Luiz Fernando Cunha Duarte

Estudo de Algoritmos em uma Chess Engine

RELATÓRIO DE PROJETO FINAL

DEPARTAMENTO DE ENGENHARIA ELÉTRICA E DEPARTAMENTO DE INFORMÁTICA

Programa de graduação em Engenharia de Computação

Rio de Janeiro
Novembro de 2021

Pontifícia Universidade Católica
do Rio de Janeiro

Luiz Fernando Cunha Duarte

Estudo de Algoritmos em uma Chess Engine

Relatório de Projeto Final

Relatório de Projeto Final, apresentado ao programa de Engenharia de Computação da PUC-Rio como requisito parcial para a obtenção do titulo de Engenheiro de Computação.

Orientador: Prof. Marco Serpa Molinaro

Rio de Janeiro
Novembro de 2021

Resumo

Cunha Duarte, Luiz Fernando; Serpa Molinaro, Marco. **Estudo de Algoritmos em uma Chess Engine**. Rio de Janeiro, 2021. 39p. Projeto de Graduação – Departamento de Engenharia Elétrica e Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Esse projeto tem como principal objetivo a implementação e avaliação de diferentes algoritmos utilizados em *chess engines*. Dessa forma, foram investigados separadamente quatro algoritmos: Minimax, Monte Carlo Tree Search, Regressão Logística e Redes Neurais Convolucionais. Duas abordagens foram adotadas para comparar tais algoritmos, a primeira visava identificar a precisão nos lances expondo tais métodos a problemas de xeque-mate em dois lances. Já a segunda buscou medir o desempenho desses nas diferentes etapas do jogo, para isso elaborou-se 20 confrontos entre cada um dos algoritmos.

Palavras-chave

Xadrez, Minimax Search, Monte Carlo Tree Search, Redes Neurais

Sumário

1
Introdução

1.1
Motivação

Garry Kasparov, um dos melhores mestres de xadrez de todos os tempos, classifica tal jogo como a *Drosophila* do raciocínio (Kasparov 2018). Isso porque tal como os geneticistas usaram essa espécie de mosca como modelo para seus estudos, muitos cientistas utilizam-se desse esporte para tentar modelar os segredos do pensamento humano.

No campo da computação, a dificuldade em traduzir o pensamento enxadrístico em algoritmos instigou grandes cientistas, como Alan Turing, que se questionava se uma máquina jogadora de xadrez poderia representar a diferença essencial entre o potencial de um computador e a mente humana. Para ampliar a compreensão sobre essa indagação, deve-se investigar as características do jogo de xadrez como fez o matemático americano Claude Shannon, que, em 1950, estabeleceu um limite inferior conservador para a complexidade da árvore de movimentos em uma partida média de 40 pares de movimentos como 10^{120}. Tal resultado, conhecido como, Número de Shannon (Shannon Number), mostra a rapidez com que a quantidade de diferentes configurações do tabuleiro cresce com o número de movimentos. Sob essa perspectiva, a capacidade de processamento computacional pura se mostra inócua para a identificação do melhor movimento para um dado cenário.

Desse modo, para se implementar uma máquina de jogar xadrez existe a necessidade de se traduzir em algoritmos a capacidade humana de descartar prontamente movimentos considerados ruins, para concentrar os esforços na identificação da jogada ótima para um determinado cenário.

1.2
Situação Atual

Até 1997, acreditava-se que os computadores não conseguiriam vencer o melhores jogadores de xadrez. No entanto, no segundo match de seis partidas o software da IBM, Deep Blue, bateu o campeão mundial Garry Kasparov. A partir desse momento, com o rápido e constante aumento da capacidade de

processamento computacional ao longo dos anos, a diferença entre humanos e máquinas nesse jogo tem se tornado gradualmente maior.

Nesse sentido, poderia-se acreditar que depois de tantos anos de esforço foi possível traduzir as características do pensamento humano para algoritmos de identificação de movimentos bons em xadrez. Porém, ao se analisar os algoritmos efetivamente usados pelo Deep Blue, pode-se concluir que os computadores não precisam pensar como os seres humanos no caso do xadrez. Com técnicas, como *Minimax Search-Tree*, *Alpha-Beta Pruning*, *Progressive Deepening*, somadas a heurísticas para a avaliação de uma configuração do tabuleiro, o software da IBM foi capaz de bater um dos melhores enxadristas da história.

Dessa forma, muitas *Chess engines* famosas, como Stockfish e Fritz, utilizaram-se desses algoritmos e de outros, além de uma vasta biblioteca de aberturas e finais. No entanto, ultimamente um outro software tem dominado o mercado de xadrez: o AlphaZero. Tal programa tem como premissa o uso de *Deep Learning* e de outras técnicas como a *Monte Carlo Tree* para gerar conhecimento sobre xadrez utilizando-se somente das regras e uma grande base de jogos.

Atualmente, portanto, com a evolução dos softwares enxadrísticos e a superioridade desses sobre os melhores jogadores da atualidade passou-se a se considerar um outro tipo de competição, entre as máquinas. Não obstante, essas competições servem como propaganda para esses programas, uma vez que os melhores jogadores buscam aqueles com maior desempenho para utilizar em seu treinamento.

1.3 Objetivo do Projeto

Este projeto tem como intuito implementar e avaliar diferentes algoritmos usados pelas chess engines presentes no mercado de xadrez. Nesse sentido, é importante ressaltar que tais programas utilizam essas diferentes técnicas combinadas ou em diferentes fazes da construção do software. No entanto, o objetivo da presente análise é isolar cada um desses procedimentos visando entender os benefícios deles juntamente com os pontos fracos nas diferentes fases do jogo. Dessa forma, foi evitada qualquer interação entre cada um dos algoritmos para evitar que possíveis pontos fracos fossem mascarados. Posteriormente, para verificar a diferença de performance foi utilizada uma combinação de dois algoritmos implementados.

Tendo em vista o objetivo do projeto, buscou-se uma biblioteca de xadrez (Python Chess) para lidar com as regras do jogo, como também a leitura de

arquivos em formatos comuns ao ambiente do xadrez, além da conexão com chess engines famosas via código. O uso de uma biblioteca externa minimizou o esforço necessário para a elaboração dos algoritmos uma vez que toda lógica do jogo de xadrez esta presente nela.

Portanto, os algoritmos avaliados foram: Minimax com Alpha-Beta Pruning (Knuth-Moore), Monte Carlo Tree Search, Regressão Logística, Redes Neurais Convolucionais (CNN), Minimax com Alpha-Beta Pruning usando a CNN como fução de avaliação, Naive e Stockfish 14. Os dois últimos não são propriamente algoritmos, entretanto funcionam como limites inferiores e superiores de desempenho respectivamente.

Os resultados obtidos mostram que o Minimax, independente da função de avaliação do tabuleiro utilizada, por basear-se na busca exaustiva, possui uma grande precisão considerando-se o horizonte estipulado pela altura máxima da árvore de busca. Tal precisão garantiu um domínio dele contra os demais algoritmos implementados. Outro resultado interessante foi o observado na CNN que teve um bom desempenho nas fases de abertura e meio-jogo, mas um desempenho ruim nos finais das partidas.

2
Minimax

O algoritmo de minimax, nesse caso, “maxmin”, é muito comum em jogos de dois jogadores cujo objetivo do participante é maximizar o ganho mínimo. Dessa forma, ele fixa a atenção no jogador 1 e tenta maximizar o valor que esse jogador obterá no jogo, assumindo que o jogador 2 jogará de maneira ótima, ou seja, tentando minimizar o score jogador 1. Para isso, o algoritmo básico consiste nos seguintes passos: expandir a árvore de possíveis movimentos dos jogadores até chagar nos possíveis fim de jogo, verificar o valor obtido pelo jogador 1 em cada fim de jogo (mais informações sobre essa função de avaliação de posição na Seção 4), propagar esses valores para os nós internos da árvore, isto é, se um no interno corresponde ao jogador 1, então ele elenca os valores obtidos com os diferentes movimentos possíveis nesse ponto (ou seja, os nos filhos) e escolhe o movimento que maximiza o valor. Analogamente, se o no interno corresponde ao jogador 2, então ele escolhe o movimento (ou equivalentemente no filho) que minimiza o valor.
A árvore de lances abaixo ilustra o funcionamento desse algoritmo na prática, na qual podemos ver que os nós brancos optam sempre por escolher o valor máximo de seus filhos, e , por outro lado, os nós pretos optam por escolher o menor valor entre os seus filhos. No caso do xadrez, cada nó representa uma configuração do tabuleiro, e os filhos dele são todas as configurações atingíveis executando-se um dos lances possíveis. Assim, o valor de cada nó é dado pela avaliação da posição.

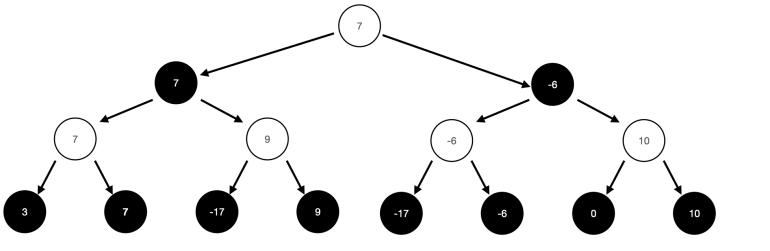


Figura 2.1: Árvore de lances.

