \;\; 10 \cdot T \leqslant V < 15 \cdot T \;\rightarrow\; D &= 5 \\
se \;\; O \neq 0 \;\; e \;\; V \geqslant 15 \cdot T \;\rightarrow\; D &= 6
\end{aligned}$$

Fórmula 5: decisão de um jogador MFS

Onde V é a relação MFS (Fórmula 1) e T é uma constante que foi testado vários valores. Para chegar em V foi necessário usar outra fórmula que obtém o mesmo resultado da relação MFS (Fórmula 1) porque o programa não oferece as variáveis exatamente como descrito na fórmula. E é a quantidade de dinheiro do adversário, P é o POT, O é o ODDS, R é o RAISE, M é a quantidade de dinheiro apostado pelo jogador e C é a CHANCE em porcentagem. Caso o V, que é o ODDS, seja igual a zero, significa começo de jogo onde o jogador não apostou nada ainda. Nesse caso, a decisão dele é sempre entrar no jogo.

3.4.4. Jean Rachlin e Gary Higgins – RH

Esses jogadores tomam a decisão discretizada definida por D:

$$V = \frac{P}{(C+1) \cdot F \cdot L \cdot (R+1)}$$

$$\begin{aligned}
se \;\; V < T \;\rightarrow\; D &= 1 \\
se \;\; T \leqslant V < 20 \cdot T \;\rightarrow\; D &= 2 \\
se \;\; 20 \cdot T \leqslant V < 50 \cdot T \;\rightarrow\; D &= 3 \\
se \;\; 50 \cdot T \leqslant V < 100 \cdot T \;\rightarrow\; D &= 4 \\
se \;\; 100 \cdot T \leqslant V < 1000 \cdot T \;\rightarrow\; D &= 5 \\
se \;\; V \geqslant 1000 \cdot T \;\rightarrow\; D &= 6
\end{aligned}$$

Fórmula 6: decisão de um jogador RH

Onde V é a relação RH (Fórmula 2), P é o POT, C é o número de vezes que alguém decidiu RAISE / BET, F é o número de jogadores que ainda vão jogar nesta rodada, L é o número de jogadores que não fugiram, R é o RAISE e T é uma constante que foi testado diversos valores. O RAISE foi alterado para com o "+1" para que não houvesse divisão por zero.

3.4.5. Aprendizado por Reforço

Esse jogador internamente tem matriz com o número de dimensões iguais ao número de dimensões do estado do jogo (Tabela 1). Cada dimensão tem uma quantidade de níveis e um intervalo correspondente a cada nível e o estado do jogo se enquadrará em um nível se o intervalo do estado corresponder àquele nível. *Por exemplo, se a dimensão POT da matriz tem 7 níveis e cada nível tem o intervalo de 13,3, um estado do jogo em que o POT esteja em 29 se enquadrará no 3º nível desta dimensão da matriz porque o 1º nível vai de 0 a 13,3 e o 2º nível vai de 13,3 a 26,6.* Caso o valor da dimensão do estado do jogo seja maior que o limite da matriz, esse valor é colocado no último nível. Além destas dimensões, a matriz tem mais duas dimensões, uma para indicar a decisão tomada e outra para indicar se o resultado foi prejuízo ou lucro. O conteúdo de cada célula da matriz é os resultados referentes àquele estado do jogo.

A ação 'treinar' irá colocar na matriz, na posição referente ao estado do jogo, as recompensas (Fórmula 4).

A ação 'tomar decisão' consulta a matriz na posição referente ao estado do jogo e obtém uma lista de pares 'decisão discretizada' / 'recompensa'. Com essa lista, toma-se uma decisão baseado na fórmula:

$$D = MAX(r(x))$$

Fórmula 7: decisão do jogador de aprendizado por reforço

Onde D é a decisão é a decisão do jogador, que é a decisão x que retorna o máximo valor na função $r(x)$. A função $r(x)$, que é a recompensa de uma decisão x, é definida por:

$$r(x) = \frac{\sum_{i=1}^{n} e(x)_i}{n}$$

Fórmula 8: recompensa de uma decisão

Onde $e(x)$ é o conjunto de recompensas em que a decisão tomada foi x. Portanto, o jogador decide a decisão que obteve uma melhor média de recompensas na proporção (exemplo na Figura 1).

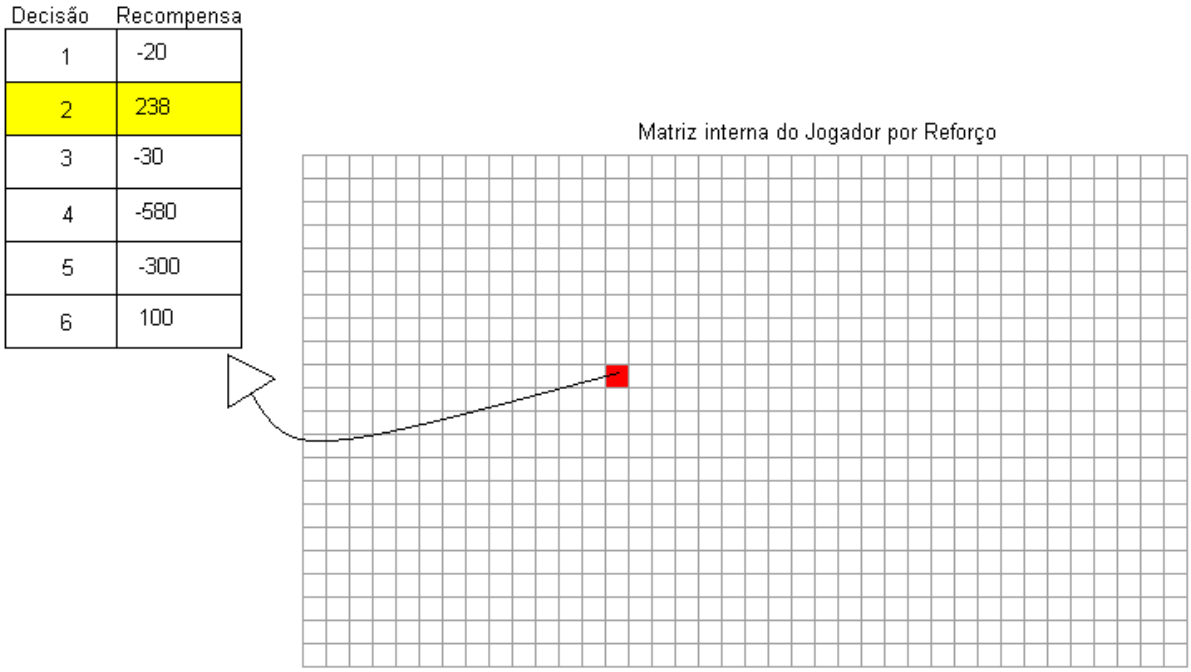

Figura 1: Exemplo de uma situação de jogo que o jogador por reforço consulta sua base de conhecimento e decide 2 que significa "continuar no jogo".

O jogador de aprendizado por reforço tem diversas constantes, diferentemente dos jogadores citados até agora que tinham no máximo 1. A matriz interna desse jogador contem nove dimensões - CHANCE, POT, RAISE, ODD, QTYRAISE, NPLAYERS, FOLLOWERS, ROUND e INGAME - que precisam de uma constante para o intervalo que cada nível da matriz representa, com exceção da dimensão ROUND, que já foi pré-definida como tamanho 2, sendo o primeiro referente a primeira rodada e o segundo referente as demais rodadas. Cada variável deveria também ter a quantidade de níveis mas, para simplificar, existe apenas duas

constantes "quantidade de níveis", uma para ODDS, POT, RAISE e CHANCE, chamado de "BIG QTY" e outra para QTYRAISE, NPLAYERS, FOLLOWERS e INGAME, chamada de "LITTLE QTY", que totaliza dez constantes. Se cada variável fossem testados apenas 5 valores diferentes, a quantidade de testes seria $5^{10} = 9.765.625$, o que impossibilita testar todos os casos. Para encontrar uma boa configuração destas constantes foi utilizado Algoritmo Genético. Os genes são essas dez constantes, onde os valores mínimo e máximo para cada uma das constantes são:

Variável	**Mínimo**	**Máximo**
CHANCE	5	25
POT	1	50
RAISE	1	25
ODD	1	10
QTYRAISE	1	10
NPLAYERS	1	10
FOLLOWERS	1	10
INGAME	1	10
BIG QTY	2	10
LITTLE QTY	2	4

Tabela 4: Limites usados pelo algoritmo genético para encontrar boas constantes para o jogador por Reforço

A função fitness é a soma da quantidade vitórias diversas disputas, cada disputa com dez *mesas,* com o Jogador por Reforço usando 10 milhões de dados preenchidos na sua matriz interna. Foram usadas sete disputas, uma contra o jogador MFS com constante T = 1, uma contra o MFS com T=5, uma contra o Jogador Aleatório, uma contra o Jogador Constante que decide "continuar", uma contra o Jogador Constante que decide "aumentar muito", uma contra o Jogador RH com constante T=0,1 e uma contra o Jogador RH com T=0,001. O resultado da função fitness é a soma ponderada de vitórias das disputas com peso 2 contra o Jogador MFS, peso 1 contra o Jogador Constante, peso 1 contra o Jogador RH e peso 4 contra o Jogador Aleatório.

A população tem tamanho 10 e a reprodução é feita na seguinte proporção: os 2 melhores sobrevivem para a próxima geração, 1 é mutação e 7 são produtos de crossover (Tabela 5).

	Pot	Chance	Raise	Odds	QtyRaise	Followers	Ingame	Nplayers	LQ	BQ
Pai 1	14	5	7	9	1	4	3	8	3	7
Pai 2	27	20	3	2	4	1	2	3	5	10
Filho	27	20	7	9	1	1	3	3	3	7

Tabela 5: Exemplo de um Crossover utilizado para ajustar o jogador por Reforço.

A função de seleção escolhida foi a stochastic uniform (MATLAB, 2008). Essa função gera uma linha onde cada indivíduo tem um pedaço da linha proporcional ao fitness. Um ponto aleatório da linha é selecionado e o indivíduo que estiver naquela faixa da linha é o primeiro pai escolhido. Os outros pais são selecionados pela distribuição uniforme de pontos pela linha, igualmente espaçados.

3.5. Probabilidade de Vencer

Para o cálculo de probabilidade de vencer foi utilizado apenas o método Monte-Carlo pela necessidade de eficiência e pela provável precisão aceitável.

4. ANÁLISE DOS RESULTADOS

O objetivo do trabalho é formular e procurar algoritmos que joguem bem Poker. Para cumprir esse objetivo, foi feito uma comparação de número de vitórias, em porcentagem, com confronto direto entre jogadores com métodos diferentes. Além dos jogadores foi analisado também o método de probabilidade de vitória, que é uma informação disponível para todos os jogadores, tanto no treinamento quanto na analise de desempenho dos jogadores. Para o jogador evolutivo foi feito uma análise um pouco maior, contendo a sua evolução e as constantes usadas.

4.1. Verificação do método Monte-Carlo

O método Monte-Carlo se mostrou eficiente em calcular a probabilidade de vencer com apenas 300 simulações de jogos. Com essa quantidade de simulações, a probabilidade de vencer é obtida com uma margem de erro de 5% aproximadamente. A probabilidade de vencer não sofre grande variação com mais simulações, como mostra o gráfico abaixo. Isso se repete em todas as situações de jogo e com qualquer quantidade de jogadores. A linha em vermelho mostra a regressão linear simples desse gráfico, mostrando que o valor se mantem constante com o aumento no número de simulações. Uma informação que não está sendo mostrada no gráfico é que depois de 100.000 simulações da mesma situação, a probabilidade de vencer se manteve em 33,545%, muito próximo ao valor obtido com 300 simulações.

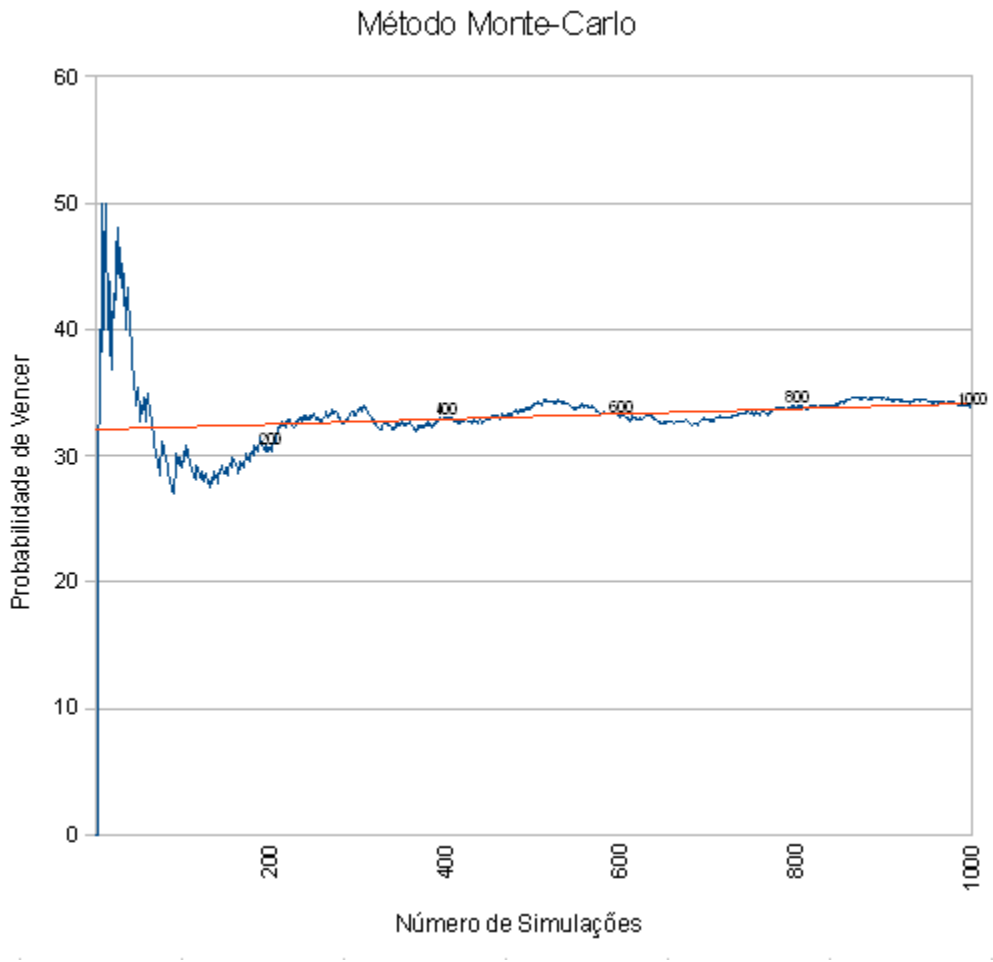

Gráfico 1: Exemplo de utilização do método Monte-Carlo para calcular a probabilidade de vencer

4.2. Quantidade de mesas necessárias para avaliar um jogador

O método Monte-Carlo também foi usado para encontrar a quantidade de jogos necessários para avaliar o desempenho de um jogador em relação ao outro. O gráfico abaixo, exemplo de resultados dos jogos em relação a quantidade de mesas jogadas, mostra que depois de 100 mesas, a proporção de vitórias permanece aproximadamente constante ao longo de mais mesas, portanto esse foi o valor utilizado para avaliação de desempenho.

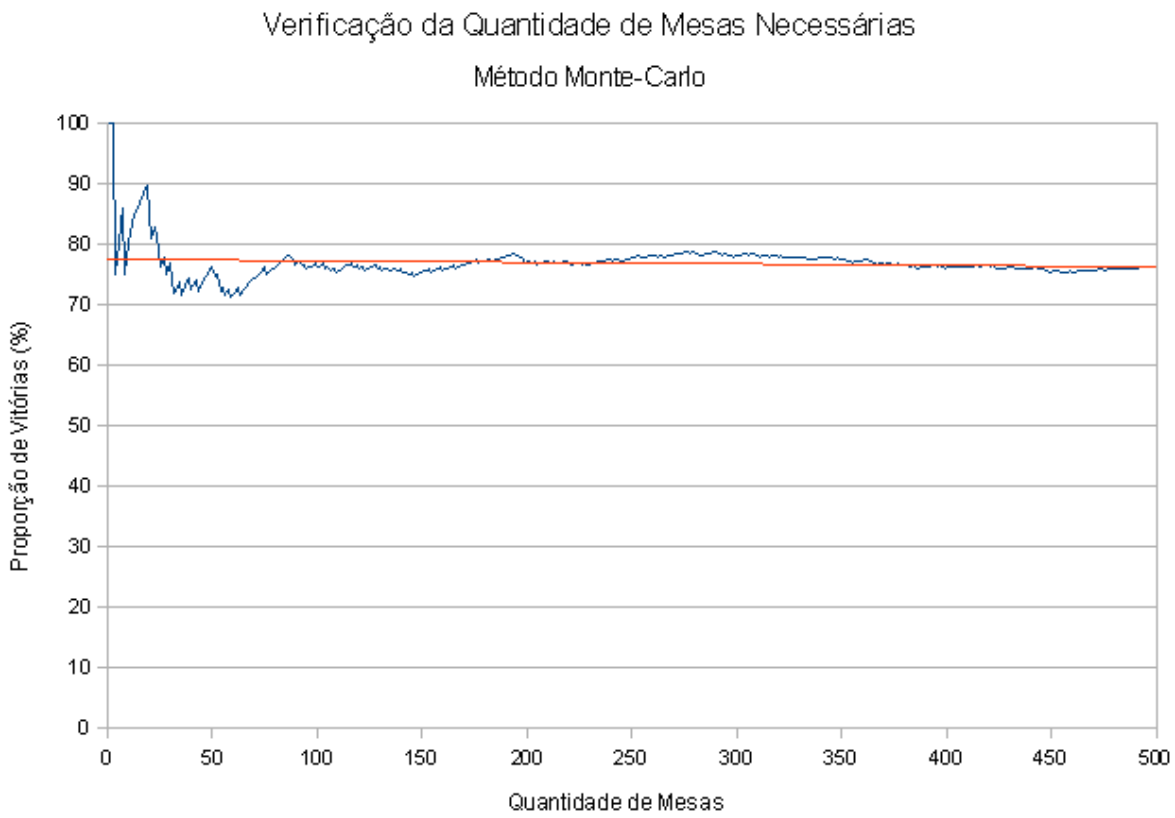

Gráfico 2: Verificação da quantidade de mesas necessárias para avaliar o desempenho de um jogador

4.3. Constantes de configuração e evolução do Jogador por Reforço

Para obter as constantes deste jogador foi utilizado algoritmo genético e tentativas através de bom senso. Uma limitação para o algoritmo genético foi a função de fitness pois foi considerado apenas 10 mesas para avaliar o desempenho de um conjunto de constantes. Segundo o Gráfico 2, uma disputa de 10 mesas é muito instável e pode ter levado a busca genética ao fracasso. Por outro lado, se fosse usado 100 mesas, o tempo para executar a busca genética seria cerca de vinte dias. A limitação da tentativa através do bom senso é a pouca quantidade de tentativas. Enquanto o algoritmo genético testou milhares de combinações de constantes, a tentativa através do bom senso se limita a poucas dezenas. Os resultados foram interessantes porque o algoritmo genético encontrou uma boa configuração de constantes para a quantidade de dados utilizada na busca. Porém, quando a quantidade de dados aumenta, o jogador piora o desempenho. A busca pelo bom senso obteve uma boa configuração de maneira gradual e contínua mas também houve piora no desempenho a partir de uma

quantidade de informação. O erro nos gráficos é a soma da proporção de derrotas da função de fitness.

