Pesquisa com Adversários: Nine Men's Morris (Grupo 34)

Francisco Ferreira (201605660)
MIEIC
Faculdade de Engenharia da Universidade do Porto
Porto, Portugal
201605660@fe.up.pt

Francisco Friande (201508213)
MIEIC
Faculdade de Engenharia da Universidade do Porto
Porto, Portugal
201508213@fe.up.pt

João Pedro Fidalgo (201605237)
MIEIC
Faculdade de Engenharia da Universidade do Porto
Porto, Portugal
201605237@fe.up.pt

Resumo—Pretende-se com este trabalho implementar um jogo, do tipo tabuleiro, para dois jogadores e resolver diferentes verso˜es desse jogo, utilizando o método de pesquisa MiniMax com cortes alpha e beta e variantes.

Index Terms—Minimax; Mexer; Colocar; Eliminar; Voar;

I. Introdução

Este trabalho foi desenvolvido no âmbito da unidade curricular de Inteligência Artificial, do 3º ano do Mestrado Integrado em Engenharia Informática e Computação, cujo objetivo é aprofundar os conhecimentos previamente adquiridos tanto nas aulas teóricas, juntamente com a abordagem de problemas práticos, sendo que o grupo optou por utilizar a linguagem C++. Sendo assim, o grupo propôs-se a implementar o jogo de tabuleiro Expert Morris com os seguintes modos: humano-humano, humano-computador e computador- computador, apresentando o jogador computador diferentes níveis de dificuldade.

O relatório está dividido nas seguintes secções: Introdução; Descrição do Problema; Formulação do Problema; Trabalho Relacionado; Implementação do Jogo; Algoritmos de Pesquisa; Experiências e Resultados; Conclusões e Perspetivas de Desenvolvimento; Referências Bibliográficas.

II. Descrição do Problema

O jogo Expert Morris caracteriza-se por ser um jogo de tabuleiro para dois jogadores cujo objetivo é reduzir o número de peças do jogador adversário para 2 ou então fazer com este seja impossibilitado de movimentar qualquer uma das suas peças no tabuleiro.

O tabuleiro de jogo está ilustrado na figura 1 em baixo.

O jogo consiste em duas fases: uma primeira fase em que cada jogador tem de posicionar cada uma das suas 9 peças no tabuleiro alternadamente conhecida como "Positioning" e uma segunda fase em que o jogador movimenta as suas peças para posições adjacentes caso estas não estejam já a ser usadas, designada de "Moving".

O seu objetivo a curto prazo é formar blocos de 3 das suas peças(designados de mills), quer na vertical quer na horizontal, o que lhe permite remover uma das peças do adversário à escolha, de forma a chegar às condições de vitória.

III. Formulação do Problema

I. Representação do estado:

A representação do estado do jogo é feita através de uma matriz de inteiros, 0, 1, 2, -1, -2 e -3. 0 (zero) representa espaços vazios que são válidos para se colocar peças, 1 jogador 2, -1 representa um sítio do tabuleiro em que não pode ser colocada nenhuma peça, representando uma união entre posições na direção horizontal, -2 é homólogo a -1 mas na direção vertical e -3 representa um local em que não pode estar nenhuma peça e que não faz nenhuma união entre posições (posição central da matriz).

Representação do estado inicial:

[[0, -1, -1, 0, -1, -1, 0],

[-2, 0, -1, 0, -1, 0, -2],

[-2, -2, 0, 0, 0, -2, -2],

[0, 0, 0, -3, 0, 0, 0] ,

[-2, -2, 0, 0, 0, -2, -2],

[-2, 0, -1, 0, -1, 0, -2],

[0, -1, -1, 0, -1, -1, 0]]

Decidimos utilizar uma matriz quadrada e com os valores

-1, -2 e -3 a indicar posições inválidas para ser mais fácil de verificar se uma peça se está a mexer corretamente e facilitar a impressão do tabuleiro de jogo e uso de algoritmos de pesquisa.

Teste Objetivo:

O teste objetivo tem duas partes, pois um jogador pode ganhar de duas formas distintas. Um jogador pode ganhar caso:

1) Consiga reduzir o número de peças do oponente para menos que 3.
2) Consiga fazer com que todas as peças do oponente não se consigam “mexer”.

Ambas as formas, porém, requerem que todas as peças a jogar tenham sido “colocadas” no tabuleiro.