Com isso, temos o seguinte pseudocódigo (Algoritmo 1) para o minimax aplicado ao xadrez. Para tornar o algoritmo concreto, foi estipulado que a

avaliação da posição atribui 1000 como o valor de uma vitória do jogador 1 e -1000 para uma derrota do jogador 1. As avaliações das posições variam nessa faixa de valores para posições em que não há xeque-mate para nenhum dos lados

Algoritmo 1: Minimax

```
MiniMax(posição, turno, profundidade)
    se profundidade = 0 ou posição é situação final então
        return Avaliação da posição, []
    fim
    se turno = "brancas" então
        maxVal ← −∞
        jogadas ← [ ]
        para todo movimento em movimentos possíveis faça
            Realiza movimento
            val,lances ← MiniMax(nova posição, "pretas", profundidade − 1)
            Desfaz movimento
            se val > maxVal então
                maxVal ← val
                jogadas ← [movimento] + lances
            fim
        fim
        return maxVal, jogadas
    fim
    senão
        minVal ← ∞
        jogadas ← [ ]
        para todo movimento em movimentos possíveis faça
            Realiza movimento
            val,lances ← MiniMax(nova posição, "brancas", profundidade − 1)
            Desfaz movimento
            se val < minVal então
                minVal ← val
                jogadas ← [movimento] + lances
            fim
        fim
        return minVal, jogadas
    fim
```

Podemos notar que a complexidade desse algoritmo cresce com a altura da árvore. Desse modo, estipulando que em cada posição tem-se b lances possíveis temos a seguinte complexidade de pior caso:

$$\sum_{i=1}^{\text{altura}} b^i \Rightarrow O(b^{altura}) \tag{2-1}$$

É importante ressaltar que devido a complexidade computacional do algoritmo para jogos complexos como o caso do xadrez, no qual a altura da

árvore é muito grande, não é possível expandir a árvore até uma posição final. Dessa forma, na verdade o que é feito é expandi-la até uma certa altura e utilizar uma *função de avaliação* para atribuir valor a esses nós folhas que não correspondam a posições finais do jogo. Falaremos mais sobres essas funções na Seção 4, e por enquanto é mais didático pensar que a árvore é expandida por completo.

3
Alpha-Beta Pruning

O *Alpha-Beta prunning* é uma extensão do algoritmo anterior (Game AIs with Minimax and Monte Carlo Tree Search) que tenta evitar expandir por completo a árvore do jogo. A ideia é ir calculando/expandindo progressivamente a árvore do jogo (exatamente como o procedimento minimax descrito acima) além de simultaneamente calcular limites inferiores e superiores α e β (respectivamente) para os nós e utilizá-los para eliminar sub-árvores mesmo antes de expandi-las. Mais precisamente, esses limites representam o seguinte:

- O α de um nó N é um valor tal que existe alguma opção de jogadas dos jogadores antes dessa posição (ou seja, os ancestrais de N) que levam o jogador 1 a uma posição onde ele pode ganhar pelo menos valor α. (Em particular, o nó N só será útil se seu valor for maior que seu α, caso contrário já tinha uma opção melhor de jogada anterior para o jogador 1.)
- O β de um nó N é um valor tal que existe alguma opção de jogadas dos jogadores antes dessa posição (ou seja, os ancestrais de N) que garante que o jogador 1 não consegue ganhar mais do que β. Intuitivamente isso quer dizer que tem opções de jogadas anteriores para o jogador 2 forçam o jogador 1 a ganhar no máximo β.

Antes de dizer como exatamente esses α's a β's são calculados, exemplificaremos como eles são utilizados para evitar a expansão de sub-árvores. Suponha que o algoritmo já tenha expandido parte da árvore do jogo, e esteja atualmente olhando para um nó N jogador 1. Suponha que o algoritmo testou um dos movimentos possíveis a partir desse no N e viu que ele obtém valor val maior que o β do nó N. Nesse momento, não é mais necessário explorar nenhum outro filho (movimento) do nó N: pela regra do minimax sabemos que o nó N terá valor maior ou igual a $val > \beta$ mas isso implica que esse jogo nunca alcançará esse nó, pois pela definição do β sabemos que tem opção anterior melhor para o jogador 2 (que tenta minimizar) que força o jogador 1 a ganhar no máximo β. Com isso, nesse momentos evitamos expandir as sub-árvores dos outros filhos de N. Analogamente, se em um no N do jogador 2 (que tenta

minimizar) encontra-se um movimento cujo valor *val* é menor que o α de N, pela regra minimax o valor de N será menor ou igual a $val < \alpha$. Porém como sabe-se que o jogador 1 tem outras opções de jogadas anteriores que o levar a valor pelo menos α, o nó N nunca será alcançado no jogo. Portanto, novamente, os outros filhos de N não precisam ser avaliados, e suas sub-árvores não são expandidas.

Para uma melhor compreensão do funcionamento do algoritmo, assim como da montagem da árvore iremos construir a árvore usada anteriormente. Porém, utilizando-se dessas cotas inferior e superior para diminuir o custo computacional. Com isso, tome como exemplo a posição inicial do xadrez que será considerada a raiz da árvore. Note que já na posição inicial existem 20 movimentos possíveis, para simplificar a análise imagine que só existem dois movimentos possíveis como indicado na figura abaixo.

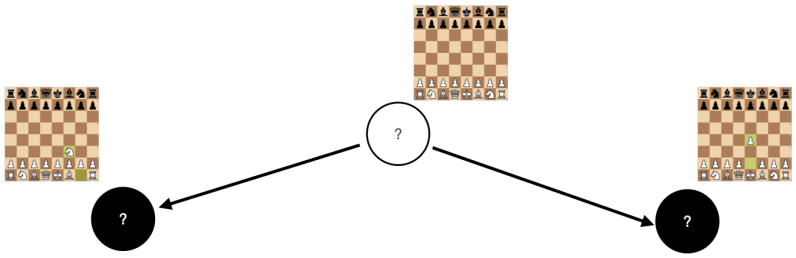


Figura 3.1: Árvore de lances.

Dessa forma, tem-se como objetivo descobrir quais dos dois lances é o melhor para o jogador de brancas. Com isso, inicialmente, iremos expandir o nó à esquerda a uma profundidade de dois níveis (mantendo a simplificação de somente dois lances possíveis). Como se pode observar na figura abaixo.

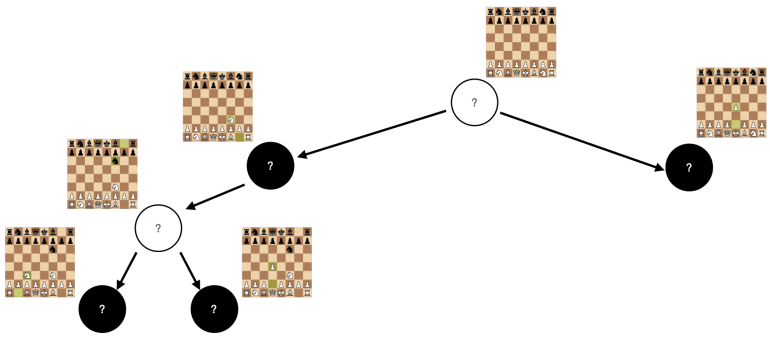


Figura 3.2: Árvore de lances.

Desse modo, chegou-se a duas posições "finais". Usando a função de avaliação no nó folha a esquerda obtemos 3 como valor para a posição. Assim,

como o pai desse nó é um nó branco, ou seja, quer maximizar o resultado obtidos podemos atualizar a cota inferior (α) desse nó para 3. Isso porque mesmo que o resultado do irmão desse nó seja pior que 3 o pai terá a oportunidade de escolher o lance que o da o melhor resultado.

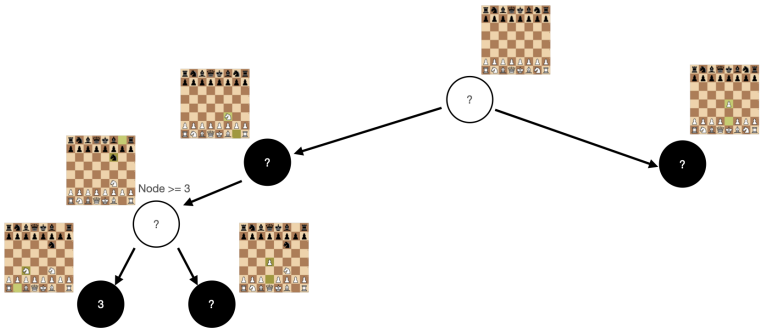


Figura 3.3: Árvore de lances.

Assim, embora tenhamos estabelecido uma cota inferior para o nó branco, ainda é necessário computar o valor do filho a direita para saber o real valor desse nó. Usando novamente a função de avaliação obtém-se o valor 7 para a posição. Com isso, o nó branco que quer otimizar o resultado do movimento irá optar pelo filho a direita e assumir o valor 7. Além de determinar o valor desse nó, pode estabelecer uma cota superior (β) para o nó avô das folhas analisadas. Isso porque ele é um nó preto e, portanto, quer minimizar os valores obtidos, o que faz com que independente da expansão da sub-árvore a direita possamos afirmar que o valor assumido por esse nó é no máximo 7.

