Universidade Federal do Rio Grande do Norte
Departamento de Engenharia de Computação e Automação
Controle Inteligente

Jogo da Velha com Min-Max

Aluno: Daniel Silva de Morais Mat.: 20170010123
Professor: Fábio Meneghetti

20 de maio de 2018
Natal/RN

Conteúdo

1 Introdução

Para resolver um problema quando não há computação clara para uma solução válida, nos voltamos para a localização do caminho (Path Finding). Há várias abordagens de busca de caminhos relacionados, como por exemplo árvores de jogos e arvore de buscas. Essas abordagens baseiam-se em uma estrutura comum, ou seja, uma árvore de estados cujo nó raiz representa o estado inicial e as arestas representam movimentos potenciais que transformam o estado em um novo estado. As buscas são desafiadoras porque a estrutura subjacente pode não ser calculada em sua totalidade devido à explosão do número de estados [1].

Assim sendo, há o algoritmo Min-Max, que é o núcleo de várias inteligências artificiais (IA) para construir decisões sobre o melhor movimento. Esse algoritmo encontra a melhor jogada para uma IA em um estado de jogo de dois jogadores, tais como jogos de Damas, Xadrez, Velha e entre outros.

Dada uma posição específica em uma árvore de jogo a partir da perspectiva de um jogador inicial, um programa de busca deve encontrar um movimento que leve à maior chance de vitória (ou pelo menos um empate). Em vez de considerar apenas o estado atual do jogo e os movimentos disponíveis naquele estado, o programa deve considerar quaisquer contra-ataques que seu oponente fará depois de fazer cada movimento. O programa assume que existe uma pontuação de função de avaliação (estado, jogador) que retorna um valor representando a pontuação do estado do jogo na perspectiva do jogador; valores inferiores (que podem ser negativos) refletem posições mais fracas.

A Figura 1 representa a estrutura de uma árvore utilizando o algoritmo Min-Max. Ela faz a avaliação de um movimento usando o algoritmo com profundidade Min-Max 1. A linha inferior da árvore do jogo contém os estados de jogo possíveis, que resultam depois que o jogador faz um movimento. Cada um desses estados do jogo é avaliado do ponto de vista do jogador inicial, e a classificação inteira é mostrada em cada nó. A primeira linha MAX contém o nó cuja pontuação é o máximo de seus respectivos filhos. Do ponto de vista do jogador inicial, este representa a melhor pontuação que ele pode atingir. No entanto, a segunda linha MIN representa as piores posições que o adversário pode forçar no jogador, portanto, suas pontuações são o mínimo de seus filhos. Cada nível alterna entre selecionar o máximo e o mínimo de seus filhos. A pontuação final demonstra que o jogador inicial pode forçar o oponente a um estado de jogo que avalia a 3.

Tendo isso em vista, este relatório tem por objetivo descrever o desenvolvimento de um programa de Jogo-da-Velha utilizando o algoritmo Min-Max. O utilizador do software joga contra o computador, que utiliza o algoritmo

como IA. A variação do nível de dificuldade é feita modificando a profundidade do Min-Max e utilizando de *hard rules*. A ideia de utilizar esse algoritmo é mostrar que podemos conseguir bons resultados, podendo ser com regras, sem ter conhecimento de toda a árvore de jogo, mas conhecendo apenas alguns estados seguintes.