Operadores:

-Operador “colocar”:

Pré-Condições: O jogador ainda não ter colocado todas as 9 peças no tabuleiro, isto não quer dizer o jogador não ter 9 peças em jogo, pois estas já podem ter sido eliminadas, e não haver nenhuma peça ainda na posição escolhida.

Efeito: É adicionada uma nova peça do jogador a jogar no local selecionado.

Custo: 1.

-Operador “mexer”:

Pré-Condições: O jogador já ter colocado todas as suas 9 peças em campo, a peça selecionada ser sua, o local para onde se quer movimentar esta peça estar livre e ser uma posição adjacente à anterior.

Efeito: A peça é movimentada para o local ocupando essa posição e deixando a anterior livre.

Custo: 1.

-Operador “voar”:

Pré-Condições: O jogador já ter colocado todas as suas 9 peças em jogo, este ter um número exato de 3 peças em campo, a peça selecionada ser uma sua e a posição para onde é suposto esta ir ser uma posição livre. A diferença deste operador para o “mexer” é que a peça pode ir para qualquer posição livre do tabuleiro.

Efeito: A peça deixa a sua antiga posição livre e passa a ocupar a nova posição.

Custo: 1.

-Operador “eliminar”:

Pré-Condições: Após o jogador executar uma jogada, “colocar”, “mexer” e “voar” se esta peça e outras duas do mesmo jogador estiverem em linha, esta linha apenas pode ser horizontal ou vertical e apenas podem estar separadas entre si pelo símbolo -1

Efeitos: O jogador que fez 3 em linha pode eliminar uma peça do adversário, esta peça do adversário não pode estar pertencer a um conjunto de 3 em linha do inimigo, pois estas não podem ser eliminadas.

Custo: 0.

IV. Trabalho Relacionado

- https://github.com/miguelgazela/nine-mens-morris
- https://www.mastersofgames.com/rules/morris-rules.htm
- https://www.geeksforgeeks.org/minimax-algorithm-in- game-theory-set-1-introduction/
- Simona-Alexandra PETCU, Stefan HOLBAN "Nine Men’s Morris: Evaluation Functions", Politehnica Univer- sity of Timisoara , Suceava, Romania, May 22-24, 2008
- Stephanie E. August, Matthew J. Shields "Adversarial Search Nine Men’s Morris, Minimax, and Alpha Beta Pruning Search Algorithms", Loyola Marymount Univer- sityOctober 25, 2015, tails module

V. Implementação do Jogo

O jogo foi implementado em C++, sendo o display do estado do tabuleiro feito através do terminal, tal como toda a jogabilidade e seleções de menu, através de impressões e leituras. O display do tabuleiro é feito sempre que há uma alteração no mesmo, nisto inclui-se mover uma peça, voar uma peça, eliminar peças ou colocar uma peça, ou seja, para um operador qualquer.

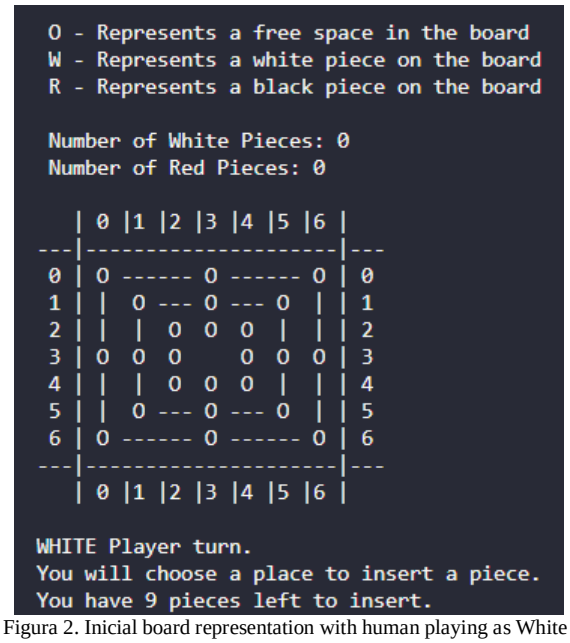

Figura 2. Inicial board representation with human playing as White

Algumas das restrições do jogo são:

- O jogador/computador não conseguem mover nem colocar peças tanto nas ligações entre posições válidas (cf Figura 3.), como fora do tabuleiro.