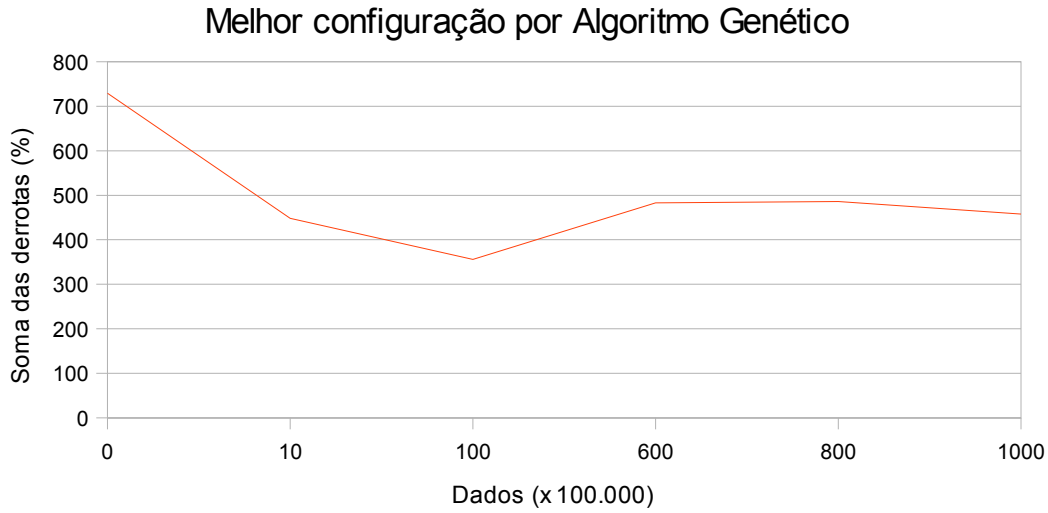

Gráfico 3: Progresso do jogador com o aumento de informação com a melhor configuração encontrada pelo algoritmo genético

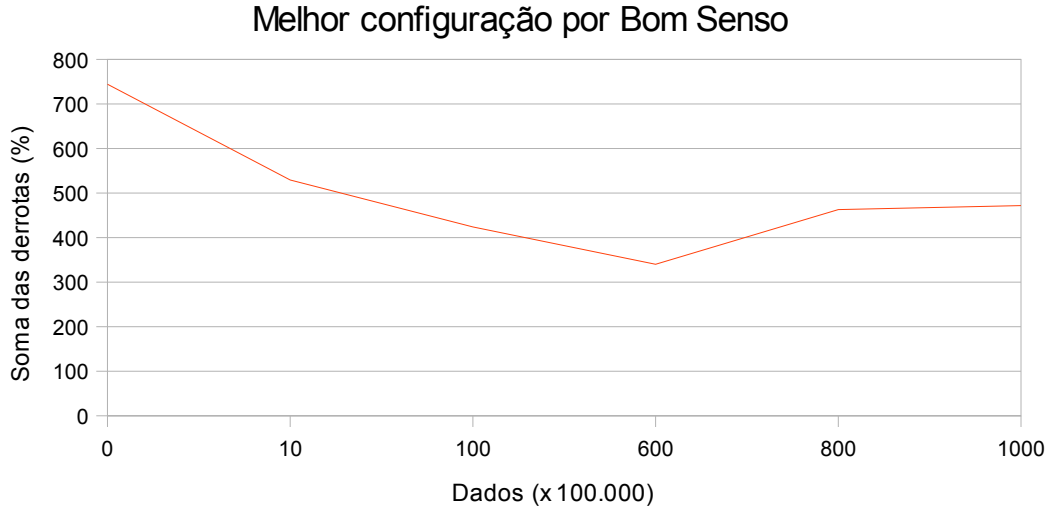

Gráfico 4: Progresso do jogador com o aumento de informação com a melhor configuração encontrada pelo bom senso

As constantes encontradas por cada método foram:

Constante	Algoritmo Genético	Bom Senso
Intervalo ODDS	9,6	2
Intervalo CHANCE	23,3	10
Intervalo POT	37,4	10
Intervalo RAISE	13,2	5
Intervalo FOLLOWERS	2,6	1
Intervalo NPLAYERS	3,5	Não usado
Intervalo QTYRAISE	3,5	Não usado
Intervalo INGAME	5,6	Não usado
BIG QTY	4	10
LITTLE QTY	2	2

Tabela 6: Constantes encontradas para ser utilizado pelo Jogador Esforço

Um retrato das decisões tomadas em diversas situações pelo jogador por esforço utilizando a configuração Bom Senso com 60 milhões de dados de informações pode ser visualizado na figura abaixo, sendo cada quadrado de cor a melhor decisão de um estado do jogo.

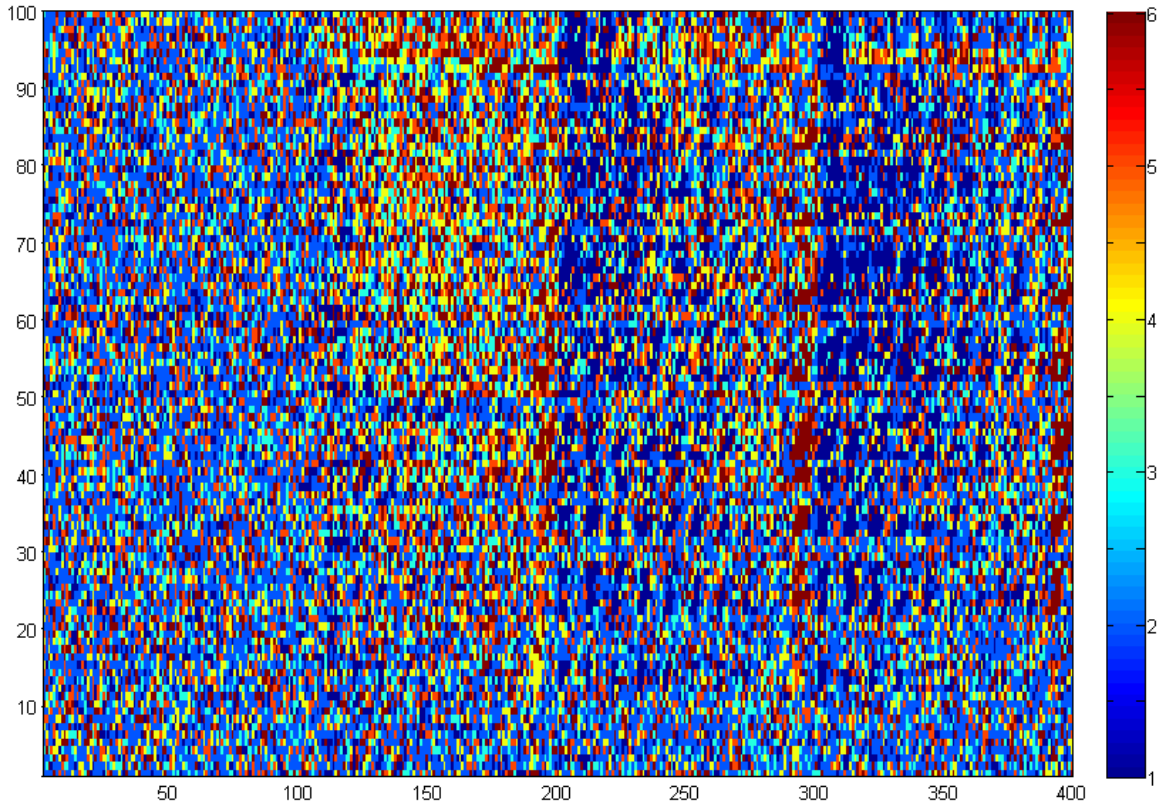

Figura 2: Exemplo do mapa de conhecimento do jogador por reforço sendo cada ponto um estado do jogo pintado com a decisão a ser tomada

4.4. Disputas

Cada confronto é formado por 200 mesas. Cada mesa contem um número par de 2 a 10 jogadores igualmente distribuídos, sendo metade jogadores de um método e a outra metade outro método, e contem até 40 jogos ou até que um jogador ganhe todo o dinheiro dos adversários, o que ocorrer primeiro. Quando cada mesa acaba, é pontuado a vitória de um dos dois métodos confrontados. Ao fim de 100 mesas é feita a proporção de vitórias de cada método. Cada jogo começa com 1000 de dinheiro para cada jogador e o Small Blind é 10 e a ordem dos jogadores é aleatória.

4.4.1. MFS x Aleatório

Para essa avaliação foi testado vários possíveis valores para a constante T do jogador MFS. MFS é um jogador que provavelmente mostraria melhores resultados que o jogador aleatório pois o MFS joga de acordo com a probabilidade de vencer e os resultados comprovaram (Gráfico 5). Isso demonstra que Poker não é somente sorte pois uma decisão inteligente melhora o desempenho do jogador.

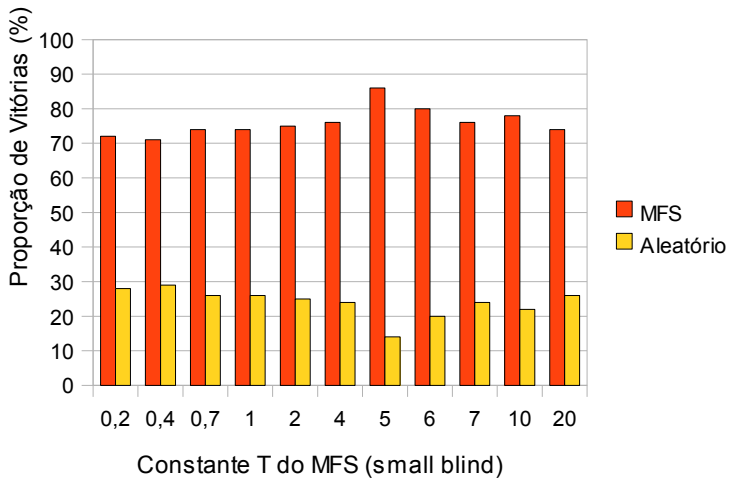

Gráfico 5: Jogador MFS x Jogador Aleatório

4.4.2. MFS x Constante

Nessa disputa foram testados apenas alguns valores principais para o MFS e todas as decisões possíveis para o jogador constante com exceção de *fugir*. Era provável que igualmente o MFS obtivesse melhores resultados por considerar a probabilidade de vencer, ao contrário do jogador constante que sempre tem o mesmo comportamento. Como o jogador MFS joga justo de acordo com a probabilidade de vencer e foge quando a probabilidade de vencer não é boa, em geral aumentar muito a aposta faz o jogador MFS fugir. Por outro lado, aceitar as aumentadas de aposta do jogador MFS é a pior coisa a ser feita pois a longo prazo ele terá ótimos ganhos por jogar exatamente de acordo com a força do seu jogo. O jogador constante obteve melhores resultados quando sua decisão era sempre aumentar a aposta porque fazia o MFS fugir sempre, mas não foi suficiente para vence-lo pois quando o aumento de aposta favorecia o MFS, este recuperava todo o prejuízo e vencia. Segue os resultados graficamente:

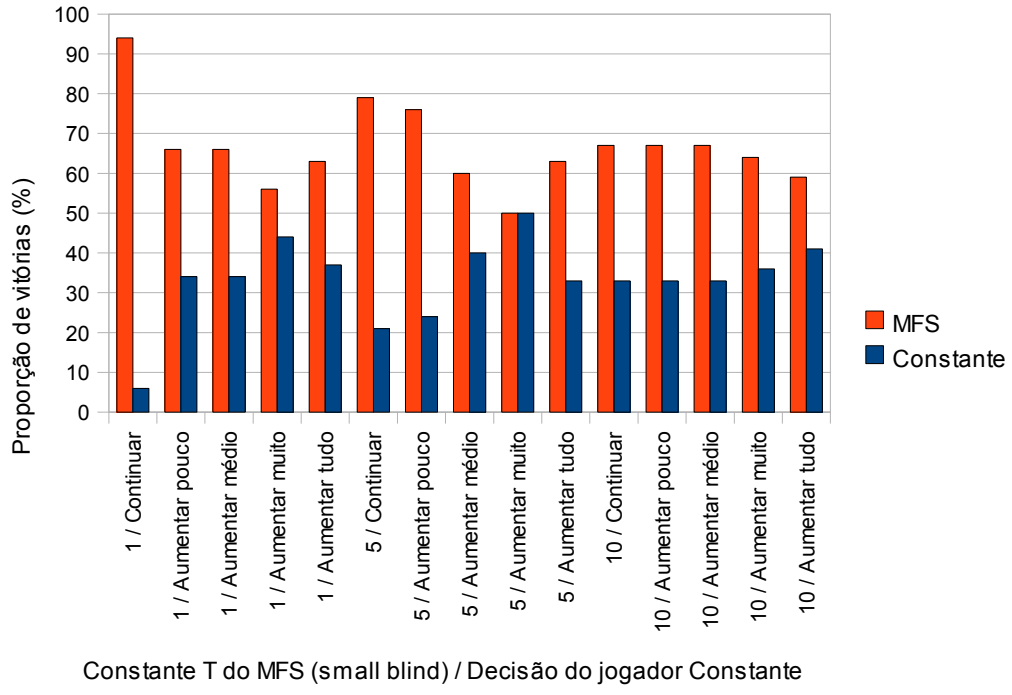

Gráfico 6: Jogador MFS x Jogador Constante

4.4.3. Aleatório x Constante

Aqui, o jogador constante usou todas as decisões possíveis com exceção de fugir. Os resultados mostraram uma insignificante vantagem para o jogador constante:

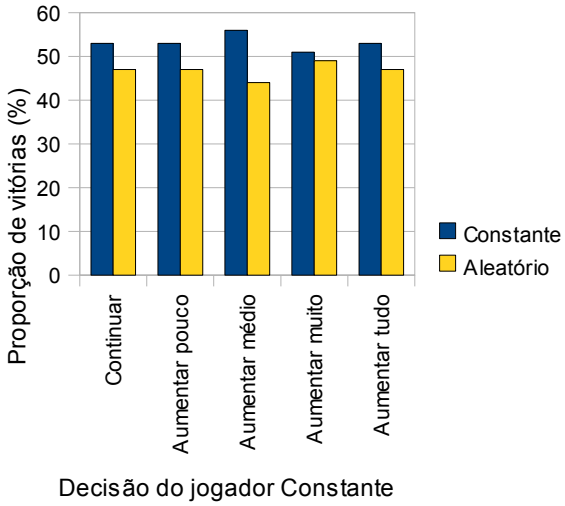

Gráfico 7:Jogador Aleatório x Jogador Constante

4.4.4. RH x Aleatório

O jogador RH joga com base no número de vezes que os adversários aumentaram a aposta, no número de jogadores que ainda tem no jogo, resumindo, em informações que podem dizer como estão os adversários. Como o jogador aleatório joga qualquer coisa, essa informação é pode ser de pouca utilidade. Mas mesmo assim, se o RH aposta mais quando ninguém está apostando, ele pode vencer porque os jogadores aleatórios fugiram. Os resultados constataram uma vantagem para o RH:

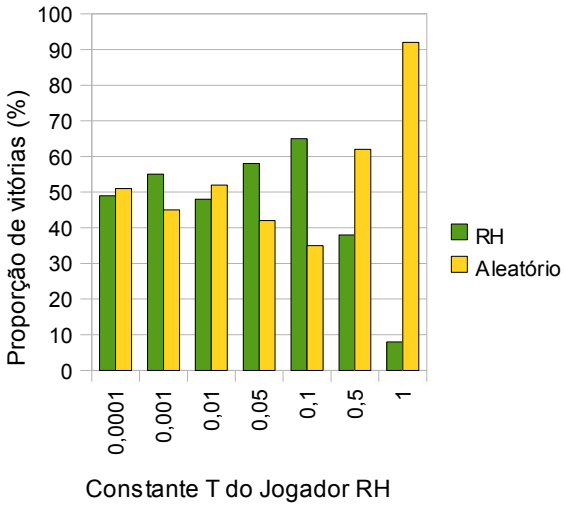

Gráfico 8: Jogador RH x Jogador Aleatório

4.4.5. RH x Constante

Nessa disputa existe um problema para o RH: como o jogador RH de acordo com a ação do adversário, um adversário que joga constante fará o jogador RH jogar constante também, fazendo com que fique imprevisível os resultados:

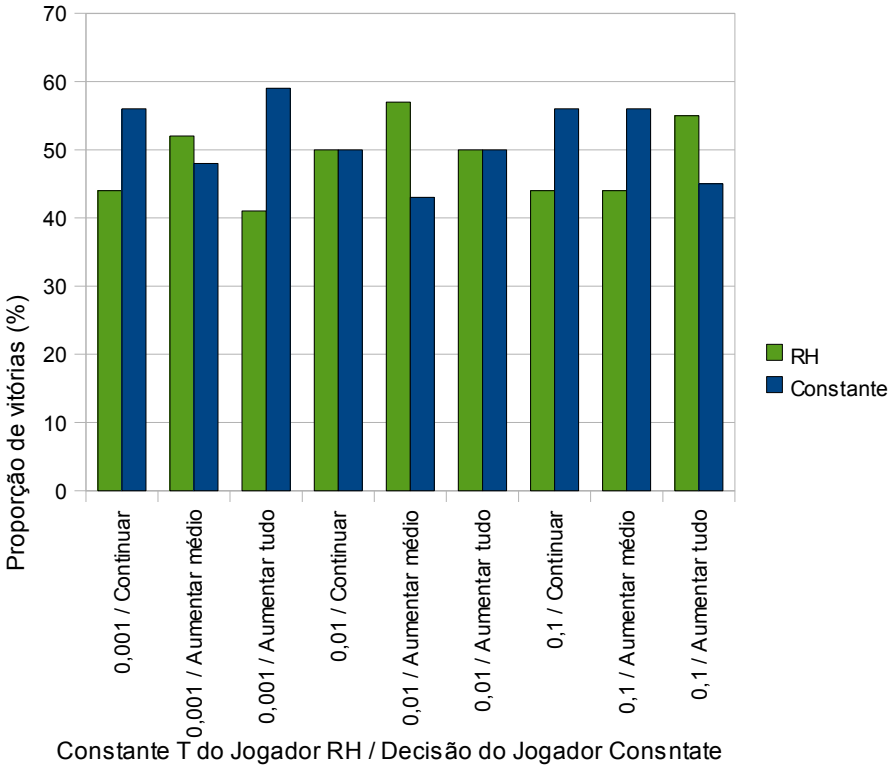

Gráfico 9: Jogador Constante x Jogador RH

4.4.6. RH x MFS

O MFS foge quando não tem muita chance de vencer e o adversário aumenta muito a aposta. Por isso, um jogador agressivo é mais forte contra o MFS do que um jogador menos agressivo. Porém, muita agressividade pode fazer com que o MFS se aproveite nas melhores situações e ganhe muito dinheiro em apenas uma rodada. Esse foi o resumo do resultado dessa disputa. O RH é mais agressivo quanto menor for a constante T. O RH é o algoritmo que poderia vencer o MFS por se aproveitar dos momentos que o MFS não está aumentando para aumentar as apostas, mas o máximo que ele conseguiu foi quase um empate. O jogador MFS utilizou apenas um valor para sua constante T, o valor "1", que experimentalmente demonstrou bons resultados:

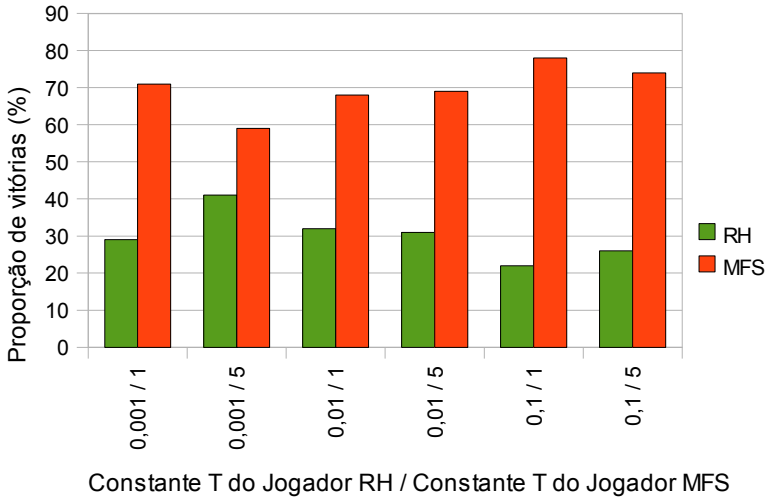

Gráfico 10: Jogador RH x Jogador MFS

4.4.7. Reforço x Aleatório

Apesar de ter sido criado com muita informação aleatória, o jogador por reforço conseguiu vencer o jogador aleatório. Talvez porque exista realmente a melhor jogada em cada situação.