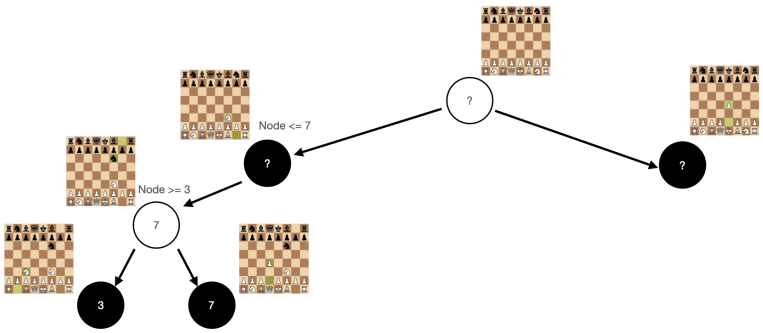


Figura 3.4: Árvore de lances.

Agora que a sub-árvore a esquerda do filho a esquerda da raiz foi computada. Para determinar o valor desse nó, deve-se computar a sub-árvore a direita dele juntamente com as cotas superior e inferior. Com isso, inicialmente

fazemos a expansão até a altura máxima estipulada como pode-se observar na figura abaixo.

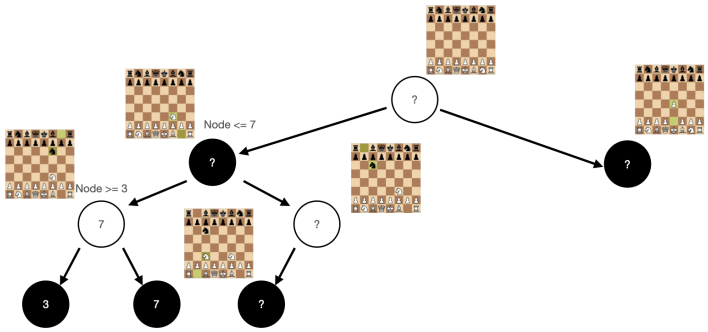


Figura 3.5: Árvore de lances.

Com isso, precisa-se avaliar o valor da folha encontrada. Usando a função de avaliação obtém-se o valor 9. Assim, pode-se determinar a cota inferior para o nó pai desta folha como 9, de maneira análoga ao que foi feito anteriormente. Além disso, visto que o avô da folha possui uma cota superior igual a 7, não é mais necessário computar a outra folha da árvore, uma vez que como o pai das folhas tem uma cota inferior de 9. Logo, independente do valor que o pai assuma ele não será escolhido por ser maior que a cota superior do avô.

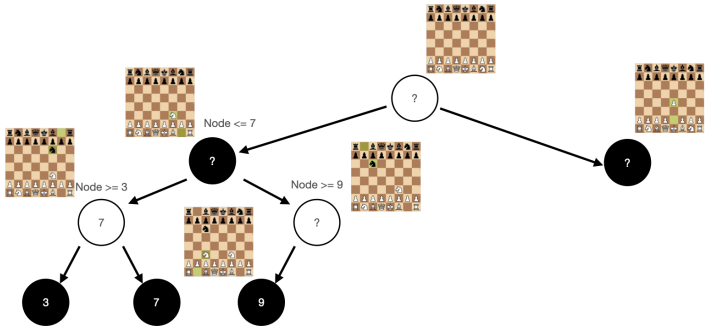
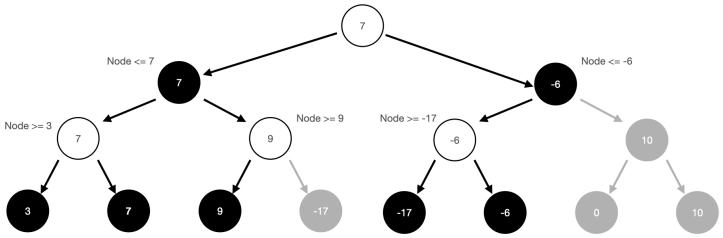


Figura 3.6: Árvore de lances.

Aplicando os passos do algoritmo no restante da árvore podemos notar que alguns nós não precisaram ser computados(em cinza). Isso auxilia a ilustrar como o uso do Alpha-Beta pruning pode reduzir a quantidade de esforço computacional.



Árvore de lances.

O detalhe de como exatamente esse procedimento, incluindo como os α's e β's são calculados, é apresentado no pseudo-código acima.

É importante notar que a adoção dessa técnica apesar de, em muitos casos, resultar em uma diminuição considerável do número de nós percorridos, ainda é possível que não seja possível eliminar nenhum nó tendo a complexidade de pior caso $O(b^{altura})$, onde b é o número de filhos de cada nó. No entanto, pode-se observar que no melhor caso a complexidade é $O(\sqrt{b^{altura}})$ (Knuth-Moore).

3.1 Ordenação de Movimentos

Note que a ordem em que os filhos de um nó são avaliados afeta o cálculo dos valores α e β. Dessa forma, pode-se tentar melhorar o desempenho dessa técnica fazendo com que o computador avalie primeiro lances mais prováveis de serem bons segundo algum critério. Note que esse método de ordenação de lances não é definitivo quanto a qualidade do lance, uma vez que se fosse não seria necessário o uso do algoritmo já que teríamos a melhor jogada vindo diretamente dele.

Um critério natural de ordenação é o seguinte: Para todo lance possível em uma dada posição, avalia-se a posição do tabuleiro após a execução desse lance utilizando-se uma *função de avaliação de estados* (mencionada anteriormente, e discutido em detalhes na próxima seção). Assim, para cada lance temos um score associado. Dessa forma, nos nós do jogador 1 (aquele que maximiza) os lances são ordenados em ordem decrescente de score, e nos nós do jogador 2 (aquele que minimiza) eles são ordenados em ordem crescente.

Tal heurística de ordenação de movimentos é inserida no algoritmo de Minimax com Alpha-Beta Pruning na parte em que se avalia a melhor resposta na posição para todos os movimentos possíveis, ou seja, com a inclusão dela, passa-se a ordenar todos os movimentos possíveis de modo que os melhores

Algoritmo 2: Minimax com Alpha Beta Pruning

```
MiniMax(posição, turno, profundidade, α, β)
    se profundidade = 0 ou posição é situação final então
    |  return Avaliação da posição, []
    fim
    se turno = "brancas" então
    |  maxval ← −∞
    |  jogadas ← []
    |  para todo movimento em movimentos possíveis faça
    |  |  Realiza movimento
    |  |  val,lances ← MiniMax(nova posição, "pretas", profundidade − 1, α, β)
    |  |  Desfaz movimento
    |  |  se val > maxVal então
    |  |  |  maxVal ← val
    |  |  |  jogadas ← [movimento] + lances
    |  |  fim
    |  |  se α < maxVal então
    |  |  |  α ← maxVal
    |  |  fim
    |  |  se β ≤ α então
    |  |  |  return maxVal, jogadas
    |  |  fim
    |  fim
    |  return maxVal, jogadas
    fim
    senão
    |  minVal ← ∞
    |  jogadas ← []
    |  para todo movimento em movimentos possíveis faça
    |  |  Realiza movimento
    |  |  val,lances ← MiniMax(nova posição, "brancas", profundidade − 1, α, β)
    |  |  Desfaz movimento
    |  |  se val < minVal então
    |  |  |  minVal ← val
    |  |  |  jogadas ← [movimento] + lances
    |  |  fim
    |  |  se β < minVal então
    |  |  |  β ← minVal
    |  |  fim
    |  |  se β ≤ α então
    |  |  |  return minVal, jogadas
    |  |  fim
    |  fim
    |  return minVal, jogadas
    fim
```

sejam avaliados primeiro.

Algoritmo 3: Heurística de ordenação de movimentos

Dados: Posição e turno
Resultado: Lista dos movimentos possíveis ordenados
dicionario_movimento_valor;
para *movimento em lances possíveis da posição* **faça**
    faz movimento;
    valor = avalia posição;
    insere movimento e valor em dicionario_movimento_valor;
fim
se *turno == brancas* **então**
    ordena movimentos em ordem decrescente de valor;
senão
    ordena movimentos em ordem crescente de valor;
fim

Em experimentos preliminares, notamos que esse critério trouxe um ganho de eficiência comparado com o algoritmo sem utilizar ordenação.

4
Função de avaliação do tabuleiro

Foi assumido, anteriormente, que a árvore de lances é expandida ilimitadamente até as posições finais. No entanto, como mencionado, isso resulta em um custo computacional muito alto. Uma possível solução para contornar esse obstáculo é truncar a expansão da árvore, limitando, com isso, a profundidade a ser explorada. Contudo, isso implica que algumas "folhas" não serão posições finais, o que não torna claro como avaliar a posição. Portanto, é necessário uma função de avaliação para um valor a posição indicando o quão boa é a posição para o jogador de brancas.

Em uma análise, superficial, pode-se pensar que a qualidade dos lances propostos por uma *chess engine* está relacionada exclusivamente aos algoritmos mencionados anteriormente. No entanto, tais algoritmos dependem para o funcionamento correto, de uma maneira de avaliar uma determinada situação do jogo, visto que é a partir de um score atribuído a cada configuração de tabuleiro que é selecionado o melhor lance a ser feito.