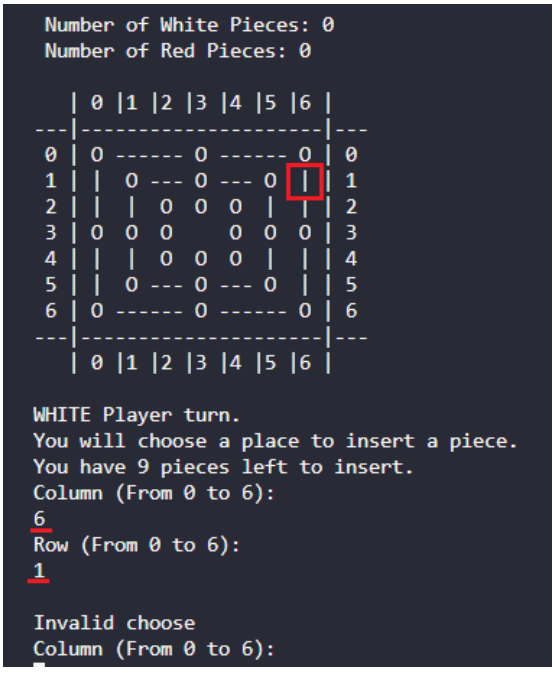

Figura 3. Invalid pick from user

- Quando um jogador apresenta 3 peças em linha (posições adjacentes na mesma direcçao, com ligações válidas ente elas), poderá remover uma peça do adversário.

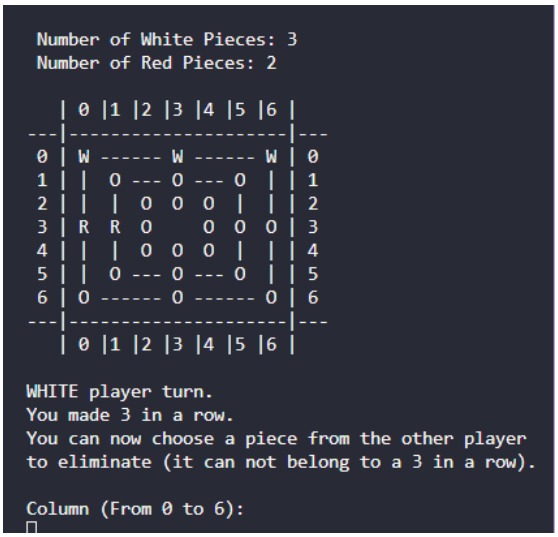

Figura 4. Remover peça do oponente

- Quando ambos os jogadores já houveram jogado as 9 peças, poderão mover as suas peças (cf Figura 5) ou, no caso de um jogador ter exactamente 3 peças, este poderá voar a sua peça (cf Figura 6).

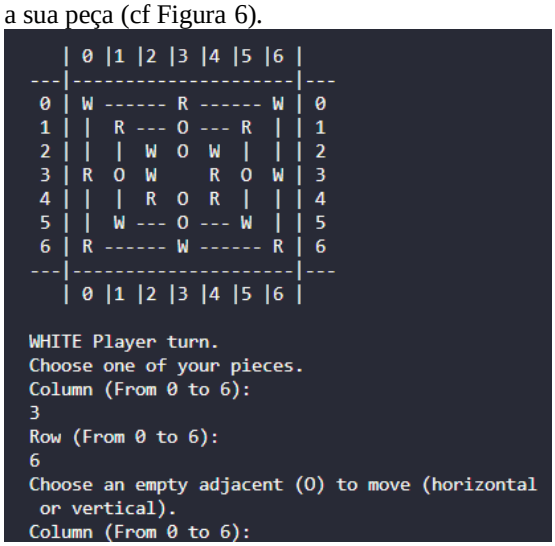

Figura 5. Mover peça

Figura 6. Voar peça

VI. Algoritmos de Pesquisa

Foi implementado um algoritmo de Minimax com cortes alfa-beta da seguinte forma:

minimax(Game game, int depth, int alpha, int beta, int maximizingPlayer, int currPlayer, int agent)

o game representa o estado do jogo, depth o nível contando de cima, alpha o valor de alpha, beta o valor de beta, maximizingPlayer o caracter representativo do jogador que vai escolher sempre a melhor jogada, currPlayer o jogador que vai executar a próxima jogada e agent a função de avaliação a ser utilizada.