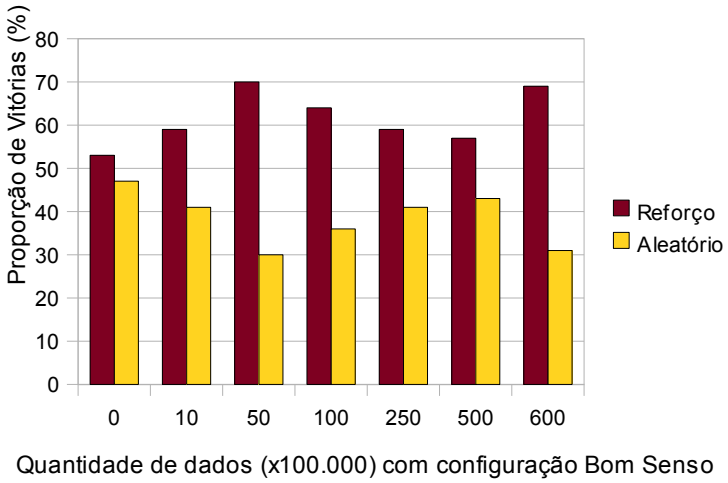

Gráfico 11: Jogador Reforço x Jogador Aleatório

4.4.8. Reforço x Constante

O jogador por reforço conseguiu aprender um jeito para vencer o jogador que sempre continua. O jogador que aumenta muito ele conseguiu melhorar o desempenho com mais informação.

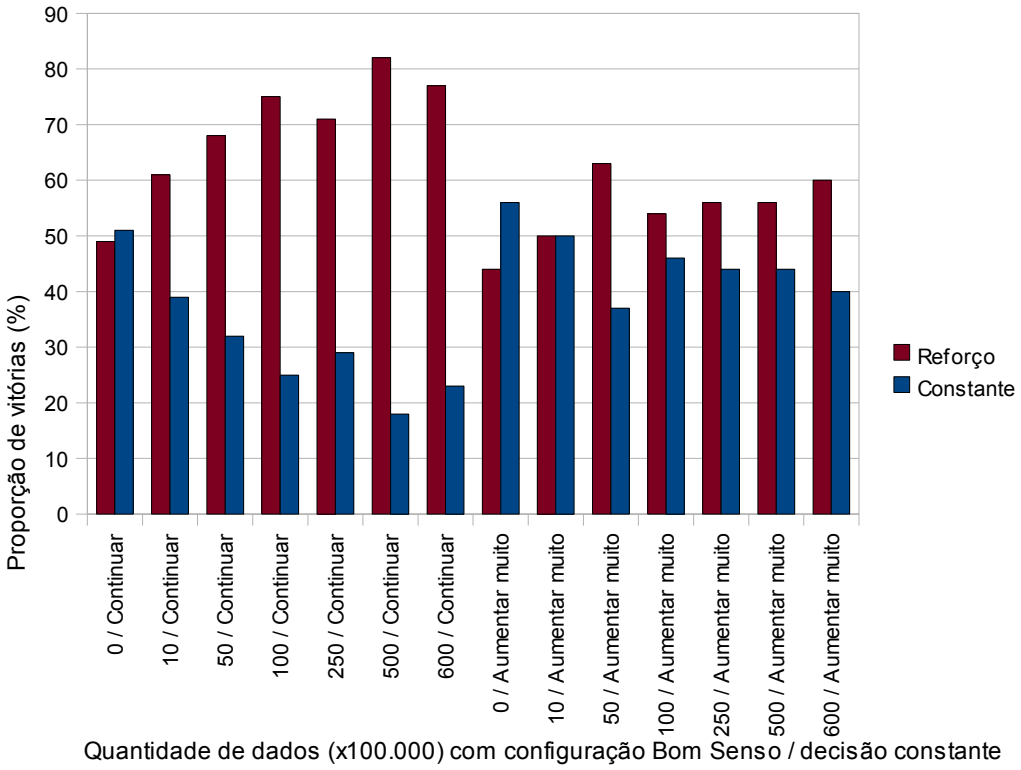

Gráfico 12: Jogador Reforço x Jogador Constante

4.4.9. Reforço x MFS

O jogador MFS, que parecia ser o jogador mais difícil de vencer, foi o mais fácil. O jogador por reforço teve uma vitória bem significativa.

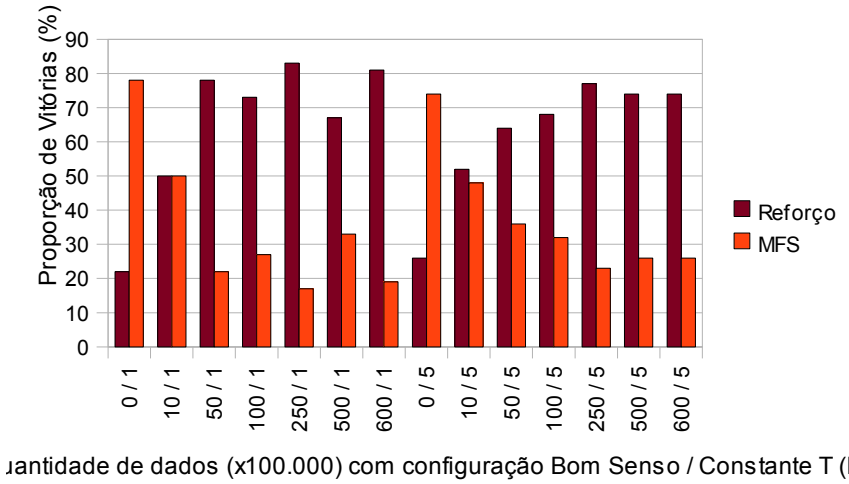

Gráfico 13: Jogador Reforço x Jogador MFS

4.4.10. Reforço x RH

O jogador RH surpreendeu, conseguindo empatar com o jogador por reforço quando configurado com a constante $T = 0{,}1$. Por algum motivo, o jogador não consegue evoluir contra este jogador.

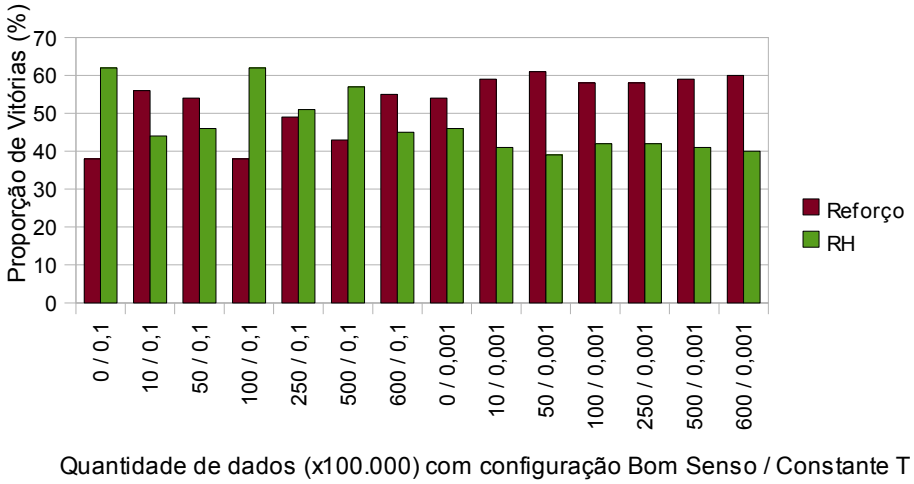

Gráfico 14: Jogador Reforço x Jogador RH

4.5.11. Disputa entre os melhores de cada

Essa disputa foi feita com 10 jogadores em jogo sendo 2 jogadores de cada tipo. O MFS tinha um jogador com constante $T=1$ e outro com $T=5$. O RH tinha um jogador com constante $T=0{,}1$ e outro com $T=0{,}001$. O jogador constante teve um jogador que decidia sempre Continuar e outro que decidia sempre Aumentar Muito. Foi jogado 500 mesas e o jogador treinado jogou melhor. Apesar do jogador RH ter tido vantagem contra o jogador por Reforço jogando individualmente, quando jogou-se em grupo, o jogador RH teve um desempenho ruim.

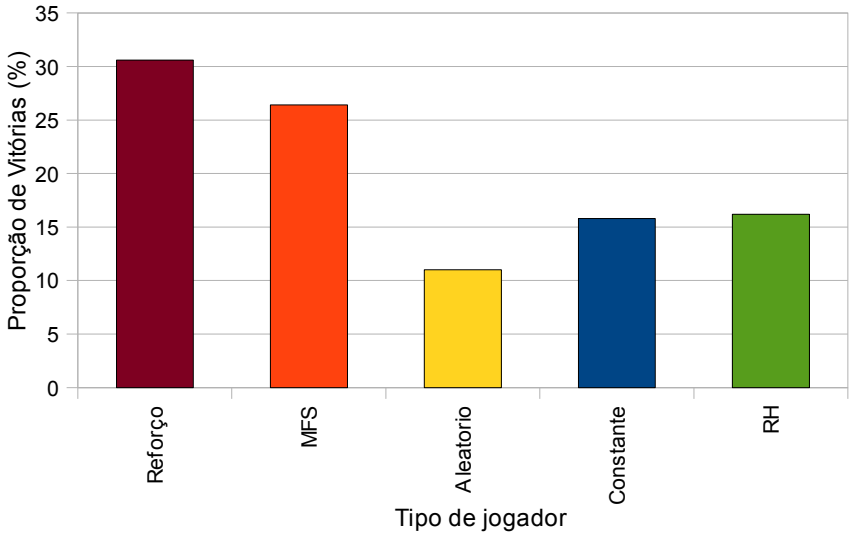

Gráfico 15: Disputa entre todos os jogadores

4.5.12. Reforço x Humano

Contra jogadores humanos bem amadores, o jogador por reforço obteve um bom desempenho. Foram testados dez *mesas* e quatro jogadores humanos diferentes e a proporção de vitórias foi de 100% para o jogador por reforço. Cada mesa tinha um jogador humano e três jogadores por reforço treinados com a configuração encontrada por Bom Senso e com 60 milhões de dados. O jogador por reforço mostrou-se bem coerente com a sua probabilidade de vencer quando decidia *fugir* ou quando decidia *apostar tudo*. Mesmo assim, não é sempre que ele joga de acordo com a probabilidade de vencer e fica difícil prever a sua jogada.

5. CONCLUSÃO

Os algoritmos testados para jogar poker tiveram desempenhos diferentes, sendo que o melhor testado foi o jogador com aprendizado por reforço, que é o jogador que fica jogando aleatoriamente até ter dados suficientes para tomar uma decisão baseado nos casos já experimentados. O jogador MFS, que parecia um excelente jogador já que poker é um jogo de aposta e probabilidade de vencer e este jogador joga baseado na aposta e na probabilidade de vencer, ficou um pouco pior que o jogador de aprendizado por reforço quando jogado todos junto mas mesmo assim obteve um bom desempenho. O método Monte-Carlo para cálculo de probabilidade de vencer foi suficiente e eficiente.

Os ajustes feitos na configuração do jogador geraram diferentes formas de progresso, mas o efeito foi o mesmo. Até uma determinada quantidade de informação, o jogador melhorou o desempenho, mas com mais informação o desempenho do jogador não melhorou mais. Esse problema aconteceu com diversas configurações e fórmulas do jogador por aprendizado. Em geral os ajustes feitos em problemas relacionados a inteligência artificial são bastante trabalhosos e pouco determinístico.

O Poker se mostrou um jogo bem desafiador de criar um bom jogador pois não existe claramente um método ótimo de vencer neste jogo o que permite utilizar muitas técnicas diferentes da Inteligência Artificial para chegar a um bom resultado. Referências de outros trabalhos mostraram a utilização de técnicas bem diferentes como Minimax, Nash Equilibrium e Redes Neurais.

Neste trabalho, dois pontos críticos foram o desempenho do algoritmo e problemas de implementação. Muito código foi reescrito em C porque a lentidão do Matlab inviabilizava a continuação do trabalho. A divisão do trabalho em etapas para testar diferentes implementações em apenas uma etapa de todo o processo foi uma solução que também economizou muito tempo. Os problemas de implementação no sistema geraram grandes prejuízos no tempo em que demorou para o trabalho ficar pronto porque todos os resultados eram descartados toda vez que algum problema no sistema era encontrado e tinha que gerar

tudo de novo, o que em alguns casos levavam dias.

Para trabalhos futuros, pode ser interessante testar algoritmos diferentes. Por exemplo, um algoritmo não discretizasse o conhecimento mas normalizasse as dimensões de forma que pudesse utilizar "K Nearest Neighbor" que classifica um dado de acordo com os casos mais próximos. Outra exemplo, utilizar métodos híbridos através de uma rede neural que identificasse qual método tem o melhor desempenho em determinada situação e utilizasse-o. Alguns trabalhos lidos procuraram uma forma de criar um jogador artificial de Poker que utilizasse informações mais detalhada do jogo, como o histórico de jogadas de cada jogador, o que pode ajudar a identificar e vencer adversários muito previsíveis como os jogadores estáticos apresentados neste trabalho.

Pode ser feita uma investigação mais aprofundada em porque o jogador não melhora continuamente com mais informação ou, pelo menos, se mantem no melhor desempenho com mais informação pode ser bastante útil para descobrir novas formas de criar jogadores mais eficientes.

6. REFERÊNCIAS BIBLIOGRÁFICAS

ECKEL, Bruce. **Thinking in C++:** Second Edition: Volume 1: Introduction to Standard C++. New Jersey: Prentice Hall, 2000. Disponível em: <http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html>. Acesso em 1 out. 2008.

FINDLER, Nicholas V.. **Studies in Machine Cognition Using the Game of Poker.** State University Of New York At Buffalo: R. J. Hanson, 1977.

HOLLAND, John H.. Genetic Algorithms and the Optimal Allocation of Trials. **Siam Journal On Computing**, Philadelphia, n. 2 , p.88-105, 3 ago. 1972.

JOHANSON, Michael Bradley. **Robust Strategies and Counter-Strategies:** Building a Champion Level Computer Poker Player. 2007. 97 f. Tese (Mestrado) - University Of Alberta, Alberta, 2007.

LUGER, George F.; STUBBLEFIELD, Willian A. **Artificial Intelligence:** Structures and Strategies for Complex Problem Solving. Harlow, England: Addison Wesley Longman, Inc., 1998.

NISAN, Noam et al. **A Course In Game Theory.** New York: Cambridge University Press, 2007.

RUSSEL, Stuart J.; NORVIG, Peter. **Artificial Intelligence:** A Modern Approach. New Jersey: Prentice Hall, 1995.

SKLANSKY, David. **The Theory of Poker.** Las Vegas Nv: Two Plus Two Pub, 1987.

SUTTON, Richard S.; BARTO, Andrew G.. **Reinforcement Learning:** An Introdution. E-Book. Disponível em: <http://www.cs.ualberta.ca/~sutton/book/ebook/the-book.html>. Acesso em: 1 out. 2008.

Outras referências (sem autor):

MATLAB (Org.). **The MathWorks:** MATLAB and Simulink for Technical Computing. Disponível em: <http://www.mathworks.com/>. Acesso em 1 out. 2008.

POKERAIWIKI (Org.). **Computer Poker Ai Wiki**. Disponível em: <http://pokerai.org/wiki/index.php/Main_Page>. Acesso em: 1 out. 2008.

POKERLOCO (Org.). **Como Jogar Poker:** Resumo. Disponível em: <http://www.pokerloco.com/pt/howtoplaypoker/index.htm>. Acesso em: 1 out. 2008.

WIKIPÉDIA MÉTODO DE MONTE CARLO (Org.). **Método de Monte Carlo.** Origem Wikipédia, a enciclopédia livre. Disponível em: <http://pt.wikipedia.org/wiki/Método_de_Monte_Carlo>. Acesso em: 1 out. 2008.

WIKIPEDIA POKER (Org.). **Poker.** From Wikipedia, the free encyclopedia. Disponível em: <http://en.wikipedia.org/wiki/Poker>. Acesso em: 1 out. 2008.

WIKIPEDIA REINFORCEMENT LEARNING (Org.). **Reinforcement Learning.** From Wikipedia, the free encyclopedia. Disponível em: <http://en.wikipedia.org/wiki/Reinforcement_learning>. Acesso em: 1 out. 2008.

7. ANEXOS

Esta sessão contém os códigos fontes implementados.

