Universidade Federal do ABC
Centro de Matemática, Computação e Cognição (CMCC)
Pós-Graduação em Ciência da Computação

Jonathan Ohara de Araujo

AGENTES INTELIGENTES PARA BATALHAS POKÉMON

Dissertação

Santo André - SP
2016

Jonathan Ohara de Araujo

AGENTES INTELIGENTES PARA BATALHAS POKÉMON

Qualificação

Qualificação apresentada ao Curso de Pós-Graduação da Universidade Federal do ABC como requisito parcial para obtenção do grau de Mestre em Ciência da Computação

Orientador: Prof. Dr. Fabrício Olivetti de França

Santo André - SP

2016

Ficha Catalográfica

de Araujo, Jonathan Ohara.
Agentes inteligentes para batalhas Pokémon / Jonathan Ohara de Araujo.
Santo André, SP: UFABC, 2016.
3p.

Jonathan Ohara de Araujo

AGENTES INTELIGENTES PARA BATALHAS POKÉMON

Essa Qualificação foi julgada e aprovada para a obtenção do grau de Mestre em Ciência da Computação no curso de Pós-Graduação em Ciência da Computação da Universidade Federal do ABC.

Santo André - SP - 2016

Prof. Dr. João Paulo Gois

Coordenador do Curso

BANCA EXAMINADORA

Prof. Dr. Fabrício Olivetti de França	Prof.Dr. André Luiz Brandão
Prof.Dr. João Paulo Gois	Prof.Dr. Denise Hideko Goya (Suplente)

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Árvore de jogos

1.1 Introdução

Nesse capítulo vamos discutir sobre tomada de decisão em jogos e árvore de decisões. Em árvores de decisões o foco será em 2 diferentes métodos de buscas: técnicas baseadas em *minimax* e técnicas baseadas em Monte Carlo. O capítulo a seguir está organizado do seguinte modo: Primeiro, a seção 1.2 irá explicar o básico da ideia de tomada de decisão e os elementos envolvidos nela. Na seção 1.2.1 é demonstrado como decisões podem ser modeladas em computação. Em 1.3 será discutido o conceito de *minimax*, algumas técnicas e melhorias. A seção 1.4 irá expor sobre o método de Monte Carlo e sua aplicação em árvores de busca.

1.2 Tomada de decisão

É muito comum associar inteligência artificial para jogos, mais especificamente inteligência artificial para jogos, com a habilidade em que os elementos do jogo (jogadores não humanos aliados ou inimigos, objetos e fases, por exemplo) têm para tomar decisões dados certas situações. Apesar disso, a tomada de decisão não é o único componente de uma inteligência artificial, podem-se enumerar outros elementos como a capacidade de avaliar as ações disponíveis e a estratégia que o personagem segue.

Tomada de decisão é o processo de escolher uma ação entre diversas possibilidades. Segundo [Harris, 1998] "Tomada de decisão é o estudo do processo de identificar e escolher alternativas baseadas nas preferências do tomador de decisões. Tomar uma decisão implica que existem diferentes escolhas para serem consideradas, e, em tal caso, não queremos apenas identificar quais dessas alternativas são viáveis, mas escolher a decisão que melhor se encaixa com as nossas metas, objetivos, desejos, valores, e assim por diante".

Ainda na teoria do processo de tomada de decisão, [Baker et al., 2002] diz que o primeiro passo na tomada de decisão é estabelecer quem é(são) o(s) tomador(es) de decisão e o(s) *stakeholder(s)* (partes afetadas ou interessadas), de modo a mitigar um possível desacordo sobre a definição do problema, metas e critérios. O processo pode ser definido nos seguintes passos:

- Definir o problema;
- Determinar os requisitos que a solução deve apresentar;
- Estabelecer objetivos que a solução do problema deve cumprir;
- Identificar alternativas que irão solucionar o problema;
- Desenvolver critérios de avaliação com base nos objetivos;
- Selecionar uma ferramenta ou método para a decisão;
- Aplicar a ferramenta ou método para selecionar a alternativa preferida;
- Validar se a solução resolveu o problema.

Em jogos (especificamente jogos digitais), segundo [Millington and Funge, 2009] a entrada da tomada de decisão é o conhecimento que tal personagem tem, e a saída é a ação a ser realizada. O conhecimento pode ser dividido em interno e externo. Conhecimento externo é a informação que o personagem tem sobre o ambiente em sua volta: posição dos outros personagens, o leiaute da fase, se um dispositivo foi ligado, a direção que um barulho veio, e assim por diante. Conhecimento interno é a informação sobre o estado interno do personagem ou pensamentos internos como: sua saúde, objetivos, seu passado, e assim por diante. Ainda segundo [Millington and Funge, 2009] a representação do conhecimento está intrinsecamente ligada com a maioria dos algoritmos de tomada de decisão.

Algumas vantagens dessa técnica são a fácil construção e a modularização

1.2.1 Árvore de decisão

Árvores de decisão é uma maneira muito clara e natural de representar uma cadeia de decisões e suas consequências. Segundo [Millington and Funge, 2009] árvores de decisão são rápidas, fáceis de implementar e simples de entender. Elas são a técnica mais simples de tomada de decisão. Outra vantagem é a questão de ser modular e muito fácil de ser construída.

Um modo muito comum de representar árvores de decisão para estratégias simples é o fluxograma. Segundo [Millington and Funge, 2009] as árvores de decisões representadas por fluxogramas têm normalmente duas respostas (sim e não, por exemplo). Na imagem

1.1 é mostrado um diagrama simples de estratégia para um inimigo, nesse fluxograma o personagem fica patrulhando até achar um inimigo, ao ver um inimigo o personagem se aproxima se necessário e o ataca.

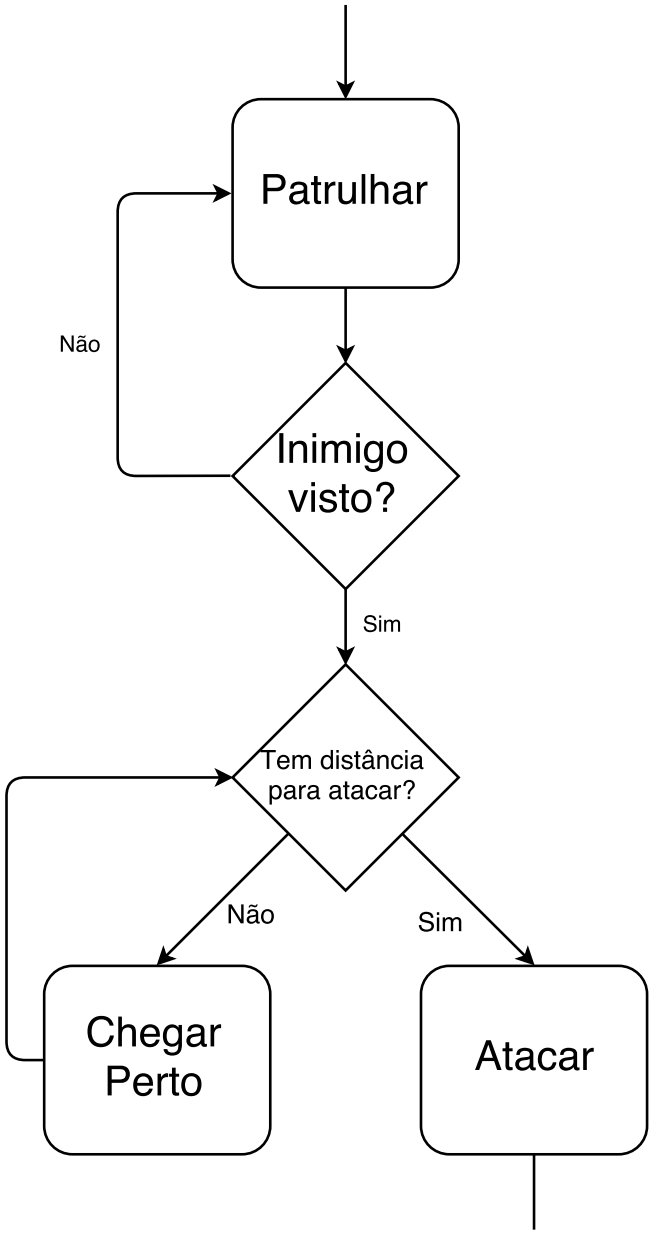

Figura 1.1: Fluxograma Simples de estratégia para inimigos.

1.2.2 Máquina de estados finitos

Outro modo de representar comportamentos de certos elementos é através de máquinas de estados finitos (do inglês Finite State Machine - FSM). [Wright, 2012] define o modelo conceitual de FSM da seguinte forma:

- O sistema (objeto que será modelado como uma FSM) deve ser descrito como uma máquina finita de estados, ou seja, o sistema pode só transitar entre diferentes estados em um escopo fechado.
- O sistema precisa ter um número finito de entradas e/ou eventos que podem disparar transações entre estados.
- O comportamento do sistema dado um ponto no tempo depende do estado atual, da entrada ou evento que ocorrerá naquele tempo.
- Para cada possível estado do sistema, o comportamento é definido para cada entrada ou evento possível.
- O sistema tem um estado inicial particular.

O modo mais comum de representar o processo de tomada de decisão em personagens nos jogos digitais é através de máquina de estados finitos ([Diller et al., 2004]). Ainda segundo os autores, desenvolvedores de jogos comerciais estão muito mais interessados em uma "ilusão de inteligência". Muitas vezes os personagens irão agir de limitados modos diferentes, por esse motivo máquinas de estados finitos funcionam muito bem, como acontece no jogo Halo da Bungie Software, onde os guerreiros da raça Convenant (inimigos) ficam no estado patrulhar até perceber (por som ou visão) um jogador, em seguida, ele irá alternar para o modo ataque ([Millington and Funge, 2009]).