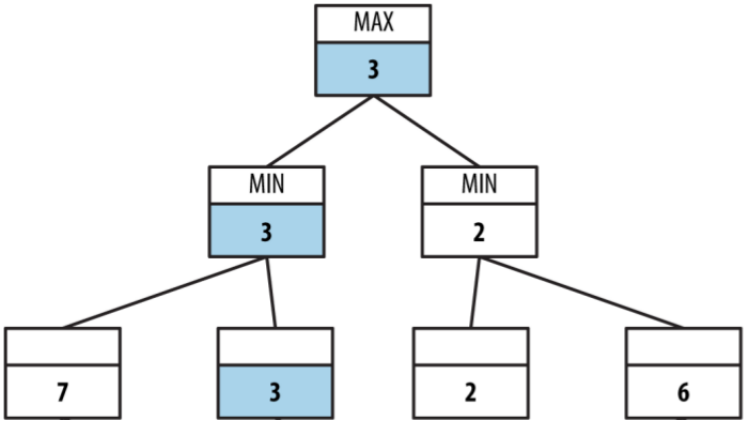

Figura 1: Exemplo de árvore de jogo com Min-Max

2 Descrição

Para o desenvolvimento do programa foi utilizada a linguagem de programação `Python 3` com `PyQt5`. O programa é dividido em 4 arquivos `.py`. Há duas classes, `tabuleiro.py` e `point.py`, que formam a estrutura básica para o jogo, enquanto `jvelha.py` é responsável de fazer as movimentações do jogo e `main.py` possui a interface e controle das fases do Jogo-da-Velha . A Figura 2 a seguir mostra a interface do jogo. Para iniciar o programa, deve selecionar quem começa e o nível. Em seguida, apertar em *jogar*. Para reiniciar, aperte em *jogar*.

Figura 2: Interface do jogo.

2.1 Implementação do Min-Max

No desenvolvimento da função `minimax(tab, prof, player)`, foi utilizada a recurção, de forma que foi possível a criação da "subárvore"e obtenção dos valores sem a necessidade de construir sua estrutura propriamente dita. A função possui 3 parâmetros, `tab`: tabuleiro com o estado atual do jogo, `prof`: profundidade da árvore que deseja alcançar e `player`: jogador atual do estado.

A seguir, temos implementação da função:

```
def minimax(self, tab, prof, player):
    if player == 1:  # 1 PC | -1 HUMANO
        best = [-1, -1, -inf]
    else:
        best = [-1, -1, +inf]
    if prof==0 or len(self.tab.getNone()) == 0:
        score = self.heuristica(tab)
        return [-1, -1, score]

    for item in tab.getNone():  # lista de None points
        tab.setLocal(item, player)
        score = self.minimax(tab, prof - 1, -player)
        tab.setLocal(item, None)
        score[0], score[1] = item.x, item.y
        if player == 1:
            if score[2] > best[2]:
                best = score  # max value
        else:
            if score[2] < best[2]:
                best = score  # min value
    return best  # retorna x y score
```

Foram utilizadas outras funções para complementar `minimax()`, como a função `heuristica(tab)` e `isVencedor(tab, x)` com `x` assumindo 1 ou -1, que são as representação na tabela para o jogador computador e o jogador humano, respectivamente.

A função `heuristica(tab)` é responsável por calcular o valor associado ao "nó"da árvore de jogo. A heurística utilizada leva em consideração a diferença entre o número de alinhamentos possíveis para cada jogador. Para calcular o alinhamento, todas as células vazias da tabela são substituídas, por vez, pelo valor associado a cada jogador e em seguida é calcada a chance de se obter o alinhamento. A seguir temos sua definição:

```
    def heuristica(self, tab):
        score  = self.alinhamento(copy.deepcopy(tab),  1)
        score += self.alinhamento(copy.deepcopy(tab), -1)
        return score
```

As outras funções são bastantes simples e fáceis de entender, então não serão descritas nesse relatório, mas seus códigos serão disponibilizados.

2.2 Estratégia

O programa é composto por 3 níveis de dificuldade, um *fácil*, um *intermediário* e um *difícil*, sendo utilizado o Min-Max e *hard rules*.

2.2.1 Nível Fácil

Para o jogo em nível *fácil* a profundidade é mínima, de forma que o computador tente otimizar apenas a próxima jogada. Desta forma, possibilita que o adversário também tenha chances de ganhar, uma vez que o desconhecimento de toda a árvore de jogo permite criar possibilidades para o adversário. A Figura 3 mostra dois exemplos de situações fáceis.