7.1. playpoker.m

Este método joga poker até que um jogador na mesa se torne campeão:

```
function [ results, winner ] =
playpoker(nplayers,playersai,maxgames,initialmoney,smallblind,database,mode
) %#ok<INUSD>
% PLAYPOKER Play a poker game.
% Play a poker game. It requests playersai their decision and make them
% learn with the results of the game.
%
% Arguments:
% nplayers - number of players that are gonna play
% playersai - players ai that are gonna play
% maxgames - max number of games
% initialmoney - initial money for each player
% smallblind - the small blind money
% database - database that receives game info
% mode - if 0, test, if 1, check victories, if 2, debug...
%
% @author Vinícius Sousa Fazio

%tic
%%
TEST_MODE = 0;
CHECKVIC_MODE = 1;
DEBUG_MODE = 2;
PLAYER_MODE = 3;

PHPLAYER=1;
PHODDS=2;
PHPOT=3;
PHRAISE=4;
PHROUND=5;
PHCHANCE=6;
PHNPLAYERS=7;
PHFOLLOWERS=8;
PHINGAME=9;
PHQTYRAISE=10;
PHDECISION=11;

PHSIZE=11;
```

```
if nargin ~= 7
    disp('erro playpoker 1');
    return;
end
if mode == TEST_MODE
    showgame=0;
    playgame(0); % test
    return;
end

%% constantes
tablesize=5;
handsize=2;
probprecision=300;

%% aloca memória
hand=zeros(nplayers,handsize);
table=zeros(tablesize,1);
chances=zeros(nplayers,1);
money=zeros(nplayers,1)+initialmoney;
moneybefore=zeros(nplayers,1);
betmade=zeros(nplayers,nplayers);
potlimit=zeros(nplayers,1);
score=zeros(nplayers,1);
prescore=zeros(nplayers,1);
fold=zeros(nplayers,1);
out=zeros(nplayers,1);
if mode == TEST_MODE
    results=createresults;
else
    results={};
end

%% inicia as variaveis de todos os jogos
showgame=0;
if mode == DEBUG_MODE || mode == PLAYER_MODE
    showgame = 1;
end
totalplayers=size(playersai,2);

if mode == CHECKVIC_MODE
    playersids=1:totalplayers;
else
    playersids=randperm(totalplayers);
end
tablecount=0;

%% joga todos os jogos
%inicia as variaveis de um jogo
tablecount=tablecount+1;
money(1:nplayers)=initialmoney;
money((nplayers+1):end)=0;
gamecount=0;
players(1:nplayers)=playersai(playersids(1:nplayers));
%joga todos os jogos
while gamecount < maxgames
    %        disp(['-game' int2str(gamecount+1)]);
    %reseta o histórico
    clear playhistory
```

```
    playhistory=zeros(nplayers*20,PHSIZE);
    playhistoryPointer=1;
    numRaises=0;
    % joga
    gamecount=gamecount+1;
    if showgame == 1
        disp(['**** JOGO ' int2str(gamecount) ' ****']);
    end
    playgame;
    if showgame == 1
        for pp=1:nplayers
            mondif=money(pp)-moneybefore(pp);
            if mondif < 0
                disp(['jogador ' int2str(pp) ' perdeu ' int2str(-mondif) '
com a mão ' int2str(prescore(pp)) ' ' hand2str(pp,5)]);
            elseif mondif > 0
                disp(['jogador ' int2str(pp) ' ganhou ' int2str(mondif) '
com a mão ' int2str(prescore(pp)) ' ' hand2str(pp,5)]);
            else
                disp(['jogador ' int2str(pp) ' não ganhou nem perdeu nada
com a mão ' int2str(prescore(pp)) ' ' hand2str(pp,5)]);
            end
        end
        disp('.                                   .');
        for pp=1:nplayers
            disp(['dinheiro do jogador ' int2str(pp) ': '
int2str(money(pp))]);
        end
        disp('.                                   .');
    end


    % salva as decisões dos jogadores em um banco de dados
    if ~isempty(database)
        for php=1:(playhistoryPointer-1)
            ph=playhistory(php,:);
            pp=ph(PHPLAYER);
            result=(money(pp)-moneybefore(pp))/smallblind;
            database.insert(ph(PHODDS),ph(PHPOT),ph(PHRAISE),ph(PHROUND),ph
(PHCHANCE),ph(PHNPLAYERS),ph(PHFOLLOWERS),ph(PHINGAME),ph(PHQTYRAISE),ph(PH
DECISION),result);
        end
    end

    %pára o jogo caso apenas um jogador consiga pagar o bigblind
    if size(money(money>=2*smallblind),1) <= 1
        %            disp('not enough players');
        break;
    end
end
%atualiza os vencedores
winner=1;
for g=2:nplayers
    if money(winner)<money(g)
        winner=g;
    end
end
winner=playersids(winner);
```

```
%%
function [ handstr ] = hand2str(num,st)
    handstr=['-- mão ' int2card(hand(num,1)) ' ' int2card(hand(num,2)) ' /
mesa'];
    for ss=1:st
        handstr=[handstr ' ' int2card(table(ss))]; %#ok<AGROW>
    end
    function [ card ] = int2card(num)
        c = fix((num-1)/4)+2;
        switch mod(num-1,4)
            case 0
                kind = 'p';
            case 1
                kind = 'e';
            case 2
                kind = 'o';
            case 3
                kind = 'c';
        end
        if c >= 2 && c <= 10
            card=[int2str(c) kind];
        end
        switch c
            case 11
                card=['J' kind];
            case 12
                card=['Q' kind];
            case 13
                card=['K' kind];
            case 14
                card=['A' kind];
        end
    end
end
%% results object
% money é quanto cada jogador tem de dinheiro
% guarda os resultados dos jogos (para teste)
    function [ r ] = createresults()
        totalplayers=size(playersai,2);
        r=struct('setgame',@setgame,'getgame',@getgame);
        resultdata=zeros(nplayers,maxgames,fix(totalplayers/nplayers));
        function [ ] = setgame(tableid,gameid,money)
            resultdata(:,gameid,tableid)=money;
        end
        function [ g ] = getgame(tableid,gameid)
            g=resultdata(:,gameid,tableid);
        end
    end

%% playgame function
%supõe-se que tenha 2 jogadores ou mais com dinheiro para o big blind.
    function [] = playgame(testInternal) %#ok<INUSD>
        if nargin == 1 % test case
            money=[1000 1 50 0];
            potlimit=[51 100 1050];
            betmade=[[50 0 0 0]; [50 0 0 0]; [50 0 0 0]; [0 0 0 0]];
            nplayersongame=3;
            doBet(1,200,0);
            if money(1) ~= 800 || sum(betmade(:,1)) ~= 151 ||
```

```
sum(betmade(:,2)) ~= 100 || sum(betmade(:,3)) ~= 99
                disp('error playgame-doBet 1.1');
            end


            nplayers=5;
            betmade=[[0 0 0 0 0];[0 0 0 0 0];[20 0 0 0 0];[10 0 0 0 0];[0 0
0 0 0]];
            fold=[0 0 0 1 0];
            out=[1 1 0 0 1];
            potlimit=[980 4020 0 0 0];
            money=[0 0 960 4010 0];
            moneybefore=[0 0 980 4020 0];
            prescore=[0 0 833927 10638331 0];
            distributePrizes();
            if money(3) ~= 990 || money(4) ~= 4010
                disp('error playgame-distributePrizes 1.1');
            end

            return;
        end

        % variaveis gerais
        betmade(:)=0;
        moneybefore(:)=money(:);

        %sorteia e distribue as cartas
        deck=randperm(52);
        table(1:tablesize)=deck(1:tablesize);
        hand(1:(nplayers*handsize))=deck((tablesize+1):
(tablesize+nplayers*handsize));

        %calcula os limites de aposta (para o caso de all-in)
        tmp=sort(money(money~=0));
        nplayersongame=size(tmp,1);
        potlimit(1:nplayersongame)=tmp(1:nplayersongame);

        %elimina os jogadores sem dinheiro
        out(:)=0;
        for jj=1:nplayers
            if money(jj) < 2*smallblind
                out(jj)=1;
            end
        end
        fold(:)=0;

        %small blind e big blind
        firstplayeronround=fix(rand()*nplayers)+1;
        firstplayer=firstplayeronround;
        lastplayer=firstplayer+2;
        while firstplayer < lastplayer
            p=rem(firstplayer,nplayers)+1;
            if out(p)==1
                lastplayer=lastplayer+1;
            else
                bet=smallblind*(3-(lastplayer-firstplayer));
                doBet(p,bet,0);
            end
            firstplayer=firstplayer+1;
```

```
        end

        %primeiro round, antes do flop
        raise=2*smallblind;
        lastplayer=firstplayer+nplayers;
        showntable=0;
        round=1;

        if showgame == 1
            disp(['round ' int2str(round)]);
        end
        playround;
        %flop, turn and river
        for showntable=3:5
            round=round+1;
            firstplayer=firstplayeronround;
            lastplayer=firstplayer+nplayers;
            if showgame == 1
                disp(['round ' int2str(round)]);
            end
            playround;
        end

        %distribui os prêmios
        for jj=1:nplayers
            prescore(jj)=pokeronerank(hand(jj,:),table);
        end

        distributePrizes();

        %guarda os resultados
        if ~isempty(results)
            results.setgame(tablecount,gamecount,money);
        end

        function [] = distributePrizes()
            for m=1:nplayers
                pot = sum(betmade(:,m));
                if pot == 0
                    continue;
                end
                for k=1:nplayers
                    if fold(k) == 0 && out(k) == 0 && potlimit(m) <=
moneybefore(k)
                        score(k)=prescore(k);
                    else
                        score(k)=0;
                    end
                end
                [ignore, order]=sort(-score);
                k1=order(1);
                count=1;
                for k=2:nplayers
                    k2=order(k);
                    if score(k1)==score(k2)
                        count=count+1;
                    else
                        break;
                    end
```

```
                end
                for k=1:count
                    o=order(k);
                    score(o)=0;
                    if out(o) == 0
                        money(o)=money(o)+(pot/count);
                    end
                end
                remaind=rem(pot,count);
                if remaind~=0
                    o=order(fix(count*rand()+1));
                    if out(o) == 0
                        money(o)=money(o)+remaind;
                    end
                end
            end

        end
        %% playround function
        function [] = playround()
            for k=1:nplayers
                chances(k)=pokerrank(handsize,shownttable,hand(k,:),table,su
m(out==0)-1,probprecision);
                if mode == DEBUG_MODE
                    disp(['-- chance do player ' int2str(k) ' = '
int2str(chances(k)) '%']);
                end
                if mode == PLAYER_MODE &&
strcmp(functiontostring(players(k).decide),'createhumanplayer/decide')
                    disp(hand2str(k,shownttable));
                end
            end

            %% apostas
            while firstplayer < lastplayer
                p=rem(firstplayer,nplayers)+1;
                m=money==0;
                mm=zeros(nplayers,1);
                for mmm=0:(lastplayer-firstplayer-1)
                    mm(rem(firstplayer+mmm,nplayers)+1)=1;
                end
                m=m&mm;
                outfold=out+fold+m;
                nplayersbetting = sum(outfold==0);
                if out(p)~=1 && fold(p) ~= 1 && money(p) > 0 &&
nplayersbetting > 1

                    alreadybet=sum(betmade(p,:));
                    realraise=raise-alreadybet;


                    startingmoney=money(p)+alreadybet;

                    realraise = min( realraise, money(p));

                    realpot=sum(sum(betmade(:,potlimit<=startingmoney)));
                    odds=0;
                    if alreadybet ~= 0
                        odds=realpot/alreadybet;
```

```
                    end

                    outfold=out+fold;
                    ingame=sum(outfold==0);

                    followers=zeros(nplayers,1);
                    for mmm=0:(lastplayer-firstplayer-1)
                        followers(rem(firstplayer+mmm,nplayers)+1)=1;
                    end
                    followers=followers&(outfold==0);
                    followers=sum(followers);

                    if mode == PLAYER_MODE &&
strcmp(functiontostring(players(p).decide),'createhumanplayer/decide')
                        disp(['bankroll: ' int2str(money(p))]);
                    end
                    decision=players(p).decide(odds,realpot/smallblind,real
raise/smallblind,round,chances(p),nplayers,followers,ingame,numRaises);

                    bet=0;
                    switch decision
                        case 1 % fold/call
                            bet=0;
                        case 2 % call
                            bet=realraise;
                        case 3 % raise low (2-4 smallblind)
                            bet=realraise+fix(rand()*(2*smallblind))
+2*smallblind;
                        case 4 % raise medium (6-18 smallblind)
                            bet=realraise+fix(rand()*(12*smallblind))
+6*smallblind;
                        case 5 % raise high (20-40 smallblind)
                            bet=realraise+fix(rand()*(20*smallblind))
+20*smallblind;
                        case 6 % raise all-in
                            bet=money(p);
                    end
                    if (bet > money(p))
                        %% allin
                        bet=money(p);
                    end
                    if money(p) > 0 && bet == 0 && realraise > 0
                        fold(p)=1;
                        if showgame == 1
                            disp(['jogador ' int2str(p) ' fugiu']);
                        end
                    end
                    playhistory(playhistoryPointer,PHPLAYER)=p;
                    playhistory(playhistoryPointer,PHROUND)=round;
                    playhistory(playhistoryPointer,PHCHANCE)=chances(p);
                    playhistory(playhistoryPointer,PHPOT)=realpot/smallblin
d;
                    playhistory(playhistoryPointer,PHODDS)=odds;
                    playhistory(playhistoryPointer,PHRAISE)=realraise/small
blind;
                    playhistory(playhistoryPointer,PHNPLAYERS)=nplayers;
                    playhistory(playhistoryPointer,PHINGAME)=ingame;
                    playhistory(playhistoryPointer,PHFOLLOWERS)=followers;
                    playhistory(playhistoryPointer,PHQTYRAISE)=numRaises;
```

```
                    playhistory(playhistoryPointer,PHDECISION)=decision;
                    playhistoryPointer=playhistoryPointer+1;

                    if decision >= 3
                        numRaises = numRaises + 1;
                    end
                    doBet(p,bet,realraise);

                    bm = sum(betmade(p,:));
                    if sum(bm > raise)
                        raise = bm;
                        lastplayer = firstplayer+nplayers;
                    end
                end
                firstplayer=firstplayer+1;
            end
        end
        function [] = doBet(p, bet, realraise)
            if showgame == 1 && fold(p) == 0
                if bet == 0
                    disp(['jogador ' int2str(p) ' pede mesa']);
                elseif bet == money(p)
                    disp(['jogador ' int2str(p) ' aumenta all-in com '
int2str(bet)]);
                elseif bet > realraise && realraise == 0
                    disp(['jogador ' int2str(p) ' aposta ' int2str(bet)]);
                elseif bet > realraise
                    disp(['jogador ' int2str(p) ' paga ' int2str(realraise)
' e aumenta ' int2str(bet-realraise)]);
                elseif bet == realraise
                    disp(['jogador ' int2str(p) ' paga ' int2str(bet)]);
                else
                    disp(['jogador ' int2str(p) ' paga misteriosamente
somente ' int2str(bet)]);
                end
            end
            %tira o dinheiro
            money(p)=money(p)-bet;
            %coloca no pot
            for i=1:nplayersongame
                pl=potlimit(i);
                b = bet;
                bm = betmade(p,i);
                if (bet + bm > pl)
                    b = pl - bm;
                end
                if b < 0
                    b = 0;
                end
                bet=bet-b;
                betmade(p,i)=betmade(p,i)+b;
                if bet == 0
                    break;
                end
            end

        end
```

```
        %% end of playgame
    end
%% end of playpoker
%toc
end
```

7.2. compute_victory.h

Este método implementa o Monte-Carlo. Ele faz simulações aleatórias e retorna o número de simulações vitoriosas.

```
/*
 * author Vinícius Sousa Fazio
 */


#include "getpower.h"

double computeRealVictoriesRandomly(int nCardsOnHand, int nCardsOnTable,
int* allyHand, int* table, int nEnemies, int gameCount) {

    int* pgc = playerGameCards;
    int* egc = enemyGameCards;
    int* adeck = deck;

    // contador dos jogos
    int iLose = 0;

    //temporários
    int i, j, c, pow;
    // coloca a mesa no jogo de todo mundo.
    for (i = 0; i < nCardsOnTable; i++) {
        pgc[i] = table[i];
        for (j = 0; j < nEnemies; j++) {
            egc[j * GAME_SIZE + i] = table[i];
        }
    }
    for (i = 0; i < nCardsOnHand; i++) {
        pgc[TABLE_SIZE + i] = allyHand[i];
    }

    // comeca a jogar
    for (c = gameCount; c > 0; c--)
    {

        // monta um baralho com as cartas que já saíram
        int totalCards = (nEnemies + 1) * HAND_SIZE + TABLE_SIZE;
        int deckP = 0;
        for (i = 0; i < nCardsOnTable; i++) {
            adeck[deckP++] = table[i];
        }
        for (i = 0; i < nCardsOnHand; i++) {
            adeck[deckP++] = allyHand[i];
        }

        // sorteia o resto do baralho
        pow = 0;
        for (i = deckP; i < totalCards; i++) {
            // O rand é bugado. Ele não sorteia todos os números.
```

```
            // Com essa gambiarra funciona.
            // os comentarios abaixo são debug
            int card = (rand() % (CARDS_QUANTITY << 2)) >> 2;
            bool repeated = false;
            //            mexPrintf ("%i | %i $ ", card, i);
            for (j = 0; j < i && !repeated; j++)
            {
                //                mexPrintf (" %i", adeck[j]);
                repeated |= card == adeck[j];
            }
            //            mexPrintf ("\n");
            adeck[i] = card;
            if (repeated)
            {
                i--;
            }
            //            if (pow++ > 100) break;
        }
        //        mexPrintf ("Contagem: %i\n", pow);


        // sorteia o resto da mao.
        for (i = nCardsOnHand; i < HAND_SIZE; i++) {
            pgc[TABLE_SIZE + i] = adeck[deckP++];
        }
        // sorteia o resto da mesa.
        for (i = nCardsOnTable; i < TABLE_SIZE; i++) {
            int card = adeck[deckP++];
            pgc[i] = card;
            for (j = 0; j < nEnemies; j++) {
                egc[j * GAME_SIZE + i] = card;
            }
        }
        // sorteia as cartas dos inimigos
        for (j = (nEnemies - 1) * GAME_SIZE; j >= 0; j -= GAME_SIZE) {
            for (i = TABLE_SIZE; i < GAME_SIZE; i++) {
                egc[j + i] = adeck[deckP++];
            }
        }

        pow = getGameCardsPower(pgc);
        for (j = (nEnemies - 1) * GAME_SIZE; j >= 0; j -= GAME_SIZE) {
            if (pow < getGameCardsPower(egc + j)) {
                iLose++;
                break;
            }
        }
    }
    return gameCount - iLose;
}
```