Essa técnica é muito empregada em jogos comerciais. Segundo [Diller et al., 2004] desenvolvedores de jogos comerciais pensam em uma inteligência artificial que entretenha o público. Em cenários que o objetivo é fazer a melhor inteligência possível, ou o objetivo é vencer os melhores humanos em um determinado desafio, tais técnicas não funcionam bem. Em cenários competitivos é possível enumerar alguns problemas dessas técnicas como: agir somente pela reatividade e a incapacidade de prever futuros movimentos ou analisar uma cadeia de movimentos em conjunto.

Ilustrar da mesma forma que o anterior

1.2~~3~~4 Árvore de jogos

conceitos (remova "coisa" do seu vocabulário)

Apesar da semelhança com árvore de decisão muitos autores como [Nijssen, 2013] e [Du and Pardalos, 1995] as tratam como ~~coisas~~ diferentes, mas, com recursos equivalentes. Uma árvore de jogo é uma representação de um estado em um jogo sequencial. Os nós representam ~~as posições do jogo~~ e as arestas representam os possíveis movimentos de um determinado estado. O nó raiz representa a ~~posição~~ inicial e os nós folhas (ou terminais) representam o fim de jogo ([Nijssen, 2013]).

↳ estado

De acordo com [aend Edward Powley et al., 2012] os jogos podem ser classificados de acordo com certas propriedades:

um estado do jogo

- **Soma-zero**: A soma das recompensas (positivas ou negativas) de todos os jogadores resulta em zero, ou seja, o que um jogador ganhou é exatamente o que o outro jogador perdeu.

- **Informação**: Completa ou incompleta, perfeita ou imperfeita. Jogos com informação completa são aqueles que o estado atual do jogo é totalmente observável por todos os jogadores. Informação perfeita se refere se o jogador tem a total informação de tudo que já ocorreu em eventos anteriores.

- **Determinista**: Se existe um fator probabilístico em certas ações (também conhecida como completude, por exemplo: incerteza sobre certas recompensas).

- **Sequencial**: Se as ações escolhidas são aplicadas sequencialmente ou simultaneamente.

- **Dicreto**: discreto ou contínuo.

Ainda segundo [aend Edward Powley et al., 2012] jogos com dois jogadores que são: soma-zero, de informação perfeita, determinísticos, discretos ou sequenciais, podem ser descritos como jogos combinatórios. Essa definição inclui jogos como: Go, Xadrez e Jogo da Velha, assim como muitos outros.

mostrar um exemplo de árvore de jogos e explicar sobre estado, recompensas, ações, ...

1.3 Minimax

O algoritmo *minimax* é comumente utilizado em jogos de dois jogadores e baseados em turnos. Jogos do mundo real normalmente envolvem uma estrutura de recompensas em que, apenas as recompensas obtidas em estados terminais (jogadas que definem quem é o vencedor) descrevem com precisão o quão bem cada jogador está se saindo ([aend Edward Powley et al., 2012]).

1.3.1 Conceito e algoritmo

O algoritmo de *minimax* tenta minimizar a possível perda máxima, para isso ele analisa todos os possíveis movimentos futuros de cada jogador. Segundo os autores [Campbell and Marsland, 1983] o algoritmo de *minimax* do seguinte modo: o algoritmo assume que existem dois jogadores chamados Max e Min, e atribui um valor para cada nó dentro da árvore do jogo (e um particular para o nó raiz) do seguinte modo: nós folhas ou terminais propagam o valor *minimax* recursivamente para todos os outros nós (passando para seu ancestral direto). Se um nó não terminal p do jogador Max for escolhido, então

o valor de p é o máximo valor dos filhos de p. Similarmente, se for o turno do jogador Min será escolhido o menor valor dos sucessores de p.

O algoritmo básico do *minimax* pode ser representado por:

Algorithm 1 Algoritmo Minimax

```
1:  procedure MINIMAX(node, depth, max)
2:      if depth = 0 OR node is terminal then
3:          return heuristic
4:      end if
5:      if max then
6:          bestValue ← −∞
7:          for each child ∈ node do
8:              value ← MINIMAX(child, depth − 1, FALSE)
9:              bestValue ← MAX(bestValue, value)
10:         end for
11:     else
12:         bestValue ← +∞
13:         for each child ∈ node do
14:             value ← MINIMAX(child, depth − 1, TRUE)
15:             bestValue ← MIN(bestValue, value)
16:         end for
17:     end if
18:     return bestValue
19: end procedure
```

Onde:

- **node** Nó atual da árvore. Na primeira chamada é passado o nó raiz da árvore, ou seja, o nó que representa o estado atual do jogo.
- **depth** Profundidade do nó (ou altura da árvore). Inicialmente é passada a altura total da árvore.
- **max** Valor sim ou não (TRUE ou FALSE) representando se a jogada analisada é do jogador **Max** ou **Min**.

Em seu trabalho, [Du and Pardalos, 1995] propuseram uma relação entre árvores *minimax* e árvore de jogos (especificamente jogos de tabuleiros). Essa relação pode ser vista na tabela 1.1.

Tabela 1.1: Árvore *minimax* e árvore de jogos

Árvore *minimax*	Árvore de jogos
Árvore	Todas possíveis configurações do tabuleiro
Nó da árvore	Configuração do tabuleiro
Arestas de Max até um nó Min	Movimento do jogador (movimento do jogador Max)
Arestas de Min até um nó Max	Movimento adversário (movimento do jogador Min)
Valor do nó	A ”nota”para certa configuração de tabuleiro
Nó folha (terminal)	Resultado do jogo (vitória, derrota ou empate)
Caminho solução	Sequência de Movimentos que leva a melhor saída

Na figura 1.2 é ilustrada a aplicação do algoritmo *minimax* para um cenário do Jogo da Velha. Na imagem o jogador Max é representado por **X** e o jogador Min por **O**. Como dito anteriormente, primeiramente a árvore é explorada (partindo da raiz) até que se chegue a nós terminais (indicando fim de jogo), em situações vitoriosas foi atribuído o valor de 10, em empates 0 e em derrota o valor -10. Em seguida, os pais desses nós folhas são preenchidos com o mínimo ou o máximo valor dos nós filhos. No primeiro nó de Min (primeiro nó da segunda fileira), por exemplo, existem dois nós filhos, um com valor 10 e outro com valor -10, por ser um nó Min é atribuído para esse nó o menor valor (no caso -10). A linha em azul mostra a melhor jogada no momento para o nó raiz.

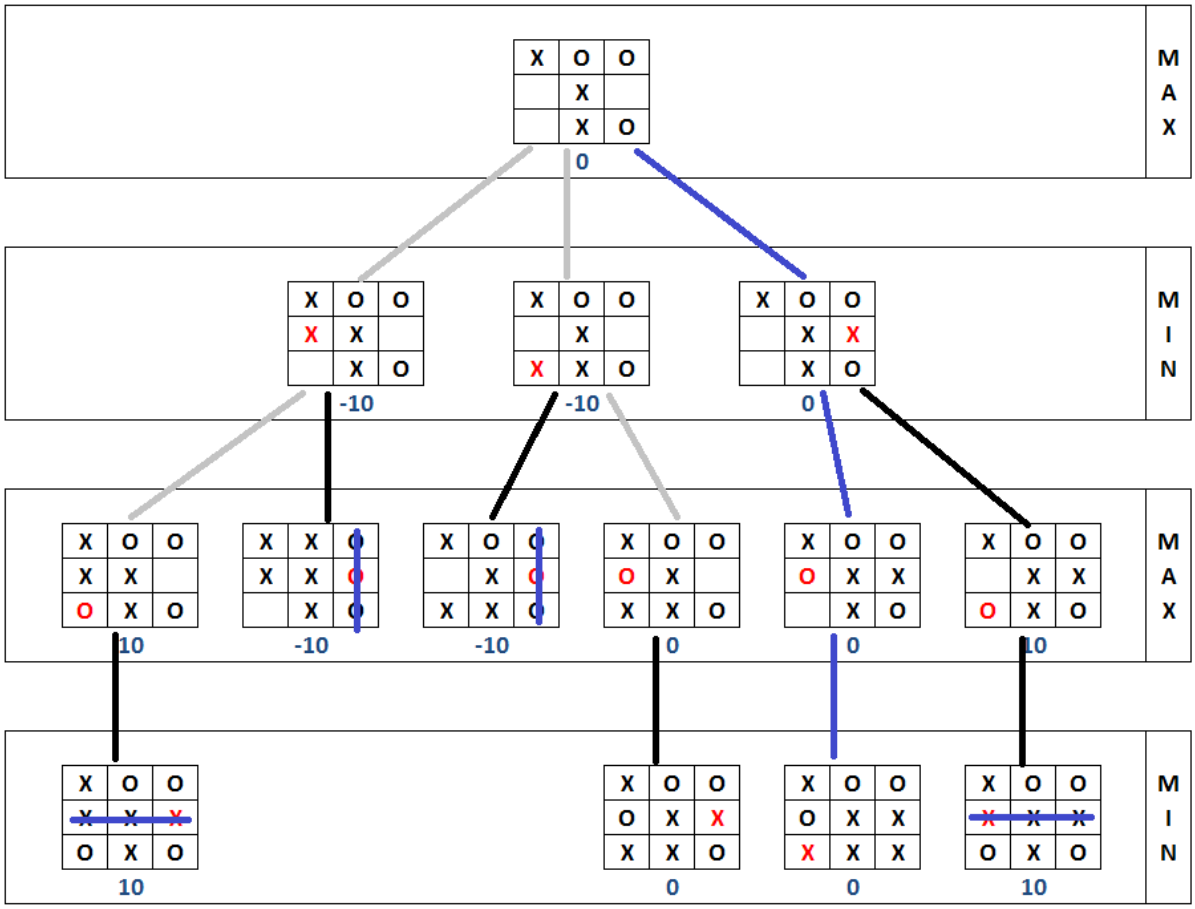

Figura 1.2: *Minimax* aplicado ao Jogo da Velha.