Visto isso, é importante para a qualidade do software que a função de avaliação consiga representar os diferentes critérios de análise de uma posição em um jogo de xadrez: diferença do número de peças, desenvolvimento das peças, estrutura de peões, etc. Com isso, pode-se imaginar que uma função que consiga traduzir e ponderar tais perspectivas de avaliação nas diversas fases do jogo, seja, ao menos, muito complexa.

4.1
Diferença ponderada do número de peças

A primeira função de avaliação leva em consideração as peças presentes no tabuleiro, as quais é atribuído um valor padrão como o exemplo dos valores utilizados nesse projeto presentes na figura abaixo - os livros de xadrez consideram tais valores como um padrão menos o atribuído aos bispos que é considerado como igual ao do cavalo na grande maioria dos autores, o que com as chess engines atuais vem sendo questionado dando-se uma ligeira preferência para os bispos. Assim, seja $V(p_i)$ o valor da peça i e P o conjunto de peças presentes no tabuleiro, o valor desse termo da função de avaliação é dado por:

$$score = \sum_{p_i \in P} V(p_i) \tag{4-1}$$

Valores das peças.

Dessa forma, pode-se observar que ao rei foi atribuído o valor 100, no entanto, isso foi uma escolha de modelagem da função uma vez que não existem posições sem o rei no tabuleiro. Isso porque o xeque-mate por definição é um ataque ao rei para o qual não existe defesa e não uma captura dessa peça. Portanto, também seria possível simplesmente excluir essa peça da função de avaliação. Outro ponto importante é que essa função atribui valor negativo às peças pretas, isso acontece porque a função é focada para o jogador de brancas. Logo, quanto mais peças pretas existirem no tabuleiro pior é para o jogador de brancas.

4.2 Desenvolvimento das peças

Nessa segunda função de avaliação, não somente a quantidade de peças é levada em consideração, mas também a posição delas peças. A cada peça é atribuída uma matriz 8x8 representando o tabuleiro de xadrez, em que cada elemento (i,j) é um score associado ao posicionamento da peça naquela casa do tabuleiro. Note que esse critério tem limitações para o final do jogo, no qual indica-se o uso de outras matrizes, no entanto a função escolhida não contempla essa distinção entre abertura, meio-jogo e final. Tais limitações tem origem nas diferentes funções das peças ao longo das fases do jogo, o que pode ser ilustrado como o exemplo do rei que na abertura e no meio-jogo tem pouca atividade mas nos finais tem um papel bastante ativo. Assim, baseando-se na implementação do artigo (**?**) temos as seguintes matrizes para as peças brancas (jogador 1).

```
[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
[ -2.0, -3.0, -3.0, -4.0, -4.0, -3.0, -3.0, -2.0],
[ -1.0, -2.0, -2.0, -2.0, -2.0, -2.0, -2.0, -1.0],
[  2.0,  2.0,  0.0,  0.0,  0.0,  0.0,  2.0,  2.0 ],
[  2.0,  3.0,  1.0,  0.0,  0.0,  1.0,  3.0,  2.0 ]
```

```
[ -2.0, -1.0, -1.0, -0.5, -0.5, -1.0, -1.0, -2.0],
[ -1.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -1.0],
[ -1.0,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -1.0],
[ -0.5,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -0.5],
[  0.0,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -0.5],
[ -1.0,  0.5,  0.5,  0.5,  0.5,  0.5,  0.0, -1.0],
[ -1.0,  0.0,  0.5,  0.0,  0.0,  0.0,  0.0, -1.0],
[ -2.0, -1.0, -1.0, -0.5, -0.5, -1.0, -1.0, -2.0]
```

```
[  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0],
[  0.5,  1.0,  1.0,  1.0,  1.0,  1.0,  1.0,  0.5],
[ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
[ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
[ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
[ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
[ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
[  0.0,   0.0, 0.0,  0.5,  0.5,  0.0,  0.0,  0.0]
```

```
[ -2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0],
[ -1.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -1.0],
[ -1.0,  0.0,  0.5,  1.0,  1.0,  0.5,  0.0, -1.0],
[ -1.0,  0.5,  0.5,  1.0,  1.0,  0.5,  0.5, -1.0],
[ -1.0,  0.0,  1.0,  1.0,  1.0,  1.0,  0.0, -1.0],
[ -1.0,  1.0,  1.0,  1.0,  1.0,  1.0,  1.0, -1.0],
[ -1.0,  0.5,  0.0,  0.0,  0.0,  0.0,  0.5, -1.0],
[ -2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0]
```

```
[-5.0, -4.0, -3.0, -3.0, -3.0, -3.0, -4.0, -5.0],
[-4.0, -2.0,  0.0,  0.0,  0.0,  0.0, -2.0, -4.0],
[-3.0,  0.0,  1.0,  1.5,  1.5,  1.0,  0.0, -3.0],
[-3.0,  0.5,  1.5,  2.0,  2.0,  1.5,  0.5, -3.0],
[-3.0,  0.0,  1.5,  2.0,  2.0,  1.5,  0.0, -3.0],
[-3.0,  0.5,  1.0,  1.5,  1.5,  1.0,  0.5, -3.0],
[-4.0, -2.0,  0.0,  0.5,  0.5,  0.0, -2.0, -4.0],
[-5.0, -4.0, -3.0, -3.0, -3.0, -3.0, -4.0, -5.0]
```

```
[0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0],
[5.0,  5.0,  5.0,  5.0,  5.0,  5.0,  5.0,  5.0],
[1.0,  1.0,  2.0,  3.0,  3.0,  2.0,  1.0,  1.0],
[0.5,  0.5,  1.0,  2.5,  2.5,  1.0,  0.5,  0.5],
[0.0,  0.0,  0.0,  2.0,  2.0,  0.0,  0.0,  0.0],
[0.5, -0.5, -1.0,  0.0,  0.0, -1.0, -0.5,  0.5],
[0.5,  1.0, 1.0,  -2.0, -2.0,  1.0,  1.0,  0.5],
[0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0]
```

Matrizes das posições das peças.

Pode-se notar que as matrizes apresentam valores positivos e negativos que indicam posições boas e ruins para se colocar a peça respectivamente. Avaliando a matriz do rei, nota-se que as posições com maior valor são aquelas em que o rei fica após realizar o roque. Isso mostra que essa função de avaliação preza por garantir a segurança do rei, juntamente com a realização do roque, tais conceitos são bem presentes nas fases de abertura e meio-jogo. Ainda sobre a matriz do rei vemos que quanto mais próximo do território adversário menor é o valor atribuído às casas, visto que o rei nessas posições está mais vulnerável a ataques.

Além disso, as matrizes das peças pretas são idênticas as das brancas invertendo-se somente a ordem das linhas.

Assim, seja P o conjunto de peças no tabuleiro, M_i a matriz associada a peça i e l_i e c_i a linha e a coluna dessa peça no tabuleiro, o score obtido por esse critério é dado por:

$$score = \sum_{p_i \in P} M_i[l_i][c_i] \tag{4-2}$$

5
Monte Carlo Tree Search

Até o momento, foram apresentados algoritmos pautados na análise exaustiva das possibilidades do jogo até uma determinada profundidade de busca, o que por um lado garante que a solução encontrada é ótima, mas por outro tem a altura da árvore de busca limitada pela capacidade de processamento dos computadores atuais. Desse modo, o método Monte Carlo Tree Search (MCTS) (MCTS Review), diferente dos demais, utiliza-se de escolhas aleatórias para amostrar diferentes partes da árvore do jogo.

Assim como nos outros algoritmo, cada nó da árvore do jogo representa uma configuração do tabuleiro. No entanto, além disso, agora armazenaremos nos nós outras informações: o número de vezes n que o nó foi visitado, e um valor t que guarda uma estimativa do "valor total" do nó.

Esses parâmetros, cujos valores iniciais são zero, são usados para a seleção de qual em nó será feita a próxima iteração do algoritmo MSTC. Para essa seleção utiliza-se a seguinte fórmula conhecida como *Upper Confidence Bound 1 (UCB1)*:

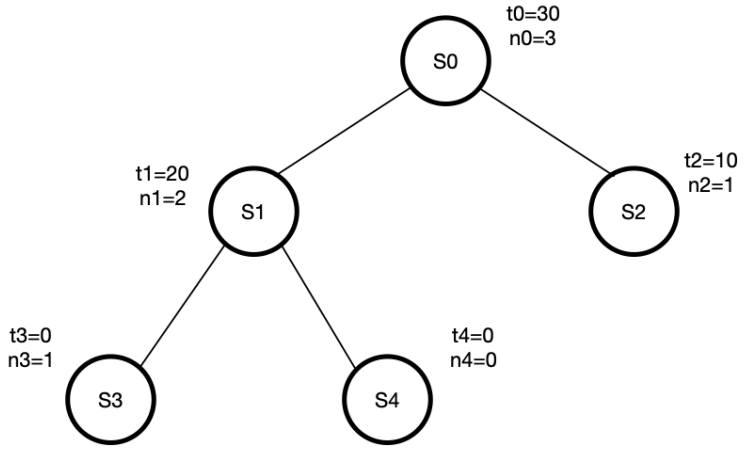
$$\text{UCB1}(s_i) = \overline{v_i} + C\sqrt{\frac{ln(n)}{n_i}} \tag{5-1}$$

na qual s_i é um estado do jogo, $\overline{v_i}$ é o valor médio do nó atual ($\frac{t_i}{n_i}$), C é uma constante, n é o número de visitas do pai do nó atual e n_i é o número de vezes que o nó atual foi visitado. Na nossa implementação foi escolhido o valor $C = 2$.