7.3. getpower.h

Este método retorna um número representando o ranking de um jogador. *Por exemplo, se o jogador tem um par com um K, J e 5 para desempate, esse jogo é transformado em um número inteiro sendo que o maior número é o maior jogo.*

```
/*
 * author Vinícius Sousa Fazio
 */

int getGameCardsPower(int* someoneGameCards) {
    /**
     *
     * return int: 4 bits (game) + 20 bits (cards) game: 0: highcard 1:
pair
     * 2: 2 pairs 3: 3 of a kind 4: straight 5: flush 6: fullhouse 7: 4 of
a
     * kind 8: straight flush
     *
     * cartas: 4 bits por carta, em ordem descendente.
     *
     * os [>> 2], [<< 2] e [& 3] são para otimizar as contas [/
     * CARD_TYPE_SIZE], [* CARD_TYPE_SIZE] e [%
     * CARD_TYPE_SIZE] respectivamente.
     */

    int nPair = 0;
    int nThreeOfAKind = 0;
    int nFourOfAKind = 0;
    bool straight = false;
    bool flush = false;
    bool straightFlush = false;
    int straightCard = 0;
    int straightFlushCard = 0;
    int power = 0;

    int* sgc = sortedGameCards;
    int* sgcs = sortedGameCardsShifted;

    //temporário
    int i, j, repeated;
    for (i = 0; i < GAME_SIZE; i++)
    {
        sgc[i] = someoneGameCards[i];
    }

    // ordena as cartas em ordem descendente
    for (i = 0; i < GAME_SIZE; i++) {
        bool changed = false;
        for (j = 0; j < GAME_SIZE - 1; j++) {
            int s1 = sgc[j];
            int s2 = sgc[j + 1];
            if (s1 < s2) {
```

```
                sgc[j] = s2;
                sgc[j + 1] = s1;
                changed = true;
            }
        }
        if (!changed) break;
    }

    for (i = 0; i < GAME_SIZE; i++) {
        sgcs[i] = sgc[i] >> 2;
    }

    // pair, 3-of-a-kind, 4-of-a-kind, full house e 2 pairs
    repeated = 0;
    for (i = 1; i < GAME_SIZE; i++) {
        if ((sgcs[i - 1]) == (sgcs[i])) {
            repeated++;
            switch (repeated) {
                case 1:
                    pairCards[nPair++] = i;
                    pairCards[nPair++] = i - 1;
                    break;
                case 2:
                    threeOfAKindCards[nThreeOfAKind++] = i;
                    threeOfAKindCards[nThreeOfAKind++] = pairCards[--
nPair];
                    threeOfAKindCards[nThreeOfAKind++] = pairCards[--
nPair];
                    break;
                case 3:
                    fourOfAKindCards[nFourOfAKind++] = i;
                    fourOfAKindCards[nFourOfAKind++] = threeOfAKindCards[--
nThreeOfAKind];
                    fourOfAKindCards[nFourOfAKind++] = threeOfAKindCards[--
nThreeOfAKind];
                    fourOfAKindCards[nFourOfAKind++] = threeOfAKindCards[--
nThreeOfAKind];
                    break;
            }
        } else {
            repeated = 0;
        }
    }

    // straight flush
    for (j = 0; j < GAME_SIZE - 3 && !straightFlush; j++) {
        int cardsInARow = 1 << 2;

        int a = sgc[j];
        for (i = j + 1; i < GAME_SIZE; i++) {
            int b = sgc[i];
            if (a - b == cardsInARow) {
                cardsInARow += 1 << 2;
                if (cardsInARow == 5 << 2) {
                    straightFlushCard = j;
                    straightFlush = true;
                    break;
                }
            }
```

```
        }
        if (cardsInARow == 4 << 2 && // 4 cartas na sequencia
        (a >> 2) == 3 && !straightFlush) // sequencia até 5
        {
            for (i = 0; i < CARD_TYPE_SIZE; i++) {
                if (sgcs[i] == CARD_A) // tem A
                {
                    if ((sgc[i] & 3) == (a & 3)) // mesmo
                        // naipe
                    {
                        straightFlushCard = j;
                        straightFlush = true;
                        break;
                    }
                } else {
                    break;
                }

            }
        }
    }

    if (!straightFlush) {

        // flush
        int nCardsOfEachType[CARD_TYPE_SIZE];
        for (i = 0; i < CARD_TYPE_SIZE; i++) {
            nCardsOfEachType[i] = 0;
        }
        for (i = 0; i < GAME_SIZE; i++) {
            int t = sgc[i] & 3; // naipe
            if (++nCardsOfEachType[t] == 5) {
                int count = 0;
                flush = true;
                for (j = 0; j <= i; j++) {
                    if (t == (sgc[j] & 3)) {
                        flushCards[count++] = j;
                    }
                }
                break;
            }
        }

        // straight
        if (!flush) {
            int cardsInARow = 1;
            bool hasA = sgcs[0] == CARD_A;
            int lastCard = sgcs[0];
            straightCard = 0;
            for (i = 1; i < GAME_SIZE && !straight; i++) {
                int newCard = sgcs[i];
                switch (lastCard - newCard) {
                    case 1:
                        switch (++cardsInARow) {
                            case 5:
                                straight = true;
                                break;
                            case 4:
                                if (hasA && newCard == 0) // seq de 'a' a
```

```
'5'
                                {
                                    straight = true;
                                }
                                break;
                            }
                            break;
                        case 0:
                            // carta repetida
                            break;
                            default:
                                cardsInARow = 1;
                                straightCard = i;
                    }
                    lastCard = newCard;
                }
            }

        }

        power = 0;
        if (straightFlush) {
            power = sgcs[straightFlushCard];
            power |= 8 << (5 * CARD_BIT_SIZE);
        } else if (nFourOfAKind == 4) {
            power = sgcs[fourOfAKindCards[0]];
            power <<= CARD_BIT_SIZE;
            power |= getOtherCards(sgcs,
            fourOfAKindCards, 4, 1);
            power |= 7 << (5 * CARD_BIT_SIZE);
        } else if ((nThreeOfAKind >= 3 && nPair >= 2)) {
            power |= sgcs[threeOfAKindCards[0]];
            power <<= CARD_BIT_SIZE;
            if (nThreeOfAKind > 3) {
                int card1 = sgcs[pairCards[0]];
                int card2 = sgcs[threeOfAKindCards[3]];
                power |= card1 > card2 ? card1 : card2;
            } else {
                power |= sgcs[pairCards[0]];
            }
            power |= 6 << (5 * CARD_BIT_SIZE);
        } else if (nThreeOfAKind >= 6) {
            power |= sgcs[threeOfAKindCards[0]];
            power <<= CARD_BIT_SIZE;
            power |= sgcs[threeOfAKindCards[3]];
            power |= 6 << (5 * CARD_BIT_SIZE);
        } else if (flush) {
            for (i = 0; i < 5; i++) {
                power <<= CARD_BIT_SIZE;
                power |= sgcs[flushCards[i]];
            }
            power |= 5 << (5 * CARD_BIT_SIZE);
        } else if (straight) {
            power = sgcs[straightCard];
            power |= 4 << (5 * CARD_BIT_SIZE);
        } else if (nThreeOfAKind == 3) {
            power = sgcs[threeOfAKindCards[0]];
            power <<= 2 * CARD_BIT_SIZE;
            power |= getOtherCards(sgcs,
```

```
            threeOfAKindCards, 3, 2);
            power |= 3 << (5 * CARD_BIT_SIZE);
        } else if (nPair >= 4) {
            power = sgcs[pairCards[0]];
            power <<= CARD_BIT_SIZE;
            power |= sgcs[pairCards[2]];
            power <<= CARD_BIT_SIZE;
            power |= getOtherCards(sgcs, pairCards, 4, 1);
            power |= 2 << (5 * CARD_BIT_SIZE);
        } else if (nPair == 2) {
            power = sgcs[pairCards[0]];
            power <<= 3 * CARD_BIT_SIZE;
            power |= getOtherCards(sgcs, pairCards, 2, 3);
            power |= 1 << (5 * CARD_BIT_SIZE);
        } else // high card
        {
            power = sgcs[0];
            power <<= CARD_BIT_SIZE;
            power |= sgcs[1];
            power <<= CARD_BIT_SIZE;
            power |= sgcs[2];
            power <<= CARD_BIT_SIZE;
            power |= sgcs[3];
            power <<= CARD_BIT_SIZE;
            power |= sgcs[4];
        }
        return power;
    }
    int getOtherCards(int* hand, int* game, int gameCards,
    int qty) {
        int i, g, cards = 0;
        for (i = 0; i < GAME_SIZE; i++) {
            bool inGame = false;
            for (g = 0; g < gameCards; g++) {
                if (game[g] == i) {
                    inGame = true;
                    break;
                }
            }
            if (!inGame && qty > 0) {
                cards <<= CARD_BIT_SIZE;
                cards |= hand[i];
                qty--;
            }
        }
        return cards;
    }
```

7.4. pokeronerank.c

Este método faz a interface com o Matlab para retornar o ranking de um jogador.

```
/*
 * author Vinícius Sousa Fazio
 */

#include "mex.h"
#define GAME_SIZE 7
#define CARD_TYPE_SIZE 13
#define CARD_A CARD_TYPE_SIZE - 1
#define CARD_BIT_SIZE 4
#define HAND_SIZE 2
#define TABLE_SIZE 5
#define CARDS_QUANTITY CARD_TYPE_SIZE << 2

int* sortedGameCards;
int* sortedGameCardsShifted;
int* pairCards;
int* threeOfAKindCards;
int* fourOfAKindCards;
int* flushCards;

#include "getpower.h"

void mexFunction( int nlhs, mxArray *plhs[],
int nrhs, const mxArray *prhs[] )
{
    double r;

    int* hand;
    double* allyHandD;
    double* tableD;
    int i;

    if (nrhs != 2){
        mexErrMsgTxt ("Número de argumentos inválido.");
        return;
    }

    allyHandD = mxGetPr(prhs[0]);
    tableD = mxGetPr(prhs[1]);

    hand = malloc(GAME_SIZE * sizeof(int));
    sortedGameCards = malloc(GAME_SIZE * sizeof(int));
    sortedGameCardsShifted = malloc(GAME_SIZE * sizeof(int));
    pairCards = malloc(GAME_SIZE * sizeof(int));
    threeOfAKindCards = malloc(GAME_SIZE * sizeof(int));
    fourOfAKindCards = malloc(GAME_SIZE * sizeof(int));
    flushCards = malloc(5 * sizeof(int));

    for (i = 0; i < HAND_SIZE; i++)
    {
```

```
        hand[i] = (int)allyHandD[i]-1;
    }
    for (i = 0; i < TABLE_SIZE; i++)
    {
        hand[i+HAND_SIZE] = (int)tableD[i]-1;
    }

    r=getGameCardsPower(hand);

    free(hand);
    free(sortedGameCards);
    free(sortedGameCardsShifted);
    free(pairCards);
    free(threeOfAKindCards);
    free(fourOfAKindCards);
    free(flushCards);

    plhs[0] = mxCreateDoubleMatrix(1,1, mxREAL);
    mxGetPr(plhs[0])[0] = r;
}
```

7.5. pokerrank.c

Este método faz a interface com o Matlab para retornar a probabilidade de vencer de um jogador pelo método Monte-Carlo.

```
/*
 * author Vinícius Sousa Fazio
 */

#include "mex.h"

#define GAME_SIZE 7
#define CARD_TYPE_SIZE 13
#define CARD_A CARD_TYPE_SIZE - 1
#define CARD_BIT_SIZE 4
#define HAND_SIZE 2
#define TABLE_SIZE 5
#define CARDS_QUANTITY CARD_TYPE_SIZE << 2

int* sortedGameCards;
int* sortedGameCardsShifted;
int* pairCards;
int* threeOfAKindCards;
int* fourOfAKindCards;
int* flushCards;
int* playerGameCards;
int* enemyGameCards;
int* deck;

#include "compute_victory.h"

void mexFunction( int nlhs, mxArray *plhs[],
int nrhs, const mxArray *prhs[] )
{
    double r;

    int nCardsOnHand;
    int nCardsOnTable;
    int* allyHand;
    int* table;
    double* allyHandD;
    double* tableD;
    int nEnemies;
    int gameCount;
    int i;

    if (nrhs != 6){
        mexErrMsgTxt ("Número de argumentos inválido.");
        return;
    }

    nCardsOnHand = mxGetScalar(prhs[0]);
    nCardsOnTable = mxGetScalar(prhs[1]);
```

```
    allyHandD = mxGetPr(prhs[2]);
    tableD = mxGetPr(prhs[3]);
    nEnemies = mxGetScalar(prhs[4]);
    gameCount = mxGetScalar(prhs[5]);

    allyHand = malloc(nCardsOnHand * sizeof(int));
    table = malloc(nCardsOnTable * sizeof(int));
    sortedGameCards = malloc(GAME_SIZE * sizeof(int));
    sortedGameCardsShifted = malloc(GAME_SIZE * sizeof(int));
    pairCards = malloc(GAME_SIZE * sizeof(int));
    threeOfAKindCards = malloc(GAME_SIZE * sizeof(int));
    fourOfAKindCards = malloc(GAME_SIZE * sizeof(int));
    flushCards = malloc(5 * sizeof(int));
    playerGameCards = malloc(GAME_SIZE * sizeof(int));
    enemyGameCards = malloc(GAME_SIZE * nEnemies * sizeof(int));
    deck = malloc(CARDS_QUANTITY * sizeof(int));;

    for (i = 0; i < nCardsOnHand; i++)
    {
        allyHand[i] = (int)allyHandD[i]-1;
    }
    for (i = 0; i < nCardsOnTable; i++)
    {
        table[i] = (int)tableD[i]-1;
    }

    r = computeRealVictoriesRandomly(nCardsOnHand, nCardsOnTable, allyHand,
table, nEnemies, gameCount);

    free(allyHand);
    free(table);
    free(sortedGameCards);
    free(sortedGameCardsShifted);
    free(pairCards);
    free(threeOfAKindCards);
    free(fourOfAKindCards);
    free(flushCards);
    free(playerGameCards);
    free(enemyGameCards);
    free(deck);

    plhs[0] = mxCreateDoubleMatrix(1,1, mxREAL);
    mxGetPr(plhs[0])[0] = ((double) (100*r)) / (double) gameCount;
}
```

7.6. test_pokerrank.c

Este método faz a interface com o Matlab para gerar os dados para demonstrar a convergencia do método Monte-Carlo.

```
/*
 * author Vinícius Sousa Fazio
 */

#include "mex.h"

#define GAME_SIZE 7
#define CARD_TYPE_SIZE 13
#define CARD_A CARD_TYPE_SIZE - 1
#define CARD_BIT_SIZE 4
#define HAND_SIZE 2
#define TABLE_SIZE 5
#define CARDS_QUANTITY CARD_TYPE_SIZE << 2

int* sortedGameCards;
int* sortedGameCardsShifted;
int* pairCards;
int* threeOfAKindCards;
int* fourOfAKindCards;
int* flushCards;
int* playerGameCards;
int* enemyGameCards;
int* deck;

#include "compute_victory.h"

void mexFunction( int nlhs, mxArray *plhs[],
int nrhs, const mxArray *prhs[] )
{
    double r;

    int nCardsOnHand;
    int nCardsOnTable;
    int* allyHand;
    int* table;
    double* allyHandD;
    double* tableD;
    int nEnemies;
    int gameCount;
    int i;
    int winCount;

    if (nrhs != 6){
        mexErrMsgTxt ("Número de argumentos inválido.");
        return;
    }

    nCardsOnHand = mxGetScalar(prhs[0]);
```

```
    nCardsOnTable = mxGetScalar(prhs[1]);
    allyHandD = mxGetPr(prhs[2]);
    tableD = mxGetPr(prhs[3]);
    nEnemies = mxGetScalar(prhs[4]);
    gameCount = mxGetScalar(prhs[5]);

    allyHand = malloc(nCardsOnHand * sizeof(int));
    table = malloc(nCardsOnTable * sizeof(int));
    sortedGameCards = malloc(GAME_SIZE * sizeof(int));
    sortedGameCardsShifted = malloc(GAME_SIZE * sizeof(int));
    pairCards = malloc(GAME_SIZE * sizeof(int));
    threeOfAKindCards = malloc(GAME_SIZE * sizeof(int));
    fourOfAKindCards = malloc(GAME_SIZE * sizeof(int));
    flushCards = malloc(5 * sizeof(int));
    playerGameCards = malloc(GAME_SIZE * sizeof(int));
    enemyGameCards = malloc(GAME_SIZE * nEnemies * sizeof(int));
    deck = malloc(CARDS_QUANTITY * sizeof(int));;

    for (i = 0; i < nCardsOnHand; i++)
    {
        allyHand[i] = (int)allyHandD[i]-1;
    }
    for (i = 0; i < nCardsOnTable; i++)
    {
        table[i] = (int)tableD[i]-1;
    }

    plhs[0] = mxCreateDoubleMatrix(gameCount,1, mxREAL);
    winCount = 0;
    for (i = 0; i < gameCount; i++){
        winCount += computeRealVictoriesRandomly(nCardsOnHand,
nCardsOnTable, allyHand, table, nEnemies, 1);
        mxGetPr(plhs[0])[i] = ((double) (100*winCount)) / (double) (i+1);
    }
    free(allyHand);
    free(table);
    free(sortedGameCards);
    free(sortedGameCardsShifted);
    free(pairCards);
    free(threeOfAKindCards);
    free(fourOfAKindCards);
    free(flushCards);
    free(playerGameCards);
    free(enemyGameCards);
    free(deck);

}
```

7.7. createconstantplayer.m

Este método cria o jogador Constante.

```
% @author Vinícius Sousa Fazio
function [ playerai ] = createconstantplayer(dec)
% Cria um jogador que joga sempre joga a mesma coisa
    playerai=struct('decide',@decide,'learn',@learn,'get',@get);
    d=dec;
    function [ decision ] =
decide(odds,pot,raise,round,chances,nplayers,followers,ingame,qtyraise)
%#ok<INUSD>
        decision=d;
    end
    function [] = get() %#ok<INUSD>
    end
    function [] = learn() %#ok<INUSD>
    end

end
```