Segundo [aend Edward Powley et al., 2012] existem duas variações do *minimax* muito comuns de serem encontradas:

- **Expectimax** generaliza *minimax* para jogos estocásticos em que as transições de estado para estado são probabilísticas. O valor de um nó é a soma dos valores dos nós filhos ponderados por suas probabilidades (possibilidade de um estado ocorrer).
- **Miximax** é semelhante ao *expectimax* de apenas um jogador, é usado principalmente em jogos com informações não precisas. Ele utiliza uma estratégia predefinida para tratar a decisão do oponente como nós de probabilidade.

1.3.2 Problemas

Em jogos com grande quantidade de escolhas e/ou com um grande número de turnos (altura de árvore) esses algoritmos não se mostram muito efetivos, pois cada escolha tem que ser avaliada até que chegue a nós terminais. Para a grande maioria dos jogos não é

viável pesquisar toda a árvore até o nó terminal como especificado pelas regras do jogo ([Campbell and Marsland, 1983]).

Podemos listar dois grandes problemas ao utilizar o *minimax* com uma árvore muito extensa. O primeiro deles é fazer toda a criação da árvore, ou seja, a simulação de possíveis jogadas até que se chegue a nós folhas (considerando que os caminhos sejam finitos). O segundo problema ocorre quando uma árvore tem um grande número de nós e uma altura muito grande, nesse caso a função de avaliação do *minimax* tem que percorrer muitos nós até chegar à raiz a partir dos nós folhas para atualizar seus valores.

1.3.3 Variações

Uma estratégia para reduzir a quantidade de nós analisados é utilizar técnicas de poda de árvores. A poda de árvore mais conhecida na literatura é a chamada *alpha-beta*. Segundo [Knuth and Moore, 1976] essa técnica é usada geralmente para aumentar a velocidade de busca sem perder informação. Nessa técnica são ignorados os nós e suas subárvores de jogadas incapazes de apresentar melhores valores do que movimentos já conhecidos. Durante a análise das jogadas são definidas duas variáveis *alpha* e *beta*. *Alpha* representa o valor máximo que o jogador Max pode fazer e *beta* a pontuação mínima do jogador Min. A cada avaliação de nó esses limites são verificados, caso o valor da avaliação não estiver entre esses limites o nó e todas suas subárvores são removidas (podadas). Esse procedimento pode ser dado pelo seguinte algoritmo (na primeira chamada alpha vale $-\infty$ e beta $+\infty$):

Algorithm 2 Algoritmo alphabeta

```
1: procedure ALPHABETA(node, depth, currentPlayer, alpha, beta)
2:     if depth = 0 OR node is terminal then
3:         return heuristic
4:     end if
5:     if max then
6:         bestValue ← −∞
7:         for each child ∈ node do
8:             bestValue ←MAX(bestValue, ALPHABETA(child, depth -1, alpha, beta,
   FALSE))
9:             alpha ← MAX(alpha, value)
10:            if beta ⩽ alpha then
11:                break
12:            end if
13:        end for
14:    else
15:        bestValue ← +∞
16:        for each child ∈ node do
17:            bestValue ←MIN(bestValue, ALPHABETA(child, depth -1, alpha, beta,
   TRUE))
18:            beta ← MIN(beta, bestValue)
19:            if beta ⩽ alpha then
20:                break
21:            end if
22:        end for
23:    end if
24:    return bestValue
25: end procedure
```

explicar passo a passo textualmente

Para resolver o problema de simular um grande número de jogadas e, ao mesmo tempo diminuir a quantidade de nós a serem analisados é possível criar avaliações de heurísticas para nós não terminais e assim delimitar a simulação da árvore até uma determinada altura. Em outras palavras, podemos definir um número máximo de altura que o algoritmo pode alcançar e, ao chegar nesse número, é retornado um valor de acordo com uma heurística.

aqui acho melhor falar ao contrário, que alguns jogos necessitam de avaliação heurística

1.3.4 Avaliação de heurística

Em jogos com pequena quantidade de nós na árvore, a avaliação de heurística não é tão importante, um exemplo desse tipo de jogo é o jogo da velha, devido ao reduzido número de nós e a pequena altura da árvore, é mais seguro explorar a árvore até o final do que criar avaliações de heurísticas para nós não terminais.

Segundo [Nielsen, 1994] avaliação heurística envolve ter um pequeno conjunto de avaliadores examinando a interface e avaliando a sua conformidade em relação ao resultado. Os autores [Christensen and Korf, 1986] discorrem que, em jogos de tabuleiro de duas pessoas a função de heurística é vagamente caracterizada pela ”força”do posicionamento de um jogador contra o outro. Pode-se dizer que uma função de avaliação de heurística tem duas propriedades ([Christensen and Korf, 1986]):

- Quando aplicado a um estado final (no caso de uma árvore, um nó folha), a avaliação de heurística tem que devolver o estado corretamente.
- O valor da função de heurística é invariável ao longo de um caminho de solução ótima.

explicar em mais detalhes, invariante a que?

1.3.5 Sistema Deep Blue

Um dos sistemas mais famosos que utilizou as técnicas citadas foi o Deep Blue da IBM. O Deep Blue é um sistema feito para jogar xadrez com técnicas baseadas em *minimax*. Além de utilizar o buscador alpha-beta, o sistema conta com uma complexa função de avaliação de nós não terminais, o sistema analisava nós com até 12 níveis de profundidade [Hsu, 1999]. O Deep Blue era composto por um *chip* chamado *Chess Chip* que é dividido em três partes [Hsu et al., 1995]:

- **Buscador alpha-beta**. Menor parte do *chip*, contendo apenas 5% de seu total, esse componente é responsável por fazer a busca na árvore de jogadas. O algoritmo utilizado é uma variante do *alpha-beta* chamado de *minimum-window alpha beta search*, essa técnica foi escolhida por não ter necessidade de ter uma pilha de valores.
- **Gerador de movimentos**. Esse componente contém o *array* bidimensional 8x8 referente a cada posição do tabuleiro do xadrez, ele também é responsável por realizar e verificar a legalidade dos movimentos de acordo com as regras do xadrez.
- **Função de avaliação**. Ocupando cerca de dois terços do *chip* esse componente é responsável pela avaliação (heurística) dos movimentos. Cada possível posição de cada peça é avaliado, essa avaliação é baseada em quatro critérios: material, posição,

segurança do rei e tempo [Works, 2016]. Material é o quanto cada peça "vale". Por exemplo, um peão vale 1 enquanto uma torre vale 5. Posição quantifica o quão bom estão posicionadas as peças, nesse cálculo uma série de aspectos é levada em conta, um deles é o número de movimentos seguros que as peças podem fazer para atacar. Segurança do rei é um aspecto defensivo da posição. Ela é determinada através da atribuição de um valor para a segurança da posição do rei, a fim de saber como fazer um movimento puramente defensivo ([Works, 2016]). Por final tempo, além de controlar o tempo da própria jogada, fazer uma jogada rápida dá ao adversário menos tempo para pensar em possíveis jogadas.

1.4 Árvore de Busca de Monte Carlo

Nessa seção vamos discutir sobre o método de Monte Carlo e sua aplicação, e sobre árvore de busca de Monte Carlo (do inglês Monte Carlo Tree Search - MCTS) uma heurística de buscas utilizada para estimar árvores de decisão, principalmente em jogos baseados em turnos e por fim veremos alguns aprimoramentos.

1.4.1 Método de Monte Carlo

O método de Monte Carlo, do qual o algoritmo MCTS foi derivado, compreende o ramo da matemática experimental que concerne com experimentos utilizando números aleatórios ([Hammersley, 2013]). Segundo [Kleij, 2010] esse método matemático se refere a utilizar amostras aleatórias com o objetivo de estimar soluções de problemas infactíveis de serem resolvidos analiticamente.

A origem métodos de Monte Carlo como ferramenta de pesquisa resulta de trabalhos em bombas atômicas durante a segunda guerra, esse trabalho envolveu uma simulação direta de problemas probabilísticos relacionados com a difusão aleatória de nêutrons em material físsil ([Hammersley, 2013]). Ainda segundo [Hammersley, 2013] a possibilidade de aplicar métodos de Monte Carlo para problemas determinísticos foi notado por Fermi, von Neuman, e Ulam e popularizado após a guerra (segunda guerra mundial).

O método de Monte Carlo possui diversas variantes, mas todas elas seguem os seguintes passos: Dado uma probabilidade de um evento acontecer chamado de p, uma quantidade de números (ou conjuntos numéricos) aleatórios são gerados, a quantidade de vezes que esse conjunto aleatório disparou o evento dividido pela quantidade de números (ou conjuntos numéricos) gerados deve ser próximo de p.