(a) (b)

Figura 3: a) O computador inicia. b) O adversário inicia.

2.2.2 Nível Intermediário

Para o jogo em nível *intermediário*, foi aumentado o nível de busca do Min-Max, de forma que ele analisa mais uma jogada, que seria a dele. Isso permitiu que alguns jogos em nível fácil fossem corrigidos. Agora o computador consegue fazer alinhamentos que não eram feitos no nível anterior. As Figuras 4 e 5 mostram situações que foram corrigidas com o aumento no nível da busca.

(a) (b)

Figura 4: a) Situação em nível fácil. b) Situação em nível intermediário.

(a) (b)

Figura 5: a) Situação em nível fácil. b) Situação em nível intermediário.

Como pode ser observado nas figuras, as situações são corrigidas de forma que o computador tende a fazer o alinhamento quando possível, ganhando o jogo. Nas situações em nível fácil, o algoritmo preferia bloquear o adversário a ganhar, mas claro que de forma ingênua, pois não é em todas as situações que isso acontece no nível fácil.

2.2.3 Nível Difícil/Imbatível

Para o jogo em nível *difícil*, foram implementadas *hard rules* jutamente com o Min-Max simples, de forma que deixaram o nível imbatível. No máximo, o adversário consegue um empate.

Para que isso fosse possível, foram analisadas 3 situações:

(i) Sempre ganhar;

(ii) Bloquear o adversário;

(iii) Bloquear triângulos.

Na primeira situação, sempre que o PC puder fazer o alinhamento ele fará, tendo vitória. Caso não, ele tentará bloquear todas as chances de alinhamento do adversário. Em seguida, o PC evitará a formação de triângulos, seja em torno da célula central, seja através dos cantos.

As duas primeiras situações são simples de entender. Já a terceira (iii), as lógicas utilizadas foram:

(a) Sempre que o adversário tiver marcado as pontas das diagonais e o centro estiver marcado, o computador marcará uma célula exatamente vizinha a uma das pontas aleatoriamente.

(b) Sempre que as células 2 e 4, 2 e 6, 4 e 8 ou 6 e 8 estivem marcadas pelo adversário, o computador marcará em 1, 3, 7 ou 9, respectivamente.

Se nenhuma das três situações forem satisfeitas, o Min-Max é acionado. Desta forma, temos a implementação:

```
def hardrules(self, tab):
    #Tentar sempre ganhar
    for item in tab.getNone():  # lista de None points
        tab.setLocal(item,1)
        if self.isVencedor(tab, 1)==True:
            tab.setLocal(item,None)
            return item
        tab.setLocal(item, None)

    #Bloquear o adversario
    for item in tab.getNone():  # lista de None points
        tab.setLocal(item, -1)
        if self.isVencedor(tab, -1) == True:
            tab.setLocal(item, None)
            return item
        tab.setLocal(item, None)
```

```
    #Bloquear triagulos em torno do centro e cantos
    if ((tab.getLocal(Point(0, 0)) == -1 and
        tab.getLocal(Point(2, 2)) == -1) or
        (tab.getLocal(Point(2, 0)) == -1 and
        tab.getLocal(Point(0, 2)) == -1)) and
        (tab.getLocal(Point(1, 1)) == 1):

        return random.choice([Point(0,1), Point(1,0),
               Point(1,2), Point(2,1)])


    if (tab.getLocal(Point(0,1))==-1 or tab.getLocal(Point(1,0))
        ==-1) and tab.getLocal(Point(0,0))==None:
        return Point(0,0)
    elif (tab.getLocal(Point(0,1))==-1 or tab.getLocal(Point(1,2))
        ==-1) and tab.getLocal(Point(0,2))==None:
        return Point(0,2)
    elif (tab.getLocal(Point(1,0))==-1 or tab.getLocal(Point(2,1))
        ==-1) and tab.getLocal(Point(2,0))==None:
        return Point(2,0)
    elif (tab.getLocal(Point(1,2))==-1 or tab.getLocal(Point(2,1))
        ==-1) and tab.getLocal(Point(2,2))==None:
        return Point(2,2)

    return None
```

Desta forma, temos a seguir as Figuras exemplificando o funcionamento no nível difícil.