7.8. creatediscreteplayer.m

Este método cria o jogador com aprendizado por reforço.

```
% @author Vinícius Sousa Fazio
function [ playerai ] =
creatediscreteplayer(stopLearn,oddsSc,chanceSc,potSc,raiseSc,qtyraiseSc,fol
lowersSc,ingameSc,nplayersSc,arraySiz,littleArraySiz)
% Cria um jogador que joga através do aprendizado por esforço
playerai=struct('decide',@decide,'learn',@learn,'get',@get);

%CUIDADO com o value de littleArraySiz e arraySiz para não estourar a
%memória

%personality paramenters
stopLearning=stopLearn;
verybigvalue=999999;
if oddsSc == -1
    oddsScaling=verybigvalue;
    oddsSize = 1;
else
    oddsScaling=oddsSc;
    oddsSize = arraySiz;
end

if chanceSc == -1
    chancesScaling=verybigvalue;
    chanceSize = 1;
else
    chancesScaling=chanceSc;
    chanceSize = arraySiz;
end

if potSc == -1
    potScaling=verybigvalue;
    potSize = 1;
else
    potScaling=potSc;
    potSize = arraySiz;
end

if raiseSc == -1
    raiseScaling=verybigvalue;
    raiseSize = 1;
else
    raiseScaling=raiseSc;
    raiseSize = arraySiz;
end

if qtyraiseSc == -1
    qtyraiseScaling=verybigvalue;
    qtyraiseSize = 1;
else
```

```
    qtyraiseScaling=qtyraiseSc;
    qtyraiseSize = littleArraySiz;
end

if followersSc == -1
    followersScaling=verybigvalue;
    followersSize = 1;
else
    followersScaling=followersSc;
    followersSize = littleArraySiz;
end

if ingameSc == -1
    ingameScaling=verybigvalue;
    ingameSize = 1;
else
    ingameScaling=ingameSc;
    ingameSize = littleArraySiz;
end

if nplayersSc == -1
    nplayersScaling=verybigvalue;
    nplayersSize = 1;
else
    nplayersScaling=nplayersSc;
    nplayersSize = littleArraySiz;
end
numrounds=2;

% odds, pot, raise, chances, round, qtyraise, followers, ingame, nplayers,
win/lose, decision
db=zeros(oddsSize,potSize,raiseSize,chanceSize,numrounds,qtyraiseSize,follo
wersSize,ingameSize,nplayersSize,2,6);
% odds, pot, raise, chances,  round, qtyraise, followers, ingame, nplayers,
win/lose, decision
qty=zeros(oddsSize,potSize,raiseSize,chanceSize,numrounds,qtyraiseSize,foll
owersSize,ingameSize,nplayersSize,2,6);

RANDOM_DECISION=0;
BEST_REC_DIFFERENCE=1;
BEST_REC_RELATION=2;
BEST_VIC_RELATION=3;
WHEEL_REC_DIFFERENCE=4;
WHEEL_VIC_RELATION=5;
WHEEL_VIC_ABSOLUTE=6;

decisionkind=BEST_REC_DIFFERENCE;

    function [ decision ] =
decide(odds,pot,raise,round,chances,nplayers,followers,ingame,qtyraise)
        if nargin == 1
            decisionkind=odds;
        elseif nargin == 0
            %não decide nada, apenas plota as decisões.
            disp(['stopLearning ' num2str(stopLearning)]);
            disp(['oddsScaling ' num2str(oddsScaling)]);
            disp(['chancesScaling ' num2str(chancesScaling)]);
            disp(['potScaling ' num2str(potScaling)]);
            disp(['raiseScaling ' num2str(raiseScaling)]);
```

```
            disp(['qtyraiseScaling ' num2str(qtyraiseScaling)]);
            disp(['followersScaling ' num2str(followersScaling)]);
            disp(['ingameScaling ' num2str(ingameScaling)]);
            disp(['nplayersScaling ' num2str(nplayersScaling)]);
            clear tmp;
            tmp=NaN(oddsSize*potSize,raiseSize*chanceSize*qtyraiseSize*foll
owersSize*ingameSize*nplayersSize*numrounds);
            k=0;
            for nplayersD=1:nplayersSize;
                for ingameD=1:ingameSize;
                    for followersD=1:followersSize;
                        for qtyraiseD=1:qtyraiseSize;
                            for round=1:2
                                for chancesD=1:chanceSize
                                    for raiseD=1:raiseSize
                                        for potD=1:potSize
                                            for oddsD=1:oddsSize
                                                k=k+1;
                                                decision=decideWithGame();
                                                tmp(k)=decision;
                                            end
                                        end
                                    end
                                end
                            end
                        end
                    end
                end
            end
            pcolor(tmp);
            shading flat;
            clear tmp;
        else
            %discretiza os argumentos
            oddsD=min(fix(odds/oddsScaling)+1,oddsSize);
            chancesD=min(fix(chances/chancesScaling)+1,chanceSize);
            potD=min(fix(pot/potScaling)+1,potSize);
            raiseD=min(fix(raise/raiseScaling)+1,raiseSize);
            qtyraiseD=min(fix(qtyraise/qtyraiseScaling)+1,qtyraiseSize);
            followersD=min(fix(followers/followersScaling)
+1,followersSize);
            ingameD=min(fix(ingame/ingameScaling)+1,ingameSize);
            nplayersD=min(fix(nplayers/nplayersScaling)+1,nplayersSize);
            round=min(round,numrounds);
            decision=decideWithGame();
        end

        function [ decision ] = decideWithGame()
            %extrai informação no banco para uma decisão
            reswin=db(oddsD,potD,raiseD,chancesD,round,qtyraiseD,followersD
,ingameD,nplayersD,1,:);
            reslose=db(oddsD,potD,raiseD,chancesD,round,qtyraiseD,followers
D,ingameD,nplayersD,2,:);
            sizwin=qty(oddsD,potD,raiseD,chancesD,round,qtyraiseD,followers
D,ingameD,nplayersD,1,:);
            sizlose=qty(oddsD,potD,raiseD,chancesD,round,qtyraiseD,follower
sD,ingameD,nplayersD,2,:);

            decision=NaN;
```

```
            %só usa decisão se o numero de decisoes é maior que o limiar
            if sum(sizwin)+sum(sizlose) > stopLearning
                switch decisionkind
                    case BEST_REC_DIFFERENCE %melhor recompensa (diferença)
em dinheiro
                        maxres=NaN;
                        for dec=1:6
                            sizw=sizwin(dec);
                            sizl=sizlose(dec);
                            if sizw + sizl > 0
                                res=(reswin(dec)-reslose(dec))/(sizw+sizl);
                                if isnan(maxres) || res > maxres
                                    maxres=res;
                                    decision=dec;
                                end
                            end
                        end
                    case BEST_REC_RELATION %melhor recompensa (relação) em
dinheiro
                        maxres=NaN;
                        for dec=1:6
                            sizw=sizwin(dec);
                            sizl=sizlose(dec);
                            if sizw + sizl > 0
                                res=((reswin(dec)+1)/(reslose(dec)+1)) *
((sizl+1)/(sizw+1));
                                if isnan(maxres) || res > maxres
                                    maxres=res;
                                    decision=dec;
                                end
                            end
                        end
                    case BEST_VIC_RELATION %melhor vitória (relação)
                        maxres=NaN;
                        for dec=1:6
                            if sizwin(dec) + sizlose(dec) > 0
                                res=(sizwin(dec)/(sizlose(dec)+1));
                                if isnan(maxres) || res > maxres
                                    maxres=res;
                                    decision=dec;
                                end
                            end
                        end
                    case WHEEL_REC_DIFFERENCE %roleta de melhor (diferença)
recompensa
                        receachdecision=[0 0 0 0 0 0];
                        for dec=1:6
                            sizw=sizwin(dec);
                            sizl=sizlose(dec);
                            if sizw + sizl > 0
                                res=fix(100*(reswin(dec)-reslose(dec))/
(sizw+sizl));
                                receachdecision(dec)=res;
                            end
                        end
                        sumreceachdecision=sum(receachdecision(receachdecis
ion > 0));
                        if sumreceachdecision == 0
                            receachdecision=receachdecision-
```

```
min(receachdecision);
                            sumreceachdecision=sum(receachdecision(receachd
ecision > 0));
                        end
                        rnddecision=rand*sumreceachdecision;
                        for dec=1:6
                            if receachdecision(dec)>0
                                rnddecision=rnddecision-
receachdecision(dec);
                                if rnddecision < 0
                                    decision=dec;
                                    break;
                                end
                            end
                        end
                    case WHEEL_VIC_RELATION %roleta de maior (relação)
vitória
                        receachdecision=[0 0 0 0 0 0];
                        for dec=1:6
                            receachdecision(dec)=fix(sizwin(dec)/
(1+sizlose(dec)));
                        end
                        sumreceachdecision=sum(receachdecision);
                        if sumreceachdecision >= 1
                            rnddecision=rand*sumreceachdecision;
                            for dec=1:6
                                rnddecision=rnddecision-
receachdecision(dec);
                                if rnddecision < 0
                                    decision=dec;
                                    break;
                                end
                            end
                        else
                            decision=1;
                        end
                    case WHEEL_VIC_ABSOLUTE %roleta de maior (absoluto)
vitória
                        sumsizwin=sum(sizwin);
                        if sumsizwin >= 1
                            rnddecision=sumsizwin*rand;
                            for dec=1:6
                                rnddecision=rnddecision-sizwin(dec);
                                if rnddecision < 0
                                    decision=dec;
                                    break;
                                end
                            end
                        end
                    case RANDOM_DECISION
                        % não faz nada pq vai tomar a decisão aleatória no
fim
                end
            end
            if isnan(decision)
                switch fix(rand*9)
                    case 0
                        decision = 1;
                    case 1
```

```
                        decision = 3;
                    case 2
                        decision = 4;
                    case 3
                        decision = 5;
                    case 4
                        decision = 6;
                    otherwise
                        decision = 2;
                end
%                decision=0;
            end
        end
    end
    function [ rdb, rqty ] = get()
        rdb=db;
        rqty=qty;
    end
    function [] = learn(argdb,argqty)
        if nargin == 0
            % não aprende. Apenas plota learn qty
            tmp=zeros(oddsSize*potSize*raiseSize*chanceSize,qtyraiseSize*fo
llowersSize*ingameSize*nplayersSize*numrounds*12);
            tmp(:)=qty(:);
            pcolor(tmp);
            shading flat;
        else
            db=argdb;
            qty=argqty;
        end
    end
end
```

7.9. createhumanplayer.m

Este método cria o jogador humano.

```
function [ playerai ] = createhumanplayer() %#ok<INUSD>
% @author Vinícius Sousa Fazio
% jogador humano

playerai=struct('decide',@decide,'learn',@learn,'get',@get);

    function [ decision ] =
decide(odds,pot,raise,round,chances,nplayers,followers,ingame,qtyraise)
%#ok<INUSD>
        disp(['odds: ' num2str(odds)]);
        disp(['pot: ' num2str(pot)]);
        disp(['raise: ' num2str(raise)]);
        disp(['round: ' num2str(round)]);
        disp(['chances: ' num2str(chances)]);
        disp(['nplay: ' num2str(nplayers)]);
        disp(['follow: ' num2str(followers)]);
        disp(['ingame: ' num2str(ingame)]);
        disp(['qtyraise: ' num2str(qtyraise)]);
        repeat=1;
        while repeat == 1
            decision=input('Escolha a jogada:');
            if decision==1 || decision==2 || decision==3 || decision==4 ||
decision==5 || decision==6
                repeat=0;
            end
        end

    end
    function [] = get() %#ok<INUSD>
    end
    function [] = learn() %#ok<INUSD>
    end
end
```

7.10. createmfsplayer.m

Este método cria o jogador MFS.

```
function [ playerai ] = createmfsplayer(argt)
% @author Vinícius Sousa Fazio
% Cria um jogador que joga matematicamente justo
    playerai=struct('decide',@decide,'learn',@learn,'get',@get);
    t = argt;
    function [ decision ] =
decide(odds,pot,raise,round,chances,nplayers,followers,ingame,qtyraise)
%#ok<INUSL>
        if (odds == 0)
            decision=2;
        else
            othermoney=(pot*(odds-1))/odds;
            mymoney=pot-othermoney+raise;
            up=(chances*othermoney)/100;
            down=((100-chances)*mymoney)/100;
            dif=up-down;
            if dif < -1 * t
                decision = 1;
            elseif dif < 1 * t
                decision = 2;
            elseif dif < 5 * t
                decision = 3;
            elseif dif < 10 * t
                decision = 4;
            elseif dif < 15 * t
                decision = 5;
            else
                decision = 6;
            end
        end
    end
    function [] = get() %#ok<INUSD>
    end
    function [] = learn() %#ok<INUSD>
    end

end
```

7.11. createrandomplayer.m

Este método cria o jogador Aleatório.

```
% @author Vinícius Sousa Fazio
function [ playerai ] = createrandomplayer() %#ok<INUSD>
% Jogador aleatório
playerai=struct('decide',@decide,'learn',@learn,'get',@get);


    function [ decision ] =
decide(odds,pot,raise,round,chances,nplayers,followers,ingame,qtyraise)
%#ok<INUSD>
        switch fix(rand*9)
            case 0
                decision = 1;
            case 1
                decision = 3;
            case 2
                decision = 4;
            case 3
                decision = 5;
            case 4
                decision = 6;
            otherwise
                decision = 2;
        end
    end
    function [] = get() %#ok<INUSD>
    end
    function [] = learn() %#ok<INUSD>
    end
end
```

7.12. createrhplayer.m

Este método cria o jogador RH.

```
% @author Vinícius Sousa Fazio
function [ playerai ] = createrhplayer(argt)
% Cria um jogador que joga de acordo com RH
    playerai=struct('decide',@decide,'learn',@learn,'get',@get);
    t = argt;
    function [ decision ] =
decide(odds,pot,raise,round,chances,nplayers,followers,ingame,qtyraise)
%#ok<INUSL>
        v = pot / ((qtyraise+1) * followers * ingame * (raise+1));
        if v < t
            decision = 1;
        elseif v < 20*t
            decision = 2;
        elseif v < 50*t
            decision = 3;
        elseif v < 100*t
            decision = 4;
        elseif v < 1000*t
            decision = 5;
        else
            decision = 6;
        end
%        disp(['v ' num2str(v) ' decision ' num2str(decision)]);
    end
    function [] = get() %#ok<INUSD>
    end
    function [] = learn() %#ok<INUSD>
    end

end
```

7.13. createdatabase.m

Este método cria a base de dados de jogadas para ser utilizada mais tarde pelos jogadores evolutivos.

```
% @author Vinícius Sousa Fazio
function [ result ] = createdatabase()
% Cria uma base de dados de jogadas e resultados
    NVARS=11;

    result=struct('get',@get,'insert',@insert);
    db = zeros(100000,NVARS);

    size=0;

%% get database
    function [ resdb,siz ] = get()
        resdb=db;
        siz=size;
    end
%% insert a new element
    function [ ndata ] =
insert(odds,pot,raise,round,chances,nplayers,followers,ingame,qtyraise,deci
sion,result)
        if size < length(db) - 1
            ndata=size;
            if nargin ~= 0
                size=size+1;
                db(size,1:NVARS)=[odds pot raise round chances nplayers
followers ingame qtyraise decision result];
            end
        else
            ndata=0;
        end
    end
end
```

7.14. traindiscrete.c

Este método treina o jogador com aprendizado por esforço.

```
/*
 * Treina o jogador discreto. Feito em C pelo maior desempenho
 * author Vinícius Sousa Fazio
 */

#include "mex.h"

#define DECISIONS 6
#define ROUNDS 2
#define WINLOSES 2


#define HODD 0
#define HPOT 1
#define HRAISE 2
#define HROUND 3
#define HCHANCE 4
#define HNPLAY 5
#define HFOLLOW 6
#define HINGAME 7
#define HQTYRAISE 8
#define HDECISION 9
#define HRESULT 10

#define VERYBIGVALUE 999999

void mexFunction( int nlhs, mxArray *plhs[],
int nrhs, const mxArray *prhs[] )
{
    // entrada
    double to, tr, tc, tp, tq, tf, ti, tn;
    int maxr, maxc, maxp, maxo, maxq, maxf, maxi, maxn;
    int maxhistory,maxh,hsize;
    mwSize *pdbsize;
    mwSize *hdbsize;
    double *qty;
    double *pdb;
    double *hdb;

    // calculo
    double odd, pot, raise, chance, round, qtyraise, follow, ingame, nplay,
result;
    int oddD, potD, raiseD, chanceD, roundD, qtyraiseD, followD, ingameD,
nplayD, decision;
    int winlose,siz;
    double newres;
    int ip;

    // constantes
    int hodd, hpot, hraise, hchance, hround, hqtyraise, hfollow, hingame,
```

```
hnplay, hdecision, hresult;
    int podd, ppot, praise, pchance, pround, pqtyraise, pfollow, pingame,
pnplay, pdecision, pwinlose;

    int i;

    // checa número de argumentos
    if (nrhs != 12){
        mexErrMsgTxt ("Número de argumentos inválido.");
        return;
    }

    // lê a entrada
    to = mxGetScalar(prhs[0]);
    tr = mxGetScalar(prhs[1]);
    tc = mxGetScalar(prhs[2]);
    tp = mxGetScalar(prhs[3]);

    tq = mxGetScalar(prhs[4]);
    tf = mxGetScalar(prhs[5]);
    ti = mxGetScalar(prhs[6]);
    tn = mxGetScalar(prhs[7]);

    to = to == -1 ? VERYBIGVALUE : to;
    tr = tr == -1 ? VERYBIGVALUE : tr;
    tc = tc == -1 ? VERYBIGVALUE : tc;
    tp = tp == -1 ? VERYBIGVALUE : tp;
    tq = tq == -1 ? VERYBIGVALUE : tq;
    tf = tf == -1 ? VERYBIGVALUE : tf;
    ti = ti == -1 ? VERYBIGVALUE : ti;
    tn = tn == -1 ? VERYBIGVALUE : tn;

    qty = mxGetPr(prhs[8]);
    pdb = mxGetPr(prhs[9]);
    hsize = mxGetScalar(prhs[10]);
    hdb = mxGetPr(prhs[11]);
    pdbsize = mxGetDimensions(prhs[9]);
    hdbsize = mxGetDimensions(prhs[11]);

    maxo=(int)(pdbsize[0]);
    maxp=(int)(pdbsize[1]);
    maxr=(int)(pdbsize[2]);
    maxc=(int)(pdbsize[3]);
    maxq=(int)(pdbsize[5]);
    maxf=(int)(pdbsize[6]);
    maxi=(int)(pdbsize[7]);
    maxn=(int)(pdbsize[8]);

    maxhistory=((int)(hdbsize[0]));
    maxh=(int)(hdbsize[1]);

    // gera constantes para auxiliar os ponteiros das bases de dados
    hodd=HODD * maxhistory;
    hpot=HPOT * maxhistory;
    hraise=HRAISE * maxhistory;
    hchance=HCHANCE * maxhistory;
    hround=HROUND * maxhistory;
    hnplay=HNPLAY * maxhistory;
    hfollow=HFOLLOW * maxhistory;
```

```
hingame=HINGAME * maxhistory;
hqtyraise=HQTYRAISE * maxhistory;
hresult=HRESULT * maxhistory;
hdecision=HDECISION * maxhistory;

i = 1;
podd=i;
i *= maxo;
ppot=i;
i *= maxp;
praise=i;
i *= maxr;
pchance=i;
i *= maxc;
pround=i;
i *= ROUNDS;
pqtyraise=i;
i *= maxq;
pfollow=i;
i *= maxf;
pingame=i;
i *= maxi;
pnplay=i;
i *= maxn;
pwinlose=i;
i *= WINLOSES;
pdecision=i;

// loop com toda a base de dados
for (i = 0; i < hsize; i++){
    // separa a informação
    odd=hdb[i+hodd];
    pot=hdb[i+hpot];
    raise=hdb[i+hraise];
    chance=hdb[i+hchance];
    round=hdb[i+hround]-1;
    nplay=hdb[i+hnplay];
    follow=hdb[i+hfollow];
    ingame=hdb[i+hingame];
    qtyraise=hdb[i+hqtyraise];

    decision=(int)(hdb[i+hdecision])-1;
    result=hdb[i+hresult];

    // detecção básica de erro na base de dados
    if (pot < 0 || raise < 0 || chance < 0 || decision < 0 ||
        decision > 5 || odd < 0 || ingame < 0 || follow < 0 ||
        nplay < 0 || follow > nplay || ingame > nplay ||
        qtyraise < 0)
    {
        mexPrintf("index %i\n",i);
        mexErrMsgTxt("Dado inconsistente");
    }

    // discretiza a informação
    oddD=(int)(odd/to);
    chanceD=(int)(chance/tr);
    potD=(int)(pot/tp);
    raiseD=(int)(raise/tr);
```

```
            roundD=(int)round-1;
            nplayD=(int)(nplay/tn);
            followD=(int)(follow/tf);
            ingameD=(int)(ingame/ti);
            qtyraiseD=(int)(qtyraise/tq);

            if(roundD>=ROUNDS) roundD=ROUNDS-1;
            if(oddD>=maxo) oddD=maxo-1;
            if(chanceD>=maxc) chanceD=maxc-1;
            if(potD>=maxp) potD=maxp-1;
            if(raiseD>=maxr) raiseD=maxr-1;
            if(nplayD>=maxn) nplayD=maxn-1;
            if(followD>=maxf) followD=maxf-1;
            if(ingameD>=maxi) ingameD=maxi-1;
            if(qtyraiseD>=maxq) qtyraiseD=maxq-1;

            //insere na base do jogador as informações
            winlose=0;
            if (result < 0) {
                winlose=1;
                result=-result;
            }
            ip=(oddD*podd)+(potD*ppot)+(raiseD*praise)+(chanceD*pchance)+
               (roundD*pround)+(followD*pfollow)+(nplayD*pnplay)+
               (ingameD*pingame)+(qtyraiseD*pqtyraise)+(winlose*pwinlose)+
               (decision*pdecision);

            siz=(int)(qty[ip]);
            qty[ip]=qty[ip]+1;
            newres=0;
            if (siz>0) {
                newres=pdb[ip];
            }
            pdb[ip]=newres+result;
        }
    }
```