1.4.1.1 Utilizando método de Monte Carlo para calcular π

Um exemplo bem difundido para utilização do método de Monte Carlo é para definir o valor de π. Primeiro é calculado a probabilidade de, ao gerar posições aleatórias dentro de uma área de um quadrado essa posição também estar dentro da área de um círculo (que está dentro desse quadrado) como mostra a figura 1.3.

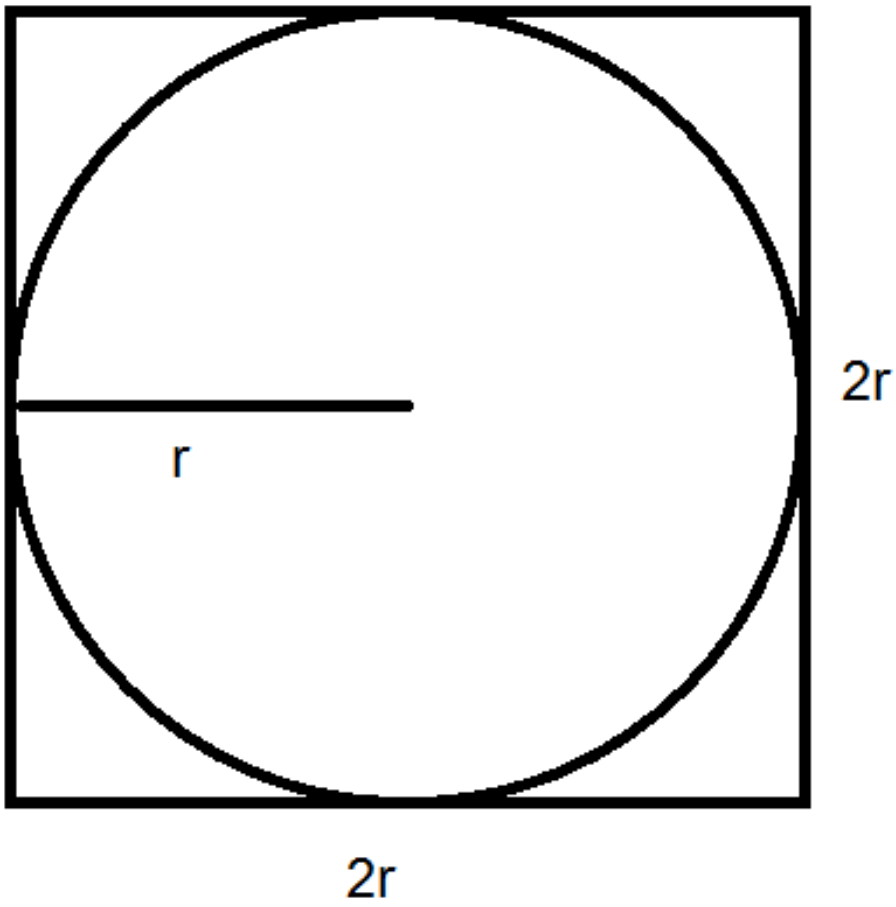

Figura 1.3: Quadrado com lado de $2r$ contendo um círculo com raio r.

A área de um quadrado é dada pela seguinte equação:

$$A = l^2 \tag{1.1}$$

Onde l é o lado do quadrado.

Já a área do círculo é dada pela seguinte equação:

$$A = \pi \times r^2 \tag{1.2}$$

Onde r é o raio do círculo.

Substituindo os valores da equação temos que área do quadrado é $4r^2$ e a área do círculo é πr^2. Logo, a chance de aleatoriamente ser escolhida uma posição dentro do quadrado e essa posição também estar dentro do círculo é dado por:

$$p = \frac{\pi r^2}{4r^2} \tag{1.3}$$

Isolando π (valor que queremos descobrir) temos a seguinte equação:

$$\pi = 4p \tag{1.4}$$

Com o auxílio de um computador é gerado 100.000 posições aleatórias. Dessa amostra 78239 ficavam dentro do círculo. Para isso utilizamos a seguinte equação:

$$x^2 + y^2 \leq r^2 \tag{1.5}$$

Onde x é a posição horizontal e y a posição vertical. A imagem 1.4 faz uma demonstração gráfica desse conjunto de números gerados, os pontos mais escuros mostram posições geradas fora do círculo e os pontos mais claros as posições geradas dentro da área do círculo.

Voltando a definição temos o seguinte: a quantidade de vezes que esse conjunto aleatório disparou o evento (78466) dividido pela quantidade de números (ou conjunto numéricos) aleatórios gerados (100.000) deve ser próximo de p ($\pi = 4p$ ou $p = \frac{\pi}{4}$). Substituindo os valores já conhecidos temos o seguinte:

$$\frac{\pi}{4} = \frac{78621}{100000} \tag{1.6}$$

Isolando π e resolvendo a equação temos:

$$\pi = \frac{314484}{100000} \tag{1.7}$$

Finalmente o valor aproximado de $\pi = 3,14484$.

1.4.1.2 Utilizando método de Monte Carlo para probabilidade de uma roleta

Com o método de Monte Carlo é bem fácil calcular probabilidades mais complexas. A imagem 1.5 representa uma roleta que pode ser dividida em 4 partes: em duas dessas partes o valor ganho é de +1, em uma dessas quatro partes o valor da recompensa é +2 e na última parte o valor é de −2. Nesse caso, vamos usar o método de Monte Carlo para

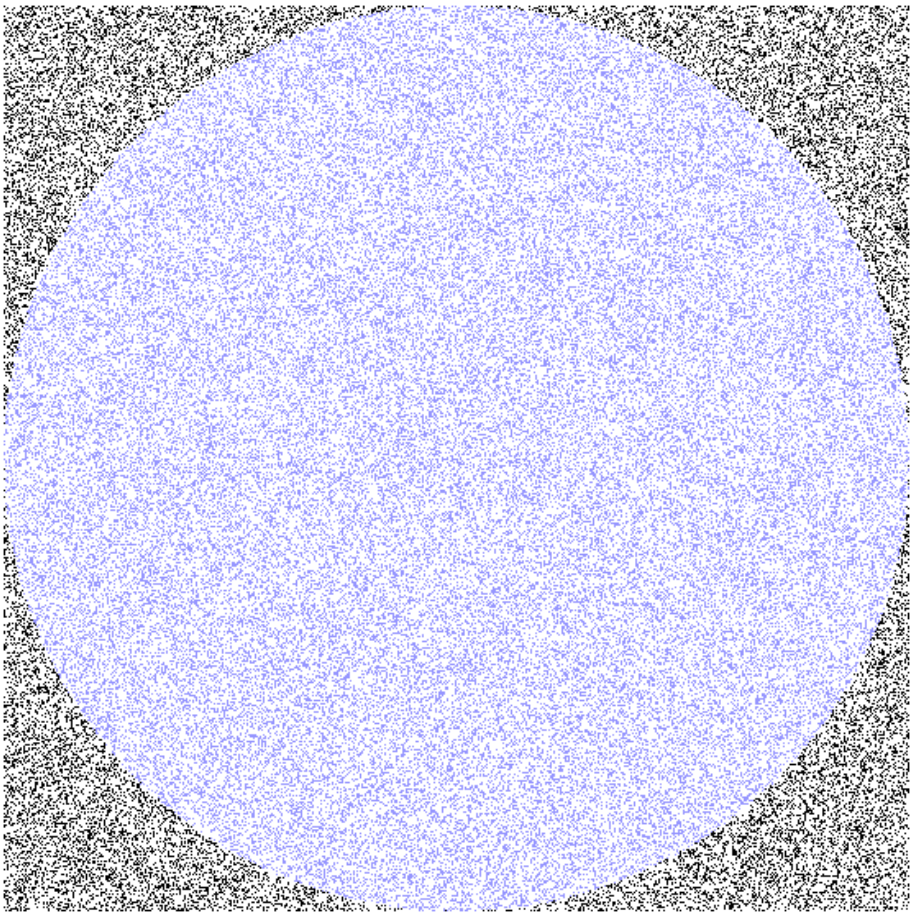

Figura 1.4: Demonstração gráfica de geração de 100.000 posições aleatórias.

descobrir qual a probabilidade de após girar 20 vezes a roleta, a soma dos pontos obtidos ser menor que zero.

Será feito 1.000.000 de simulações, em cada simulação será girado a roleta 20 vezes e os pontos serão somados. Após as simulações temos que em 64.020 das simulações o resultado da soma resultou em valores negativos. Voltando a definição temos:

$$p = \frac{64020}{100000} \tag{1.8}$$

Ou seja, após rodar 20 vezes a roleta da figura 1.5 a chance da soma dos pontos ser negativa é de $0,06402$.

1.4.2 Avaliação de nós utilizando Monte Carlo

O primeiro contato do método de Monte Carlo com árvore de busca foi na avaliação de nós ([Nijssen, 2013], [Winands et al., 2008]). Essa avaliação era chamada de avaliação de Monte Carlo (do inglês Monte Carlo Evaluation - MCE).