A função começa por verificar se o jogo está acabado, ou seja, se um jogador perdeu, ou se o valor de depth atinge 0, retorna a avaliação do estado do jogo. Em seguida verifica se é o turno do maximizingPlayer, pois, se for, vai escolher a jogada com melhor valor, caso contrário a com menor valor. Dentro de ambas as verificações é verificado se é suposto colocar uma peça ou mover, se for para colocar vai criar um vetor com tamanho 2 com o valor da jogada e onde vai inserir a peça; caso seja para mover uma peça o vetor irá ter tamanho 3, em que o primeiro elemento é o valor, o segundo a peça selecionada e em terceiro para onde vai ser movida. Para ambos os casos, inserir ou mover, é verificado se faz 3 em linha, se isto acontecer vai ter de eliminar uma peça, e vai correr todas as peças que pode eliminar e criar vetores com mais 1 de tamanho em relação ao que foi dito anteriormente, em que os últimos elementos são as coordenadas da peça a eliminar. Depois, é verificado se essa jogada tem maior/menor valor que o maximo/minimo, caso isto aconteca substitui esse pelo seu valor.

Como implementação de cortes alpha/beta, é ainda verificado, para o caso de ser maximizingPlayer. Se o valor da jogada é superior ao alpha, se for para o outro player verifica se é inferior ao beta. Para executar os cortes verifica se o beta é menor ou igual que alpha, se isso acontecer dá break do loop dessas jogadas.

Funções de avaliação:

Nesta representação o 1 representa o maximizingPlayer e o 2 ou outro jogador

```
int evaluation1(Game game, int player)
{


if(game.checkLose(switchPlayer(player)))
        return 1;
    if(game.checkLose(player))
        return -1;
    return 0;
}

int evaluation2(Game game, int player)
{
    int value = 100 * evaluation1(game,
player) + game.getnrPlayerPieces(player) -
game.getnrPlayerPieces(switchPlayer(player
));
    return value;
}

int evaluation3(Game game, int player)
{
    int value = 10 * evaluation2(game,
player) + game.count2inRow(player) -
game.count2inRow(switchPlayer(player));
```

```
    return value;
}
```

VII. Experiências e Resultados

Foram executados vários testes de eficiência do algoritmo para os diferentes agentes de forma a comparar os custos da solução em cada um e os resultados a nível temporal. É de notar que todos os tempos apresentados incluem a impressão dos tabuleiros, para efeitos demonstrativos, de modo a que todos os valores sejam comparados já com a agravante referida. Os testes foram de tempo médio de execução de jogadas e de número de jogadas.

	P1	P2
Agent	3	1
Depth	4	4
Winner		-
Number or plays	12	11
Average time per play (s)	0.152071	0.08414

	P1	P2
Agent	3	2
Depth	4	4
Winner		-
Number or plays	15	14
Average time per play (s)	0.001493	0.027628

	P1	P2
Agent	2	3
Depth	4	1
Winner		
Number or plays	19	18
Average time per play (s)	0.156801	0.000667

	P1	P2
Agent	2	3
Depth	4	2
Winner		
Number or plays	20	19
Average time per play (s)	0.093477	0.002038

	P1	P2
Agent	1	2
Depth	4	4
Winner		
Number or plays	15	16
Average time per play (s)	0.066698	0.083194

	P1	P2
Agent	1	2
Depth	4	1
Winner		
Number or plays	20	21
Average time per play (s)	0.052196	0.000337

Como se pode verificar, a função de avaliação com melhor desempenho é a 3ª, pois ganhou todas, independentemente da profundidade utilizada no Minimax, exceto quando usada a profundidade mínima nesta contra a máxima no oponente (usando agente2).

VIII. Conclusões e Perspetivas de Desenvolvimento

Com este trabalho conseguimos aprender um pouco mais sobre a inteligência artificial, nomeadamente o uso de um algoritmo bastante usado, que é o Minimax e com a execução de testes para verificação de qual o resultado mais eficiente. Relativamente ao desenvolvimento futuro do projeto recorreremos mais uma vez à linguagem de C++, com a perspetiva de desenvolvimento a passar pela elaboração do jogo em multiplayer respeitando as suas regras e condições de vitória e posteriormente pela implementação dos algoritmos.

Referências Bibliográficas

[1] Stuart Russel and Peter Norvig, “Artificial Intelligence: A Modern Approach”, Third Edtition, Pearson Education Inc., 2010, ISBN: 978-0-13-604259-4.

[2] Stuart Russel and Peter Norvig, “AimaCode - Code for the Book Artificial Intelligence: A Modern Approach", 2019, [online], available at: https://github.com/aimacode , consulted on March 2019.