7.15. filldatabase.m

Este procedimento faz diversos jogos apenas para preencher a base de dados.

```
% @author Vinícius Sousa Fazio
while 1==1
    if rand < 0.05
        simulatealldispute;
    end

    tic
    disp('treinando');

    clear;
    initialmoney=1000;
    smallblind=10;
    maxgames=1000;
    to=2;
    tc=10;
    tp=10;
    tr=5;
    tf=1;
    tn=-1;
    tq=-1;
    ti=-1;
    n=10;
    ln=2;
    clear pd;
    clear pdb;
    clear qty;
    nepochs=1;
    pd=creatediscreteplayer(0,to,tc,tp,tr,tq,tf,ti,tn,n,ln);
    [pdb,qty]=pd.get();
    while exist (['simdatabase' int2str(nepochs) '.mat'], 'file') ~= 0
        clear database;
        clear hdb;
        clear hsiz;
        load(['simdatabase' int2str(nepochs)]);
        [hdb,hsiz]=database.get();
        traindiscrete(to,tr,tc,tp,tq,tf,ti,tn,qty,pdb,hsiz,hdb);
        nepochs=nepochs+1;
    end
    clear database;
    database=createdatabase();

    clear simplayers;
    for i=1:10
        simplayers(i)=pd; %#ok<AGROW>
    end
    for i=11:15
        simplayers(i)=createconstantplayer(2); %#ok<AGROW>
    end
    for i=16:20
        simplayers(i)=createconstantplayer(5); %#ok<AGROW>
```

```
    end
    for i=21:25
        simplayers(i)=createmfsplayer(1); %#ok<AGROW>
    end
    for i=26:30
        simplayers(i)=createmfsplayer(5); %#ok<AGROW>
    end
    for i=31:35
        simplayers(i)=createrhplayer(0.1); %#ok<AGROW>
    end
    for i=36:40
        simplayers(i)=createrhplayer(0.001); %#ok<AGROW>
    end
    for i=41:50
        simplayers(i)=createrandomplayer(); %#ok<AGROW>
    end

    disp('jogando');

    nplayers=2;
    while 1==1
        nplayers=nplayers+1;
        if nplayers>10
            nplayers=2;
        end
        playpoker(nplayers,simplayers,maxgames,initialmoney,smallblind,data
base,999);
        if database.insert() == 0
            break;
        end
    end

    disp(['salvando epoca ' int2str(nepochs)]);
    save (['simdatabase' int2str(nepochs)], 'database');
    disp(['salvo epoca ' int2str(nepochs)]);
    toc

end
```

7.16. newsimulationdispute.m

Este método foi criado para fazer as disputas entre dois jogadores de tipos diferentes.

```
% @author Vinícius Sousa Fazio
function [ simulation ] = newsimulationdispute( pd ,mfs,rh,constantdecision
)

    playerdiscrete=pd;
    playerhuman=createhumanplayer();
    playermfs=createmfsplayer(mfs);
    playerrh=createrhplayer(rh);
    playerrandom=createrandomplayer();
    playerconstant=createconstantplayer(constantdecision);

    simulation = struct('dispute',@dispute);

    function [ winb ] = dispute (a, b, qty)
        function [ p ] = getplayer(pid)
            if strcmp(pid, 'discrete')
                p = playerdiscrete;
            elseif strcmp(pid, 'mfs')
                p = playermfs;
            elseif strcmp(pid, 'rh')
                p = playerrh;
            elseif strcmp(pid, 'constant')
                p = playerconstant;
            elseif strcmp(pid, 'random')
                p = playerrandom;
            elseif strcmp(pid, 'human')
                p = playerhuman;
            end
        end
        playera = getplayer(a);
        playerb = getplayer(b);
        wina=0;
        winb=0;
        stylegame=1;
        if strcmp(a,'human') || strcmp(b,'human')
            stylegame=3;
        end

        for i=1:qty
            nplayers=mod(i,5)*2+2;
            if strcmp(a,'human')
                simplayers(1)=playera; %#ok<AGROW>
                for j=2:nplayers
                    simplayers(j)=playerb; %#ok<AGROW>
                end
            elseif strcmp(b,'human')
                simplayers(1)=playerb; %#ok<AGROW>
                for j=2:nplayers
                    simplayers(j)=playera; %#ok<AGROW>
                end
```

```
            else
                for j=1:(nplayers/2)
                    simplayers(j)=playera; %#ok<AGROW>
                end
                for j=(nplayers/2+1):nplayers
                    simplayers(j)=playerb; %#ok<AGROW>
                end
            end
            [ig,w]=playpoker(nplayers,simplayers,40,1000,10,{},stylegame);
            if strcmp(functiontostring(simplayers(w).decide),[ 'create' a
'player/decide' ])
                wina=wina+1;
            elseif strcmp(functiontostring(simplayers(w).decide),[ 'create'
b 'player/decide' ])
                winb=winb+1;
            end
%            disp ([a ': ' num2str(wina*100/i) ' ' b ': ' num2str(winb*100/
i)])
        end
        disp ([a ': ' num2str(wina*100/qty) ' ' b ': '
num2str(winb*100/qty)])
    end

end
```

7.17. newsimulationdispute2.m

Este método faz a disputa entre todos os jogadores.

```
% @author Vinícius Sousa Fazio
disp('taining');
clear simplayers
to=2;
tc=10;
tp=10;
tr=5;
tf=1;
tn=-1;
tq=-1;
ti=-1;
n=10;
ln=2;
st=600;
clear pd;
clear pdb;
clear qty;
nepochs=1;
pd=creatediscreteplayer(0,to,tc,tp,tr,tq,tf,ti,tn,n,ln);
[pdb,qty]=pd.get();
while exist (['simdatabase' int2str(nepochs) '.mat'], 'file') ~= 0 &&
nepochs <= st
    clear database;
    clear hdb;
    clear hsiz;
    load(['simdatabase' int2str(nepochs)]);
    [hdb,hsiz]=database.get();
    traindiscrete(to,tr,tc,tp,tq,tf,ti,tn,qty,pdb,hsiz,hdb);
    nepochs=nepochs+1;
end
disp('playing');
simplayers(1)=pd; %#ok<AGROW>
simplayers(2)=pd; %#ok<AGROW>
simplayers(3)=createmfsplayer(1); %#ok<AGROW>
simplayers(4)=createmfsplayer(5); %#ok<AGROW>
simplayers(5)=createrandomplayer(); %#ok<AGROW>
simplayers(6)=createrandomplayer(); %#ok<AGROW>
simplayers(7)=createconstantplayer(2); %#ok<AGROW>
simplayers(8)=createconstantplayer(5); %#ok<AGROW>
simplayers(9)=createrhplayer(0.1); %#ok<AGROW>
simplayers(10)=createrhplayer(0.001); %#ok<AGROW>

wins=[0 0 0 0 0 0 0 0 0 0];
for jj=1:500
    [ig,w]=playpoker(10,simplayers,40,1000,10,{},1);
    wins(w)=wins(w)+1;
    disp(int2str(jj));
end
wins
disp ('end');
```

7.18. simulatealldispute.m

Este método foi usado para ajustar o jogador com aprendizado por reforço. Ele faz simulações de disputa contra todos os outros tipos de jogadores e retorna um "fitness".

```
% @author Vinícius Sousa Fazio

function [ winb ] = simulatealldispute(args, nst)
%     for amfs=[0.2 0.4 0.7 1 2 4 5 6 7 10 20]
%         sim=newsimulationdispute(0,amfs,0,2);
%         disp(['mfs=' num2str(amfs)]);
%         sim.dispute('random','mfs',100);
%     end
%     disp '-------------------------';
%     for aconstant=2:6
%         sim=newsimulationdispute(0,1,0,aconstant);
%         disp(['constant=' num2str(aconstant)]);
%         sim.dispute('constant','random',100);
%     end
%     for amfs=[1 5 10]
%         for aconstant=2:6
%             sim=newsimulationdispute(0,amfs,0,aconstant);
%             disp(['mfs=' num2str(amfs) ' constant=' num2str(aconstant)]);
%             sim.dispute('constant','mfs',100);
%         end
%     end
%     disp '-------------------------';
%     for arh=[0.0001 0.001 0.01 0.05 0.1 0.5 1]
%         disp(['rh=' num2str(arh)]);
%         sim=newsimulationdispute(0,1,arh,0);
%         sim.dispute('random','rh',100);
%     end
%     for arh=[0.001 0.01 0.1]
%         for aconstant=[2 4 6]
%            disp(['rh=' num2str(arh) ' constant=' num2str(aconstant)]);
%             sim=newsimulationdispute(0,1,arh,aconstant);
%             sim.dispute('constant','rh',100);
%         end
%     end
%     for arh=[0.001 0.01 0.1]
%         for amfs=[1 5]
%             disp(['rh=' num2str(arh) ' mfs=' num2str(amfs)]);
%             sim=newsimulationdispute(0,amfs,arh,0);
%             sim.dispute('mfs','rh',100);
%         end
%     end
%     disp 'compiling...';
%     mex (which ('traindiscrete.c'));
    to=9.6;
    tc=23.3;
    tp=37.4;
    tr=13.2;
    tf=2.6;
```

```
    tn=3.5;
    tq=3.5;
    ti=5.6;
    n=4;
    ln=2;

    if 1 == 1
        to=2;
        tc=10;
        tp=10;
        tr=5;
        tf=1;
        tn=-1;
        tq=-1;
        ti=-1;
        n=10;
        ln=2;
    end
    st=9999;
    np=100;
    if nargin == 1
        to=args(1);
        tc=args(2);
        tp=args(3);
        tr=args(4);
        tf=args(5);
        tn=args(6);
        tq=args(7);
        ti=args(8);
        n=fix(args(9));
        ln=fix(args(10));

        st=800;
        np=100;
    end
    if nargin == 2
        st=nst;
        np=100;
    end
    disp (['                                        STOP = ' int2str(st)]);
    disp
'------------------------------------------------------------------------------'
;
    tic
    clear pd;
    clear pdb;
    clear qty;
    nepochs=1;
    pd=creatediscreteplayer(0,to,tc,tp,tr,tq,tf,ti,tn,n,ln);
    [pdb,qty]=pd.get();
    while exist (['simdatabase' int2str(nepochs) '.mat'], 'file') ~= 0 &&
nepochs <= st
        clear database;
        clear hdb;
        clear hsiz;
        load(['simdatabase' int2str(nepochs)]);
        [hdb,hsiz]=database.get();
        traindiscrete(to,tr,tc,tp,tq,tf,ti,tn,qty,pdb,hsiz,hdb);
        nepochs=nepochs+1;
```

```
    end

    disp(['@@ RECDIF to=' num2str(to) ' tc=' num2str(tc) ' tp=' num2str(tp)
' tr=' num2str(tr) ' tf=' num2str(tf) ' tn=' num2str(tn) ' tq=' num2str(tq)
' ti=' num2str(ti) ' n=' num2str(n) ' ln=' num2str(ln) ' mfs=1/5 cnt=2/5
rh=0.1/0.001']);
    clear sim;
    sim=newsimulationdispute(pd,1,0.1,2);
    winb = 0;
    winb = winb + 2 * sim.dispute('discrete','mfs',np);
    winb = winb + 4 * sim.dispute('discrete','random',np);
    winb = winb + sim.dispute('discrete','rh',np);
    winb = winb + sim.dispute('discrete','constant',np);
    clear sim;
    sim=newsimulationdispute(pd,5,0.001,5);
    winb = winb + 2 * sim.dispute('discrete','mfs',np);
    winb = winb + sim.dispute('discrete','constant',np);
    winb = winb + sim.dispute('discrete','rh',np);
    disp(['result: ' num2str(winb)]);
    clear pd;
    clear pdb;
    clear qty;
    clear database;
    clear hdb;
    clear hsiz;

    toc
end
```

7.19. simulatehuman.m

Este procedimento foi usado para jogar o jogador com aprendizado por reforço contra humanos.

```
% @author Vinícius Sousa Fazio
echo off;

to=9.6;
tc=23.3;
tp=37.4;
tr=13.2;
tf=2.6;
tn=3.5;
tq=3.5;
ti=5.6;
n=4;
ln=2;
st=999;

tic
clear pd;
clear pdb;
clear qty;
nepochs=1;
pd=creatediscreteplayer(0,to,tc,tp,tr,tq,tf,ti,tn,n,ln);
[pdb,qty]=pd.get();
while exist (['simdatabase' int2str(nepochs) '.mat'], 'file') ~= 0 &&
nepochs <= st
    clear database;
    clear hdb;
    clear hsiz;
    load(['simdatabase' int2str(nepochs)]);
    [hdb,hsiz]=database.get();
    traindiscrete(to,tr,tc,tp,tq,tf,ti,tn,qty,pdb,hsiz,hdb);
    nepochs=nepochs+1;
end

disp(['@@ RECDIF to=' num2str(to) ' tc=' num2str(tc) ' tp=' num2str(tp) '
tr=' num2str(tr) ' tf=' num2str(tf) ' tn=' num2str(tn) ' tq=' num2str(tq) '
ti=' num2str(ti) ' n=' num2str(n) ' ln=' num2str(ln) ' mfs=1/5 cnt=2/5
rh=0.1/0.001']);
clear sim;
sim=newsimulationdispute(pd,1,0.01,2);
%sim.dispute('human','discrete',1);
```

Algoritmos para um jogador inteligente de Poker

Vinícius Sousa Fazio[1]

[1]Departamento de Informática e Estatística – Universidade Federal do Santa Catarina (UFSC)
Caixa Postal 476 – 88.040-900 – Florianópolis – SC – Brasil

sfazio@inf.ufsc.br

Abstract. *Poker is a game of bluffing and probability of winning. The goal was to create and searh for algorithms that could learn to play well Poker. The evolving player created uses reinforcement learning where the game was divided in a huge state matrix with decision and reward's data. The player takes the decision with best average reward on each state. To compare the player's efficience, there were disputes with players that take decision based on formulas. The results were shown graphically on this paper.*

Resumo. *Poker é um jogo de blefe e probabilidade de vencer. O objetivo foi inventar de procurar algoritmos que aprendessem a jogar bem Poker. O jogador evolutivo criado utiliza aprendizado por reforço onde o jogo foi dividido em uma grande matriz de estados com dados de decisões e recompensas. O jogador toma a decisão com a melhor recompensa média para cada estado. Para comparar a eficiência do jogador, várias disputas com jogadores que tomam decisões baseado em fórmulas foram feitas. Os resultados foram mostrados graficamente no trabalho.*

1. Introdução

Ninguém sabe ao certo a origem do Poker (WIKIPEDIA POKER, 2008). O jogo mais antigo conhecido que tenha as características de blefe, "vence quem tem a melhor mão" e aposta é do século XV e de origem alemã, chamado Pochspiel. Há registros de pessoas jogando um jogo chamado As Nas no século XIX, que é muito parecido com poker e usa vinte cartas. Alguns historiadores descordam da origem do poker como uma variação do As Nas e acreditam vir de um jogo francês chamado Poque. O ator inglês Joseph Crowell relatou que o jogo era jogado em New Orleans em 1829, com um baralho de vinte cartas e quatro jogadores apostando qual "mão" era a mais valiosa. Logo após isso, o baralho inglês de 52 cartas foi utilizado e durante a Guerra Civil Americana as regras ficaram mais parecidas com as que são utilizadas hoje. Em torno de 1925 começou-se a jogar o poker com cartas comunitárias, que é a modalidade usada neste trabalho. O poker e seus jargões fazem parte da cultura americana. Em 1970 começaram os campeonatos mundiais de poker nos Estados Unidos. Em 1987, o poker com cartas comunitárias foram introduzidos em cassinos da Califórnia e se tornaram a modalidade de poker mais popular até hoje. Em 1998, um filme com o tema de poker chamado "Cartas na Mesa", em inglês "Rounders", foi lançado nos Estados Unidos.

Poker é um jogo de risco ou blefe. O espírito do jogo é conseguir convencer o adversário que o seu jogo é mais forte que o dele e tentar adivinhar se o jogo dele é mais forte que o seu. O convencimento é através de apostas. Se você não apostar que seu jogo é melhor que o do seu adversário, o seu adversário vence sem precisar mostrar o jogo. Também é conhecido como jogo de mentiroso, assim como o "truco" no Brasil.

As regras do jogo são bem simples e isso foi um dos motivos de ter se tornado tão popular (POKERLOCO, 2008). O jogador não possui informações suficientes para tomar a decisão ótima. Ou seja, é um jogo de informação incompleta e essa é uma característica de muitos problemas relacionados a área de Inteligência Artificial.

Um jogador com muita sorte consegue ganhar todos os jogos sem precisar tomar decisões inteligentes, mas esses casos são raros. Para a maioria das jogadores, Poker envolve alguma decisão

inteligente. Isso explica porque bons jogadores de poker ganham mais que jogadores medianos e porque existem jogadores de Poker profissionais. Algumas pessoas afirmam que Poker é um jogo puramente de sorte mas para outras, que escreveram muitos livros a respeito, como David Sklansky, não. Existem muitos livros e artigos publicados a respeito, dezenas de softwares sendo comercializados em que o único propósito é tomar decisões inteligentes em Poker e muitos sites de Poker Online proíbem o uso destes softwares por dar muita vantagem aos jogadores que os utilizam.

Muitos jogadores artificiais de poker, também conhecidos como pokerbots, já foram desenvolvidos. Existem até torneios (JOHANSON, 2007) de pokerbots, como o 2007 AAAI Computer Poker Competition No-Limit event ocorrido no ano de 2007.

Para cumprir o objetivo de formular algoritmos que tomem decisões inteligentes em poker foi implementado cinco algoritmos de diferentes jogadores. O primeiro foi o jogador aleatório, que toma decisão aleatória, sem levar em conta o estado do jogo. O segundo foi o jogador constante, que toma uma decisão constante, também sem se importar com o estado do jogo. O terceiro, nomeado MFS, foi utilizar uma fórmula que faz uma relação linear com algumas informações do jogo, que são: com a quantidade de dinheiro apostado, a quantidade de dinheiro que possa vir a receber caso vença, a probabilidade de vencer e a probabilidade de perder. O quarto, nomeado RH, foi também utilizar outra fórmula que é uma relação linear de outras informações do jogo, que são: quantidade de dinheiro que receberá caso vença, a quantidade de jogadores que ainda vão tomar alguma decisão, a quantidade de jogadores não fugiram, a quantidade de dinheiro que precisa pagar para continuar no jogo e a quantidade de vezes que alguém aumentou a aposta. O quinto, nomeado jogador de aprendizado por reforço, foi armazenar milhões de decisões e consequências em diferentes estados do jogo através de simulações e, quando for tomar uma decisão, consultar essa base de dados por todas as decisões feitas em um estado semelhante e tomar a decisão que obteve o melhor resultado, na média. Este último jogador precisava de muitas constantes para ajustar a "semelhança do estado do jogo" e para encontrar estas constantes foi utilizado Algoritmo Genético.

Para medir o desempenho dos jogadores foi feito uma comparação de quantidade de vitórias disputando as diferentes estratégias uma com a outra, além de jogadores humanos testarem os jogadores artificiais. As disputas foram feitas utilizando diferentes configurações dos jogadores artificiais.

A justificativa para esse trabalho não é apenas criar um bom jogador de poker. A motivação é testar e comparar diferentes algoritmos de para jogar poker e testar um novo algoritmo de aprendizado para solucinar um problema com informação incompleta em um domínio relativamente simples que é o poker. Uma boa solução pode servir de inspiração para resolver problemas mais genéricos relacionados a barganha, que é o espírito do poker, como a bolsa de valores.