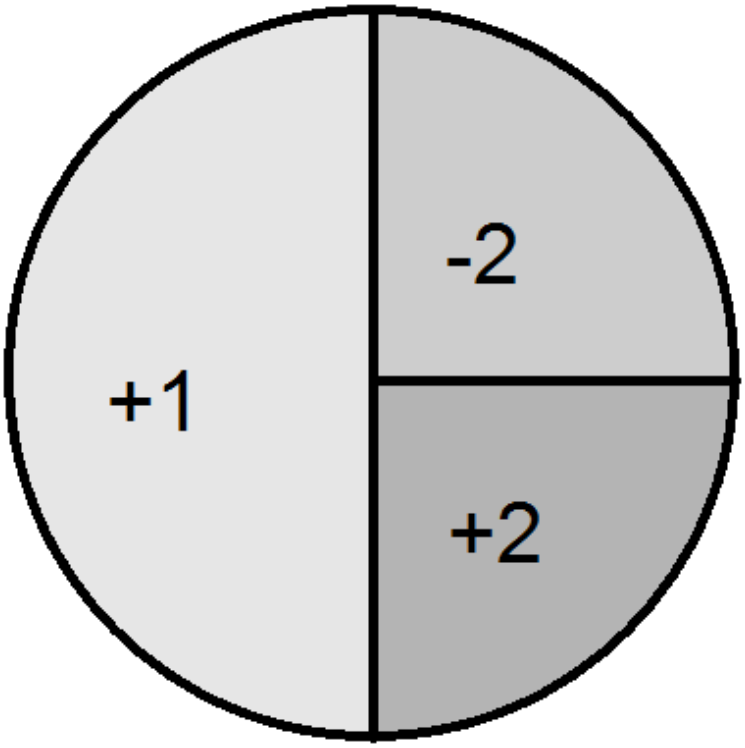

Figura 1.5: Distribuição de recompensas de uma roleta.

Em vez de usar recursos dependentes do domínio, tais como a mobilidade ou vantagem de um material, uma série de *playouts* (chamado também de amostras, lançamentos, reprodução ou simulações) é iniciado a partir da posição a ser avaliada. O *playout* é uma sequência de movimentos (semi-) aleatórios. O *playout* encerra quando o jogo está terminado e um vencedor pode ser determinado de acordo com as regras do jogo. A avaliação de uma posição é determinada por combinação dos resultados de todos os *playouts* usando uma função de agregação estatística. Isso pode ser, por exemplo, a pontuação média do jogo ([Nijssen, 2013]).

Por não exigir muito conhecimento do domínio ela funciona muito bem em jogos onde não existe uma função de avaliação de nós intermediários muito bom (como em Go por exemplo), como no trabalho de [Brügmann, 1993] onde o MCE foi aplicado a um tabuleiro de 9x9. A grande desvantagem dessa técnica é a quantidade de tempo que ela leva para avaliar um determinado nó, pois precisa chegar até um nó folha, fazendo a técnica eficaz apenas em árvores com profundidade de nós baixa. Uma derivação direta dessa abordagem é a árvore de busca de Monte Carlo.

1.4.3 Monte Carlo em árvores de busca

Por décadas a busca alpha-beta tem sido a abordagem padrão usada em programas de jogos soma-zero de dois jogadores como xadrez e damas (entre outros). Ao passar dos

Esse jogo é particularmente desafiador para os algoritmos de busca em árvore pois:

anos, muitas melhorias foram propostas para essa solução. Porém, em alguns jogos onde é difícil construir uma função de avaliação precisa (e.g., Go) a abordagem alpha-beta foi mal sucedida ([Winands et al., 2008]).

O algoritmo de busca de Monte Carlo ganhou muita notoriedade por apresentar uma alta competitividade ao enfrentar os melhores jogadores do jogo Go. Go é um jogo muito difícil para computadores jogar: tem alto fator de ramificação, a uma árvore profunda, e não existe nenhuma função de validação de heurística confiável para nós não terminais ([aend Edward Powley et al., 2012]). Ainda segundo os autores, nos últimos anos MCTS tem alcançado grande sucesso em muitos jogos específicos, jogos generalistas, complexos planejamentos do mundo real, otimização e controle de problemas, e se tornou uma importante ferramenta do pesquisador de inteligência artificial.

Outro grande ponto do algoritmo de busca de Monte Carlo é o resultado positivo que ele obteve em jogos com informação incompleta e/ou imperfeita como nos trabalhos de [Cai and Wurman, 2005] e [Misra, 2013].

você está falando de busca de monte carlo ou mcts? decida-se e siga uma linha de raciocínio

O MCTS é um método para encontrar soluções ótimas em um determinado domínio utilizando de amostras aleatórias no espaço de decisão e construindo uma árvore de busca de acordo com os resultados ([aend Edward Powley et al., 2012]). A árvore de busca de Monte Carlo é um algoritmo baseado em simulação, frequentemente usado em jogos. A ideia principal é iterativamente rodar simulações do nó raiz da árvore até um nó terminal, de maneira incremental crescendo a árvore onde o nó raiz é o estado atual do jogo ([Tak et al., 2014]).

O algoritmo de MCTS é um processo iterativo que ocorre até que um determinado critério de parada seja alcançado. Segundo [aend Edward Powley et al., 2012], geralmente essa iteração é limitada por algum recurso computacional como tempo, memória ou algum limitador de números de iteração.

Basicamente o MCTS é uma árvore que cresce de modo incremental e assíncrono como mostra a figura 1.6. O que define a estratégia de como a árvore irá expandir são as seguintes políticas:

- **Política de árvore**: é a estratégia de qual nó escolher, essa política tem que balancear escolher nós pouco visitados e aprofundar nós que parecem promissores ([aend Edward Powley et al., 2012]). Essa política procura um **nó expansível**, ou seja, um nó não folha e que ainda não foi visitado.

- **Política padrão**: é a estratégia de como rodar as simulações até sua conclusão. Um caso bem simples de política padrão é fazer movimentos aleatórios.

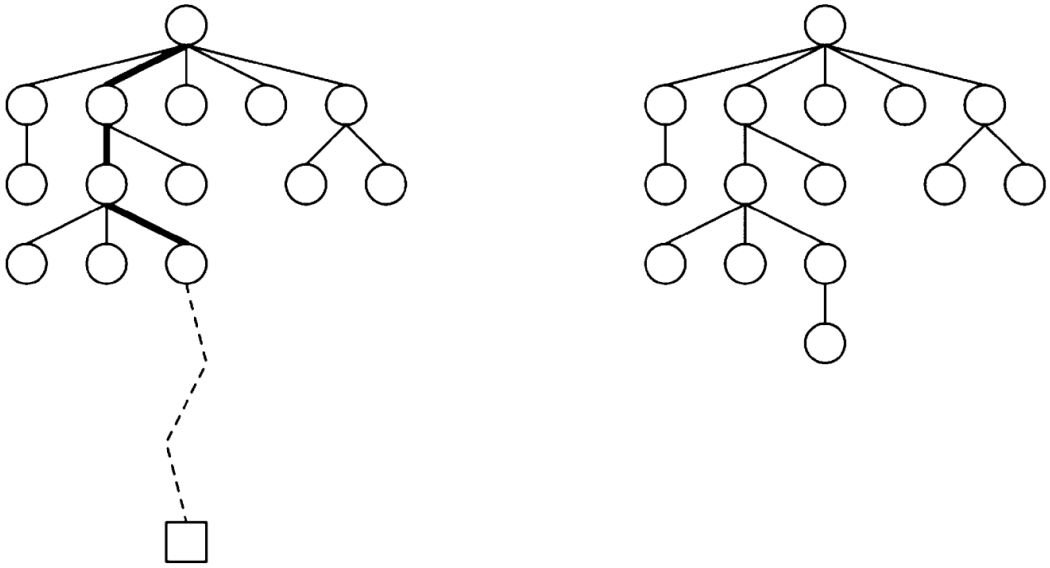

Figura 1.6: Representação Básica do MCTS [Baier and Drake, 2010].

Pode-se descrever cada iteração do algoritmo em quatro passos básicos:

- **Seleção** Começando pelo nó raiz da árvore, um nó filho é escolhido de acordo com a **política de árvore** até que seja encontrado um nó expansível.
- **Expansão** É adicionado um ou mais nós filhos no nó escolhido de acordo com as possíveis ações naquele estado.
- **Simulação** Simula o jogo a partir dos novos nós gerados (geralmente é apenas um nó) na fase de expansão até que se alcance o fim de jogo. As escolhas das ações a partir desse novo nó é definido pela **política padrão**. Conhecimentos sobre o jogo podem ser adicionados para a escolha dessas ações fazendo a simulação mais realística ([Nijssen, 2013]).
- **Retropropagação** O resultado (valor) da simulação é propagado para todos os pais do nó gerado até o nó raiz, atualizando as estatísticas de todos seus ancestrais. Normalmente os valores para os resultados do jogo são: vitória 1, empate $\frac{1}{2}$ e derrota 0.

A imagem 1.7 mostra o processo completo do MCTS. Na imagem, V é o valor da recompensa (vitória, empate ou derrota) de um nó folha e Δ é o resultado das estatísticas do nó (combinação de recompensas positivas e negativas).