Uma iteração do MCTS pode ser dividida em quatro fases: seleção, expansão da árvore atual, simulação e retropropagação.

5.1
Seleção

Na fase de seleção, deve-se selecionar uma das folhas da árvore atual. Para isso, deve-se partir da raiz e enquanto não se obtém um nó folha deve-se selecionar dentre os filhos do nó atual aquele que tem o maior UCB1, verificar se ele é uma folha e caso não seja repetir esse procedimento a partir dele.

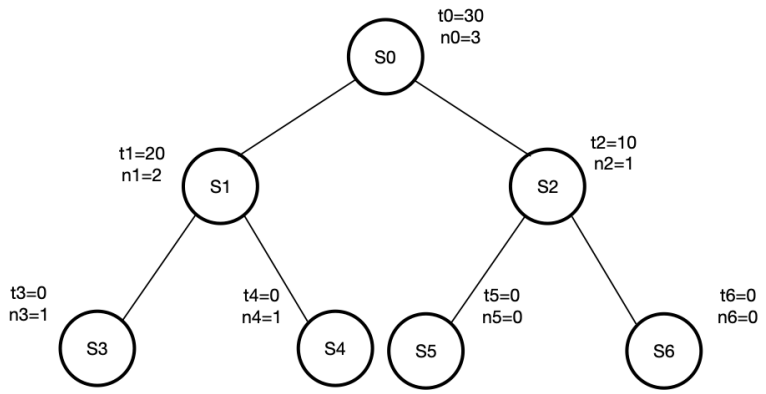


Uma árvore de MCTS genérica.

Assim, na árvore acima deve-se começar pela raiz e calcular o UCB1 para seus filhos S1 e S2. Com isso, obtemos UCB1(S1) = 11,48 e UCB1(S2) = 12,10 e como deve-se escolher o filho com que maximiza o UCB1, escolhe-se o nó S2 como próximo a ser explorado. Como S2 é um nó folha, a fase de seleção nesse exemplo é encerrada.

5.2 Expansão da árvore atual

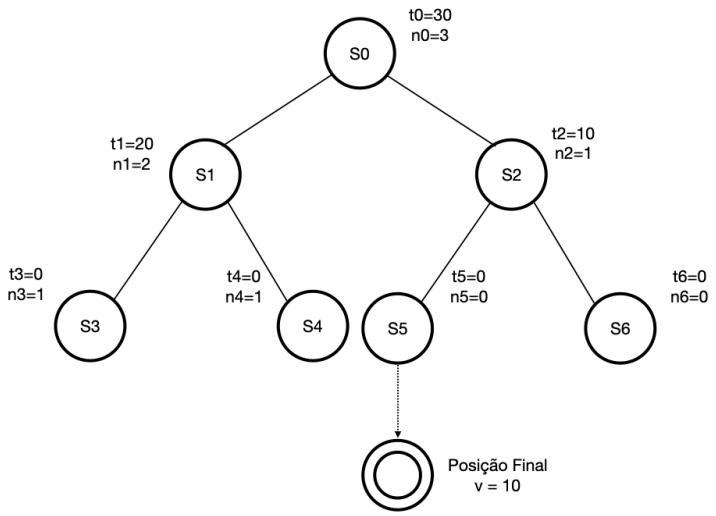
A fase de expansão ocorre quando o nó folha encontrado na fase de seleção já foi visitado pelo menos uma vez. Ela consiste em, a partir do nó atual, selecionar todas as ações possíveis e adicioná-las à árvore e selecionar arbitrariamente um dos nós adicionados para a fase simulação, visto que todos tem o mesmo valor de UCB1. Para a árvore usada na seção anterior, a partir do nó S2 pode-se imaginar que existem duas outras ações que virarão os nós S5 e S6. Como a escolha do nó é arbitrária, pode-se escolher o nó S5 para prosseguir para a fase de simulação. Assim, a árvore MCTS anterior assume a seguinte configuração.



A árvore de MCTS após a expansão.

5.3 Simulação

A fase de simulação ocorre em duas situações: após uma expansão ou quando um nó folha que nunca foi visitado é escolhido na fase de seleção. Nela, partindo-se do nó atual, simula-se aleatoriamente ações do jogo até chegar em uma posição final do jogo. Assim, vemos o valor dessa posição final do jogo e tal valor será usado na fase de retropropagação. Na figura abaixo, ilustramos essa operação considerando que a posição final obtida com essas escolhas aleatórias tem valor 10.

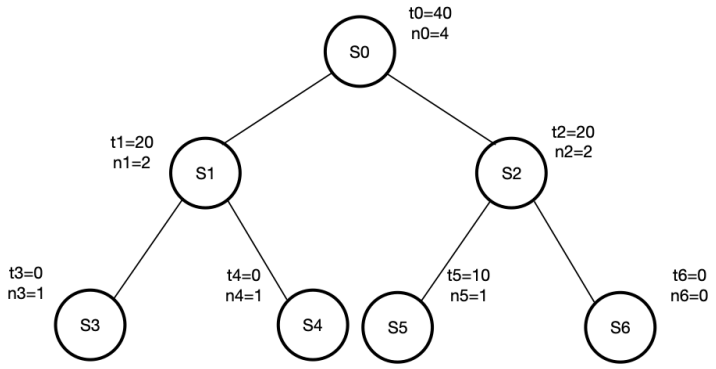


A árvore de MCTS após a simulação.

Note que as ações tomadas na simulação não são adicionadas à árvore, por isso o uso de uma seta tracejada.

5.4 Retropropagação

Nessa fase, o resultado obtido pela simulação é propagado para **todos os ancestrais** do nó N a partir do qual ela foi realizada. Desse modo, para cada ancestral de N deve-se acrescentar um ao número de visitas que ele teve e somar o valor obtido da simulação ao valor total do nó. Seguindo no nosso exemplo, árvore assume a seguinte configuração final.

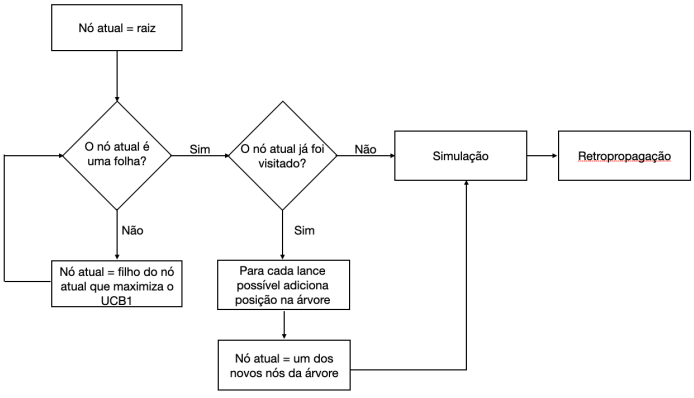


A árvore de MCTS após a retropropagação.

Assim, pode-se notar que os nós S5, S2 e S0 tiveram os seus valores totais e de visitas atualizados.

5.5 Esquema geral de funcionamento do algoritmo

Com as seções anteriores pode-se ter uma noção de como o algoritmo funciona. Dessa forma, para facilitar a compreensão das transições entre as fases do algoritmo o esquema abaixo visa elucidar a anatomia de uma iteração do MCTS.



Iteração do MCTS.

No esquema acima pode-se perceber as fases do algoritmo e identificar, por exemplo, as duas situações que levam a fase de simulação. Tal esquema está concretizado no pseudocódigo abaixo.

Algoritmo 4: Monte Carlo Tree Search

```
EXPANDIR(posição)
    para todo movimento em movimentos possíveis faça
    |  Adiciona o movimento como filho da posição na árvore de busca
    fim
SIMULAÇÃO(posição)
    aux ← posição
    enquanto aux possui lances possíveis faça
    |  Movimento ← sorteia movimentos possíveis
    |  aux ← Executa Movimento
    fim
    return Resultado do jogo na situação aux
MONTECARLOTREESEARCH(posição, turno, iterações)
    raiz ← posição
    EXPANDIR(raiz)
    enquanto iterações ≠ 0 faça
    |  atual ← Seleciona o nó folha que tem o maior valor de UCB1
    |  se atual já foi visitado então
    |  |  EXPANDIR(atual)
    |  |  atual ← Seleciona um dos filhos do nó atual
    |  fim
    |  valor ← SIMULAÇÃO(atual)
    |  Retropropagar o valor e atualizar o número de visitas dos nós percorridos
    |   iterações ← iterações − 1
    fim
    se turno = "brancas" então
    |  return Filho do nó raiz com maior t
    fim
    return Filho do nó raiz com menor t
```

5.6 A implementação do MCTS para o xadrez

Tendo em vista um esquema geral de funcionamento do MCTS, deve-se entender quais mudanças ou detalhes são necessários para a implementação desse algoritmo para a realidade do xadrez. Na fase de simulação, jogar o jogo aleatoriamente até uma posição final pode acabar ficando muito custoso. Desse modo, jogar aleatoriamente até uma certa profundidade e depois avaliar a posição usando a função de avaliação anterior parece ser uma opção com uma performance de tempo melhor. No entanto, fica-se refém da qualidade da função de avaliação da posição.