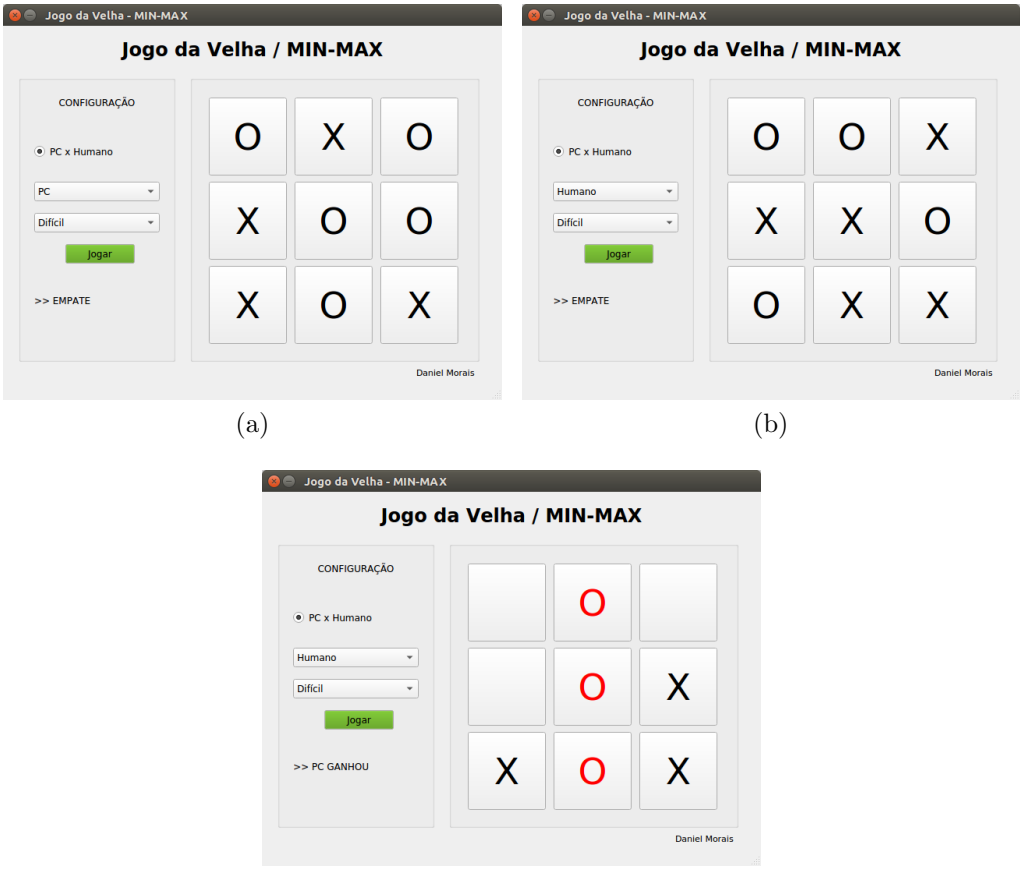

Figura 6: a) Empate. b) Empate. c) PC ganhou.

3 Conclusão

Desta forma, podemos concluir que o objetivo do trabalho proposto foi cumprido, tendo seus resultados validados com os jogos. Além disso, demostrando que é possível encontrar soluções sem a necessidade de ter conhecimento de toda a árvore de jogo.

Referências

[1] George T. Heineman Gary Pollice Stanley Selkow. *Algorithms in a Nutshell*. 2016.