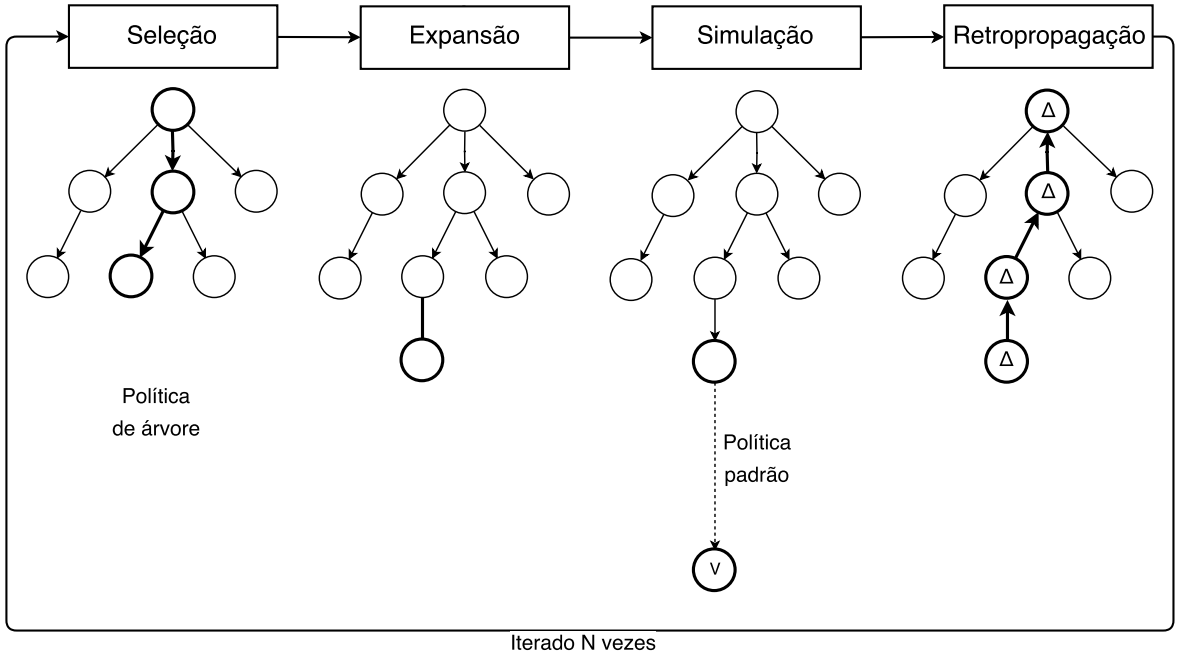

Figura 1.7: MCTS esquema.

figuras, tabela e algoritmos são flutuantes, você não sabe se estarão antes ou depois do texto, sempre referência pelo numero: Alg.3

Os algoritmos a seguir mostram o algoritmo do método principal, do método selecionar e simular respectivamente.

Algorithm 3 Algoritmo MCTS - método principal

```
1: procedure MCTS(noAtual)
2:     while recurso computacional do
3:         noSelecao ← SELECIONAR(noAtual)
4:         noExpansao ← EXPANDIR(noSelecao)
5:         valorResultado ← SIMULAR(noExpansao)
6:         RETROPROPAGAR(noExpansao, valorResultado)
7:     end while
8:     return MELHORFILHO(noAtual)
9: end procedure
```

Algorithm 4 Algoritmo MCTS - método Selecionar

```
1: procedure SELECIONAR(no)
2:     while no.filhos > 0 do
3:         no ← POLITICAARVORE(no)
4:     end while
5:     return no
6: end procedure
```

Algorithm 5 Algoritmo MCTS - método Simular

```
1: procedure SIMULAR(no)
2:     acao ← no.acao
3:     while not acao.isFimDeJogo do
4:         EXECUTAR(acao)
5:         acoes ← LERACOES( )
6:         acao ← ESCOLHERACAOALEATORIA(acoes)
7:     end while
8:     valor ← 1/2
9:     if acao.isVitoria then
10:        valor ← 1
11:    else if acao.isDerrota then
12:        valor ← 0
13:    end if
14:    return valor
15: end procedure
```

~~Onde:~~ V. de comentários anteriores

- *noAtual*: Nó que representa o estado atual do jogo. No primeiro turno é passado o nó raiz da árvore.
- MELHORFILHO(no): Escolhe o melhor nó filho. O conceito de melhor filho varia entre as diferentes implementações do MCTS.

Ao finalizar o MCTS existem quatro critérios de como escolher o melhor filho ([Schadd, 2009]):

- **Filho máximo**: Seleciona o filho com valor mais alto.
- **Filho robusto**: Seleciona o filho com o maior número de visitas.

- **Filho máximo-robusto**: Selecione o filho que tem tanto o maior número de visita e o valor mais alto. Se não existir esse filho, é recomendável continuar a busca até achar um filho máximo-robusto do que escolher um filho com baixo número de visitas.
- **Filho seguro**: Selecione o filho com menor chance de ter resultado negativo.

1.4.4 Upper Confidence Bound for Trees (UCT)

A implementação mais comum do algoritmo MCTS é o *Upper Confidence Bound for Trees* (UCT) ([aend Edward Powley et al., 2012]).

Um dos grandes dilemas na definição de política de árvore é como balancear as escolhas, entre explorar novos nós para fugir de máximos locais e aprofundar de uma subárvore existente podendo achar melhores resultados ou garantindo uma maior robustez na avaliação de um nó. Segundo [Auer et al., 2002] o Upper Confidence Bound for Trees (UCT) resolve essa questão de maneira ótima até um fator constante.

O algoritmo UCT pode ser decomposto em duas partes: MCTS e Upper Confidence Bounds (UCB). Sua primeira implementação formal foi em 2006 por Kocsis e Szepesvari [Kocsis and Szepesvári, 2006]. O algoritmo UCB entra como uma política de árvore para implementação de MCTS.

1.4.4.1 Upper Confidence Bounds (UCB)

O algoritmo mais tradicional de Upper Confidence Bounds é o UCB1 que foi inicialmente aplicado para o problema *multiarmed bandit* ([Auer et al., 2002]). Segundo [Auer, 2002] o termo *multiarmed bandit problem* ou problema dos caça-níqueis (ou mais precisamente "problema dos K caça-níqueis") reflete o problema de um apostador em uma sala com várias máquinas de caça-níquel. Em cada tentativa o apostador tem que decidir em qual máquina ele vai jogar. Para maximizar o ganho total ou recompensa, sua escolha se baseia em recompensas anteriormente coletadas em cada máquina.

O UCB1 pode ser definido pela seguinte equação:

$$UCB1 = \overline{x}_j + \sqrt{\frac{2 \ln n}{n_j}} \tag{1.9}$$

Onde:

- : j: a escolha que está sendo analisada. No exemplo uma máquina caça-níquel específica.

- : $\overline{x}_j$: a média da recompensa paga pela máquina j.
- n_j: a quantidade de vezes que foi jogado na máquina j.
- n: A soma de quantas vezes foi jogado em cada máquina.

Nessa equação fica claro como está distribuído o fator exploração ($\sqrt{\frac{2\ln n}{n_j}}$) e o fator aprofundamento ($\overline{x}_j$). A imagem 1.8 ilustra essa separação.

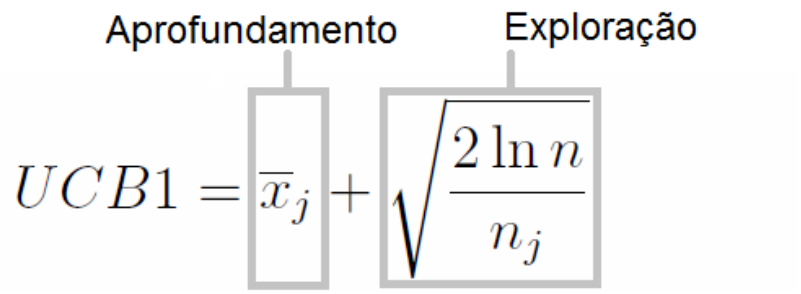

Figura 1.8: UCB1 exploração e aprofundamento na equação.

Por esse motivo é comum encontrar na literatura a equação UCB1 da seguinte forma:

$$UCB1 = \overline{x}_j + C\sqrt{\frac{2\ln n}{n_j}} \tag{1.10}$$

Onde C é uma constante que regula o valor da exploração, ou seja, caso queira aumentar a exploração basta $C > 1$, caso queira diminuir a exploração $1 > C > 0$.

1.4.4.2 Política de árvore UCB1

A aplicação do UCB1 como política de árvore funciona do seguinte modo: no processo de seleção, a escolha pode ser modelada com um problema *multiarmed bandit* independente. A utilização do MCTS com qualquer política de árvore UCB é chamada de UCT.

Uma variação proposta por [Kocsis et al., 2006] chamada de *plain UCT* provou ter resultados muito superiores ao UCT comum. Ainda segundo [Kocsis et al., 2006] com essa modificação a chance de selecionar o melhor movimento converge em 1 e, através de testes empíricos foi observado que é mais rápido que outros algoritmos de busca em árvore como o *alpha-beta*.

A equação do *plain UCT* é definida da seguinte forma:

$$UCT = \overline{x}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}} \tag{1.11}$$

Onde o valor de C_p é sugerido como $\frac{1}{\sqrt{2}}$ e pode ser ajustado para mais ou menos para regular o nível de exploração.