Desse modo, para tomar uma decisão de qual lance jogar a partir de uma dada posição. Fixa-se a raiz da árvore do MCTS nesta, e realiza-se o número estipulado de iterações do algoritmo. Assim, ao fim da iterações haverá uma árvore de movimentos montada, para escolher o melhor lance basta analizar os filhos da raiz e optar por aquele que possui o maior valor total no caso de brancas ou o menor valor total no caso de pretas. Note que, para cada lance uma nova árvore é computada.

6
Algoritmos de aprendizado de máquina

Os algoritmos descritos até o momento, utilizam-se ou de uma busca exaustiva (Minimax) ou do comportamento assintótico de simulações (Monte Carlo Tree Search) para encontrar qual é o melhor movimento a ser feito na posição. Tais métodos não tem a intenção de tentar identificar se um lance é bom em si, mas sim comparar as diferentes possibilidades e de acordo com algum critério/valor selecionar a melhor opção.

Outra abordagem possível, de certo modo, mais similar com a maneira de um enxadrista pensar é tentar adquirir um conhecimento sobre o xadrez de modo a não necessitar mais de comparações para identificar se um lance é "bom" ou "ruim". Existem diversos métodos baseados em aprendizado de máquina que pode ser utilizados para fazer tal (The elements of Statistical Learning).

Neste trabalho, propomos dois procedimentos para realizar a classificação de lances em "bons" e "ruins"'baseados em *regressão logística* e *redes neurais*. Além da diferença de modelos utilizados, esses procedimentos também utilizam representações diferentes do tabuleiro/lances (*features*) e são treinados de forma diferente. Apresentamos agora mais detalhe sobre eles.

6.1
Procedimento baseado em regressão logística

O estudo de modelos de regressão logística (Introductory Econometrics: A Modern Approach) tem sido de grande utilidade tanto para a predição de dados quanto para explicação dos modelos. Isso acontece porque tais modelos tem boas propriedades estatísticas e garantem uma grande explicabilidade dos parâmetros estimados. Assim, poderia-se tentar entender elementos que compõe um bom lance no xadrez a partir da análise dos parâmetros estimados.

Para treinar uma regressão para classificar lances em "bons" ou "ruins", são apresentados dados rotulados da forma $(estado, lance, classificação)$, onde $estado$ é o estado atual do tabuleiro, $lance$ é um lance possível a partir de $estado$ a ser classificado, e $classificação \in \{bom, ruim\}$ é a classificação "real" desse lance. Por exemplo, a base de dados (Regression Dataset) possui uma coleção de tais dados anotados obtidos de jogos anteriores, onde todos os lances

que o jogador que ganhou a partida executou são classificados como “bons” e todos os lances que o vencedor escolheu não fazer são classificados como “ruins”.

A representação de $(estado, lance)$ que utilizamos foi retirada de (How I coded my own Python chess engine) e consiste em um vetor de 192 posições. As primeiras 64 posições representam as casas do tabuleiro. Se existe uma peça na casa a letra inicial dessa é colocada na posição, se a peça for branca a letra é maiúscula, caso contrário minúscula, se não houver peça ‘None’ é colocado. Tais colunas por serem categóricas recebem um tratamento via multi-hot encoding para as fases de treinamento. As próximas 64 posições (65-128) representam a origem de um movimento, sendo 1 se for a origem e 0 caso contrário. Finalmente, da posição 130 até a 192 temos representados o destino do movimento, adotando-se 1 para caso a casa seja o destino do movimento e 0 caso contrário.

a1	b1	...	h8	from_a1	from_b1	...	from_h8	to_a1	to_b1	...	to_h8

.

Estrutura do vetor de representação.

6.2 Redes neurais convolucionais

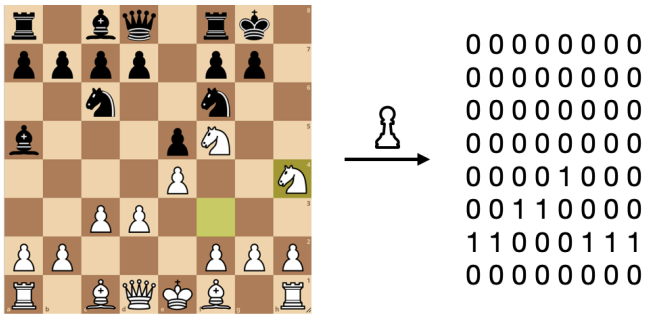
As redes neurais atualmente tem sido cada vez mais usadas para diferentes propósitos: processamento de linguagem natural, identificação de objetos, predição de séries temporais, entre outros. Dessa forma, devido a essa vasta gama de utilidades existem também múltiplas arquiteturas de rede, cada qual serve a um devido objetivo. Nessa perspectiva, as redes neurais convolucionais (A Comprehensive Guide to CNNs) mostram-se bastante efetivas para jogos complexos como Go (CNN plays Go), que é conhecido pelo seu raciocínio lógico abstrato. O resultado desse estudo mostra que redes neurais convolucionais(CNNs) se treinadas com arquiteturas apropriadas e datasets válidos podem capturar muito da experiência de aprendizado humano em tarefas lógicas complexas.

O sucesso da CNN para o caso Go pode ser atribuído ao fato que cada movimento nesse jogo adiciona uma única peça ao tabuleiro o que basicamente significa a mudança de um pixel. Com isso, observa-se que a diferença na representação do tabuleiro após uma jogada é suave, constante e quase sempre ligada aos importantes padrões observados pela rede, o que contribui para a consistência das classificações feitas.

Porém, como discutido anteriormente a avaliação de uma posição no xadrez leva em consideração a iteração de diversas peças, golpes táticos a curto

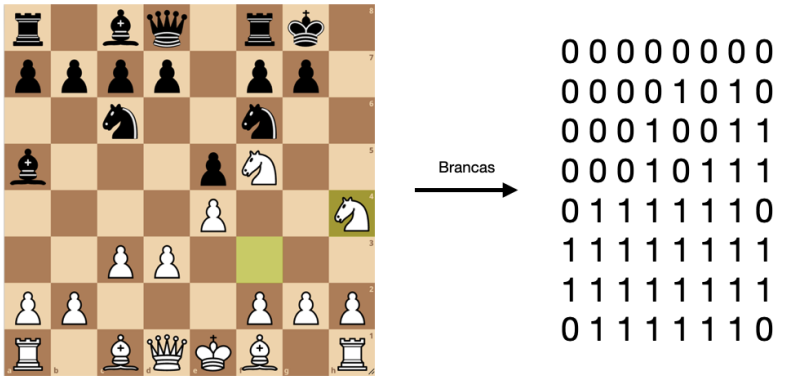
prazo que desencadeiam estratégias para o fim do jogo. Essa característica do xadrez faz com que o reconhecimento de padrões tenha que captar as nuanças da posição e como o posicionamento específico das peças pode levar a vantagens. Outro aspecto diferente do xadrez é que uma jogada no xadrez produz uma mudança bem mais aguda do que no caso do Go o que dificulta ainda mais o reconhecimento de padrões. Tais características mostram que o aprendizado do xadrez é muito centrado no conhecimento do domínio.

Visto isso, com o intuito de facilitar o aprendizado desses detalhes que devem captados para executar-se uma boa avaliação, aqui neste trabalho propomos utilizar CNNs em xadrez da seguinte forma. Primeiro, não vamos classificar um *lance* em "bom/ruim", mas sim queremos computar o *valor numérico* de uma posição, que é extraído da avaliação feita pela chess engine Stockfish. Aqui para representar uma posição utilizamos 12 matrizes 8x8, uma para cada peça (preta/branca), que indica suas ocorrências no tabuleiro. No exemplo abaixo, tem-se a um exemplo de matriz representando a posição dos peões brancos no tabuleiro.



Segunda proposta de representação.

Além disso, temos mais duas matrizes que representam as casas atacadas pelas peças brancas e pretas. Tais matrizes, aparentemente "intrusas", auxiliam a rede neural a medir a atividade das peças de cada lado, ou seja, quanto mais casas marcadas como 1 nessas matrizes mais ativas estão as peças do lado que ela representa. Vale ressaltar que essas matrizes são robustas a diferenças materiais entre os lados, o que quer dizer que independente de um lado ter mais peças que o outro existe a possibilidade das peças do lado em desvantagem material estarem mais ativas o que pode diminuir a desvantagem ou até mesmo demonstrar uma vantagem.



Matriz de casas atacadas pelas peças brancas.

Para o treinamento da rede CNN, geramos posições aleatórias de um jogo de xadrez. Para isso, partindo-se da posição inicial do jogo escolheu-se um número aleatório de lances pertencentes ao intervalo [1, 200]. Assim, após a escolha de quantos lances serão jogados, executa-se uma simulação análoga a usada no MCTS para gerar uma posição. Tal posição, finalmente, é avaliada pela engine Stockfish 14, por meio de uma chamada ao software usando a interface da biblioteca Python-Chess, recebendo um resultado numérico o qual será usado como label. Dessa forma, a rede tentará entender como essa engine avalia as posições. Portanto, para decidir qual é o melhor movimento a ser realizado considera-se todas as posições alcançáveis a partir dos lances possíveis e escolhe-se aquela com melhor valor de avaliação da CNN (no caso das brancas o maior valor e no das pretas o menor valor).