2. Metodologia

Foi desenvolvido um programa de jogo de poker na modalidade no-limit que aceita jogadores genéricos, que pode ser um jogador humano, que pega as entradas do teclado, quatro jogadores estáticos – constante, aleatório, MFS e RH, e um jogador evolutivo – aprendizado por reforço. Todos esses jogadores jogam entre si milhares de vezes e de diversas formas possíveis: mesas entre 2 a 10 jogadores e com jogadores aleatoriamente escolhidos tanto no treinamento como na verificação de desempenho dos jogadores. Em todos os jogos, o valor de dinheiro inicial foi 1000 e o valor do small blind foi 10.

2.1. Treinamento

O treinamento dos jogadores deveria ser rápido para se tornar viável a tentativa de diversas formas de treinamento. A estratégia utilizada para o treinamento com este objetivo foi dividir em duas etapas onde a primeira etapa é feita apenas uma vez e a segunda etapa, que necessita de muitos ajustes, é feita muitas vezes.

A primeira etapa foi registrar um histórico de jogos, decisões e recompensas. Nessa base foi registrado todos as decisões de jogadores com diferentes métodos em vários jogos e a recompensa

daquela decisão, que só é preenchida no final do jogo e é replicada para todas as ações que levaram àquele recompensa. A informação contida nessa base é a seguinte:

Tabela 1. dimensões do estado do jogo

Nome	**Descrição**
POT	Soma da quantidade de dinheiro apostado por todos
ODDS	Relação do POT com o dinheiro apostado
RAISE	Quantidade de dinheiro que precisa usar para continuar no jogo
CHANCE	Probabilidade de vencer o jogo quando todos mostram as cartas
ROUND	Em que rodada - pre-flop, flop, turn ou river - o jogo se encontra
FOLLOWERS	Número de jogadores que ainda vão decidir na rodada atual
INGAME	Número de jogadores que não fugiram nem saíram
NPLAYERS	Número de jogadores
QTYPLAYERS	Quantidade de BET / RAISE feito por todos

As informações contidas no estado do jogo ainda não comentadas são o ODDS, o FOLLOWERS, o INGAME, o NPLAYERS e o QTYRAISE. ODDS é apenas a relação do dinheiro do POT em relação ao dinheiro apostado, por exemplo, um ODDS igual a 5 significa que tem no POT 5 vezes o valor apostado pelo jogador. O FOLLOWERS indica quantos jogadores ainda vão decidir algo. Isso porque o último a decidir tem vantagem em relação ao primeiro a decidir porque o último já sabe quem aumentou e quem não. INGAME indica quantos jogadores ainda não fugiram. NPLAYERS indica quantos jogadores estão jogando, independente de ter fugido, falido ou ainda estar em jogo. O QTYRAISE indica a quantidade de vezes que algum jogador aumentou a aposta e é importante porque jogos que houveram muito aumento de aposta pode indicar que muitos jogadores estão com boas chances de vencer.

Depois de preenchida essa base de dados com milhares de decisões, foi iniciado a segunda etapa, que consiste em passar para todos os jogadores informações de treinamento que são essas informações do histórico de jogos. Os jogadores estáticos ignoram essa informação assim como os humanos. Um exemplo de uma linha de informação dessa base de dados: primeira rodada, chance de vencer de 30%, POT de 50, ODDS de 3, é necessário pagar 20 para continuar no jogo, foi tomada a decisão de RAISE 40, recompensa de 50.

2.2. Implementação dos Jogadores Artificiais

Para tomar uma decisão, cada jogador tem como entrada o estado do jogo, que é multidimensional e suas dimensões são as mesmas informações guardadas na base de dados, que são: POT, ODDS, RAISE, CHANCE, ROUND, FOLLOWERS, INGAME, NPLAYERS, QTYRAISE.

Para simplificar a implementação e o treinamento dos jogadores, foi estabelecido que cada jogador tem como saída da ação 'tomar uma decisão' um valor discreto entre 1 e 6, que é a decisão discretizada:

Tabela 2. decisão discretizada

Código	**Descrição**
1	Fugir

2	Continuar no jogo
3	Aumentar pouco: equivalente a 2 a 4 small blind
4	Aumentar médio: equivalente a 6 a 18 small blind
5	Aumentar muito: equivalente a 20 a 40 small blind
6	Aumentar tudo: all-in

2.2.1. Jogadores Aleatórios

Esses jogadores tomam a decisão discretizada aleatória na seguinte proporção: 11% para cada decisão sendo que a decisão de continuar o jogo tem a proporção de 44%.

2.2.2. Jogadores Constantes

Esses jogadores tomam sempre a mesma decisão discretizada.

2.2.3. Jogadores MFS – Mathematically Fair Strategy

Esses jogadores (FINDLER, 1977) tomam a decisão discretizada definida por na fórmula:

$$se \;\; O = 0 \;\rightarrow\; D = 2$$

$$E = \frac{P \cdot (O-1)}{O}$$

$$M = P - E + R$$

$$V = \frac{C \cdot E - (100-C) \cdot M}{100}$$

$$\begin{aligned}
se \;\; O \neq 0 \;\; e \;\; V < -T \;\rightarrow\; D &= 1 \\
se \;\; O \neq 0 \;\; e \;\; -T \leqslant V < T \;\rightarrow\; D &= 2 \\
se \;\; O \neq 0 \;\; e \;\; T \leqslant V < 5 \cdot T \;\rightarrow\; D &= 3 \\
se \;\; O \neq 0 \;\; e \;\; 5 \cdot T \leqslant V < 10 \cdot T \;\rightarrow\; D &= 4 \\
se \;\; O \neq 0 \;\; e \;\; 10 \cdot T \leqslant V < 15 \cdot T \;\rightarrow\; D &= 5 \\
se \;\; O \neq 0 \;\; e \;\; V \geqslant 15 \cdot T \;\rightarrow\; D &= 6
\end{aligned}$$

Onde V é a relação MFS e T é uma constante. M é a quantidade de dinheiro do adversário, P é o POT, O é o ODDS, R é o RAISE, E é a quantidade de dinheiro apostado pelo jogador e C é a CHANCE em porcentagem e D é a decisão discretizada. Caso o O, que é o ODDS, seja igual a zero, significa começo de jogo onde o jogador não apostou nada ainda. Nesse caso, a decisão dele é sempre entrar no jogo.

2.2.4 Jogadores RH – Jean Rachlin e Gary Higgins

Esses jogadores (FINDLER, 1977) tomam a decisão discretizada definida por :

$$V = \frac{P}{(C+1)\cdot F\cdot L\cdot(R+1)}$$

$$\begin{aligned} se\ \ V < T &\ \rightarrow\ D = 1 \\ se\ \ T \leqslant V < 20\cdot T &\ \rightarrow\ D = 2 \\ se\ \ 20\cdot T \leqslant V < 50\cdot T &\ \rightarrow\ D = 3 \\ se\ \ 50\cdot T \leqslant V < 100\cdot T &\ \rightarrow\ D = 4 \\ se\ \ 100\cdot T \leqslant V < 1000\cdot T &\ \rightarrow\ D = 5 \\ se\ \ V \geqslant 1000\cdot T &\ \rightarrow\ D = 6 \end{aligned}$$

Onde V é a relação RH, P é o POT, R é o número de vezes que alguém decidiu RAISE / BET, F é o número de jogadores que ainda vão jogar nesta rodada, L é o número de jogadores que não fugiram, R é o RAISE, T é uma constante e D é a decisão discretizada.

2.2.5 Jogadores com Aprendizado por Reforço

Jogadores com aprendizado por reforço (SUTTON; BARTO, 2008, WIKIPEDIA REINFORCEMENT LEARNING, 2008) tem uma matriz com o número de dimensões iguais ao número de dimensões do estado do jogo. Cada dimensão tem uma quantidade de níveis e um intervalo correspondente a cada nível e o estado do jogo se enquadrará em um nível se o intervalo do estado corresponder àquele nível. *Por exemplo, se a dimensão POT da matriz tem 7 níveis e cada nível tem o intervalo de 13,3, um estado do jogo em que o POT esteja em 29 se enquadrará no 3º nível desta dimensão da matriz porque o 1º nível vai de 0 a 13,3 e o 2º nível vai de 13,3 a 26,6.* Caso o valor da dimensão do estado do jogo seja maior que o limite da matriz, esse valor é colocado no último nível. Além destas dimensões, a matriz tem mais duas dimensões, uma para indicar a decisão tomada e outra para indicar se o resultado foi prejuízo ou lucro. O conteúdo de cada célula da matriz é os resultados referentes àquele estado do jogo.

No treinamento, a decisão e a consequência, que é o dinheiro ganho ou perdido, é salvo na posição referente ao estado do jogo.

Na hora de tomar uma decisão, a matriz na posição referente ao estado do jogo é consultada e obtém uma lista de pares *decisão discretizada* e *recompensa*. Com essa lista, toma-se uma decisão que obtiver a maior média de recompensas por jogo.

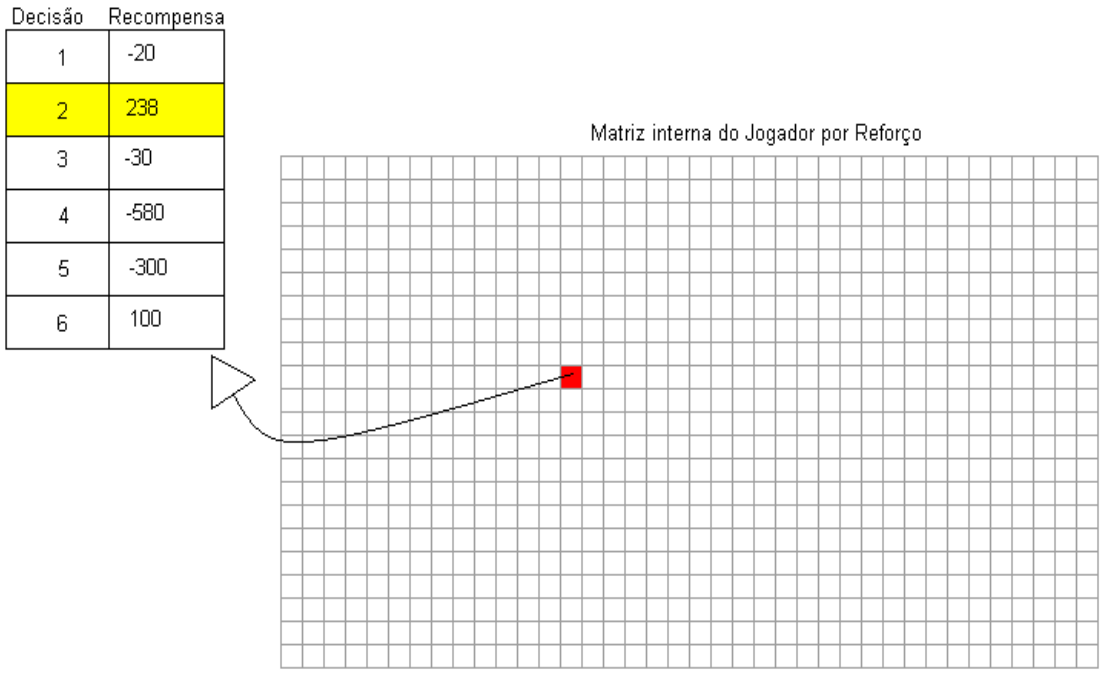

Figura 1. Exemplo de uma situação de jogo que o jogador por reforço consulta sua base de

conhecimento e decide 2, que significa “continuar no jogo”.

O jogador de aprendizado por reforço contem várias constantes que precisam de ajustes. Elas representam o intervalo que cada nível da matriz interna desse jogador que contem as nove dimensões - CHANCE, POT, RAISE, ODD, QTYRAISE, NPLAYERS, FOLLOWERS, ROUND e INGAME, com exceção da dimensão ROUND, que já foi pré-definida como tamanho 2, sendo o primeiro referente a primeira rodada e o segundo referente as demais rodadas. Cada dimensão deveria ter duas constantes: uma para o intervalo e outra para a quantidade de níveis mas, para simplificar, existe apenas duas constantes “quantidade de níveis”, uma para ODDS, POT, RAISE e CHANCE, chamado de “BIG QTY” e outra para QTYRAISE, NPLAYERS, FOLLOWERS e INGAME, chamada de “LITTLE QTY”, que totaliza dez constantes. Se cada variável fossem testados apenas 5 valores diferentes, a quantidade de testes seria $5^{10}=9.765.625$, o que impossibilita testar todos os casos. Para encontrar uma boa configuração destas constantes foi utilizado Algoritmo Genético (HOLLAND, 1972, RUSSEL; NORVIG, 1995)..

A função fitness do algoritmo genético é a soma da quantidade vitórias diversas disputas, cada disputa com dez mesas, com o Jogador por Reforço usando 10 milhões de dados preenchidos na sua matriz interna. Foram usadas sete disputas, uma contra o jogador MFS com constante T = 1, uma contra o MFS com T=5, uma contra o Jogador Aleatório, uma contra o Jogador Constante que decide “continuar”, uma contra o Jogador Constante que decide “aumentar muito”, uma contra o Jogador RH com constante T=0,1 e uma contra o Jogador RH com T=0,001. O resultado da função fitness é a soma ponderada de vitórias das disputas com peso 2 contra o Jogador MFS, peso 1 contra o Jogador Constante, peso 1 contra o Jogador RH e peso 4 contra o Jogador Aleatório. A população tem tamanho 10 e a reprodução é feita na seguinte proporção: os 2 melhores sobrevivem para a próxima geração, 1 é mutação e 7 são produtos de crossover. Cada gene é uma constante e a população inicial é aleatória. A função de seleção escolhida foi a stochastic uniform (MATLAB, 2008). Essa função gera uma linha onde cada indivíduo tem um pedaço da linha proporcional ao fitness. Um ponto aleatório da linha é selecionado e o indivíduo que estiver naquela faixa da linha é o primeiro pai escolhido. Os outros pais são selecionados pela distribuição uniforme de pontos pela linha, igualmente espaçados.

	Pot	Chance	Raise	Odds	QtyRaise	Followers	Ingame	Nplayers	LQ	BQ
Pai 1	14	5	7	9	1	4	3	8	3	7
Pai 2	27	20	3	2	4	1	2	3	5	10
Filho	27	20	7	9	1	1	3	3	3	7

Figura 2. Exemplo de uma situação de jogo que o jogador por reforço consulta sua base de conhecimento e decide 2, que significa “continuar no jogo”.

de jogadores que não fugiram, *R* é o RAISE, *T* é uma constante e *D* é a decisão discretizada.

3. Resultados

Não houve um algoritmo predominantemente vencedor. Houve apenas um que venceu, na média, mais que outro em uma disputa de todos contra todos. Mas em disputas individuais, que é um método contra outro, houve o caso do Reforço vencer o MFS que venceu o RH que venceu o Reforço, o que não deixou de continuar sendo o reforço o jogador favorito por ter vencido a maioria das disputas individuais. O jogador MFS obteve resultados muito bons em disputas individuais e os outros jogadores tiveram os piores resultados. Durante os experimentos o jogador MFS obteve resultados muito bons e demorou até que um jogador evolutivo conseguisse vencê-lo.

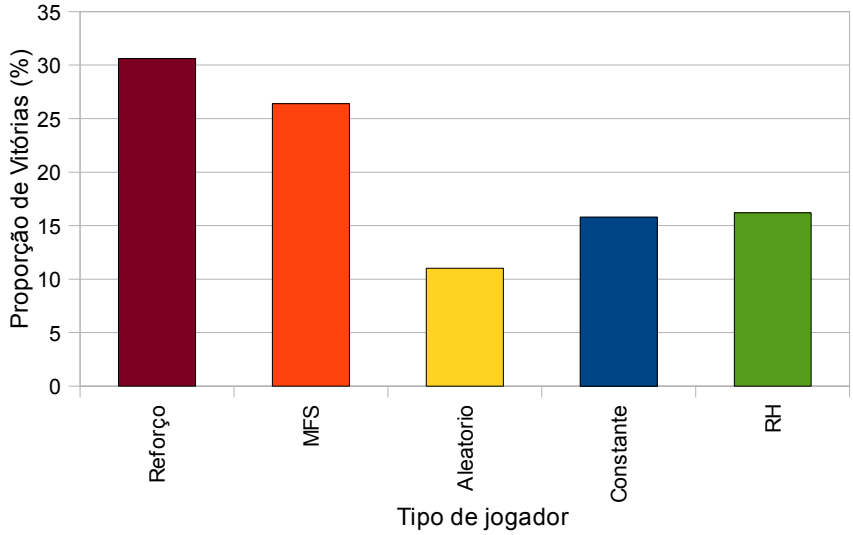

Figura 3. Resultado da disputa entre todos os jogadores

4. Conclusão

Os resultados foram abaixo da expectativa, que era do jogador por reforço ganhar muito mais que perder dos outros tipos de jogadores, que são decisões de fórmulas simples. Todos os resultados foram obtidos de simulações artificiais e uma base de dados com jogadas humanas pode aumentar a chance de vitória dos algoritmos evolutivos. Um problema em melhorar os algoritmos é o tempo de processamento necessário pois milhares de jogos precisam ser feitos para evoluir um jogador e pode demorar muito até verificar se um método é melhor que outro. Outro problema é que métodos artificiais que vencem métodos artificiais não necessariamente jogam bem contra jogadores humanos e uma bateria de teste com jogadores humanos é ainda mais demorado. Mesmo assim o resultado final foi um jogador que jogou razoavelmente bem.

Referências

FINDLER, Nicholas V.. **Studies in Machine Cognition Using the Game of Poker.** State University Of New York At Buffalo: R. J. Hanson, 1977.

HOLLAND, John H.. Genetic Algorithms and the Optimal Allocation of Trials. **Siam Journal On Computing**, Philadelphia, n. 2 , p.88-105, 3 ago. 1972.

JOHANSON, Michael Bradley. **Robust Strategies and Counter-Strategies:** Building a Champion Level Computer Poker Player. 2007. 97 f. Tese (Mestrado) - University Of Alberta, Alberta, 2007.

MATLAB (Org.). **The MathWorks:** MATLAB and Simulink for Technical Computing. Disponível em: <http://www.mathworks.com/>. Acesso em 1 out. 2008.

POKERLOCO (Org.). **Como Jogar Poker:** Resumo. Disponível em: <http://www.pokerloco.com/pt/howtoplaypoker/index.htm>. Acesso em: 1 out. 2008.

RUSSEL, Stuart J.; NORVIG, Peter. **Artificial Intelligence:** A Modern Approach. New Jersey: Prentice Hall, 1995.

SUTTON, Richard S.; BARTO, Andrew G.. **Reinforcement Learning:** An Introdution. E-Book. Disponível em: <http://www.cs.ualberta.ca/~sutton/book/ebook/the-book.html>. Acesso em: 1 out. 2008.

WIKIPEDIA POKER (Org.). **Poker.** From Wikipedia, the free encyclopedia. Disponível em: <http://en.wikipedia.org/wiki/Poker>. Acesso em: 1 out. 2008.

WIKIPEDIA REINFORCEMENT LEARNING (Org.). **Reinforcement Learning.** From Wikipedia, the free encyclopedia. Disponível em: <http://en.wikipedia.org/wiki/Reinforcement_learning>. Acesso em: 1 out. 2008.