1.4.4.3 Algoritmo UCT

Baseado no algoritmo MCTS previamente explicado, vamos fazer algumas alterações para implementar o UCT. Primeiro o código do método principal:

Algorithm 6 Algoritmo UCT

```
1: procedure UCT(noAtual)
2:     while recurso computacional do
3:         noSelecao ← SELECIONAR(noAtual)
4:         noExpansao ← EXPANDIR(noSelecao)
5:         valorResultado ← SIMULAR(noExpansao)
6:         RETROPROPAGAR(noExpansao, valorResultado)
7:     end while
8:     return UCB1(noAtual)
9: end procedure
```

Método UCB1:

Algorithm 7 Algoritmo UCT - UCB1

```
1: procedure UCB1(no)
2:     melhorValor ← −∞
3:     melhorFilho ← null
4:     for each child ∈ no do
5:         valor ← child.value + C √(2 ln no.visitas / child.visitas)
6:         if valor > melhorValor then
7:             melhorValor ← valor
8:             melhorFilho ← child
9:         end if
10:    end for
11:    return melhorFilho
12: end procedure
```

Método Selecionar:

Algorithm 8 Algoritmo UCT - Selecionar

```
1: procedure SELECIONAR(no)
2:     while no.filhos > 0 do
3:         no ← UCB1(no)
4:     end while
5:     return no
6: end procedure
```

E por fim o retropropagar:

Algorithm 9 Algoritmo UCT - Retropropagar

```
1: procedure RETROPROPAGAR(no, valor)
2:     no.visitas ← no.visitas + 1
3:     no.valor ← no.valor + valor
4:     if no.pai not null then
5:         RETROPROPAGAR(no.pai, valor)
6:     end if
7: end procedure
8:
```

1.4.4.4 Exemplo UCT

Dado o algoritmo explicado na subseção anterior, vamos executar um exemplo. Partindo da seguinte árvore da figura 1.9 será executada uma iteração passo a passo. Na imagem, os valores dentro do nó representam valor (soma das recompensas) / visitas (quantas vezes o nó foi visitado).

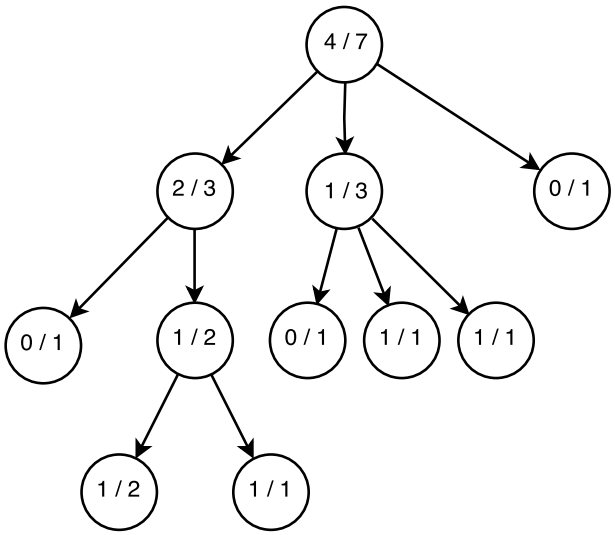

Figura 1.9: MCTS Exemplo - Árvore inicial.

Começando pelo nó raiz chamamos a função UCT (será utilizado $C = \sqrt{2}$). Aplicando o método UCB1 a todos os filhos do nó raiz a árvore fica do seguinte modo 1.10 (na imagem os valores de UCB1 estão do lado esquerdo da árvore).

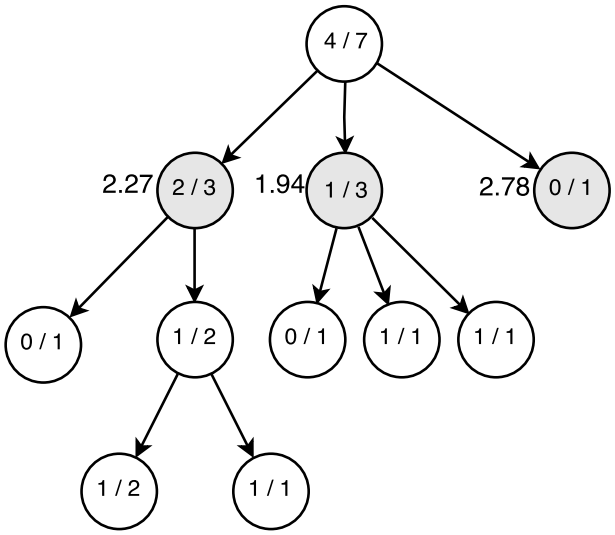

Figura 1.10: MCTS Exemplo - UCB1 aplicado.

O nó com maior valor de UCB1 não tem filhos, ou seja, ele é expansível. Expandido ele e fazendo a simulação teremos a árvore como indica na figura 1.11 (a simulação resultou em vitória, por isso o valor 1).

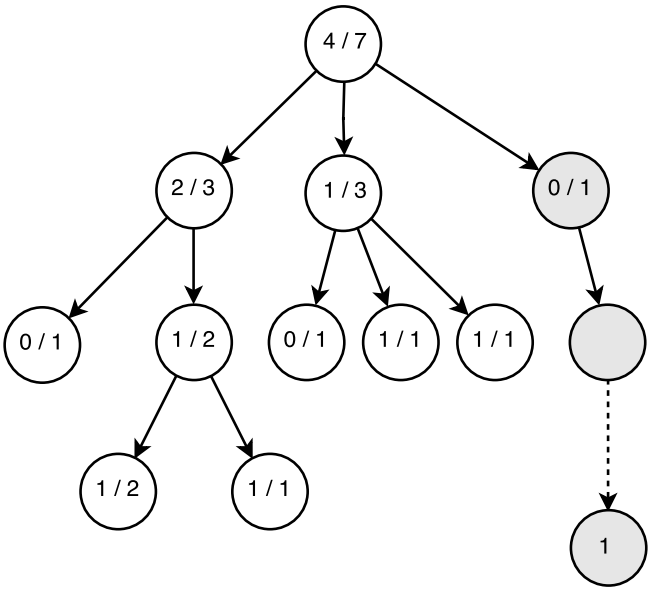

Figura 1.11: MCTS Exemplo - Expansão e simulação.

O último passo da nossa iteração é a retropropagação. A imagem 1.12 mostra como ficou a árvore depois da última adição e simulação.

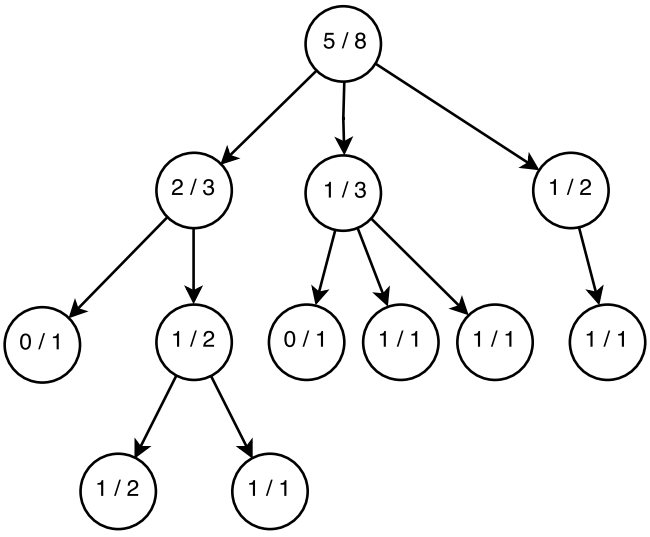

Figura 1.12: MCTS Exemplo - Expansão e simulação.

Após isso voltamos para o início do laço e, caso ainda tenha recurso computacional, começamos novamente utilizando a árvore atualizada.

1.4.5 Grupos de movimentos

Um aprimoramento para o UCT para quando se tem muitas ações parecidas é a chamada grupos de movimentos. Proposto por [Childs et al., 2008] essa melhoria diminui consideravelmente a quantidade de ramos da árvore Monte Carlo. A técnica consiste em criar uma nova camada onde todas as possíveis ações são separadas em grupos e o UCB1 é usado para selecionar qual grupo será escolhido. A imagem a 1.13 ilustra esse processo.

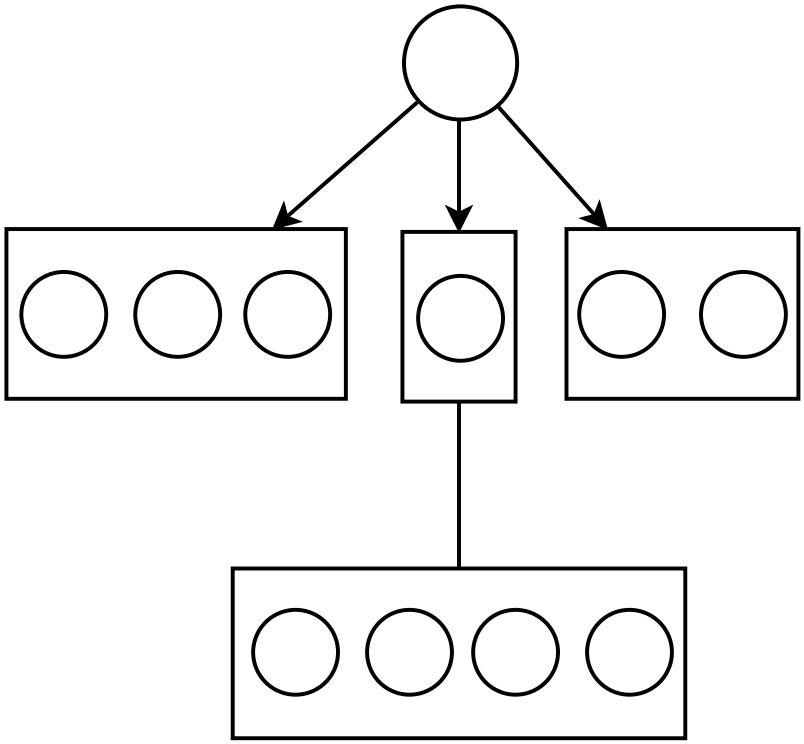

Figura 1.13: MCTS grupos de movimentos.