7
Experimentos computacionais e avaliação dos algoritmos

Neste projeto, implementamos os algoritmos descritos acima para jogar xadrez a avaliamos suas performances. Além disso, incluímos dois outros algoritmos aos testes para servir como "cotas inferior e superior" para o desempenho dos algoritmos estudados:

- **Naive:** escolhe-se aleatoriamente dentre os lances possíveis o movimento a ser executado
- **Stockfish 14.1:** chess-engine open-source (Stockfish)

Com isso temos um total de 6 algoritmos testados:

1. Minimax (com Alpha-Beta pruning)
2. Minimax usando CNN como função de avaliação
3. MCTS
4. Regressão Logística
5. CNN (rede neural convolucional)
6. Naive
7. Stockfish

Os experimentos em questão foram executados em um MacBook Pro 2015 com 16 GB de memória RAM e processador de 4 núcleos Intel Core i7 Quad-Core de 2.2 GHz com 256 KB de memória cache L2 por núcleo e 6 MB de memória cache L3 compartilhada. A linguagem de programação utilizada para a implementação dos algoritmos foi Python, e as principais bibliotecas usadas foram python-chess (Python Chess), TensorFlow (TensorFlow) e Numpy (NumPy).

7.1 Descrição dos parâmetros utilizados nos algoritmos

Antes de apresentar os resultados dos experimentos, reportamos os parâmetros utilizados em cada um dos algoritmos. Primeiro, em todos os algoritmos (que precisam de tal) utilizamos como função de avaliação uma média ponderada das funções de avaliação "diferença ponderada do número de peças" descrita na Seção 4.1 e "desenvolvimento de peças" descrita na Seção 4.2 (foi atribuído um peso de 40% à primeira e de 60% à segunda).

Sobre os parâmetros dedicados de cada algoritmo temos:

- **Minimax.** Utilizamos altura máxima da árvore busca como 4, a sempre foi utilizado o Alpha-Beta Pruning para um melhor desempenho em termos de tempo computacional. Além disso, com o intuito de potencializar a técnica de pruning foi utilizada uma técnica de ordenação dos movimentos discutida na seção 3.1 e a função de avaliação descrita acima.
- **Minimax com CNN como função de avaliação.** Utilizamos altura máxima da árvore busca como 4, a sempre foi utilizado o Alpha-Beta Pruning para um melhor desempenho em termos de tempo computacional. Além disso, com o intuito de potencializar a técnica de pruning foi utilizada uma técnica de ordenação dos movimentos discutida na seção 3.1.
- **MCTS.** Como o tempo para encontrar o lance está diretamente ligado ao número de iterações do algoritmo, após alguns testes foram escolhidas 3000 iterações como o padrão usado. Além disso, como jogar aleatoriamente até uma posição final é custoso os número de passos aleatórios da fase de simulação foi limitado a uma profundidade de 6 lances, após isso é feita a avaliação da posição utilizando a função de avaliação descrita acima.
- **Regressão logística.** Foi treinada na base de dados (Regression Dataset) usando 9 batches de tamanho 100000, e tem 192 variáveis dependentes, as colunas da representação do tabuleiro da seção 6.1.
- **CNN.** Foi adotada uma arquitetura que consiste em 4 camadas convolucionais com 32 filtros cada, tamanho de kernel igual a 3 e função de ativação Relu. Além dessas camadas, foi adicionada uma camada para redimensionar os dados (Flatten) e a ela mais duas camadas densas. A primeira tem como tamanho da saída 64, representando as casas do tabuleiro de xadrez e função de ativação Relu. Já a segunda, tem 1 como o tamanho da saída e usa a função sigmoid como função de ativação.

A rede neural foi treinada usando usando 100 épocas, batches com 2048 elementos e usando EarlyStopping.

– **Stockfish.** Foi dado 0.1 segundos para tempos de busca profundidade de busca igual a 1 como uma tentativa de limitar a sua precisão.

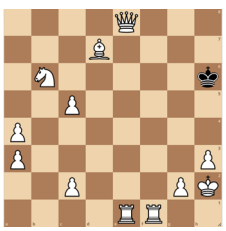
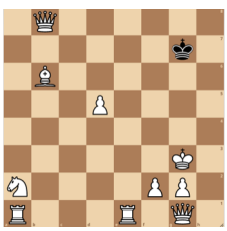
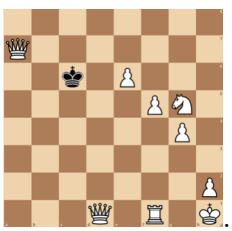
7.2 Competição entre algoritmos

O primeiro teste executado é uma competição em que todos os algoritmos jogam contra todos os outros 20 partidas, sendo 10 de brancas e 10 de pretas. Dessa forma, por exemplo o Minimax jogará 20 partidas contra o MCTS, 20 partidas contra o Naive, 20 partidas contra o Stockfish e assim por diante. Isso porque à exceção do MCTS e do Naive, todos os outros algoritmos utilizados são determinísticos, ou seja, dada uma mesma configuração eles sempre irão reagir da mesma forma. Com isso, assumindo-se a configuração inicial do jogo de xadrez os algoritmos só conseguiriam jogar duas partidas distintas fruto da mudança de lados, isso porque dada uma mesma posição inicial e os mesmos lados os algoritmos jogam sempre os mesmos lances. Para evitar e contornar esse obstáculo, sem dar vantagem a nenhum dos algoritmos as partidas começaram a partir de uma posição, retirada de aberturas conhecidas e cada lado terá a oportunidade de jogar tanto de brancas quanto de pretas a partir dessa posição inicial. Tais posições inicias foram escolhidas a partir de um arquivo no formato PGN que contém todas as aberturas conhecidas, com isso a partir de um script esse arquivo foi transformado para o formato JSON que torna mais fácil a leitura do mesmo pelo programa de testes.

Durante a execução dos testes, notou-se a necessidade de se limitar o número de jogadas das partidas para alguns algoritmos. Isso aconteceu porque eles não demonstraram capacidade de arrematar o jogo em posições finais, o que fazia com que algumas partidas durassem mais que 15 minutos. Tendo isso em mente, quando a partida atinge 100 lances, ou seja, 100 jogadas das brancas e 100 jogadas das pretas é declarado empate forçado. Assim, a tabela de resultados abaixo deve ser entendida da seguinte maneira. A primeira coluna determina o confronto entre algoritmos, as demais colunas referem-se ao desempenho do primeiro algoritmo mencionado ao longo das 20 partidas. Por exemplo, na primeira linha da tabela observa-se o confronto entre a CNN e o Stockfish, com isso podemos ver que a CNN teve 0 vitórias, 0 empates, 0 empates forçados e 20 derrotas nessa duelo.

Confrontos	Vitórias	Empates	Empates Forçados	Derrotas
Minimax+CNN x Minimax	2	0	13	5
Minimax+CNN x Stockfish	0	0	0	20
Minimax+CNN x Naive	20	0	0	0
Minimax+CNN x MCTS	20	0	0	0
Minimax+CNN x Regressão Logística	20	0	0	0
CNN x Stockfish	0	0	0	20
CNN x Minimax	0	0	0	20
CNN x MCTS	6	0	14	0
CNN x Regressão Logística	5	2	13	0
CNN x Naive	10	0	10	0
MCTS x Stockfish	0	0	0	20
MCTS x Minimax	0	0	0	20
MCTS x Regressão Logística	20	0	0	0
MCTS x Naive	20	0	0	0
Regressão Logística x Stockfish	0	0	0	20
Regressão Logística x Minimax	0	0	0	20
Regressão Logística x Naive	0	0	20	0
Minimax x Stockfish	0	0	0	20
Minimax x Naive	20	0	0	0
MCTS(16) x MCTS(6)	0	5	9	6
Stockfish x Naive	20	0	0	0

Ao se analisar o desempenho da CNN nos confrontos com os demais algoritmos, pode-se notar que ela obteve um desempenho sólido, sendo superada somente pelo Minimax e o Stockfish. Um ponto a se destacar é o grande número de empates forçados, o que a primeira vista não parece dizer muito sobre o nível de aprendizado do algoritmo sobre o xadrez. Porém, analisando-se as posições finais das partidas onde se foi declarado empate pode-se notar que o aprendizado das noções fundamentais da abertura e do meio jogo foi bem sucedido , visto que na maioria dos casos a CNN tinha vantagens consideráveis, como por exemplo, duas damas extras, frutos da promoção de peões ou ter capturado todas a peças do adversário como pode-se observar nas seguintes posições.

Empates forçados CNN (brancas).

Como visto nas posições 'finais' das partidas, a rede neural teve uma grande superioridade contra os algoritmos Naive, Regressão Logística e MCTS. O que mostra que embora ela tenha aprendido os conceitos de abertura e meio jogo, pouco foi aprendido sobre os conceitos de fim de jogo como a articulação das peças para forçar xeque-mate. Isso torna-se ainda mais evidente quando observam-se os dois empates da CNN contra a regressão logística.

Empates forçados CNN (pretas).