1.5 Algoritmos de busca em árvore

O algoritmo de *minimax* por décadas foi a melhor opção para criar agentes competitivos em jogos combinatórios. Com o grande número de pesquisadores estudando o tema diversos aprimoramentos e técnicas foram criadas para melhorar diversos pontos da técnica base (como o alpha-beta). O grande ponto de notoriedade do *minimax* foi em 1997 quando o sistema Deep Blue venceu o Kasparov (campeão mundial) no xadrez, o que na época parecia muito difícil devido a alta complexidade do xadrez.

Apesar do sucesso, o *minimax* nunca teve muito sucesso competitivamente em jogos com ainda mais possibilidades de jogadas como o Reversi ou Go. O MCTS vem ganhando

bastante espaço na área de pesquisa, pois ele não tem a necessidade de uma função de heurística precisa como o alpha-beta e apresentam resultados quase tão bons quanto ele ([Kocsis et al., 2006]). O grande feito do MCTS foi em outubro de 2015 quando o AlphaGo sistema de desenvolvido pela Google DeepMind derrotou o campeão mundial Lee Sedol no Go (tabuleiro de 19x19) ([Mind, 2016]). O AlphaGo utiliza MCTS combinado com redes neurais para reduzir a profundidade e largura efetiva da árvore de busca [Silver et al., 2016].

Na imagem 1.14 (adaptado de [XKCD, 2016]) mostra a relação da dificuldade dos jogos com capacidade do computador derrotar profissionais em alguns jogos.

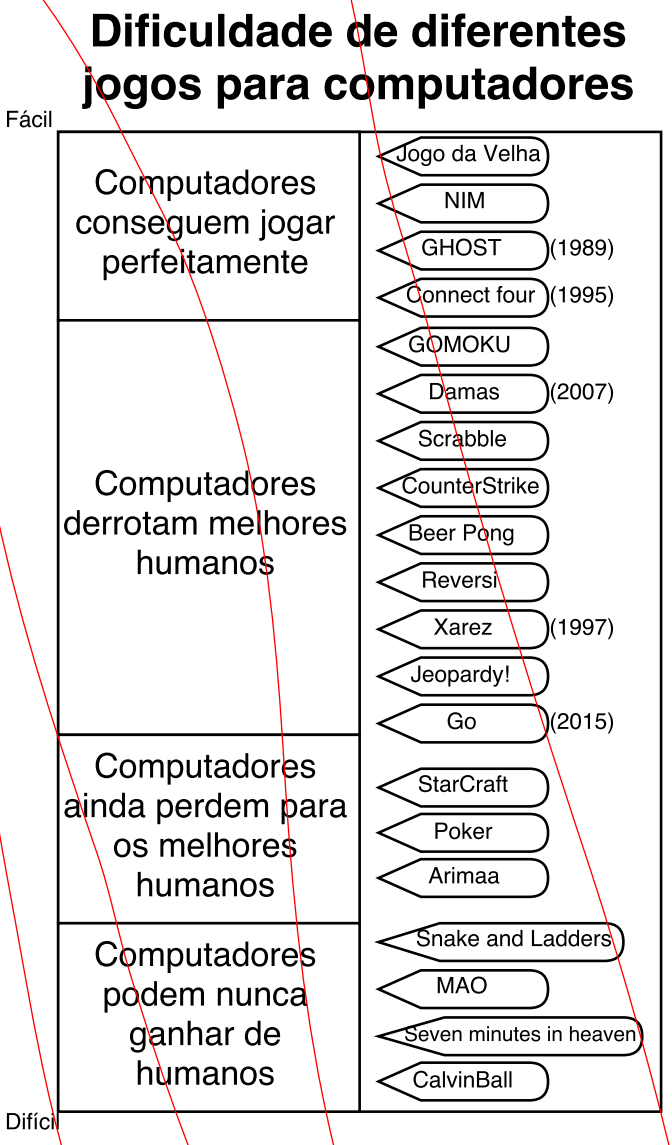

Figura 1.14: Dificuldade dos jogos e IA (adaptado de [XKCD, 2016]).

1.7 Considerações Finais

Referências Bibliográficas

[aend Edward Powley et al., 2012] aend Edward Powley, C. B., Whitehouse, D., Lucas, S., Cowling, P. I., Tavener, S., Perez, D., Samothrakis, S., Colton, S., and et al. (2012). A survey of monte carlo tree search methods. *IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI*.

[Auer, 2002] Auer, P. (2002). Using confidence bounds for exploitation-exploration tradeoffs. *Journal of Machine Learning Research*, 3(Nov):397–422.

[Auer et al., 2002] Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256.

[Baier and Drake, 2010] Baier, H. and Drake, P. D. (2010). The power of forgetting: Improving the last-good-reply policy in monte carlo go. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):303–309.

[Baker et al., 2002] Baker, D., Bridges, D., Hunter, R., Johnson, G., Krupa, J., Murphy, J., and Sorenson, K. (2002). *Guidebook to Decision-Making Methods*. Department of Energy, USA.

[Brügmann, 1993] Brügmann, B. (1993). Monte carlo go technical report.

[Cai and Wurman, 2005] Cai, G. and Wurman, P. R. (2005). Monte carlo approximation in incomplete information, sequential auction games. *Decis. Support Syst.*, 39(2):153–168.

[Campbell and Marsland, 1983] Campbell, M. S. and Marsland, T. A. (1983). A comparison of minimax tree search algorithms. *Artificial Intelligence*, 20(4):347–367.

[Childs et al., 2008] Childs, B. E., Brodeur, J. H., and Kocsis, L. (2008). Transpositions and move groups in monte carlo tree search. In *2008 IEEE Symposium On Computational Intelligence and Games*, pages 389–395. IEEE.

[Christensen and Korf, 1986] Christensen, J. and Korf, R. E. (1986). A unified theory of heuristic evaluation functions and its application to learning. In *Proceedings of the 5th*

National Conference on Artificial Intelligence. Philadelphia, PA, August 11-15, 1986. Volume 1: Science., pages 148–152.

[Diller et al., 2004] Diller, D. E., Ferguson, W., Leung, A. M., Benyo, B., and Foley, D. (2004). Behavior modeling in commercial games. *Behavior Representation in Modeling and Simulation (BRIMS)*, 68.

[Du and Pardalos, 1995] Du, D. and Pardalos, P. (1995). *Minimax and Applications*. Nonconvex Optimization and Its Applications. Springer US.

[Hammersley, 2013] Hammersley, J. (2013). *Monte Carlo Methods*. Monographs on Statistics and Applied Probability. Springer Netherlands.

[Harris, 1998] Harris, R. (1998). Introduction to Decision Making. http://www.virtualsalt.com/crebook5.htm. Acessado em: 24-07-2016.

[Hsu et al., 1995] Hsu, F., Campbell, M.S., H., and A.J.J (1995). Deep blue system overview. *Proceedings of the 9th ACM Int. Conf. on Supercomputing*, pages 240–244.

[Hsu, 1999] Hsu, F.-H. (1999). Ibm's deep blue chess grandmaster chips. *IEEE Micro*, 19(2):70–81.

[Kleij, 2010] Kleij, A. (2010). Monte carlo tree search and opponent modeling through player clustering in no-limit texas hold'em poker. *M. Sc. University of Groningen, Netherlands*.

[Knuth and Moore, 1976] Knuth, D. E. and Moore, R. W. (1976). An analysis of alpha-beta pruning. *Artificial intelligence*, 6(4):293–326.

[Kocsis and Szepesvári, 2006] Kocsis, L. and Szepesvári, C. (2006). Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer.

[Kocsis et al., 2006] Kocsis, L., Szepesvári, C., and Willemson, J. (2006). Improved monte-carlo search. *Univ. Tartu, Estonia, Tech. Rep*, 1.

[Millington and Funge, 2009] Millington, I. and Funge, J. (2009). *Artificial Intelligence for Games, Second Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition.

[Mind, 2016] Mind, G. D. (2016). AlphaGo - Google DeepMind. https://deepmind.com/alpha-go. Acessado em: 01-08-2016.

[Misra, 2013] Misra, S. (2013). Markov chain monte carlo for incomplete information discrete games. *Quantitative Marketing and Economics*, 11(1):117–153.

[Nielsen, 1994] Nielsen, J. (1994). Usability inspection methods. chapter Heuristic Evaluation, pages 25–62. John Wiley & Sons, Inc., New York, NY, USA.

[Nijssen, 2013] Nijssen, J. (2013). Monte-carlo tree search for multi-player games.

[Schadd, 2009] Schadd, F. C. (2009). *Monte-Carlo search techniques in the modern board game Thurn and Taxis*. PhD thesis, Maastricht University.

[Silver et al., 2016] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489.

[Tak et al., 2014] Tak, M. J., Lanctot, M., and Winands, M. H. (2014). Monte carlo tree search variants for simultaneous move games. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–8. IEEE.

[Winands et al., 2008] Winands, M. H. M., Björnsson, Y., and Saito, J.-T. (2008). *Monte-Carlo Tree Search Solver*, pages 25–36. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Works, 2016] Works, H. D. B. (2016). How Deep Blue works. https://www.research.ibm.com/deepblue/meet/html/d.3.2.shtml. Acessado em: 07-02-2016.

[Wright, 2012] Wright, D. R. (2012). Finite state machines. *CSC215 Class Notes. Prof. David R. Wright website. N. Carolina State Univ. Retrieved July*, 14:1–28.

[XKCD, 2016] XKCD (2016). Game AIs. http://www.xkcd.com/1002/. Acessado em: 01-08-2016.