Dessa forma, usando o vocabulário enxadrístico, a CNN, de pretas, afogou o rei das brancas, ou seja, fez com que não houvesse movimentos legais possíveis para as brancas executarem, contudo sem dar xeque-mate o que pelas regras do xadrez é considerado empate. O que corrobora a falta de coordenação das peças para finalizar a partida. Além disso, atentando-se para o número de lances das partidas em que a CNN saiu vitoriosa pode-se notar que em sua maioria foram partidas curtas (menos de 40 lances), o que em muitos casos ainda pode-se considerar como uma fase de meio-jogo demonstrando-se, portanto, que poucos conceitos de fim de jogo foram aprendidos. Isso se deve a alguns fatores principais: a insuficiência de posições de final de jogo na base de dados, o uso de poucas épocas para no treinamento, e que as chess engines tendem a ter uma precisão menor nos finais de jogos, o que afeta o aprendizado da rede devido as labels dos dados serem geradas por uma dessas engines.

O MCTS mesmo tendo sido superado pela CNN no duelo entre eles apresenta uma melhor capacidade de forçar o xeque-mate, visto que contra a regressão logística, por exemplo, ele obteve 100% de vitórias enquanto a CNN empatou algumas vezes como visto anteriormente. Essa capacidade mostra que a partir de muitas simulações aleatórias se pode, no caso do xadrez, encontrar linhas que consigam forçar uma situação de xeque-mate. Isso, no entanto, não demonstra a precisão desse algoritmo para encontrar o melhor lance para a corrente situação, visto que por realizar várias simulações o lance escolhido foi aquele que apresentou maior número de cenários positivos, o que não exime

a possibilidade de existirem refutações únicas a tal movimento que por essa característica não são previstas pelo método.

Fazendo-se uma análise de sensibilidade sobre o tamanho de passos dados na etapa de simulação pode-se perceber que o aumento do número de passos aleatórios de 6 para 16 resultou em um desempenho pior, como pode ser visto no resultados dos jogos entre o MCTS(16) e o MCTS na tabela. Isso acontece porque com mais passos para frente maior o número de possíveis configurações, e como foi mantido fixo o número de iterações do algoritmo, tal número não foi suficiente para reproduzir um comportamento assintótico.

A regressão logística apresentou o pior desempenho entre todos os algoritmos, mostrando-se incapaz de vencer o Naive. Além disso, em muitas partidas usando-se o Stockfish para analisar a posição final observa-se que o Naive está em uma situação melhor do que a da regressão, o que demonstra que as variáveis explicativas utilizadas e conjunto de dados foram insuficientes para o aprendizado de conceitos básicos do jogo.

O Minimax, de todos os algoritmos implementados, foi aquele que obteve o melhor desempenho, perdendo somente para o Stockfish, o que era esperado antes do início dos testes. Isso porque fazer uma análise exaustiva com uma profundidade de busca de tamanho 4 acaba captando possíveis ameaças e encontrando a melhor resposta para elas, o que não acontece por exemplo com o MCTS que por basear-se numa simulação probabilística acaba optando pelo lance com melhor valor esperado de avaliação e, com isso, perdendo a capacidade de encontrar essas linhas específicas que podem ser decisivas para a partida. Vale ressaltar que a mudança na função de avaliação proposta pela CNN não surtiu em um aumento de desempenho, isso porque como visto anteriormente a CNN tem uma precisão menor na avaliação de finais, o que explica os resultados do confronto entre os dois algoritmos o qual mostrou uma leve vantagem para o Minimax com a função de avaliação do tabuleiro proposta.

Finalmente, os algoritmos adicionais, Stockfish e Naive, cumpriram o papel de limites superiores e inferiores. Uma vez que o primeiro ganhou todas as partidas disputadas e o segundo embora não tenha perdido todas a partidas acabou sendo um importante fator para identificar a capacidade dos adversários.

7.3 Precisão em problemas de mate em 2

Nesse teste, todos os algoritmos foram confrontados com 220 problemas de mate em 2 movimentos e foi observado a porcentagem de acertos dos lances

que levavam ao xeque-mate. Tal tipo de problema é muito comum para o treinamento de enxadristas, e consiste em uma posição na qual sabe-se que um dos lados pode executar um xeque-mate em dois lances. Para isso, a base de dados (Mate in 2 puzzles) que estava no formato de texto contendo o FEN da posição inicial e o PGN da solução (sequência de lances que resulta em no xeque-mate) foi transformada para um arquivo JSON cuja chave é o FEN da posição e o valor é o PGN de resposta.

Como o objetivo do teste era avaliar a precisão foram escolhidos problemas de mate em 2 visto que são os menores em termos de árvore de busca o que agiliza a execução dos testes, sem prejudicar a medida da precisão de cada algoritmo. Para considerar que o algoritmo acertou verifica-se se os lances feitos por ele são iguais ao da sequência descrita na solução (o que acontece na maioria dos casos, uma vez que esses problemas geralmente só tem uma solução), ou se a sequência resulta em xeque-mate. Não há interesse nesses testes em identificar se o algoritmo conseguiria ganhar a partida a partir da posição dada, desse modo, somente são analisados os dois lances feitos por ele.

Dessa forma, após a realização dos testes obtiveram-se os seguintes resultados presentes da tabela abaixo.

Algoritmos	Acertos
Naive	8
Regressão Logística	9
MCTS	21
CNN	44
Minimax	220
Minimax+CNN	220
Stockfish	220

Com isso, podemos notar que o Minimax, por realizar uma busca exaustiva, sempre acaba encontrando a sequência ótima de lances. A rede neural convolucional tem uma porcentagem de acertos de 20%. Isso acontece porque durante o treinamento ela captou mais a capacidade de desenvolver as peças, e conquistar o centro do tabuleiro, que são objetivos nas fases de abertura e meio-jogo. Contudo, ela não adquiriu a coordenação das peças para dar xeque-mate que é mais comum nos finais das partidas. Com 9,5% de acertos o MCTS, figura como terceiro dos algoritmos em termos de acertos. A baixa precisão desse algoritmo se dá ao fato de que ele se baseia em diversas simulações aleatórias para entender se um lance é bom, o que não é suficiente para a resolução desses problemas. Isso acontece porque um lance numa sequência de mate é indefensável, ou seja, não existe movimento do adversário

que evite o fim do jogo. Assim, o MCTS, por se basear nas simulações escolhe o lance que na vasta maioria dos casos é superior, não exime a possibilidade remota de ter uma sequência de movimentos do adversário que "salve" o xeque-mate. Por último, com a mesma porcentagem de acertos que do Naive, que escolhe lances aleatórios, encontra-se a regressão logística. Esse baixo desempenho está atrelado as escolhas das variáveis explicativas que se avaliadas sozinhas não possuem um significado muito claro, o que talvez pudesse ser resolvido adicionando-se interações entre as variáveis. Além disso, o xadrez possui uma vasta gama de óticas a se considerar ao fazer a avaliação de uma posição, o que torna difícil para a regressão captar todas elas.

Referências bibliográficas

[A Comprehensive Guide to CNNs] SAHA, S.. **A comprehensive guide to convolutional neural networks-the eli5 way**, Dec 2018.

[CNN plays Go] CLARK, C.; STORKEY, A. J.. **Teaching deep convolutional neural networks to play go**. CoRR, abs/1412.3409, 2014.

[Game AIs with Minimax and Monte Carlo Tree Search] MUENS, P.. **Game ais with minimax and monte carlo tree search**, Jan 2020.

[How I coded my own Python chess engine] MAI, E.. **Machine learning: How i coded my own python chess engine**, Jan 2021.

[Introductory Econometrics: A Modern Approach] WOOLDRIDGE, J. M.. **Introductory Econometrics: A Modern Approach**. ISE - International Student Edition. South-Western, 2009.

[Kasparov 2018] KASPAROV, G.. **Chess, a drosophila of reasoning**. Science, 362(6419):1087–1087, 2018.

[Knuth-Moore] KNUTH, D.; MOORE, R.. **An analysis of alpha-beta priming**. 2002.

[MCTS Review] SWIECHOWSKI, M.; GODLEWSKI, K. ; ET AL.. **Monte carlo tree search: A review of recent modifications and applications**. CoRR, abs/2103.04931, 2021.

[Mate in 2 puzzles] **25,000 chess puzzles**.

[NumPy] HARRIS, C. R.; MILLMAN, K. J.; VAN DER WALT, S. J. ; ET AL.. **Array programming with NumPy**. Nature, 585(7825):357–362, Sept. 2020.

[Python Chess] `https://python-chess.readthedocs.io/en/latest/`.

[Regression Dataset] MAI, E.. **Chess moves: A collection of 1.6 million chess moves labeled as good/bad**, 2018.

[Shannon Number] WIKIPEDIA CONTRIBUTORS. **Shannon number — Wikipedia, the free encyclopedia**, 2021. [Online; accessed 13-April-2021].

[Stockfish] **Stockfish 14.1**.

[TensorFlow] ABADI, M.; AGARWAL, A. ; ET AL.. **TensorFlow: Large-scale machine learning on heterogeneous systems**, 2015. Software available from tensorflow.org.

[The elements of Statistical Learning] HASTIE, T.; TIBSHIRANI, R. ; ET AL.. **The Elements of Statistical Learning**. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.